A Platform-Centric UML-/XML-Enhanced HW/SW Codesign Method for the Development of SoC Systems

A Thesis
Presented to
The Academic Faculty

by

Chonlameth Arpnikanondt

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in Electrical and Computer Engineering

Georgia Institute of Technology
May 2004

A Platform-Centric UML-/XML-Enhanced HW/ SW Codesign Method for the Development of SoC Systems

Approved by:

Dr. Vijay K. Madisetti, Advisor

Dr. Russell M. Mersereau

Dr. Sudhakar Yalamanchili

April 6, 2004

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my thesis advisor, Dr. Vijay Madisetti, for his guidance and support over the course of my Ph.D. pursuit. Thank you for your patience. Thank you for believing in me!

I also would like to thank all of the faculty members who have helped to serve on the defense and proposal committees: Drs. Altunbasak, Barnwell, Lim, Mersereau, Ramachandran, L. Wills, and Yalamanchili. My special thanks go out to Drs. Mersereau, and Yalamanchili for their help serving on the Thesis Reading committee, and Dr. Lim for agreeing to serve on the defense committee on such a short notice.

Mohamed Ben Romdhane, Lennon Dung, and Tom Egolf—thank you all for being a good mentor.

Additionally, I would like to thank the staff members at the Center for Signal and Image Processing (CSIP): Charlotte Doughty, Christy Ellis, Kay Gilstrap, Keith May, and Sam Smith. Without their assistance over many different occasions, it would be hard to complete all the requirements for this thesis.

I would love to send a special thanks to the folks at Little Bangkok—yes, all of you! Thank you all for being my friends, and supporters throughout this long and tedious process. I might not have said it before, but I owe you folks a lot! Thank you for helping get me here today.

A special thanks go out to all the members of my family. Thank you for being so patient. Lastly, it's you, HON-HON! Thank you for your love and understanding. You have given me all the confidence in the world! Thank you!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

As today's real-time embedded systems grow increasingly ubiquitous, rising complexity ensues as more and more functionalities are integrated. Market dynamics and competitiveness further constrict the technology-to-market time requirement, consequently pushing it to the very forefront of consideration during the development process. Traditional system development approaches could no longer efficiently cope with such formidable demands, and a paradigm shift has been perceived by many as a mandate.

This thesis presents a novel platform-centric SoC design method that relies on a platform-based design to expedite the overall system development process. The proposed approach offers a new perspective towards the complex systems design paradigm, and is able to attain the desired paradigm shift through extensive reuse and flexibility. It offers a unified communication means for all sectors involved in the development process: Semiconductor vendors can use it to publish their platform specifications; Tool vendors can use it to develop and/or enhance relevant tools such as an architecture selector or a HW/SW partitioner; System developers can use it to efficiently develop the system.

Key technologies are identified, namely the Extensible Markup Language (XML) and the Unified Modeling Language (UML), that realize the proposed approach. This thesis extends XML to attain a standard means for modeling, and processing a large amount of reusable platform-related data. In addition, it utilizes UML's own extension mechanism to derive a UML dialect that can be used to model real-time systems and characteristics. This UML dialect, i.e. the UML profile for Codesign Modeling Framework (UML-CMF), remains compliant to the UML standard, and is useful for real-time modeling in general.

A sub-profile within the UML profile for Codesign Modeling Framework is also developed so as to furnish a means for efficient modeling of platforms, and that can be seamlessly integrated with other real-time modeling capabilities offered by the UML-CMF. Such an effort yields a robust UML-compliant language that is suitable for a general platform-based modeling and design.

Last but not least, this thesis defines a comprehensive requirements specification process, based on UML, that helps capture and analyze informal customers' requirements and transform them to a formal, functional requirements specification. A practical use of the proposed approach is also demonstrated through a powerful case study that applies the approach to develop a digital camera system. The results are comparatively presented against the SpecC approach in terms of cost metrics based on the Constructive Cost Models (COCOMO II.2000).

Chapter 1

Introduction

The semiconductor industry is a very lucrative market. Its sale in the year 2002 alone grossed an approximate 150 billion USD—30 percent of which comes from microprocessors, DSPs, microcontrollers, and programmable peripheral chips [1]. A recent forecast by Dataquest predicts an estimated 168 billion USD in semiconductor sale by the end of the year 2003 [130]. Along with such a huge market share, however, has come an increase in system complexity. It is estimated that by the year 2010 the expected transistor count will approach 3 billion with a corresponding expected CPU speed of over 100 GHz, and transistor density of about 660 million transistors/cm^2 [2]. Consequently, such an increase in complexity will result in an increase in power dissipation, cost, and especially technology-to-market time.

In [2], it is argued that computer products would eventually progress from large, general-purpose, impersonal static forms to portable, personal, flexible, market-targeted forms. Personalization, flexibility, and quick time to market would dictate a quickturn design methodology. Time to market for new architectures would no longer be measured in years, but in months. Design cycle would have to decrease or become the bottleneck for future progress. The time-to-market requirement, coupled with other design constraints such as design flexibility, cost, real-time requirements and rigid form factor (e.g. size, weight and power dissipation) represent a formidable challenge that designers of current systems must overcome. Today the need for a paradigm shift in the design method has become more and more pronounced and demanding.

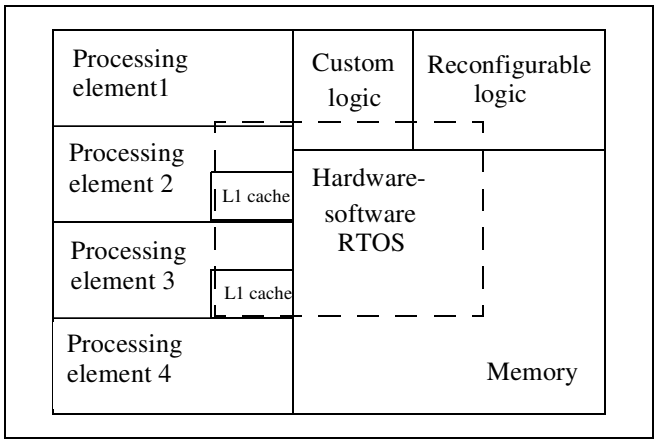


Figure 1.1: Typical System-on-a-chip Architecture

1.1 Typical Systems

A distributed, real-time, embedded system-on-a-chip (SoC) with reconfigurable logic and multiple processing elements sharing a common memory, like that shown in Figure 1.1, has been catching on rapidly and is likely to become very common in the near future [3]. This assertive conclusion stems from many different factors, where the more important ones are listed below.

- In terms of sale volume, embedded processors currently have outsold PC processors by a distant margin [1]. A wide range of applications, especially in the area of wireless and portable devices, has attributed to a tremendous demand for embedded processors. Given a rate of progress in IC technologies today, a *real-time embedded SoC* will find even more suitable applications in the future.

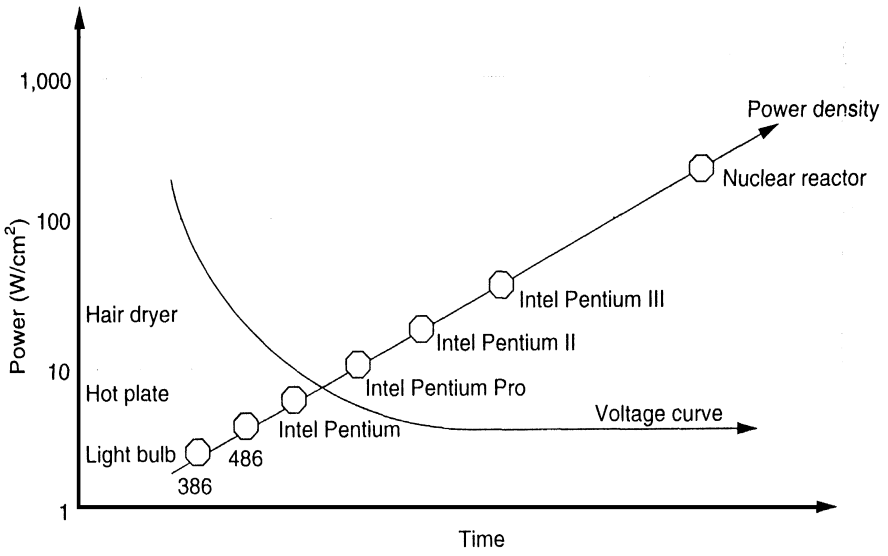


Figure 1.2: Power Density Curve

- Current power density data [4] show that power tends to double every eighteen months. Herring [2] discusses that power dissipation will have a major role in determining what systems would look like in the future. The power density extrapolation as depicted in Figure 1.2 shows that the dissipated heat will eventually approach those of a nuclear power (250 watts/cm^2). With such an increase in power dissipation, the costs associated with packaging and thermal dissipation will dwarf any savings achieved with higher transistor densities. Noise and coupling issues will also become even more difficult to solve as frequencies increase. Consequently a *distributed system* that contains several smaller, slower, heat-manageable processors is going to become a preferable choice to a powerful single-processor system with a hard-to-solve power dissipation problem.

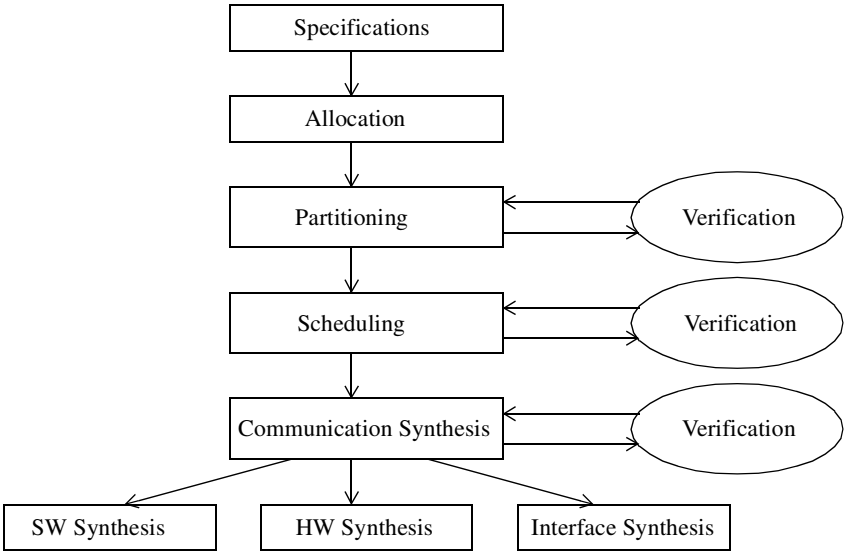


Figure 1.3: Generic Hardware/Software Codesign Process Flow

- As the IC technology progresses from micron to submicron to molecular levels, the cost of producing mask sets will skyrocket [31]. A new 300nm, 0.13µ or 0.10µ high-volume manufacturing plant today cost about $3.5 billion [32]. As such, chip designers will tend to produce a chip suitable for more than one application. Coupled with the fact that performance flexibility and application flexibility are so essential to the success of an embedded system [2, 5], a system with a high degree of *reconfigurability* will not be uncommon in the near future.

This dissertation will focus upon a design method that is more suitable for such systems and their predominant requirements such as minimal time-to-market, real-time environment, flexibility and various form-factor constraints. A paradigm shift in current

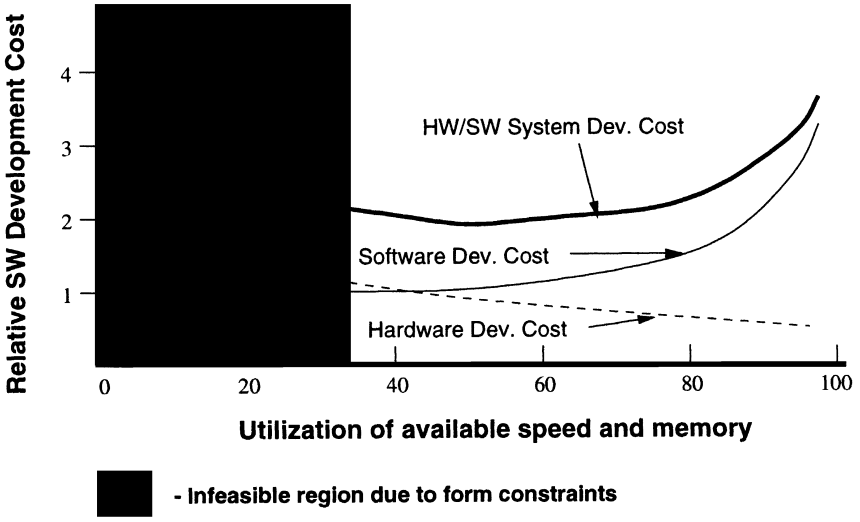
codesign approaches is imperative. Such design approaches that derive from a processor running sequential code have fast become a legacy and uncharacteristically tedious when dealing with the development of systems today. An ever-growing system complexity and pressure to keep the technology-to-market time to the minimum only further dictate the necessity for an improvement in current codesign practices.

1.2 Traditional Codesign Methods

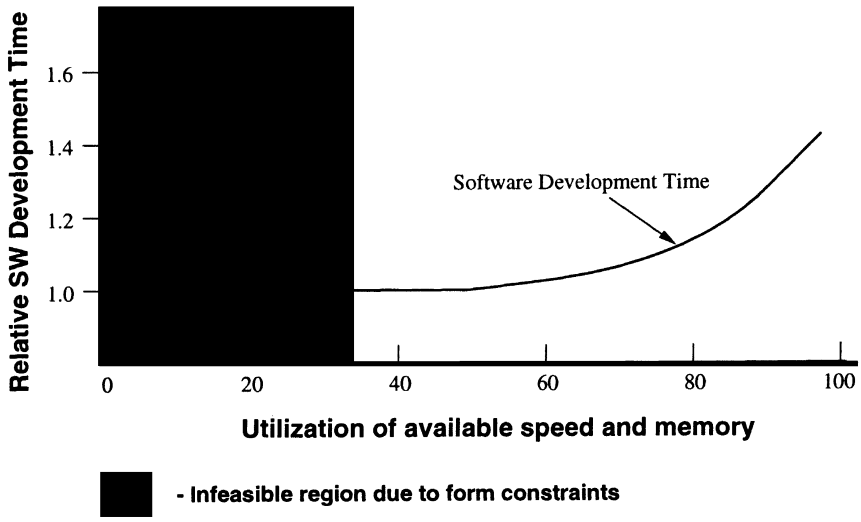
Hardware-software codesign refers to a *concurrent* and *cooperative* design of hardware and software in a system. As shown in Figure 1.3, a process of codesign usually involves four main tasks: *architecture selection*, *hardware/software partitioning*, *tasks scheduling* and *communication synthesis*. The design process flows in a *waterfall* style, typically commencing with a set of specifications. This approach, however, suffers from numerous limitations:

- Use of written requirements
- Lack of a collaborative hardware-software codesign environment
- Limited architectural exploration
- Lack of distributed, real-time support
- Inability to cope with very complex SoC systems

Using written requirements to specify system functions and constraints fosters ambiguity in interpretation and does not facilitate customer interaction, thus leading to increased design iterations and low customer satisfaction. Current industrial practice commonly relies upon designer experience and *ad hoc* techniques to select an architecture and allocate algorithm functionality [6]. This approach severely limits a designer's ability to explore the design space in order to improve the design.



(a)



(b)

Figure 1.4: The effect of hardware constraints on: (a) HW/SW prototyping costs (b) software schedule

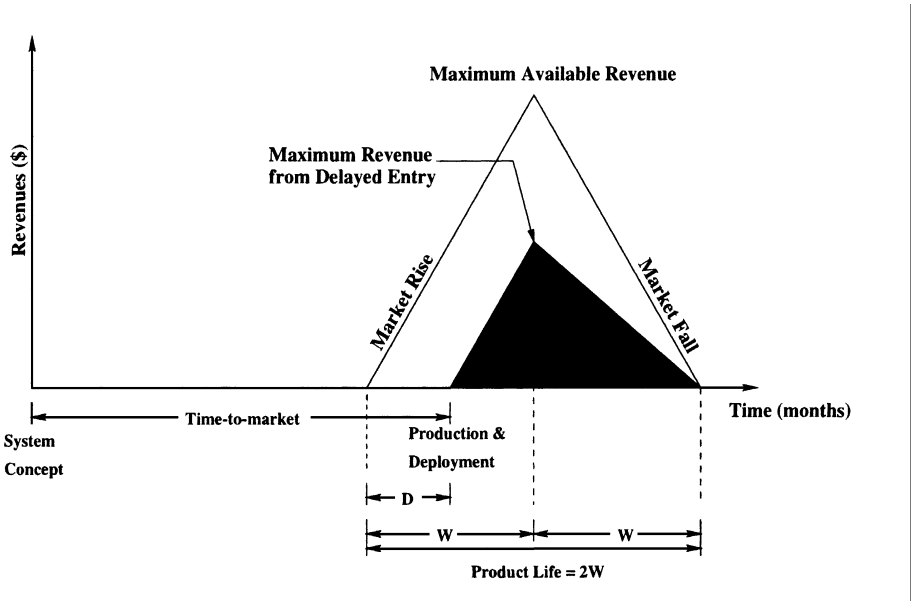


Figure 1.5: A typical time-to-market cost model

To minimize overall system costs, traditional system-level design and test approaches attempt to minimize hardware costs, subject to performance constraints. Nonetheless, these approaches overlook an important characteristic of software prototyping. Parametric studies based on historical project data show that designing and testing software is difficult if margins of slack for hardware CPU and memory resources are too restrictive [6]. Figure 1.4 [7] depicts the software prototyping principle. Graphs (a) and (b) show that software dominates system development cost and time when CPU and memory utilization are high. However as developers reduce resource utilization by adding extra hardware resources, software cost and schedule tend to decrease drastically. SoC and/or distributed-processing chip-building techniques are historically not the cause of production delays. Software availability, support and knowledge base are the bane of product schedules [2].

While constraining the HW/SW architecture is detrimental to software development cost, the corresponding effect on development time can be even more devastating. Time-to-market costs can often outweigh design, prototyping, and production costs of commercial products. Figure 1.5 illustrates a model developed by ATEQ Corporation [8] of the economic impact of delayed market introduction. This simple model quantifies lost revenue from delayed market entry (d) based on the product's total projected revenue and the duration of the market window (W). The unshaded region of the triangular revenue curve signifies this revenue loss. When the product life cycle is short, being late to the market often mean disaster.

Friedrich et.al. [5] reports that, in most embedded applications, the use of general-purpose operating system platforms is not applicable because it is too expensive. Embedded system requirements such as processor performance, memory, and cost are so variable that a general-purpose operating system cannot meet all the needs. Other approaches, such as avoiding an operating system altogether by implementing all the functionalities directly, or by developing an in-house operating system, can limit flexibility and be costly. However, a survey suggests that these approaches are used by 66 percent of the embedded systems in Japan, primarily because a suitable alternative does not readily exist. As such, providing a real-time operating system support with a high degree of reconfigurability becomes very essential to the development of systems today.

Most current HW/SW codesign methods commence with a set of specifications that is usually described by some kind of a formal language. However, as the design process goes from the gate to register-transfer level (RTL), from RTL to the instruction set, and from the instruction set to system level design where formalism often does not exist, problems start to ensue [9]. Furthermore traditional HW/SW codesign approaches still rely heavily on a simulation-based technique, where designers go top-down until getting the design, then describing it in some language, simulating it, and figuring out what needs to be done. With such an approach, decisions made during the design process are usually based solely on designers' experiences. Also traditional codesign practices require more

time to correct constraint violations because they often are checked very late in the design process. Given the complexity and demanding schedules of today's commercial systems, these *blind, design-first-constraint-checked-last* approaches tend to yield a sub-optimal result.

1.3 Tools Integrated Environment

As traditional HW/SW codesign methods fail to address many issues involved in the development of SoC systems today, the first generation of integration tools have emerged that attempts to aid systems developers in coping with an increasing system complexity. Such tools normally run on a single underlying semantic backbone, or a single model of computation [9].

In a tools-integrated environment, the design is usually captured into some kind of a unified representation, e.g. Specification and Description Language (SDL), or Codesign Finite State Machine (CFSM). During the design flow, which virtually still adheres to that of the traditional codesign approach, this unified representation behaves as an input to a collection of different tools, and often change during each design stage to incorporate more and/or new information into the design. The Corsair design flow [10] illustrated in Figure 1.6 is a good representation of this codesign approach. Other tools-integrated approaches include such methods as POLIS [11] and Coware [138].

Although the tools integration method provides for designers a better automated and integrated HW/SW codesign environment, its tendency to adhere too much to the traditional codesign practice proves to be the downside. Its aim to tackle the complexity of today's SoC systems will eventually find a bottleneck in its ineffectiveness to raise a level of design abstraction that is only acceptable to the traditional codesign method. Also, like its traditional precursor, the lack of focus on *flexibility* makes it unattractive for current SoC systems that are so application-oriented and highly driven by technology-to-market time.

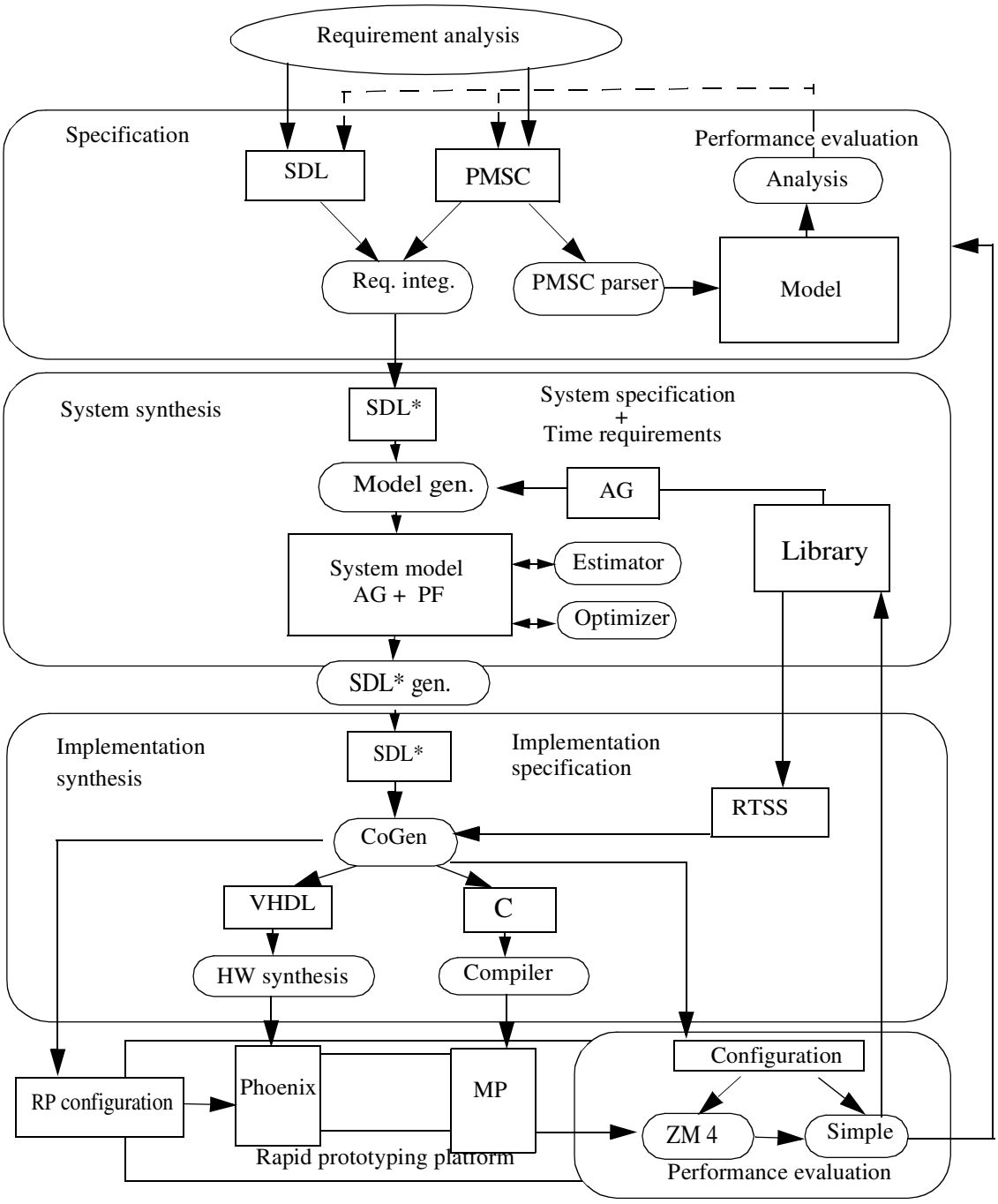


Figure 1.6: The Corsair Design Flow

1.4 Problem Statement

It is evident from the previous discussion that most existing HW/SW codesign methods fail to address satisfactorily the issues involved in the development of real-time, embedded SoC systems. As the system complexity grows in accordance with Moore's law, and as a pressure to minimize the technology-to-market time increasingly overwhelms, the necessity for a paradigm shift in HW/SW codesign practices becomes a mandate [2]. Flexibility dictates that a processor be designed with the absolute highest general-purpose performance possible [9]. More and more systems developers will turn to build a system out of such processors to meet tight time-to-market constraints and flexible application requirements [2]. Constraints must be fully addressed to ensure reliability. In addition, as semiconductor manufacturers continue to define new methods and new ways to build systems, it is desirable for systems developers to be able to incorporate such technological advances into their existing design approaches. Nevertheless, the fact stands that the semiconductor market is too vast and too dynamic for current HW/SW codesign approaches to readily keep pace with its demands. It is also an extremely competitive market, where such a slack often proves costly to systems manufacturers.

Motivated by all of the reasons above, this dissertation attempts to improve upon the traditional codesign method and the tools-integrated approach. It aims to raise a design abstraction level as well as to improve the various vital aspects essential to the success of a quickturn, yet reliable, development of SoC systems.

1.4.1 Technical Problem

As discussed in DeBardelaben [6], a combinatorially significant number of alternatives exist in the implementation of embedded systems. Compounded by increasing complexity, the problem of implementing such systems becomes even more difficult. Embedded system requirements are also invariably diverse and often conflicting. Some key requirements include such attributes as battery life, portability, security, connectivity, user interface, application compatibility, universal data access, and cost. This list of requirements pre-

sents an enigma for the CPU and system developer. While battery life, portability, and cost require simple, application-specific solutions, universal data, security, and user interfaces require the ability for higher performance. These requirements call not only for specific digital performance requirements, but also for specialized analog capability to permit better interaction with the analog-centric human user. CPU and system optimization for one set of requirements will cause unacceptable design trade-offs in other areas. For example, architecture cannot be designed solely for high-overhead, general-purpose performance, or it will sacrifice battery life, portability, and cost.

The objective of this thesis is, therefore, to develop a systematic approach and guideline that can be used as a design framework to assist system developers in:

- *carrying out the SoC design with quickness and correctness,*
- *exploring architecture and design space so that optimal decisions can be made, and*
- *dealing with an ever-growing complexity.*

The efficiency of this research is to be comparatively evaluated using the Constructive Cost Modeling technique, COCOMO II.2000 [19].

1.4.2 Technical Challenges

Designing a distributed real-time embedded SoC is a difficult task by its own right for there are often many conflicting requirements to be reckoned with. To support system developers, however, with an efficient codesign method that aids a wide spectrum of such design can prove even more difficult. There are numerous technical challenges facing researchers that attempt to do so. Some of the more formidable tasks are listed as follows:

1. Determining a suitable approach to facilitate the system developer in dealing with an invariably diverse set of requirements as well as the increasing complexity, while improving upon the technology-to-market time,

2. Developing a design environment that fosters the use of a wide variety of state-of-the-art design tools and techniques, and is adaptive to frequent technological changes,
3. Developing a method that has a good appeal towards semiconductor vendors, EDA tool vendors and system developers.

The complexity of these tasks are primarily attributable to the following factors:

- *Large size of the design space*. A combinatorially significant number of architectural and functional alternatives exist in the implementation of embedded SoC systems. Available components often vary in cost, performance, modifiability, reliability, power, size, and design effort. In addition, there are a lot of communication elements (buses, crossbars), communication protocols (Bluetooth, PCI), and interconnect topologies (ring, linear, mesh, tree) to choose from. Further compound by various combinations of requirements, the design space that system developers must explore become enormously expansive.
- *Complexity and constraints imposed on design time and cost*. The year 2002 has seen information appliances outsold PCs by a wide margin [1]. This new market encompasses small, mobile, and ergonomic devices that provide information, entertainment, and communications capabilities to consumer electronics, industrial automation, retail automation, and medical markets. These devices require complex electronic design and system integration delivered in the short time frames of consumer electronics. The system design challenge of at least the next decade is the dramatic expansion of this spectrum of diversity and the shorter and shorter time-to-market window [13].
- *Tools and techniques*. Vissers [9] argues that semiconductor manufacturers will design systems, rather than system houses design processors. The semiconductor sector is going to do more codesign, with a lot of knowledge that was previously at the system house being either handed off to, or moving towards the semiconductor

company. As such, semiconductor manufacturers will tend to define new methods and new ways to build systems, and the tools for system design will need to be based on the same design approach. Given such a trend, the design method must be able to provide a unified environment for tools and techniques from both the semiconductor and the system houses.

- *Necessity for a paradigm shift*. Traditional codesign approaches is no longer a viable choice for handling the complexity of today's embedded SoC systems. For example, system developers using traditional codesign approach often make design decisions blindly and *a priori* for there is a lack in the availability of presimulated data and/or cost-estimation tools. Techniques such as standardization, cosimulation, coverification, cosynthesis, reuse, etc. need a refreshing re-examination. Systems complexity dictates that a design abstraction level be raised higher. The need for a paradigm shift in codesign approaches becomes eminent. A new method, to be useful, must satisfactorily address these requirements associated with current embedded SoC systems.

- *Disagreement on a standard practice*. While adhering to a standard design practice and/or a standard set of tools actually helps, it is extremely unlikely that there will ever be a unanimous agreement on any one design standard. Marketing strategies, legacy designs and many other opposing factors are often weighed in heavily. Consequently, many excellent standard and non-standard design approaches will continue to co-exist. To make use of them to the fullest, the design method must act like a system of such tools and techniques that allows them to work in unison within the same environment. This integration effort will also prove to be difficult.

1.4.3 A Solution to the Problem

To effectively address the issues in codesign, developers have to raise a design abstraction level so as to better foster extensive reuse *and* make a better use of cutting edge tools and techniques. In addition, requirements must be systematically taken into account to better

reflect the various needs and constraints that are pertinent to the systems development. This thesis presents a design approach, called a *platform-centric SoC design method*, that is motivated by the systems development model as depicted in Figure 1.7.

The development model in Figure 1.7 views the systems development environment as consisting of three separate domains: a *reusable object* domain, a *platform* domain, and a *product design* domain. The reusable object domain provides a library (or libraries) of *logical* platform-compliant tool and component entities, whose *physical* whereabouts could virtually be anywhere accessible through their logical counterparts in the library. Such use of a logical library within the reusable object domain allows various forms of platform-compliant tools and components to be uniformly abstracted and represented, and pre-characterized data compactly modeled and efficiently utilized in the platform and the product design domains. Chapter 4 of this dissertation details such a logical library, called a *library of platform objects* (LPO), that constitutes the reusable object domain. This thesis utilizes the Extensible Markup Language (XML) to model the logical LPO database. XML permits the power of many existing Internet technologies to be harnessed that could potentially lead to an efficient exchange of data—be it knowledge, application programs, or design components. When carefully designed, it is also possible for a reusable platform object and/or application program to insert itself into a reusable object domain as an LPO module, thus making the library *scalable* and as expansive as the Internet itself.

On the other hand, the platform concept [96] helps expedite the design process by reducing the degree of freedom in the development of SoC systems, without absolutely relinquishing system flexibility and performance. The platform domain comprises such platforms that are tailored for various specific purposes, e.g. workstation, low-power handset, VDO, or multimedia. For each platform, the platform vendor would normally also specify and/or supply compatible tools and components in the reusable object domain that can be used to develop the final product.

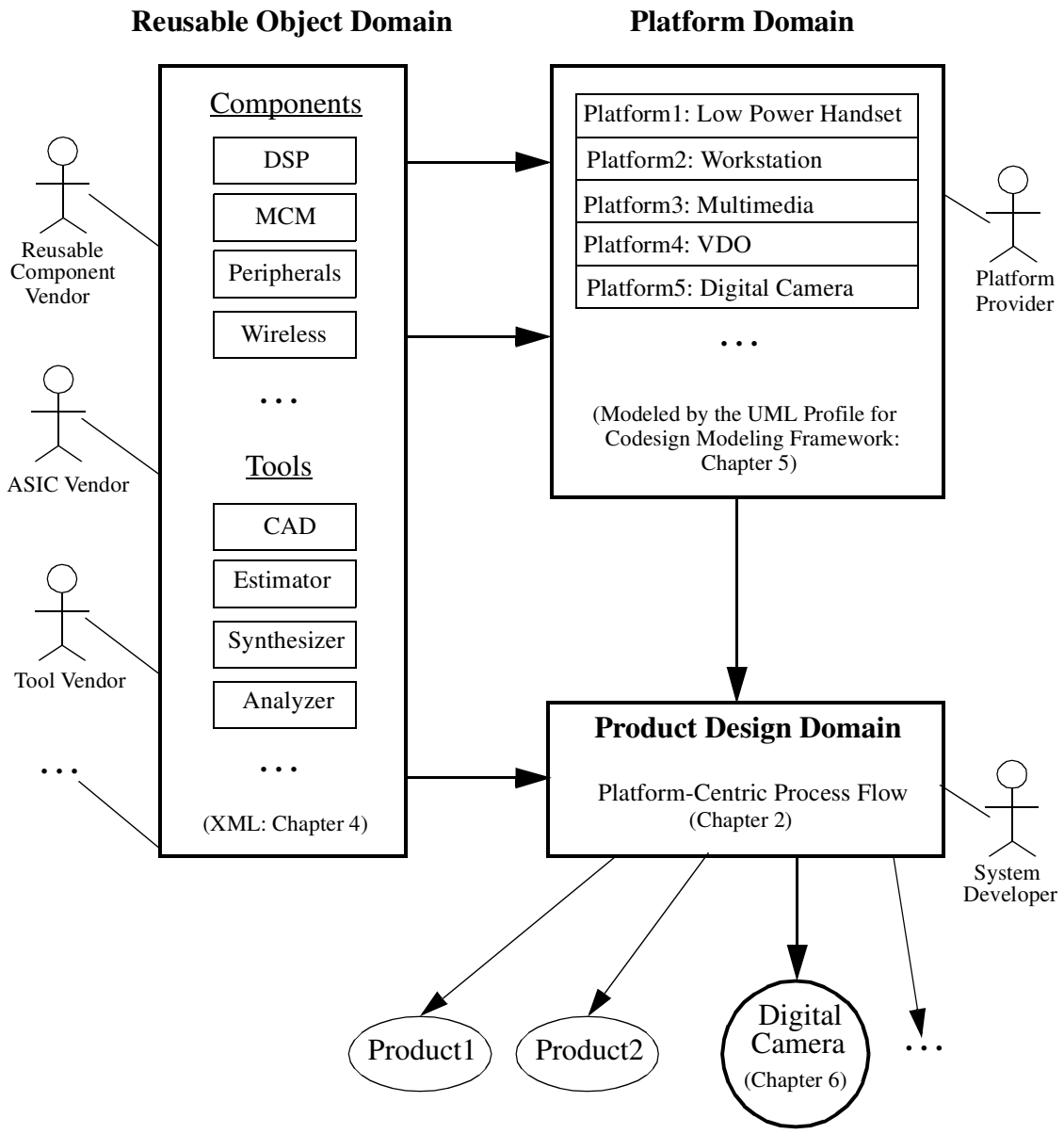


Figure 1.7: The enhanced system development model. Such a model provides a basis for the proposed approach.

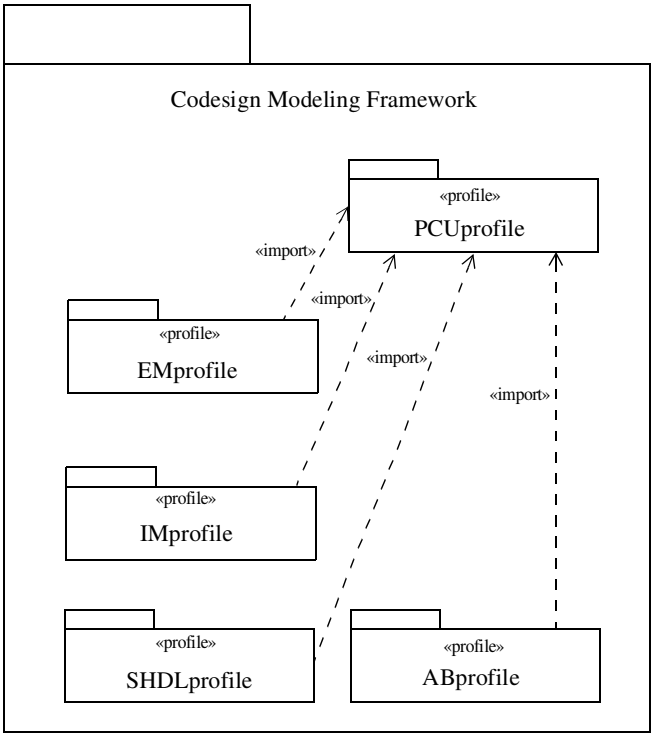


Figure 1.8: The UML Profile for Codesign Modeling Framework (*see Chapter 5*).

The system development phase that renders the final products takes place in the product design domain. Within this domain, the system developer employs the platform in the platform domain and the platform-compliant tools and components in the reusable object domain to derive the target architecture—and eventually the final product. The product design process is described in detail in Chapter 2; it is then applied to develop a simplified digital camera system as an application case study in Chapter 6.

The Unified Modeling Language (UML) plays an integral role. By itself, UML furnishes modeling capabilities for handling real-time requirements in software systems.

Chapter 5 of this thesis further enhances such capabilities via UML's stereotypes and tag values such that, by using the proposed approach, (1) HW design and HW/SW codesign are robust and possible, (2) platforms could be modeled efficiently, (3) the system developer could conveniently model interrupts and exceptions—the characteristics that are vital to most real-time embedded systems, and (4) the enhanced UML for the proposed approach would furnish an interface facility to the reusable object domain, as well as the product design domain. The extended UML elements provided for by this thesis are packaged together in the *UML Profile for Codesign Modeling Framework* as depicted in Figure 1.8, and detailed in Chapter 5.

The platform-centric SoC method is aimed at the design of today's SoC systems with emphasis on real-time, embedded systems. The approach provides a guideline and a SoC design environment that promotes an integration of state-of-the-art tools and techniques necessary for the development of the systems. It renders a new and better perspective towards codesign approaches, while also raising a level of design abstraction. Because the configurable platform objects are designed off-cycle, they contribute to a general improvement in development time. By incorporating their usage, the overall method strikes a balance between total design flexibility and minimal time-to-market.

Within a platform-centric environment, timing behaviors and other constraints (e.g. size, weight) are more predictable. Detailed functional specification derived in the analysis phase will be *mapped* directly to the target architecture, which, in turn, is constructed using platform-compatible hardware and software components, much like how a personal computer system is built by selecting from a menu of different available options. At the core of the platform-centric approach lie the Unified Modeling Language (UML) and a library of platform objects (LPO) that, together, effectively raises the design abstraction level and promote faster development time without incurring additional costs. The approach also permits a seamless integration of tools and techniques that aid analysis and synthesis processes as well as design automation.

Borrowed from the software engineerings field and adapted to better suit the requirements for the development of real-time embedded SoC systems, UML represents a force that drives the development process flow right from where system requirements are analyzed and captured until the desirable implementation model results. Use of UML as a unified representation of the system under development eases the constraints processing and requirements analyzing processes. Its object-orientedness allows complexity to be effectively handled. UML uses its own constraint capturing mechanism and the supplementary Object Constraint Language (OCL) in dealing with a wide variety of constraints. This, in effect, along with the UML modeling capability serve as the underpinnings for the derivation of the platform-independent functional specifications with timing requirements. The UML profile for schedulability, performance, and time specification [29], while has yet to be fully standardized, is useful for modeling real-time systems for analysis and synthesis purposes. The semi-formal nature of UML practically bridges the analysis-synthesis gap to yield a better design flow. UML framework for hardware and software unified modeling of real-time, embedded systems will be presented.

The library of platform objects (LPO), on the other hand, provides a collection of system platforms, i.e. *platform objects (PO)*, that are further governed by a set of rules and requirements specific to the proposed platform-centric SoC design method. Each platform object represents an abstraction of a common *configurable* architecture along with its related components, and tools and techniques specific to that platform. The eXtensible Markup Language (XML) can be used to provide a concrete facility for realizing the LPO and managing the complexity resulted from a huge database of extremely diverse LPO modules. XML also promotes information interchange which can culminate to the virtually unbounded sharing of platform object member modules. Furthermore, there exists an extensive collection of public-domain and/or open-source tools for XML and other related technologies that can be used to enhance XML's capability.

1.5 Organization of Dissertation

In this chapter issues in codesign method are introduced. The chapter briefly describes the technical challenges facing system developers and summarizes the proposed solution to the problem. The remainder of this dissertation presents a more thorough examination on the problem and the proposed approach.

Chapter 2 describes the proposed platform-centric SoC design method in detail. It illustrates the design flow and discusses each main step in the design process. Definition of a platform as originally defined by Sabbagh [96], as well as the platform-based and platform-centric design approaches, are presented. The chapter concludes by comparing the proposed approach with previous related work.

Chapter 3 lays out the technological background for the proposed SoC design method. Whereas the platform technology is discussed in Chapter 2, this chapter gives an overview of the other two fundamental technologies: the Unified Modeling Language (UML) and the Extensible Markup Language (XML). The chapter begins with an introduction to UML as a modeling tool very well perceived within the software engineerings community. It is followed by a discussion on an attempt by the Object Management Group (OMG) to empower UML for the development of real-time embedded software—an effort which will eventually culminate to a design framework known as the UML Profile for Schedulability, Performance, and Time Specification [29]. Thereafter, an overview of XML and a few other related Internet technologies ensue.

Chapter 4 outlines the structure of the library of platform objects (LPO), as well as furnishes a comprehensive guideline and requirements specification that a platform object must possess in order to be scalable and compatible with the proposed approach. Essential elements for each platform object, e.g. architecture blueprint, XML-based self-described modules, platform managing tool, etc., are also discussed in detail.

Chapter 5 provides a detailed treatment of UML extensions for the development of real-time embedded systems. The chapter starts with a layout of the Codesign Modeling Framework hierarchy that encompasses five other sub-profiles—the generic utility profile (*PCUprofile*) the Exception Modeling profile (*EMprofile*), the Interrupt Modeling profile (*IMprofile*), the Synthesizable Hardware Description Language profile (*SHDLprofile*), and the Architecture Blueprint profile (*ABprofile*). Each of these profiles furnishes a design framework that is specifically tailored for the proposed approach, and can robustly cope with the characteristics and requirements essential for the development of real-time embedded SoC systems. The chapter, then, proceeds to discuss the domain concept for each sub-profile, followed by the description of the corresponding stereotypes.

Chapter 6 applies the platform-centric SoC design method to the development of a simplified digital camera system so as to demonstrate the use and the robustness of the proposed approach. Specifically, the Nios development board is used to mimic the digital camera system where raw image data are read from a charge-coupled device (CCD), and then JPEG encoded and stored into memory. The chapter begins with an overview of the Altera's Nios system, followed by the actual system development process that explicitly demonstrates the use of the proposed approach. A quantitative evaluation is then presented that compares the development cost of the proposed platform-centric SoC design method against the SpecC method, using the COCOMO II.2000 cost estimation model [19].

Chapter 7 concludes the dissertation with the thesis contributions and a discussion of future directions for this research.

Chapter 2

Platform-Centric SoC Design Method

In this chapter, a novel platform-centric SoC design method is introduced. The chapter begins with an overview of the platform concept and platform-based design. Thereafter it formally defines the platform-centric approach, and discusses the detailed design process flow as well as how the UML and the library of platform objects (LPO) assists in contributing to the robustness of the proposed approach. The UML and the LPO provides a tools-integrated environment that can enable the best of the existing tools and techniques to work together in one single environment. Because knowledge in the Internet age is ubiquitous, the proposed approach encourages a collaboration of existing state-of-the-art tools and techniques, including those of open-source and public-domain, as well as proprietary schemes and standardized approaches. To conclude, the chapter compares the proposed approach with previous work in the area (Section 2.3) and discusses a possible collaboration between the proposed approach and the SystemC approach (Section 2.4).

2.1 The Platform Concept

In the proposed platform-centric SoC method, the library of platform objects (LPO) and the UML work collaboratively to enhance the system development process. While UML manifests itself as a powerful solution to designing and managing complex SoC systems, the scalable LPO aids the developer by elevating the design abstraction level as well as providing a set of pre-characterized constraints and knowledge-based environment for the system developer. This section introduces the platform concept.

2.1.1 Introduction to Platforms

Sabbagh [96] defines platforms as fully functional families of products, each of which is characterized by a set of commonalities, and is specified and implemented in such a way that allows itself and its capabilities to be further customized and re-targeted for specific actual end products. Examples of platforms are abundant, across diversely different application areas. For instance, the Boeing 777 passenger doors, each of which has a different set of parts with subtly different shapes and sizes for its position on the fuselage, are built out of the same platform whose 98% of all door mechanisms are common [96]. PC platforms, which evolve around the x86 instruction set architecture, a full set of buses, legacy support for the interrupt controller, and the specification of a set of communication devices [14], represent other examples.

Even though commonality in platforms often compromises product performance and hinders innovation and creativity, it expedites the overall process of developing end products. A typical platform could spawn scores of products that are more quickly and economically upgradeable through an upgrade of the platform itself. Such advantages are greatly desirable in the development of embedded SoC systems today, where quick time-to-market and ease of upgradeability are the dominating factors. Sangiovanni-Vincentelli and Martin [14] argues that design problems are pushing IC and system companies away from full-custom design methods, towards designs that they can assemble quickly from pre-designed and pre-characterized components. In addition, because platforms can potentially yield high-volume production from a single mask set, manufacturers will tend to be biased towards platform utilization to counter the ever increasing mask and manufacturing setup costs. These platform benefits have become even more attractive as the present state of advances in IC technologies results in more readily acceptable system performances that suit a wide range of applications.

Although the notion of platforms has existed for years, only recently has it drawn a great deal of interest from the electronic systems design community. Currently a number of system platforms exist that can roughly be classified as follows [70]:

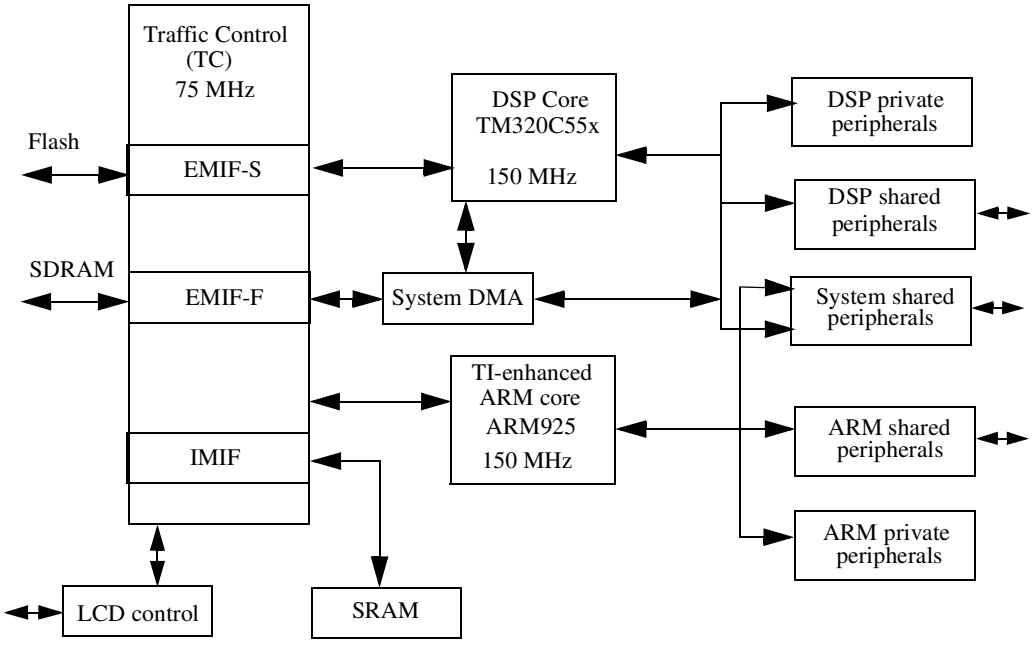


Figure 2.1: A simplified 289-pin TI's OMAP5910 platform architecture, which has the packaging size of 12x12 mm^2 (based on a figure in [70]).

- *Full-system platform*. Platforms in this category often are complete with respect to hardware and software architectures that full applications can be implemented upon, and are generally composed of a processor, a communication infrastructure, and application-specific blocks. Some also utilize FPGAs to attain better flexibility. Examples of full-system platforms include Philips' Nexperia, TI's OMAP multimedia platform (Figure 2.1), Infineon's M-Gold 3G wireless platform, Parthus' Bluetooth platforms, ARM's PrimeXsys wireless platform, Motorola's Black, Green and Silver Oak, Altera's Excalibur, Quicklogic's QuickMIPS and Xilinx' Virtex-II Pro.
- *Quasi-system platform*. Platforms in this category generally do not specify full hardware and software architectures so as to provide more flexibility for system

developers to re-target them for a wider range of applications. Such platforms as Improv Systems, ARC, Tensilica and Triscend focus more on the ability to configure processors, while others such as Sonics' SiliconBackplane and PalmChip's CoreFrame architectures provide neither a processor nor a full application, but rather define interconnect architectures that full systems can be built upon instead.

The system platform must also include the tools that aid the designer in mapping an application onto the platform in order to optimize cost, efficiency, energy consumption, and flexibility.

2.1.2 Platform-based Design for Embedded SoC Systems

The basic idea behind the platform-based design approach is to avoid designing a chip from scratch. The utilization of platforms limits choices, thereby providing faster time-to-market through extensive reuse, but also reducing flexibility and performance compared with a traditional ASIC or full-custom design approach. Goering [70] surveys how the platform-based design is defined, and presents them as follows:

- The definition and use of an architectural family, developed for particular types of application domains, that follows constraints that are imposed to allow very high levels of reuse for hardware and software components (*Grant Martin, Cadence).*
- The creation of a stable microprocessor-based architecture that can be rapidly extended, customized for a range of applications and delivered to customers for quick deployment (*Jean-Marc Chateau, ST Microelectronics*).
- An integration-oriented design approach emphasizing systematic reuse, for developing complex products based upon platforms and compatible hardware and software virtual components (VCs), intended to reduce development risks, costs and time-to-market (*Virtual Socket Interface Alliance's (VSIA) platform-based design development working group*).

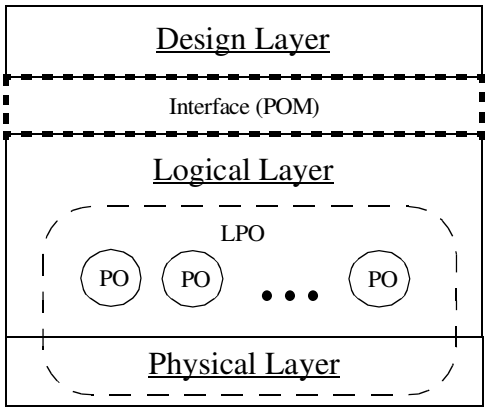


Figure 2.2: Logical model of the platform-centric environment

2.2 Platform-Centric SoC Design Approach

The platform-centric method is an enhanced version of a platform-based design approach. It provides an enhanced tools-integrated environment to assist the designer in coping with the complexity and the various requirements of today's real-time, embedded SoC systems. The approach follows the design flow illustrated in Figure 2.3.

Definition 2.1: Platform-centric SoC Design Method

A reuse-intensive, software-biased, and analysis-driven codesign approach that relies upon the UML-/XML-enhanced tools-integrated environment and the use of platforms to develop a feasible[1] SoC system quickly and correctly.

Figure 2.2 illustrates the layered architecture of the platform-centric environment. In the physical layer reside the actual hardware and software components, as well as associated design tools, all of which provide the necessary resources for the development of SoC systems.

1. Borrowed from Operations Research, the term *feasible* means "*constraints conforming*." It is to note that a feasible design is not necessarily *optimal*. Nor is it necessarily the best design achievable.

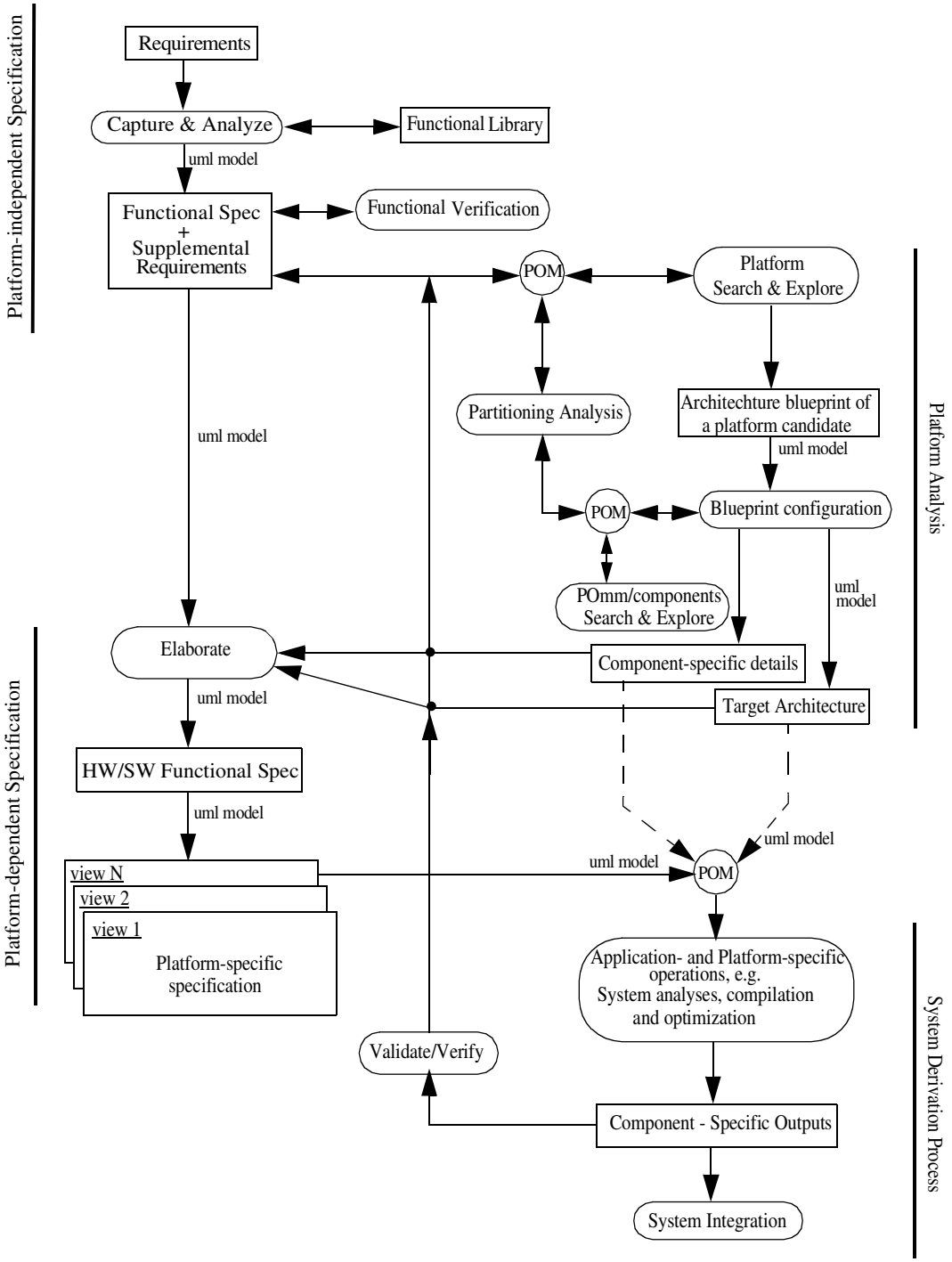


Figure 2.3: The platform-centric SoC method design flow

Definition 2.2: Platform Object (PO)

For any platform that resides in the physical layer and is a member of a particular library of platform objects (LPO) associated with a platform-centric SoC design method, there always exists a corresponding platform object (PO) in the logical layer that models such a platform.

The logical layer represents a pool of platform models called platform objects (PO), as well as models of other related entities that implement the various platform objects and/or that are platform-compatible and can be used later to build systems based on the proposed approach. Effectively these platform objects and their affiliated modules, such as an abstract representation of the platform that provides an architectural reference for the system developer, or various additional auxiliary information, which uniquely defines and identifies such an abstract representation and the relationship with its counter-part in the physical layer, form a library of platform objects (LPO). It is the LPO modules in the logical layer with which system developers have direct contact, and not those in the physical layer.

The LPO in the logical layer can be envisaged as a database of the characteristics of platforms and their affiliated modules, and thus, is suitable to be uniformly represented using the Extensible Markup Language (XML). The use of XML also brings forth other benefits, the most important of which arguably is the ability to blend well into the Internet. This capability makes the LPO boundary to be as expansive as the Internet itself, and the LPO member modules to be virtually locationally unrestricted. Furthermore, the LPO can potentially inherit many characteristics of the Internet technologies which can result in each of its PO behaving like a *directory* that can dynamically grow and shrink with respect to the *contents*, i.e. the PO member modules (POmm), present at the time of search. The LPO that can change dynamically in such manner is said to be *scalable*.

System development activities take place in the design layer. The developer accesses the resources in the logical and physical layers through a pre-defined interface

called *platform object manager (POM)*. This interface often is a software tool member of the LPO.

The platform-centric SoC design approach promotes an enhanced tools-integrated environment while also raising a level of design abstraction. Each platform object (PO) represents an abstraction of common *configurable* architectures along with their affiliated components and tools that belong to the platform domain.

A configurable platform is pre-designed off-cycle, often optimized for a specific application domain. The platform's common architecture model, the *blueprint*, fosters the concept of scalability and parametrizability; it allows candidate components to be added in or taken out without affecting other candidates. Besides mitigating the architecture selection problem, such a characteristic can help the system developer avoid lower-level hardware-dependent programming, while, at the same time, posing additional requirements for the PO member modules (POmm). As such, a library of platform objects is a logical collection of pre-designed, pre-characterized platforms that are further governed by a set of rules and requirements specific to the proposed approach. Chapter 4 discusses these rules and requirements in detail.

The proposed design flow commences with the requirements capturing and processing step that results in the platform-independent functional specification as well as the specification for timing and other requirements. The system developer then applies an estimation technique on the resultant functional specification to acquire general performance estimates and uses them to further help select the target architecture. With the target architecture information available, the developer can subsequently derive the detailed platform-dependent specification, which may contain a collection of different analysis models, e.g. concurrency model, subsystem and component model, etc. This specification along with the information relevant to the selected architecture are passed back to the appropriate LPO modules to be further analyzed, realized and integrated. Detailed discussion regarding each main stage in the proposed approach is presented as follows.

2.2.1 Platform-independent Specification

This stage chiefly concerns with the derivation of the functional specification that is still independent of any platform instance binding. Unlike most current codesign approaches that begin with a formal specification of the system, the proposed platform-centric method starts with the requirements capturing process. Kotonya and Sommerville [35] define a *requirement* as a statement of a system service or constraint. A service statement describes how the system should behave with regard to the environment; whereas, a constraint statement expresses a restriction on the system's behavior or on the system's development.

The task of requirements capturing typically involves two main subtasks, namely, *determining*, and *analyzing* the requirements as imposed by the customer [36]. During the requirements determination subtask, the developer determines, analyzes and negotiates requirements *with* the customer. It is a concept exploration through, *but not limited to*, UML's Use Case diagrams. The customer involvement in the requirements capturing process is highly recommended. Once all the requirements are determined, the analysis subtask begins that aims at eliminating contradicting and overlapping requirements, as well as keeping the system conforming to the project budget and schedule. The functional requirements are then modeled using Use Case diagrams; while, those with non-functional characteristics, e.g. speed, size, reliability, robustness, portability, standards compliance, ease of use, etc., may be captured into the supplementary requirements specification (usually a note or textual description) later to be processed and incorporated into the functional specification as constraints [97].

As established techniques in the OO analysis for specifying data and functions, Use Case and Class diagrams play a major role in deriving the platform-independent functional specification. Once all classes are identified, detailed functional implementations ensue. Various UML's diagrams can be used to describe the system characteristics. For example, Sequence and/or Activity diagrams can capture concurrent and/or sequential interactions; or, Component diagrams can depict components connection within the system. The resulting functional specification can have the class methods implemented using

a programming language of choice. This specification can later be verified for behavioral correctness and handed off to the next stage.

2.2.2 Platform Analysis

The main objective of this stage is to select the target architecture from the library of platform objects (LPO). Each PO is categorized by its applicable area(s) of use. The developer selects a candidate platform most likely to be successful for the application at hand based on information such as the associated datasheet and the requirements specification obtained in the previous stage. Then the configuration of the selected platform object follows that results in the target architecture model.

The configuration process starts by acquiring an architecture blueprint specific to the chosen platform object by means of the *platform object manager (POM)*—a front-end tool equivalent to the *interface* concept in Figure 2.2. The *architecture blueprint* is *a collection of logical PO member modules (POmm), each of which corresponds to a physical component in the platform*. The relationship among the POmm in the blueprint is captured using the UML's Class diagrams—the *ABprofile* framework to be discussed in Chapter 5. Figure 2.4 (a) illustrates the architecture blueprint of the simplified TI's OMAP platform shown in Figure 2.1.

In order to construct the target architecture model, the developer may utilize estimation tools especially optimized for the platform object to provide estimates of design metrics such as execution time and memory requirements for each part of the platform-independent specification. Estimation can be performed either statically by analyzing the specification or dynamically by executing and profiling the design description. The estimation tools take as the input the platform-independent specification, in XML Metadata Interchange (XMI) format which is defined as part of UML, and generate estimation results to be used by the developer and/or by the architecture-selection search algorithm. These results can further be validated against the requirements either manually or using validation tools.

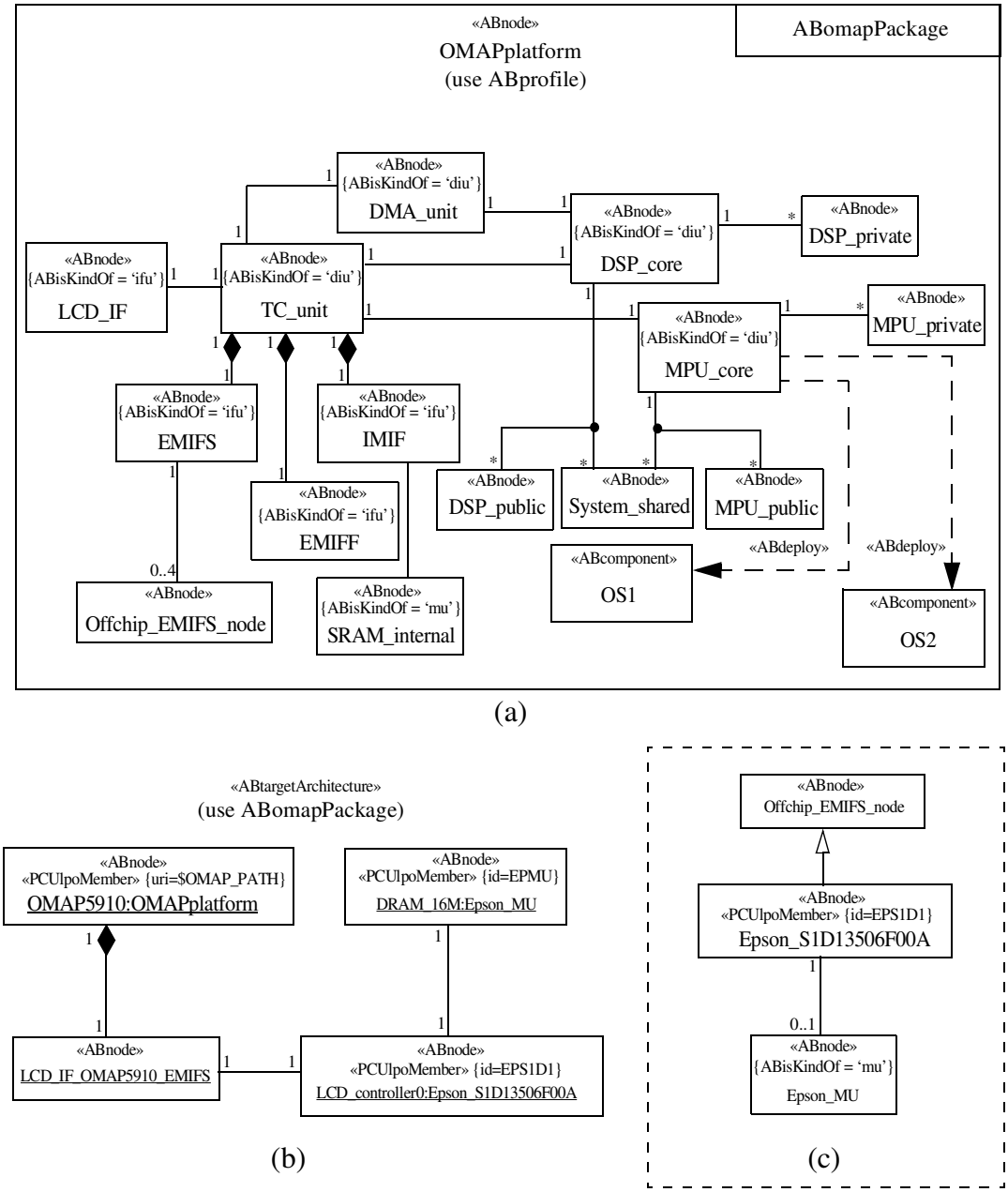


Figure 2.4: TI's OMAP architecture blueprint which (a) depicts the abstract representation of the platform architecture, and is used by POmm suppliers as a reference model, and by the developer to construct the target architecture (b). Each link in the object diagram (b) represents a pre-defined communication. The DRAM object comes as a derivative requirement when instantiating the LCD controller POmm module (c).

The platform characteristics dictate that a microprocessor or microprocessors be pre-selected and pre-optimized for specific areas of application. This places a considerable constraint on the tasks of architecture selection and system partition, which consequently could result in a faster development time trying to achieve a feasible system. Also such characteristics could permit estimation tools and profiling techniques to perform more accurately, as well as attribute to a smaller search space for the automated algorithms. In the proposed approach, the tasks of selecting the target architecture and partitioning the system are interwoven. The developer can partition the system and/or select the candidate hardware components manually, or by using various architecture selection techniques such as *simulated annealing*, *genetic algorithms*, and *tabu search*. Consult [23], [91] for detailed treatments of these algorithms on the architectural selection process. In [92], [93], [94], other related algorithms are presented.

It is to note that, in developing a system where flexibility and technology-to-market time are the predominant factors, software often carries more weight than the hardware counterpart. A study by Edwards [37] based on Amdahl's Law [38], which limits the possible overall speedup effects obtainable by accelerating a fraction of a program, suggests that it is not worth the effort to attain a substantial speedup for an arbitrary program by moving part of it to hardware. To support his claim, Edwards shows that if a part of the program which accounts for as much as 90% of the execution time is moved to an ASIC, it is still not possible to achieve more than a speedup factor of 10, even under ideal circumstances where the ASIC executes infinitely fast and communication cost is disregarded.

Even though the use of nearly any hardware/software partitioning algorithm is arguably viable for the proposed approach, the developer should always be aware of the increasing weight of software in today's SoC systems development. The platform-centric approach addresses this issue in two ways. Firstly, by utilizing the platform concept as the basis for the design, the proposed approach allows the system developer to concentrate harder on the software development process. Secondly, by utilizing UML as the unified representation of the system under development, the proposed approach allows the task of

developing a software system to be handled more robustly by one of the best tools in the software engineering discipline. In Chapter 5, the UML profile for Codesign Modeling Framework is presented that helps bridge the gap between the platform concept and UML, and effectively, hardware and software.

Once all the hardware components are chosen, they will be (1) instantiated with the blueprint to derive the target architecture, and (2) associated with the hardware-bound software modules, and (3) assigned the proper parameter values either manually by the designer or automatically either by the POM or an accompanying configuration tool. At the end of this stage yields the target architecture which can be used to derive *hardware-software functional specification*. The target architecture is essentially represented by the concrete instance of the architecture blueprint. Figure 2.4 (b) illustrates the logical representation of the simplified OMAP5910 Video architecture as discussed in [95]. This Video architecture utilizes a LCD controller POmm module, as shown in Figure 2.4 (c), and a 16-MByte DRAM as permissible by the LCD controller.

2.2.3 Platform-dependent Specification

The XML-based library of platform objects (LPO) is information-rich. Each component in the LPO is self-descriptive; it supplies the developer and the corresponding PO managing tool (POM) with information such as identity, design properties (weight, dimension, times, etc.), tool-related information, and so on. One of the information sets it may carry is the description of hardware-dependent software routines that can be used by the developer to avoid low-level coding of hardware dependent routines. Consider the LCD_controller0:Epson_S1D13506F00A object in Figure 2.4 (b). This object is represented by a specific XML description of the Epson_S1D13505F00A instance—with all the required parameters configured, and pre-characterized data captured. Within this XML description exists a section that possibly describes hardware-dependent software routines, a corresponding device driver and the location where their implementation may be found. The availability of these functions facilitates the software development process by helping

the system developer avoid implementing specific hardware-interface routines—it virtually hides communication-related details at a lower level of design abstraction.

Also, the platform-dependent specification stage is often iterative per se. During this stage, the developer first derives the hardware-software functional specification. Thereafter, additional models, e.g. concurrency model, schedulability model, etc., may be developed and analyzed (in the next stage), whose results are back-annotated for further elaboration of the hardware-software functional specification. Such activities may proceed iteratively, as shown in Figure 2.3, until all required system characteristics are determined, at which point the implementable platform-dependent specification results that can now be realized and integrated into a full system prototype.

2.2.4 System Derivation Process

The System Derivation process often goes iteratively, hand in hand, with the other stages; it furnishes an execution domain for the platform-dependent models resulted from the preceding Platform-dependent stage. The tools and tasks involved within this stage vary considerably ranging from detailed analyses, to validations and verifications, to hardware and software syntheses, to system integration. Software modules may be compiled, optimized and saved in a microprocessor-downloadable format, and hardware-bound modules may be synthesized and saved into some common format like the EDIF netlist file. Or a schedulability analysis tool may take a partially specified schedulability model and completes it for the developer—a process known as *parameters synthesis*. At the end of this stage, accurate timing and other constraints information can be obtained and verified/validated that leads to the system integration.

At this stage, the platform object manager (POM) simply functions as an auxiliary tool that supplies the system developer with necessary information to aid a smooth flow of the development process. In its simplest form, it behaves just like a web browser with all relevant information compiled and readily retrievable. A more sophisticated POM may

permit other tools to be called and operated within itself, rendering a highly integrated environment of tools and information.

2.3 Comparison with Previous Research

Table 2.1 shows how the proposed approach compares with previous related research. The general framework for the hardware/software cosynthesis approach often involves a predetermined hardware architecture consisting of a particular microprocessor and an ASIC, and the hypothesis that if only the behavior could be partitioned between these components, the remainder of the design process would automatically be done by the high-level synthesis tool and the compiler [23]. A representation of the cosynthesis approach includes the work by Ernst, et.al. [39]. They describe the COSYMA (COSYnthesis of eMbedded Architectures) system which is used to extract, from a given program, a part that could be implemented in an ASIC. COSYMA uses a novel software-oriented HW/SW partitioning algorithm that identifies critical operations in an instruction stream and moves those operations from software to hardware. Gupta and De Micheli [40] work with a similar architecture, but take the inverse approach. Instead of trying to accelerate a software implementation, their Vulcan cosynthesis system aims at reducing the cost of an ASIC design by moving less time-critical parts to a processor—checking timing constraints and synchronism as it does so. Yen and Wolf [41, 42] also propose efficient algorithms for cosynthesizing an embedded system's hardware engine, consisting of a heterogeneous distributed processor architecture, and application software architecture, consisting of the assignment and scheduling of tasks and communication of an embedded system. Although these cosynthesis approaches represent state-of-the-art efforts on codesign methods, they do have certain limitations—some of the more notable ones are:

Table 2.1: Feature support of current codesign approaches. The survey approaches include the Model-based [43,44], POLIS [11], Corsair [10], SpecC [20], SystemC [131], Chip-in-a-day [46].

Features[a]	Model Based	POLIS	Corsair	SpecC/ SystemC	Chip-in-a-day	Proposed Approach
Mechanism for customer's requirements determination and analysis						☺☺
Mechanism for constraints capturing and validation	☺	☺	☺☺	☺		☺☺
Mechanism for real-time handling		☺	☺☺	☺☺		☺☺
Unified representation	☺☺	☺☺	☺☺	☺☺	☺☺	☺☺
Elevation of design abstraction level	☺☺	☺	☺	☺	☺	☺
Translation and elaboration	☺	☺☺	☺☺	☺☺	☺☺	☺☺
Tools integration environment		☺☺	☺☺	☺☺	☺☺	☺☺
Feasibility for large-scale systems design	☺☺	☺☺	☺☺	☺☺		☺☺
Suitability for wide range of application	☺	☺☺		☺☺		☺☺
Being adaptive to technology-specific approaches						☺☺
Integrated documentation						☺☺

a. ☺☺ = supported; ☺ = partially supported; BLANK = not supported

- In the COSYMA and Vulcan systems, the fixed architecture is presumed without elaborating on such a decision. This limits the usefulness of the approaches.
- Yen and Wolf's approach supports only bus-oriented architectural topologies, which do not scale well for large application.
- Automated architectural selection is only partially supported in Yen and Wolf's approach, and not at all in COSYMA and Vulcan systems.None is able to raise design-abstraction level and/or capable of capturing customer's requirements. Neither is there a support for real-time handling mechanism.

The model-based approach [43, 44] fosters a late-partitioning, late-technology binding philosophy. Its basic supposition is that models serve as design blueprints for developing systems. At its core, the model-based approach employs a modeling technique to capture systems behaviors at different levels of abstraction. The resulting models are then subject to validation and stepwise refinement process. A validated model is simulated in a specific set of experimental conditions to verify its adherence to the initial requirements, constraints, and design objectives. Technology assignment is then carried out from the verified model specifications. The approach handles the complexity issue via the use of design modularity and hierarchy where the designer constructs models from elementary building blocks that are connected into larger blocks in a hierarchical manner. Nevertheless, the approach suffers from several limitations:

- The approach does not explicitly specify the synthesis process.
- Lack of support for translation process and tools makes it insufficient to handle the requirements of systems today.
- There is no real-time handling mechanism, nor the capability to capture customer's requirements.

The POLIS [11] codesign method addresses the issues of unbiased specification and efficient automated synthesis through the use of a unified framework, with a unified hardware-software representation. POLIS is an example of the tools-integrated environment that relies on performance estimates to drive the design, and on automation techniques to complete the tasks at each design step. The integral idea behind POLIS is the Codesign Finite State Machine (CFSM). The CFSM provides a unified input for the tools within the POLIS environment. POLIS supports automated synthesis and performance estimation of heterogeneous design through the use of Ptolemy [45] as the simulation engine. Such ability allows POLIS to provide necessary feedback to the designer at all design steps. A simple scheme for automatic HW/SW interface synthesis is also supported.

Similar to POLIS is the Corsair [10] integrated framework method. The Corsair framework contains several tools for the automatic implementation of formally specified embedded systems. Based on the extended specification language Specification and Description Language with Message Sequence Chart (SDL/MSC), the approach supports system synthesis, implementation synthesis and performance evaluation for rapid prototyping. The data from performance evaluation are back-annotated to support the estimation tool during the system synthesis step.

Even though POLIS and Corsair represent a general improvement for the design of complex embedded SoC systems, several limitations still exist:

- The CFSM graph used in the POLIS framework provides very little support for real-time handling mechanism.
- POLIS's automatic interface generation scheme is still very primitive. Much work has to be done for it to be able to cope with large-scale systems design.
- Although the tools in the Corsair environment are very well integrated, it only works for a fixed architecture. The approach gives no insight why this particular architecture is selected.
- Both methods do not furnish a process for capturing customer's requirements.

The SpecC method [20], on the other hand, is based on a specify-explore-refine paradigm. It is a unified language, IP-centric approach aimed at easing the problems caused by heterogeneous design. The SpecC language can be employed to describe both hardware and software behaviors until the designer attains the feasible implementation model later on in the design process, hence, promoting an unbiased hardware/software partitioning for the system under development. The formalism of the SpecC language allows for efficient synthesis. SpecC provides support for exceptions handling mechanism, and is capable of capturing information about timing constraints explicitly within its constructs. Limitations of the SpecC method are in the following areas:

- SpecC only supports the capturing and propagation of timing constraints, but not other constraints. This makes the tasks of constraints validation more difficult.
- The approach does not specify a process and a mechanism to systematically handle customer's requirements.

Another language-based HW/SW codesign approach that could have a significant impact on how SoC systems are developed relies instead on the SystemC [131] language. Based entirely on C++, SystemC provides a unified, IP-centric environment for specifying and designing SoC systems, and is capable of modeling systems at different abstraction levels from the transaction-accurate level to the bus-cycle accurate level to the RTL level for hardware implementation—making it well-suited for an interface-based approach that fosters cosimulation and/or coverification of heterogeneous systems under development. The SystemC's support for executable specification modeling within a single language facilitates HW/SW codesign while, simultaneously, easing the cosimulation/coverification tasks. In addition, its generalized communication model permits an iterative refinement of communication through the concepts of *ports, channels,* and *interfaces*[1]. Other features of SystemC include fixed-point modeling capability, and easy integration of existing C/C++ models.

Nonetheless, SystemC does have its own limitations that manifest in the following areas:

- Like the SpecC approach, SystemC does not specify a process and a mechanism to systematically handle customer's requirements.
- SystemC is relatively new, especially as a HDL. The current version (SystemC 2.0) has yet to provide support for hardware synthesis.
- The interface-based approach adopted by SystemC singularly might not suffice to tackle current issues in the development of SoC systems.

1. Borrowed from SpecC [20], the *channel* and *interface* concepts are first supported in SystemC 2.0.

The chip-in-a-day approach [46] pioneered by the University of California, Berkeley's Wireless Research Center (BWRC) represents an early prototype for the more promising platform-based design [14]. The approach uses Simulink to capture a high-level data flow and control flow diagrams. Based on precharacterized hardware components, it implements data path macros directly using a tool such as Synopsys's Module Compiler, while the control logic is translated into VHDL and synthesized. In this approach, algorithms are mapped directly into hardware that derives its parallelism not from multiple CPUs, but from a multitude of distributed arithmetic units. Although the current results claim to be two to three orders of magnitude more efficient in power and area than architectures based on software processors, the chip-in-a-day approach finds its limitations in the following:

- It still cannot improve upon the execution time.
- It remains to be seen if this approach can support a much more complex real-time embedded SoC design.

2.4 Other Embedded Design Approaches using UML

The platform-based design concept [14] upon which the chip-in-a-day approach is based also spawns an inception of the proposal for the Embedded UML profile [133] whose objective is to merge real-time UML and HW/SW codesign together. Embedded UML coalesces various existing ideas currently used in real-time UML and HW/SW codesign practices. It is conceived by its initiators as a UML profile package which is "suitable for embedded real-time system specification, design, and verification [133]."

In a nutshell, Embedded UML models the system using a collection of reusable communicating blocks similar to ROOM's capsule concept [134]. Interfaces and channels, that are the extensions of ROOM's port and connector notations, are used for communication specification and refinement. Within these interfaces and channels, UML stereotypes can be defined for communication and synchronization services. Other UML models such

as Use Case and Sequence diagrams provide means for specifying testbenches and test scenarios, while a combination of well-defined State diagram semantics and Action semantics [133] serves as a driving force for code generation, optimization, and synthesis. In addition, the extended Deployment diagrams, called the Mapping diagrams, may be employed to model *system platforms* [14] and furnish the platform-dependent refinement paradigm for performance analysis, and optimized implementation generation.

Even though no detailed implementation of the Embedded UML is available at the time of this writing, a careful perusal over its proposal reveals a few interesting facts. The Embedded UML profile and the UML profile for Codesign Modeling Framework (see *Chapter 5*) bear some resemblance as per their underlying objectives—each of which attempts to furnish a means to model and to develop platform-based real-time embedded systems. Certain minute implementation details of the two profiles differ tentatively. While the Embedded UML resorts to the ROOM's real-time modeling approach [134] and utilizes capsule-based reusable blocks as the basic design units, the Codesign Modeling Framework profile relies on the UML profile for Schedulability, Performance, and Time Specification [29] and perceives all design entities, including communication links and protocols, as reusable objects in the LPO. Yet another fundamental difference between the two profiles exists that possibly comes as a consequence of discrepancy in the viewpoint towards how each of them should be conceived. Just like Vissers [9], this thesis believes that semiconductor manufacturers will design systems, rather than system houses design processors. Hence, it conceives the UML profile for Codesign Modeling Framework such that, when employed by the proposed method and/or the system developer, it allows for easy malleability to new technologies. This is to contrast with the Embedded UML profile whose approach seems to come from the opposite direction where system houses are the primary forces in the development process. Under such a circumstance, it is likely that new methods and new ways defined by the semiconductor sector to build systems could possibly prevent the Embedded UML from attaining its full effectiveness—mainly due to the generalization of the Embedded UML package. As a *preliminary* assessment, the approach embraced by this thesis should eventually be more robust in a long run as it is

designed to better adapt to technological changes, and to be a more specialized package without totally relinquishing generality.

Proposed as part of the Yamacraw Embedded Software (YES) effort at the Georgia Institute of Technology, the YES-UML [135] represents yet another UML extensions package that aims at furnishing the system-level unified representation for the development of today's embedded systems. The YES approach fosters the concept of extreme reuse, where the development process encompasses both the Application Engineering and the Domain Engineering arenas, and the efficiency gain results from the application of the *economies of scope* [136] that suggest the development of a family of products rather than a single product as traditionally practiced.

The YES-UML is perceived as a complete system integrating notations whose objective is to "address the multiple-language, multiple-analysis problem of embedded systems design by combining together levels of abstraction and heterogeneous conceptual models [137]." Owing to its unified representation capability, systems analysts can deal directly with the UML models at the front-end, whereas systems designers can utilize their conventional analysis tools seamlessly at the back-end through the XMI interface. The proposed YES-UML extensions encompass the following modeling capabilities for [135]:

- Behaviors of analog interfaces within the embedded specification,
- Real-time related behaviors within the specification,
- Early identification of size, weight, and power (SWAP) constraints,
- Hardware/software interfaces, and
- The development of an executable specification capable of expressing concurrency in the real-time system being described.

It is clear from this discussion that, like the Embedded UML, the YES-UML also bears a number of similarities to the Codesign Modeling Framework profile even though its intended development paradigm is larger in scope than that of the proposed platform-

centric approach. Due to lack of implementation details, no useful comparison between the YES-UML and the Codesign Modeling Framework can be made at this time.

2.5 A Perspective on Collaboration with Non-Platform Approaches

Being a graphical language, UML works well for the proposed approach to help promote reuse at a high abstraction level—specifically at the *platform* level where the system could be quickly assembled using pre-designed, pre-characterized platform components. The proposed approach normally benefits from such platform-component reuse, as well as the use/reuse of knowledge brought forth by the XML technologies, to expedite the overall SoC systems development process. Nevertheless, the utterly diverse requirements of today's SoC systems make it nearly impossible that a desirable platform component would always exist for the system developer when applying the proposed platform-centric SoC design method. As a result, this section discusses a possible collaboration of the platform-centric approach and other non-platform approaches that could spawn an efficient sub-process within the proposed approach, and satisfactorily address this very issue.

A programming language such as SpecC and SystemC is particularly well-suited for implementing the functionality of the UML models. The reasons are that (1) both of them are object-oriented, as is UML, which could result in a convenient mapping scheme between the model and the implementation, and *vice versa*, and (2) like UML, they can uniformly represent hardware and software in a single language. SystemC, in particular, permits a large repertoire of C/C++ reusable modules to be incorporated should need arise. As a result, SystemC manifests itself as a possible language of choice for the proposed approach for the implementation of the UML models.

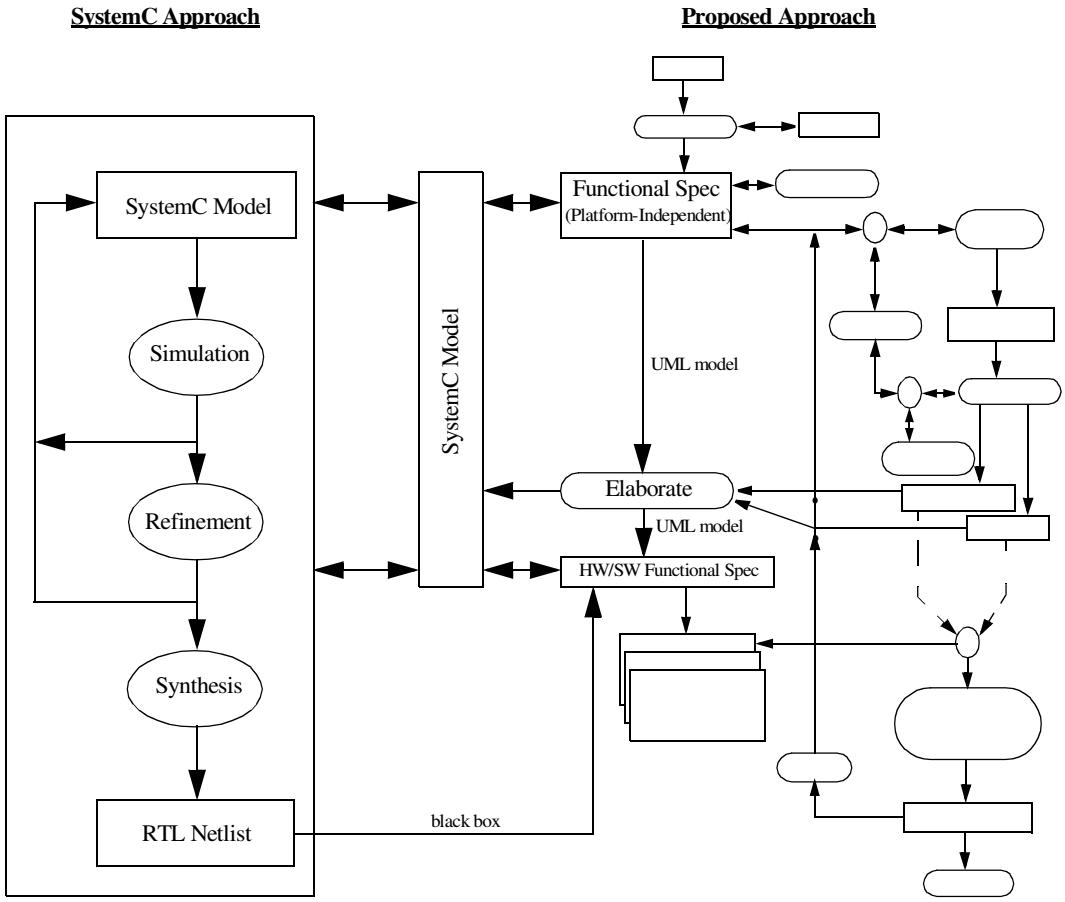


Figure 2.5: Collaborative usage model for the proposed platform-centric approach and the SystemC approach (adapted from [132])

Figure 2.5 illustrates a possible collaborative scheme between the proposed approach and the SystemC approach. In this figure, SystemC is used to implement the functionality of the platform-independent specification resulted from the requirements analysis. In a typical platform-centric development process flow, further analysis on this specification (in the *platform analysis* phase) will help determine the target architecture as well as the partition of hardware and software modules. At this point, if there exists no desirable hardware component such that the developer has to implement it as part of the design cycle, s/he can conveniently apply the SystemC iterative refinement approach to

the platform-independent functional model—yielding either the SystemC HW/SW model or the RTL netlist which can be inserted back into the platform-centric environment. Such a collaborative approach presented herein provides a safeguard for the proposed platform-centric method where it can still benefit from a unified design environment and extensive reuse at the *interface* level even when the *platform* level design is not possible.

One of the major contributions of the proposed platform-centric SoC design method is the information-rich environment that promotes easy and effective architecture selection process, while at the same time allows a wide range of state-of-the-art tools and technique to co-exist collaboratively, with minimal modification required. In addition, the proposed approach defines a UML framework for the design of real-time embedded SoC systems that begins with customer's requirements determination and analysis, and results in a detailed functional specification. This thesis asserts that such contributions bring about a desirable paradigm shift along with an improvement in the overall efficiency of the proposed SoC design approach.

The next chapter gives a technological overview of the essential background related to the Unified Modeling Language (UML) and the Extensible Markup Language (XML). It commences with an introduction to UML, followed by a discussion on the UML Profile for Schedulability, Performance, and Time Specification [29]. Thereafter, an overview of XML and a few other related Internet technologies ensue.

Chapter 4 outlines the structure of the library of platform objects (LPO), as well as furnishes a comprehensive guideline and requirements specification that a platform object must possess in order to be scalable and compatible with the proposed approach. Chapter 5 provides a detailed treatment of UML extensions for the development of real-time embedded systems. Chapter 6 demonstrates the cost-effectiveness and the robustness of the proposed approach via an application case study, i.e. the development of a simplified digital camera system. A quantitative evaluation against the SpecC method, using the COCOMO II cost estimation model follows that concludes the chapter.

Chapter 3

UML and XML

This chapter presents a comprehensive introduction to the Unified Modeling Language (UML) and the Extensible Markup Language (XML) technologies—the very foundation of the proposed platform-centric SoC design method. To begin, the chapter explains UML and its various diagrams as traditionally conceived within the software engineering community where UML is originated. It also discusses UML usage scenarios, extensibility mechanisms and possible mappings of UML notations to other programming languages such as Java, and C/C++. Thereafter, it gives an overview of the UML Profile for Schedulability, Performance, and Time Specification. The chapter concludes with a discussion of XML and related technologies that are essential for the robust implementation of a scalable and Internet-oriented library of platform objects (LPO) for the proposed method.

3.1 Unified Modeling Language

In the proposed platform-centric SoC design method, UML not only serves to drive the development process from its earliest stage of requirement capturing and analysis to the later stage of acquiring the correct implementation models, but it also provides a common input format, i.e. XMI, for the collection of tools in the library of platform objects (LPO). The UML extension mechanism makes it very flexible to be adapted to better suit new tools and techniques. In addition, UML can be used for documenting the design.

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing and documenting the artifacts of both software and non-software systems. It represents a collection of the best engineering practices that have proven successfully in the modeling of large and complex systems [24]. UML bases most of its foundation on the works of Booch, Rumbaugh, and Jacobson, but its reach has come to encompass a greater expanse than originally perceived by its creators [25]. The language has successfully undergone the standardization process with the Object Management Group (OMG) and become widely received by the industry.

The UML is a simple and generic notation made of model elements that can be used to model requirements for design of the system. Mathew [26] describes a UML model as the basic unit of development which is highly self-consistent and loosely coupled with other models by navigation channels. A model is not directly visible to users. It captures the underlying semantics of a problem, and contains data accessed by tools to facilitate information exchange, code generation, navigation, etc. UML models are represented graphically. Many different perspectives can be constructed for a model—each shows all or part of the model and is portrayed by one or more diagrams. Table 3.1 lists the models and diagrams as defined by UML.

Table 3.1: UML Models and Diagrams

UML Models	UML Diagrams
1. Class model 2. State model 3. Use case model 4. Interaction model 5. Implementation model 6. Deployment model	1. Class diagrams 2. Sequence diagrams 3. Collaboration diagrams 4. Object diagrams 5. Statechart diagrams 6. Activity diagrams 7. Use case diagrams 8. Component diagrams 9. Deployment diagrams

Use case diagrams describe what a system does from the standpoint of an external observer. They are closely connected to *scenarios*—an example of what happens when someone or something interacts with the system. A use case is a summary of scenarios for a single task of goal. An *actor* is who or what initiates the events involved in that task. The connection between actor and use case is called a *communication* association. A use case diagram can also be viewed as a collection of actors, use cases and their communications. Use case diagrams are helpful in such tasks as communicating with clients, and capturing and determining requirements.

A class diagram gives an overview of a system by depicting classes and the relationships among them. Class diagrams are static, which means they display what interacts but not what happens when they do interact. Some useful relationships are, for example, *association, aggregation, composition* and *generalization*. A number of attributes can be adorned on an association: a *role nam*e to clarify the nature of the association, a *navigability* to shows which direction the association can be traversed, a *multiplicity* to govern the number of possible instances of the class associated with a single instance on the other end, and so on. Mandatory elements in each class diagram consist of classes, associations, and multiplicities. Figure 3.1 illustrates a subset of UML notations that are often used in this dissertation and elsewhere.

To simplify complex class diagrams, classes can be grouped into packages. A package is a collection of logically related UML elements. A package is said to depend on another if changes in the other could possibly force changes in the first. *Dependency* is denoted by a dotted arrow, with the arrow originates from a client package, i.e. a package which depends upon the other package (the supplier) where the arrow terminates. Another UML mechanism that assists in simplifying complex class diagrams is the object diagrams. Object diagrams show instances instead of classes, and are useful for explaining small pieces with complicated relationships, especially recursive relationships.

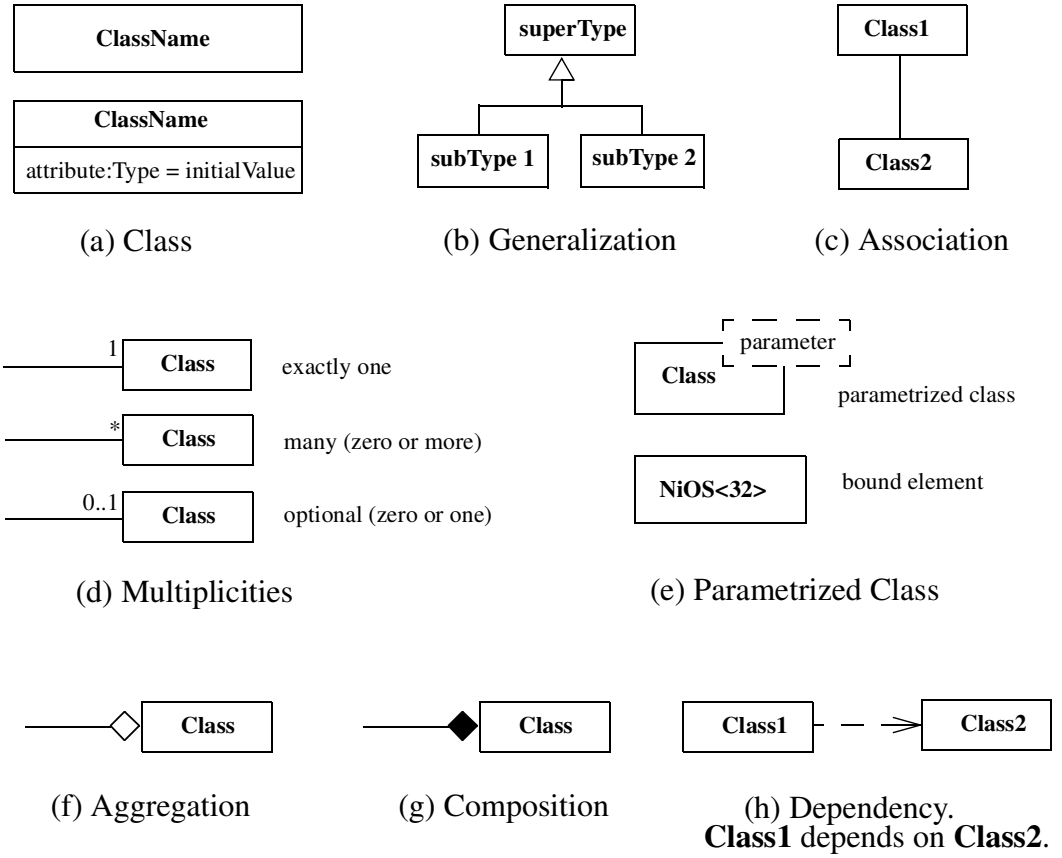


Figure 3.1: Summary of UML notations

While class and object diagrams are static model views, sequence, activity, statechart, and collaboration diagrams are of dynamic characteristics. A sequence diagram details how operations are carried out, what messages are sent, and when. Sequence diagrams are organized according to time. The time progresses vertically down the page. The objects involved in the operation are listed from left to right according to when they partake in the message sequence. A message in a sequence diagram is *asynchronous* if it allows its sender to send additional messages while the original is being processed. Similar to sequence diagrams, collaboration diagrams convey the same information but instead focusing on object roles rather than times that messages are sent. In collaboration diagrams, object roles are the vertices and messages are the connecting links with proper sequence numbers on them.

Objects have behaviors and state. The state of an object depends on its current activity or condition. A statechart diagram shows the possible states of the object and the transitions that cause a change in state. States in statechart diagrams can be nested. Related states can be grouped together into a single *composite* state. Nesting states is necessary when an activity involves concurrent or asynchronous subactivities. While a statechart diagram focuses attention on an object undergoing a process, an activity diagram focuses on the flow of activities in a single process. Out of each activity comes a single *transition*, connecting that activity to the next activity. A transition may *branch* into two or more exclusive transitions, or *merge* to mark the end of the branch. A transition may also *fork* into two or more parallel activities. These parallel threads can later *join* to form a single transition.

The last two diagrams defined by UML are component and deployment diagrams. Component diagrams are physical analogs of class diagram. Deployment diagrams display the physical configurations of software and hardware. The physical hardware in a deployment diagram is made up of *nodes*, which, in turn, can contain *components*.

UML is more complete than other languages in its support for modeling complex systems, and is particularly suited for capturing real-time embedded systems. The major features of UML suitable for modeling real-time embedded systems include [28]:

- an object model (incorporating data attributes, state, behavior, identity and responsibility) allowing the structure of the system to be captured,
- use case scenarios—allowing key outputs to be identified from system in response to user input,
- behavioral modeling—use of statechart diagrams help facilitate the dynamic modeling of the system's behavior,
- packaging—providing mechanisms to organize elements of the modeling into groups,
- representations for concurrency, communication and synchronization for modeling real-world entities,
- model of physical topology—using deployment diagrams to show the devices and processors which comprise the system,
- support for object-oriented *patterns* and *frameworks*—allowing common solutions to common problems to be represented.

In the context of object oriented technology, patterns can be thought of a problem-solution pair. They capture the static and dynamic structures and collaborations among key participants of successful solutions to problems that arise when building applications in a particular domain [67]. As opposed to frameworks that support reuse of detailed design and code, patterns facilitate reuse of successful software architectures and designs by relying on two key principles [67]: (1) separation of interface from implementation, and (2) substitution of variable implementations with common interfaces. Gamma, et.al., show in their pioneering work on design patterns [68] how patterns solve design problems and how they explicitly capture expert knowledge and design trade-offs, and make this expertise more widely available.

For further readings on UML, UML's specification by OMG [24] and the user guide by Booch, Rumbaugh and Jacobson [30] provide a detailed treatment of the subject. For patterns, the Design Patterns book by Gamma, et.al. [68] is a good starting point for interested readers. Quick tutorials can be found at [66, 67], while [64, 65] represents the very foundation of design patterns as conceived by Christopher Alexander during the late 1970s [66].

The following subsections discuss UML's general extensibility mechanisms that include topics on constraints and Object Constraint Language (OCL), tagged values, and stereotypes. Then, a discussion on UML-to-code translation, as well as an overview of the UML profile for Schedulability, Performance and Time Specification [29] are presented. This section owes a large part of the UML diagrams description to a good tutorial by TogetherSoft [27]. Appendix C summarizes the various UML notations.

3.1.1 Constraints and Object Constraint Language (OCL)

To quote [98]: "A constraint is some additional restriction (above the usual UML well-formedness rules) applied against a modeling element." UML constraints always appear enclosed in a pair of curly braces ({}) and can be placed inside text notes. Certain kinds of constraints, e.g. {subset}, {ordered}, {xor}, etc., are predefined, whereas others can be user-defined (see UML Specification [24] for details). The interpretation of user-defined constraints are tool-dependent. In fact, as stated in the specification, it is expected that individual tools would provide one or more languages in which formal constraints could be written [24].

As the predefined UML constraints are not comprehensive enough to handle all aspects of a system specification, and user-defined constraints can potentially result in ambiguities, UML decidedly includes with its specification a *formal* constraint description language—the Object Constraint Language (OCL) [24, 99]—that can come in handy when need arises.

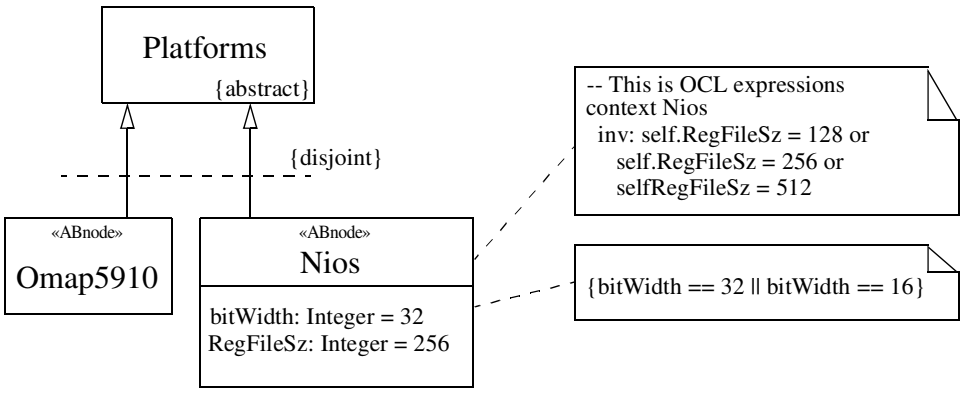


Figure 3.2: Demonstrative use of UML extensibility mechanisms

OCL expressions are typically used to specify invariant conditions, as well as pre- and post-conditions that must hold for the system being modeled. They can also be used to describe guard conditions, to specify constraints on operations, as well as to provide a navigation language. OCL expressions can reside in text notes, or can be hidden by tools. Because an evaluation of an OCL expression is instantaneous, it never alters the state of the system. Figure 3.2 demonstrates the use of UML constraints and OCL expressions.

3.1.2 Tagged Values

To avoid inundating UML models with an excessive number of graphical notations that often result in developmental ineffectiveness, detailed UML element properties may be captured using such mechanisms as *attributes*, *associations* and *tagged values*.

A tagged value is a keyword-value pair of type String (in the UML specification, a keyword is actually called a *tag*) that permits arbitrary information to be attached to any kind of model element so as to provide semantic guidance for back-end tools such as code generators, and report writers [24], to name a few. Each tag represents a particular kind of

property applicable to one or many kinds of model elements. Similar in nature to UML constraints, a tagged value also always appears within a pair of curly braces ({}). When more than one tagged value are to be specified, a comma (,) is used as a delimiter to separate them. When a tag appears without an accompanying value, it represents a standard shorthand notation for {isTagname = true}. As an example in Figure 3.2 above, {abstract} is a tagged value that is semantically identical to {isAbstract = true}.

3.1.3 Stereotypes

Among the core extensibility mechanisms furnished by UML, stereotypes are probably the most powerful and extensively used construct. When applied to a model element, it subclasses that model element, inheriting the attributes and relationships, but exhibiting specific intent such that an unambiguous interpretation can be rendered when processed by tools. Just like any model element, a stereotype can have constraints and tagged values attached to itself.

Stereotypes are usually shown as text imbedded in a pair of guillemets. In Figure 3.2, for example, «ABnode» is a stereotype. So are «ABcomponent», «ABIpoMember», «ABmap» and «ABdeploy» in Figure 2.4. Where the use of guillemets is not possible, the text strings << and >> are used.

3.1.4 UML to Code Mapping

Even though UML's lack of rigorous formalism has been a subject of criticism by many, it has also been demonstrated by quite a few UML tool vendors [100, 101, 102, 103, 104, 105] that such a shortcoming does not really hinder the developments of features like code generation, reverse engineering (where a tool constructs UML models based on existing code), as well as round-trip engineering, which further tries to regenerate the source code when the model is modified.

Following simple rules like those shown in Table 3.2, the mappings between UML and a popular OO programming language such as Java have evolved to become a standard feature supported by almost all commercial UML tools, plus a couple of free ones. With some modifications, UML to code mappings for other programming languages, e.g. Ada, Real-Time Java, Delphi, CORBA IDL, optimized C/C++ for embedded application, are also possible.

3.1.5 UML Profile for Schedulability, Performance and Time Specification

Currently undergoing the final stage prior to being standardized by OMG, this real-time profile aims to bridge a gap between the real-time and UML communities by providing capabilities for modeling real-time systems and characteristics, such as quality of services (QoS) that often are non-functional, yet essential to the development of real-time systems. The profile furnishes a design framework that (1) enables the construction of models that could be used to make quantitative predictions about the QoS characteristics, and that (2) facilitates communication of design intent among developers in a standard way [29].

Table 3.2: General rules for the mappings between UML models and Java

UML Constructs	Java Constructs
1. Attribute	1. Instance variable
2. Operation	2. Method
3. Abstract class	3. Abstract class
4. Interface	4. interface keyword
5. Package	5. Package declaration
6. Subclass (Generalization)	6. extend keyword
7. Realization	7. implement keyword
8. «use» or Dependency	8. import clause
9. Multiplicity	9. Array
10. Role	10. Instance variable of type Class that is associated with the Role

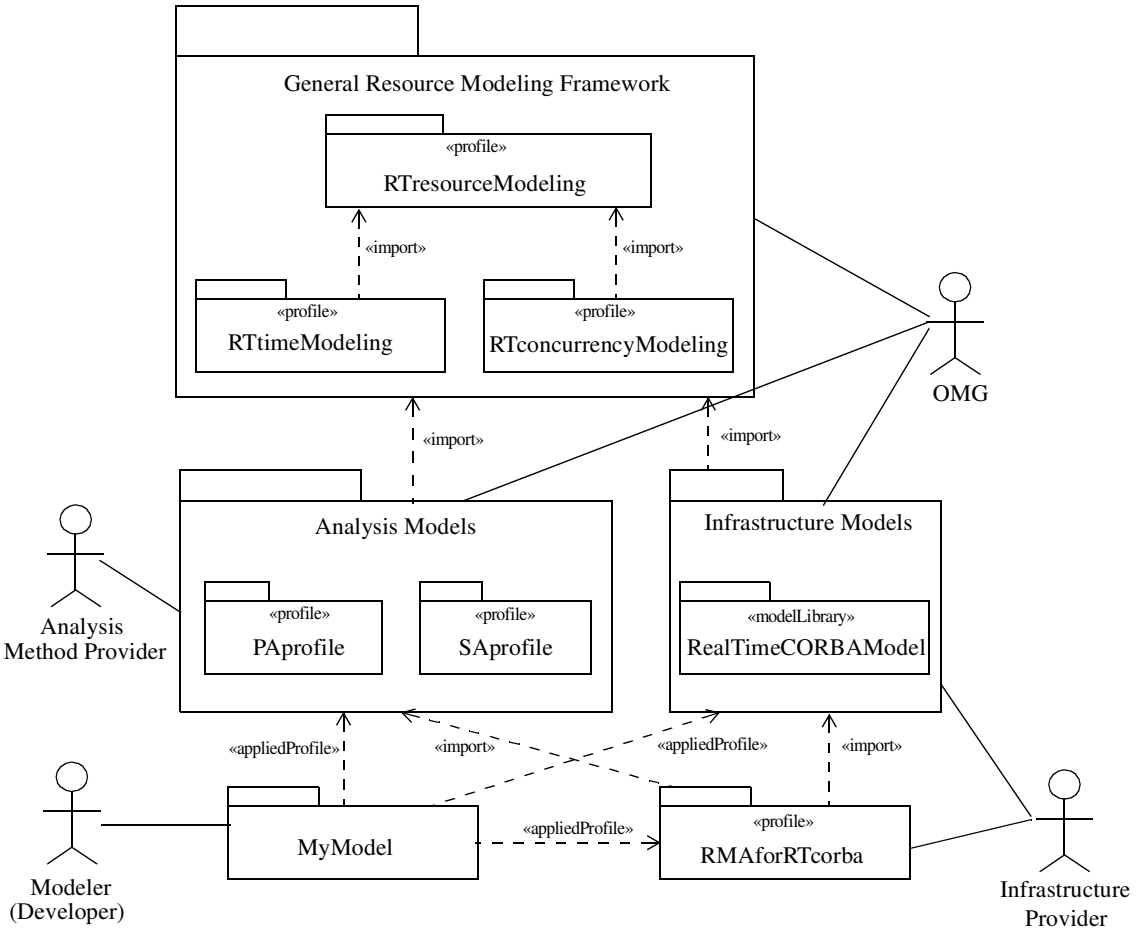


Figure 3.3: Structure of the UML Profile for Schedulability, Performance and Time Specification

A UML profile, shown as a package notation adorned with «profile», is a collection of predefined stereotypes that is used to tailor UML for a specific problem domain. As such the UML profile for Schedulability, Performance and Time Specification (or, contractively writing, *the UML real-time profile*) provides the *system developer*, as well as tool vendors, e.g. an *analysis method provider*, or *an infrastructure provider*, with a means to effectively model real-time applications for analysis and synthesis via a predefined set of RT-related stereotypes.

At the core of this profile is the General Resource Modeling framework which encompasses the RTresourceModeling, RTtimeModeling, and RTconcurrencyModeling profiles that define the concept and the modeling of resources, time and concurrency, respectively. Along with these three packages, the UML real-time profile also include the basic frameworks for schedulability and performance analyses, i.e. the SAprofile and the PAprofile packages, as well as the Real-Time CORBA infrastructure model. Figure 3.3, which is adapted from [29] and [106], illustrates the structure of the UML real-time profile that also depicts the interactions among different active entities (represented by stick figures) and profile packages.

The UML real-time profile's general resource model deals specifically with various aspects of resources modeling in the real-time domain. It details a comprehensive treatment on resources-sharing where mechanisms for modeling activeness and protection are defined. Services as provided by the resources are either exclusive or non-exclusive, with various quality of service (QoS) characteristics imposed upon them.

The time model described in the RTtimeModeling profile is a specialized resource model, and thus, naturally inherits all the rudimentary concept defined in the RTresource-Modeling package. Using the RTtimeModeling profile, the developer can specify time value, instant, and duration, as well as associate time with actions and events where moment of occurrence and duration are of importance. It is to be noted, nonetheless, that the RTtimeModeling profile only permits one clock to be defined.

The last package in the general resource framework, the RTconcurrencyModeling profile, copes primarily with the concurrency aspects of modeling real-time applications. Using this profile, the concept of active objects and threads can be modeled conveniently. The profile resorts to the message passing scheme for communicative means. From the profile viewpoint, a call action in the sender initiates a message transfer that subsequently triggers a behavior in the receiver. The modeler can specify call actions to be *synchronous* or *asynchronous*, and operations or receptions of messages to be *immediate* or *deferred*.

In addition to the profiles discussed earlier, the UML real-time profile also contains two other native packages that provide basic frameworks for modeling real-time applications for further schedulability and performance analyses. These two profiles do not, by and large, support all available techniques and algorithms, but rather only a representative subset of those. It is expected that more specialized profiles for schedulability and performance analyses, as well as syntheses, can later be supplied by tool vendors.

3.2 Extensible Markup Language

The Extensible Markup Language (XML) is the other technological underpinnings for the proposed platform-centric SoC design method. XML provides a concrete facility for realizing the LPO concept and managing the complexity resulted from a huge database of extremely diverse LPO modules. XML also promotes information exchange which can culminate to the virtually unbounded sharing of PO member modules (POmm). Furthermore, there exists an extensive collection of public-domain and/or open-source tools for XML and other XML-related technologies that can be used to enhance its capability.

3.2.1 Introduction to Markup Languages and XML

Markup languages are all about describing the form of the document—that is, the way the content of the document should be interpreted [47]. A *markup language* is a notation for writing text with markup *tags*, where the tags are used to indicate the structure of the text [48]. Tags have names and attributes, and may also enclose a part of the text.

One of the most predominant markup languages these days is the Hyper-Text Markup Language (HTML). HTML is an *application* of the Standard Generalized Markup Language (SGML), which is a rather complicated *superset* of XML. HTML has a fixed set of markup tags, and is primarily used for layout on the Web. Being a hyper-text language, a HTML document can contain links to other documents, text, sound, images, and various other resources. However, it says nothing about the content of the data.

Both SGML and HTML heavily influence the development of XML [49]. XML is a semantic language that allows text to be meaningfully annotated. It is designed to separate syntax from semantics to provide a common framework for structuring information and allow tailor-made markup tags for any imaginable application domain [48]. XML also supports internationalization (Unicode) and it is platform-independent.

One of the functions of XML is the storage of data. The technologies that dominate the market for data storage are relational databases that manage traditional data types such as numbers and text [50]. XML goes beyond the boundary of traditional relational database technologies by offering the potential to process smart data, or i.e. self-describing data. XML documents are, by their nature, ideal for storing databases [51]. A standardized interface to XML data is defined through W3C's Document Object Model (DOM) [52], which provides a CORBA IDL interface [53] between applications exchanging XML data. The clearly defined format provided by XML helps make the data readily transferable to a wide range of hardware and software environments. New techniques in programming and processing data will not affect the logical structure of the document's message [51]. If more detail needs to be added to the document, the model can be updated and new markup tags added where required in the document instance. If a completely new style is required then the existing document model can be linked to the new one to provide automatic updating of document structures. For these reasons, a claim exists that XML brings about a revolution in communication and information distribution across the Web and within intranets [50].

3.2.2 Conceptual View of XML

An XML document is an *ordered, labeled* tree [48]. The leaf nodes of an XML tree are *terminal* elements that contain actual data in the form of text strings. Although a character data leaf node can be declared EMPTY, it is usually non-empty and must be non-adjacent to other leaf nodes. Other kinds of XML leaf node exist, that include:

- processing instructions—annotations for various processors,
- comments, and
- a schema declaration, i.e. Document Type Definitions (DTD) or XML-Schema.

In XML tree terminology, *non-terminal* elements always contain subelements, or child nodes, that can be grouped as *sequences* or *choices*. A sequence defines the order in which subelements must appear. A choice gives a list of alternatives for subelements. Sequences and choices can contain each other. Each non-terminal element node can be labeled with a name—often called element type—and a set of attributes, each consisting of a name and a value.

By adopting the Extended Backus-Naur Form (EBNF) for its syntax (see Section 1.4 of the official W3C XML Recommendation [55]), XML is designed to be easy to use with modern compiler tools [54]. The EBNF defines a set of rules, where every rule describes a specific fragment of syntax. A document is valid if it can be reduced to a single, specific rule, with no input left, by repeated application of the rules. XML is defined by an EBNF grammar of about 80 rules.

An XML document normally consists of three types of markup tags, the first two of which are optional:

1. An XML processing instruction identifying the version of XML being used, the way in which it is encoded, and whether it references other files or not, e.g.,

```
<?xml version="1.0" encoding="UTF-8" standalone="yes">
```

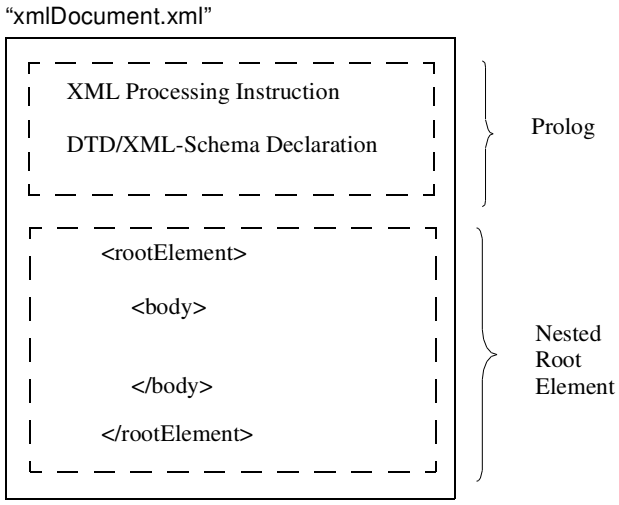


Figure 3.4: Structure of an XML document

2. A schema declaration that either contains the formal Document Type Definitions (DTD) markup declaration in its internal subset (between square brackets) or references a file containing the relevant markup declaration (the external subset), e.g.:

   <!DOCTYPE poModule SYSTEM "http://www.lpo.com/dtds/poModule.dtd">

3. A fully-tagged document instance which consists of a root element, whose element type name must match that assigned as the document type name in the document type declaration (DTD), within which all other markup is nested.

The term *prolog* is used to describe the section where the XML processing instructions and a schema declaration may optionally appear. Figure 3.4 illustrates the XML document structure.

A *well-formed* XML document must contain exactly one *element*. This single element can be viewed as the *root* of the document. Elements can be nested, and attributes can be attached to them. Attribute values must be in quotes, and tags must be balanced, i.e. they must always be explicitly opened and closed. Empty element tags must either end with a /> or be explicitly closed.

The structure of an XML document is defined by a schema language. Currently the W3C XML Recommendation [55] supports two types of a schema language, namely, Document Type Definition (DTD) and XML-Schema [56]. While the XML-Schema is by far more powerful than the DTD, it is also more complicated and still is very immature. Tool support for the XML-Schema is meager. As of today, it remains to be seen whether or not the XML-Schema will be able to completely replace the DTD as stated in its objective. In implementing the LPO, only the DTD will be used for it is more stable and sufficiently powerful. The DTD to XML-Schema conversion tools are readily available. Figure 3.5 shows an example of a DTD file used to describe the structure of the PO register file.

Also used in SGML, the DTD provides a facility to specify a set of tags, the order of tags, and the attributes associated with each. A well-formed XML document that conforms to its DTD is called *valid*. A DTD is declared in the XML document's prolog using the !DOCTYPE tag. It is actually within the DTD that the *sequence* and *choice* grouping of non-terminal child nodes is specified. XML tags are defined using the DTD's ELEMENT tag, while the attributes associated with each XML tag are defined using ATTLIST. A DTD's terminal element can be of types parsed character data (#PCDATA), EMPTY or ANY. A terminal element declared as ANY will not be parsed; thus, it can contain subelements of any declared type, including character data. It is to note that an XML document is case-sensitive.

```
<!-- Define abstract type for each module -->
<!ENTITY % typeFile SYSTEM "file:///C:/Data/Thesis/XML/Test/abstractType.txt">
<!ENTITY % abstractTypeList "(%typeFile;)">

<!ELEMENT poRegfile (self, pom, blueprint, poSchema, poInfo?, polif+)>

<!-- Start with self here -->
<!ELEMENT self (id, poID, uri)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT poID (#PCDATA)>
<!ELEMENT textField (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT uri (#PCDATA)>

<!-- the POM, PO Manager -->
<!ELEMENT pom (id, uri)>
<!-- Architecture blueprint -->
<!ELEMENT blueprint (uri)>
<!-- Location of the PO schema file -->
<!ELEMENT poSchema (uri)>
<!-- Information about the PO -->
<!ELEMENT poInfo (info)+>
<!ELEMENT info (subject, textField)>
<!ELEMENT subject (#PCDATA)>

<!-- Main Entry... consisting of toolModule and componentModule -->
<!ELEMENT polif (name, uri)>

<!-- Attribute declaration -->
<!-- This DTD uses a MIME type for the fieldType attribute -->
<!ATTLIST textField fieldType CDATA "text/plain">
<!ATTLIST textField isImported (yes | no) "no">
<!ATTLIST polif moduleKind (component | tool) #REQUIRED>
<!ATTLIST polif abstractType %abstractTypeList; #REQUIRED>
<!-- moduleID should match an id as defined in each corresponding xml file -->
<!ATTLIST polif moduleID ID #REQUIRED>
```

Figure 3.5: An example of a Document Type Definition (DTD) file. When written as a separate file, the DTD file contains the content of the DTD declaration as nested between the square brackets within the <!DOCTYPE [DTD]> element.

3.2.3 XML Extensions and Applications

As of late 1998, the XML design effort was re-chartered under the direction of an XML Coordination Group and XML Plenary Interest Group to be carried out in five new XML working groups [57]: XML Schema Working Group, XML Fragment Working Group, XML Linking Working Group (XLink and XPointer), XML Information Set Working Group, and XML Syntax Working Group. These working groups were designed to have close liaison relationships with the W3C's Extensible Stylesheet Language (XSL) Working Group and Document Object Model (DOM) Working Group. Under such concerted collaborations, several XML-based technologies have since emerged. These technologies make XML even more powerful and attractive as a convenient tool to realize the LPO concept. Some of these XML extensions and applications that could be used to enhance the proposed platform-centric SoC design method include:

- *Namespaces*. The use of namespace helps avoid name clashes when the same tag name is used in different contexts. Namespaces can be defined in any element, and appear as a *prefix* before an element or an attribute name, separated by a colon. Their scope is the element in which they are defined. Unfortunately, namespaces and DTDs do not work well together [49]; they bear no special meaning for the DTD. The DTD views a namespace as an integral part of a tag name without differentiating it from the local name. To better harness the power of namespaces, XML-Schema should be used.

- *Addressing and linking*. XML extends HTML's linking capabilities with three supporting languages: Xlink [58] which describes how two documents can be linked; XPointer [59] which enables addressing individual parts of an XML document; and, XPath [60] which is an underlying technology for XPointer for specifying location paths. These technologies, together with the namespace technology, enable XML documents, e.g. the POmm, to integrate very well into the World Wide Web. As a result, data exchanges are promoted in such a way that can ultimately lead to one big database that spans the entire Internet and intranet spaces.

- *WWW Resources*. The Resource Description Framework (RDF) [61] is an XML application that works with metadata, i.e. data that describes data; it expresses XML data in the canonical XML format for ease of identification. RDF is a language that describes WWW resources, such as title, author and modification date of a Web page, as well as anything that can be identified on the Web [62]. It provides a common framework for expressing WWW information in such a way that it can be exchanged between applications without loss of meaning. One widely used RDF today is the Dublin Core, which is seen by many as a way of standardizing RDF for Web resources [47]. As the LPO can also span the WWW domain, the Dublin Core may be used to ease the POmm identification process. More information about the Dublin Core can be found at *http://www.purl.org/dc*.

Of these XML extensions and applications, XPath plays a major role in an efficient implementation of the LPO. Because the LPO is composed mainly of XML data, XPath becomes an obvious choice for data querying and retrieving operations. In addition, other XML-based technologies such as XPointer, XQuery [72] and XSL/XSLT [71], that can potentially contribute to the success of the LPO, also base their implementations on or around XPath. The applicable usages of these technologies may involve using (1) XPointer to access remote XML content, (2) XQuery to process XML data query, or (3) XSL/XSLT to automatically render XML data into another format more suitable for human understanding.

3.2.3.1 XPath

A W3C standard, XPath is a syntax for defining parts of an XML document [73]. It comes pre-defined with a library of standard functions that helps cope with strings, numbers and Boolean expressions.

From XPath's point of view, an XML document is a tree view of nodes. XPath uses *location path expressions* similar to the traditional file path to identify nodes in an XML document. A location path expression results in a node-set that matches the path.

A location path consists of one or more location steps, separated by a slash (/), and it can be absolute or relative. An absolute location path starts with a slash and a relative location path does not. The location steps are evaluated in order, one at a time, from left to right. Each step is evaluated against the nodes in the current node-set. If the location path is absolute, the current node-set consists of the root node. If the location path is relative, the current node-set consists of the node where the expression is being used. Location steps consist of an axis, a node test and zero or more predicates. The syntax for a location step takes the form: *axisname::nodetest[predicate]*. For example, in the expression child::po[last()], child is the name of the axis, po is the node test and [last()] is a predicate. XPath 1.0 supports only four expression types and seven node types. The expression types consist of *node-set, string, number*, and *boolean*; whereas, the node types encompass *document, element, attribute, comment, text, namespace* and *processing instructions*.

To make XPath expressions easier to use, abbreviations are defined for frequently used XPath syntax. Table 3.3 lists these abbreviations. Table 3.4 demonstrates the use of some XPath expressions on the LPO register file below.

Table 3.3: XPath syntax abbreviations

Expression	Abbreviation
self::node()	.
parent::node()	..
child::childname	childname
attribute::childname	@childname
/descendant-or-self::node()/	//
[position()=3]	[3]
[position()=last()]	[last()]

```
<?xml version="1.0"?>
<!DOCTYPE lpoRegfile SYSTEM "file:///C:/Data/Thesis/XML/Test/lpoRegfile.dtd">
<lpoRegfile>
    <self>
    <name>Example LPO</name>
    <id>lrf001</id>
    <uri>file:///C:/Data/Thesis/XML/Test/lpoRegfile.xml</uri>
    <textField>This is a description of this lpoRegfile</textField>
    </self>
    <po>
    <name>po-NiOS</name>
    <uri>file:///C:/Data/Thesis/XML/Test/poRegfile_nios.xml</uri>
    </po>
    <po>
    <name>po-OMAP</name>
    <uri>file:///C:/Data/Thesis/XML/Test/poRegfile_omap.xml</uri>
    </po>
</lpoRegfile>
```

A number of public-domain tools exist for XPath. For example, the Apache's Xalan [74, 75] contains an XPath engine that can be used with C++ or Java. The Java-based Jaxen [76] and Microsoft's MSXML [77] provide some alternatives. The examples presented in Table 3.4 are evaluated using MSXML 3.0. Complete reference to XPath can be found at the W3C website [60]. ZVON [78] and W3Schools [73] also contain excellent resources on XPath and other XML-related topics. As it is beyond the scope of this dissertation to cover the whole spectrum of XML technologies in detail, interested readers should consult appropriate references for further information. Chapter 4, however, will discuss how each XML technologies presented above fit into the LPO implementation scheme.

Table 3.4: Examples of the XPath expressions

Path	Description
/lpoRegfile/self/*	returns all children of the *self* node, i.e. *name, id, uri, textField*
@*	returns all attributes of the context node
string(//id/.)	returns string value of the *id* node, i.e. *lrf001*
//attribute::isImported	returns the *isImported* attribute in the document, i.e. *isImported = "no"*
/descendant::po[1]	returns the first *po* node in the document
string(//@fieldType)	returns the string value of the *fieldType* attribute in the document, i.e. *text/plain*
po[name and uri]	returns all the *po* nodes within the context that contain both *name* and *uri* elements
//self/textField[@isImported = "no"]	returns all *textField* nodes that have the *isImported* attribute values set to "no"
count(//uri)	returns number of the *uri* node appearances, i.e. 3

Chapter 4

Library of Platform Objects

This chapter is dedicated to a detailed discussion of the XML specification for realizing a library of platform objects (LPO) for the proposed platform-centric SoC design method. It starts by describing desirable LPO characteristics from the conceptual domain viewpoint. From within this domain, requirements for the LPO are identified. Thereafter, the chapter maps these requirements onto the XML domain, and defines the XML specification, as well as identifies the roles that are expected from the system developer, the platform object member modules (POmm) developer, and the POmm themselves. It provides a general but precise guideline on how a LPO should be implemented. Requirement levels are indicated by keywords according to the guideline furnished by the Internet Engineering Task Force (IETF)'s RTF2119 [63], which is also summarized in Table 4.1.

4.1 Conceptual Viewpoint

This section looks at a LPO analytically, and identifies characteristics that are desirable for the proposed platform-centric SoC design method. Once identified, these characteristics serve as the basis for deriving the XML specification. In general, a LPO behaves like a data warehouse for the developer—it permits data to be stored, inventoried, searched, and retrieved. The specification addresses such requirements, as well as ensures compatibility among data from different providers, and among every individual LPO and the proposed approach.

Table 4.1: Summary of the requirement levels as specified in IETF's RFC2119

Keywords	Meanings
must, required, shall	an absolute requirement
must not, shall not	an absolute prohibition
should, recommended	there may exist valid reasons in particular circumstances to ignore a particular item
should not, not recommended	there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful
may, optional	truly optional

Definition 4.1: Library of Platform Objects (LPO)

A collection of pre-designed, pre-characterized system platforms that further meet a governing set of rules and requirements specific to the proposed platform-centric SoC design method.

Definition 4.2: Platform Object (PO)

A configurable/reconfigurable, pre-designed, pre-characterized system platform that (1) supports *fast* and *correct* system construction via a set of well-defined communication infrastructure, and (2) carries with it a set of compatible *platform object member modules (POmm)*.

4.1.1 LPO in Principle

A LPO is a *distributed and scalable database*. As such, it is only appropriate to, first and foremost, identify with these principal LPO characteristics that make the concept novel and beneficial for the proposed platform-centric SoC design method.

Database-wise, from the platform-centric approach's point of view, a LPO involves three main categories of data:

- **Component information**. Data in this category conveys various information about component characteristics, as well as component-specific knowledge for the very purpose of assisting the developer in efficiently deploying its self. The data shall include, at the minimum, the component identification and its whereabouts, the corresponding UML model, the characteristics similar to those appeared in a databook, and potentially the availability of component-specific software routines, as well as any possible tool(s) association. Knowledge-based information, such as design guideline, user manual, official specification, etc., is also expected.
- **Tool information**. Because platforms are often designed with certain application domains in mind, full potential of a platform-based design such as the proposed approach can usually be benefited when a platform is accompanied by a specially-tailored tool suite. For this reason, tools, and not just components, are integral to the robustness of the platform-centric approach. To treat tools and components equally, the method views tools also as LPO modules that can convey information, just like their component counterparts. Information held by tool modules is not as diverse as that of component modules, and shall include information about their availability and whereabouts, as well as knowledge-based information such as user guide and manuals.
- **Platform information**. A platform often is configurable using both hardware and software components. In addition, because of its specialized nature, it is suitable for a platform provider to also supply a template that can serve as a guideline for the developer when trying to construct a desirable target architecture from the platform. As such, various characteristics related to this template can be collected and grouped, and the template placed in a LPO.

A LPO is scalable. Once constructed, it is open to an addition of new modules and/or a removal of current ones—given that such an addition or removal does not conflict

with the requirements for its existence. It is perceived that a LPO could behave like a distributed database, with platforms (PO) and platform components (POmm) physically residing across the Internet and the developer only locally maintains the logical representations of these entities. Owing largely to the Internet technologies, such distributedness and scalability potentially renders better manageability, maintainability and upgradeability for a LPO—a feature that could very well be attractive for both the system developer and the platform and component providers. Under such a scenario, the system developer could be completely up-to-date with a current set of available platform technologies and components; whereas, the platform and component providers could be in total control of the services such as component usage, upgrade, and maintenance.

4.1.2 Identity

One of the questions with a LPO behaving like a distributed database is, "whose responsibility is it to first initiate the existence of a LPO?" Because a LPO is primarily a database of platforms whose existences further induce the existence of platform components, it should be the platform providers that are responsible for the initiation. When a platform is installed, it shall check to see if a LPO exists that can be identified. If not, it creates one and makes it known publicly such that subsequent insertion of platforms will not have to perform the same task redundantly.

The ability to check an existence of a LPO at the time of platform installation implies that there must locally exist a mechanism that keeps track of platform availability as well as any relevant information necessary for identifying and locating platforms. This mechanism represents a logical LPO space that provides the mapping to their physical counterparts during the development process.

Definition 4.3: Logical LPO space

A logical representation of a LPO that resides locally, where the developer has complete access to, and providing links to its physical equivalents

Definition 4.4: Physical LPO space

A physical LPO space defines a locationally unrestricted domain where a LPO and its data may reside. This spatial domain is confined by the Internet boundary, and includes any local space as well. As such, it is a superset of a logical LPO space.

4.1.3 Scalability

A LPO grows or shrinks dependently on an *attachment* or *removal* of LPO modules, i.e. platforms and/or platform components. An attachment of a LPO module onto a LPO requires, as a prerequisite, that the attaching module physically exists. Its whereabouts is locationally unrestricted—it may be locally present, or remotely accessible via the Internet; it attaches itself onto a LPO by creating a logical instance of itself and imprinting that instance onto the logical LPO space such that a trace to its physical counterpart is possible and *complete*, i.e. trace information from the logical LPO space can bring about complete access to information stored remotely at the physical LPO space. An act of removal is essentially the reverse concept of an attachment. However, it is only necessary to remove a logical module from the locally located logical LPO space—effectively detaching a link between the physical counterpart and the logical LPO. To promote scalability, physical LPO modules can also make their presences known and identifiable to Internet search engines.

An installer or an uninstaller can be useful for attaching and removing logical LPO modules, be they platforms or components. The use of an installer/uninstaller does also offer another benefit; it makes a LPO more user-friendly and attractive, resulting in a higher frequency of LPO utilization. In *The Selfish Class* [69], the authors suggest through the *Work Out Of The Box* pattern that a higher rate of software object utilization usually results in a better rate of survival. This principle applies to the use of default arguments as well.

4.1.4 Operations

LPO-related operations, chiefly *read* data and *process* data, are performed by LPO tools. Each LPO tool must be able to read needed data from the XMI inputs and/or the platform components, and then process them according to its intended task.

A LPO may also furnish the developer with a user-interface tool that makes interactions between the developer and the LPO efficient. This tool shall support the Create/ Read/Update/Delete (CRUD) operations, similar to the basic operations found in most Object Oriented and Query languages, to help manage local modules in the logical LPO space. Moreover, because of the information-rich nature of a LPO, the user-interface tool should also be able to process data contents and present them in a human-readable format, in much the same way as web browsers do.

By observing how a LPO comes into existence, it is obvious that each platform must also carry with it a user-interface tool. Since it is possible that a LPO can host more than one platform, there can very well be more than one user-interface tool within a LPO—each one comes with and belongs to each platform, and can be tailored specifically to suit the platform's characteristics. The platform that first initiates the LPO creation shall lend the service of its user-interface tool to help manage modules in the locally-located logical LPO space.

4.1.5 Interactions

The platform-centric SoC design method relies heavily on two mechanisms, namely, the UML and a LPO. These mechanisms belong to two different application domains; UML applications are allowed to request cross-domain services from a LPO. Because of its nativeness to UML and its being based on XML just like the LPO, it is envisaged that the XML Metadata Interchange format (XMI) could be a natural choice that provides a standard means for representing inter-domain communications. Operations and components

provided by a LPO constitutes a processing power and development resources to drive the design flow for applications modeled in the UML domain.

Utilizations of LPO components during the development process, either in the context of target architecture construction or application design, are achieved through the use of UML's Package which permits these components to be imported into the UML application domain for further reuse. As such, it is imperative that each LPO component provide a UML package whose contents contain links to itself in a LPO, and may describe the behaviors and characteristics that will promote the reuse of its own self in the UML domain.

4.2 XML Viewpoint

This section describes a LPO from an implementation point of view, where XML promises to be a convenient, yet effective, tool for realizing the library. To begin, the mapping from the conceptual viewpoint to the XML equivalents are presented. The actual implementation of a LPO then ensues that lays out the general structure, as well as the structures of the schema documents, including relevant tag definitions, that constitute the physical artifacts of the LPO.

4.2.1 Mapping of Conceptual LPO to XML Equivalents

Not all the LPO characteristics from the conceptual domain can be explicitly mapped one-to-one into the XML domain. However, by imposing certain rules to those with no direct mapping, XML could suffice to completely realize all the characteristics of the LPO.

The logical LPO space as defined in Definition 4.3 can be regarded as a locally-located database that logically represents a larger set of real data whose locations could be anywhere in the physical LPO domain. When physical LPO modules are attached to or removed from a LPO, this local database is modified to reflect such changes. In the XML domain, such a local database manifests itself as a register file.

Definition 4.5: Register File

An editable document that holds information about the identity and whereabouts of either the platform objects (PO) or PO member modules (POmm), as defined by the *lpoRegfile* and *poRegfile* schemas, respectively.

4.2.1.1 LPO Register File

The existence of the LPO register file signifies the existence of a LPO. It is created when the first platform object is installed. Each subsequent PO installation updates the LPO register file in order to declare its presence in the LPO (via the logical LPO space).

Axiom 4.1 For the Library of Platform Objects L, L exists if and only if the corresponding LPO register file R_L exists.

Axiom 4.2 Let S_{RP} be a set of PO register files in L. R_L exists if and only if S_{RP} is a non-empty set.

Axiom 4.3 From Axioms 4.1 and 4.2, it follows that L exists if and only if S_{RP} is a non-empty set.

4.2.1.2 PO Register File

The existence of the PO register file signifies the existence of a platform object (PO). It is created when the platform object (PO) is installed. Each subsequent POmm installation updates the PO register file in order to declare its presence in the PO.

Axiom 4.4 For the platform object P, P exists if and only if the corresponding PO register file R_P exists.

Tag syntax definitions of both the LPO and PO register files are defined in XML schema documents, namely *lpoRegfile.dtd* and *poRegfile.dtd* (see Appendix D). The register files *should* only be updated through the installation/uninstallation processes.

4.2.1.3 Auxiliary Information

Because XML that implements a LPO depends on schema documents to define tag syntax and semantics, present also as part of the library is the fourth data category—the *auxiliary information*.

In addition to schema documents, the category actually encompasses anything at all that cannot be classified into components, tools or platform information, but are present either as requirements for proper functioning of a LPO or as auxiliary entities for efficiency gains. Other auxiliary information may include user-defined definitions to be used by the schema documents, or run-time configuration files used by the user-interface software, as well as the register files.

4.2.1.4 Structure of LPO

Figure 4.1 illustrates the structural organization of a LPO. UML's *Aggregation* notations depicted as straight lines with hollow diamonds, are used to represent hierarchical containment relationships, where a module on the hollow diamond end contains the other module. Straight lines connecting modules simply show *Association* relationships among them.

Definition 4.6: Platform Object Member Module (POmm)
A member of a particular platform object that is used to design and construct a system. Four kinds of module exist in a platform object, corresponding to four LPO data categories: *component*, *tool*, *architecture blueprint* (or platform information), and *auxiliary.*

Definition 4.7: Architecture Blueprint (AB)
Sometimes referred to only as *blueprint*, it is an abstract, logical view of the PO architecture(s). It corresponds to the *template* notion introduced in Section 5.1.1.

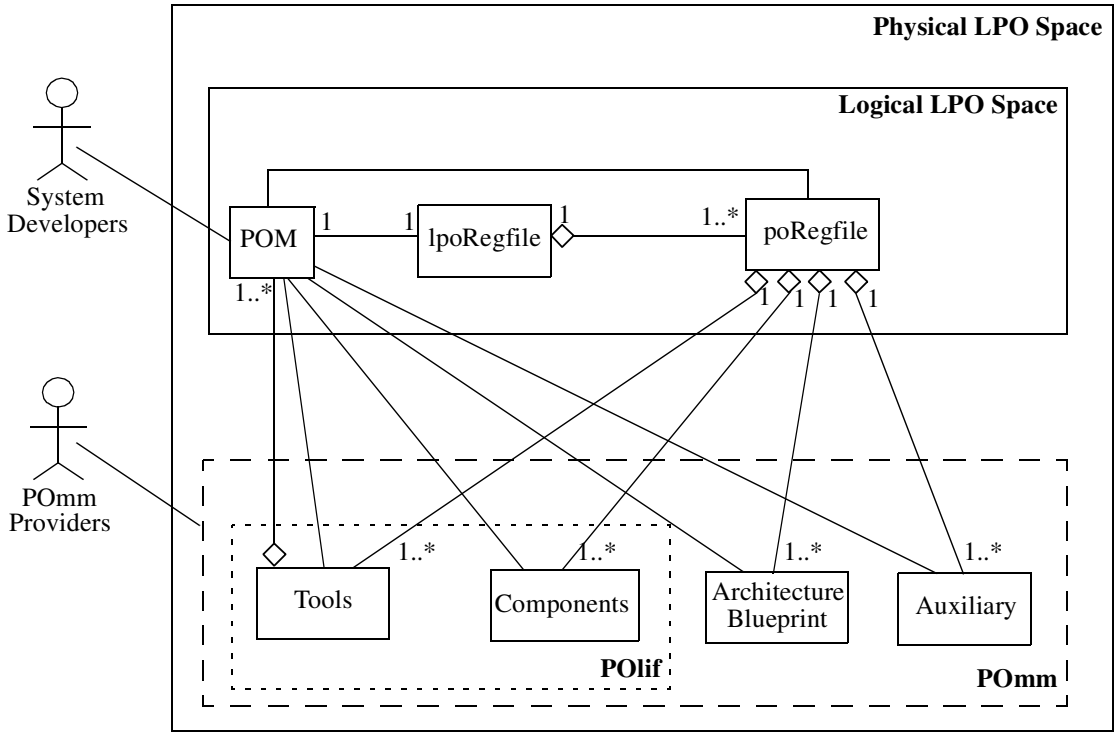


Figure 4.1: Structural organization of the LPO

Definition 4.8: Platform Object Logical Interface (POlif)

A collective term used to describe the POmm modules, i.e. *POmm/tools* and *POmm/components*, that provide a logical gateway to the corresponding physical entities existing elsewhere in the physical LPO space.

Definition 4.9: Platform Object Manager (POM)

A software managing tool whose *main* task is to provide for the system developer a user interface to the member modules of the same PO it belongs to. It corresponds to the concept of user-interface tool described in Section 5.1.4.

As shown in Figure 4.1, a POM should be capable of providing the sole point of access to the PO domain it belongs to. By further elaborating on POM functionalities, a

seamlessly unified tool environment for the proposed approach could possibly be attained. Figure 4.1 also roughly shows the interactions between the LPO and its environment, i.e. system developers and POmm providers.

4.2.1.5 Tag Syntax and Semantics

A LPO can become more efficient if its XML tag syntax and semantics possess a certain degree of flexibility that enable them to be tailored specifically for each platform. Nonetheless, for a LPO to function for the proposed platform-centric SoC design method, a common set of LPO tag syntax and semantics must be agreed upon.

To promote (1) a possible sharing of multi-PO POlifs, and (2) the use of a common POM to perform basic services for all the POs, all register files in the LPO shall resort to one common set of XML tag syntax and semantics. Also, all XML documents within each PO domain—specifically, the POlif domain—must utilize the same syntax and semantics. In addition, all XML and XMI documents must be valid when checked against appropriate schema files.

Let P be a platform object in a LPO L. Let $D_u \in P$ be a set of XMI documents, $D_p \in P$ be a set of XML documents in the POlif, $S \in P$ be a set of the schema documents associated with each elements in D_p, and μ be an official OMG's XMI schema. Also, let *isValid(x, y)* be a function that returns TRUE if an XML document *x* is *valid* when checked against a schema *y*, and returns FALSE otherwise.

Axiom 4.5 A platform object P can only effectively contribute to the proposed platform-centric SoC design method if, (1) $\forall d_u \in D_u$, *isValid*(d_u,μ) is TRUE, *and* (2) $\forall s \in S$ and $\forall d_p \in D_p$, *isValid*(d_p, *s*) is TRUE.

Furthermore, for any active PO, there must exist at least three schema documents at all time: the *polif* and *poRegfile* schemas, and the XMI schema as defined in UML [24].

4.2.1.6 Platform Objects

Platform objects are designed off-cycle. By so doing, the enhanced flexibility offered by the proposed approach, or any platform-based design, is somewhat compromised. In return, however, the platform-centric approach achieves greater potential to attain the feasible design correctly and more quickly by pre-designing, pre-characterizing certain aspects of the system, as well as providing guidance both in the forms of constraints and related information so that the system developer can make better design decisions at all stages of the design.

A PO enters the LPO by updating the LPO register file to record its presence. Upon entry, various tasks *may* also need to be carried out and proper values configured. These configuration tasks *may* involve:

- determining and selecting existing design tools, and their execution paths,
- determining design tools that must be installed,
- determining the type of a computing platform the POM will be deployed,
- determining if a LPO has already existed.

An installer and/or a clear installation instruction *should* always be provided to ease the installation process.

Axiom 4.6 Let L be a LPO, and n be the number of POs in L. Then if there exists L, then L must contain at least one PO as its library member, i.e. $exist(L) \rightarrow (n \geq 1)$.

At installation time, the PO checks for the existence of the LPO register file to determine whether or not a LPO exists. If it finds no LPO register file, it creates one and updates the file to declare its presence. Then it creates its own PO register file and performs necessary updates to reflect the existence of the POmm modules that are vital to its existence. These POmm modules include:

- The architecture blueprint (AB), and POmm/components that build it,
- Associated POmm/tools, including the PO managing software (POM),
- Schema documents
- Knowledge-based information

It is expected that most of the PO for the proposed approach will come as a result of modifying existing system platforms. Two broad classes of system platforms are likely to become common in the LPO:

1. Full-application platform, which allows full applications to be built on top of hardware and software architectures. In general, the blueprint will consist of a processor and a communication infrastructure. The POmm/components are composed mainly of application-specific blocks that will probably share POmm/tools. Examples include Philips' Nexperia [79] and TI's OMAP multimedia platform [80].
2. Fully programmable platform, which typically consists of a FPGA and a processor core. Communication infrastructure is often synthesized along with the core, on an as-needed basis, during configuration. Examples include Altera's NiOS [81], Quicklogic's QuickMIPS [82], and Xilinx's Virtex-II Pro [83].

A choice of PO communication infrastructure often plays a vital role in achieving high performance and great flexibility. Besides easy integration, desirable I/O subsystems should also take into account scalability and parametrizability. CAN [85], FlexRay [86] and I^2C [87] represent a subset of current cutting edge embedded system I/O technologies that can potentially be used to build a platform. An I/O subsystem that allows its power to be configured [88] is also an attractive choice. In any case, proper documentation *should* always be exercised.

4.2.1.7 Architecture Blueprint (AB)

Because a PO either often comprises of a family of processors or is fully programmable, it can be affiliated with more than one target architecture. A blueprint is an abstract, logical view of these architectures.

Working with a blueprint instead of the physical model consisting of UML nodes and components can be more attractive for the system developer. It is a convenient means to represent a set of all possible target architectures attainable per platform. Moreover, it provides a simple yet powerful mechanism for dealing with the issue of hardware implementation that, otherwise, would not be satisfactorily addressed were the developer to adhere to the UML physical model. As will be shown later in Section 5.6, homomorphic mapping between the logical model of the chosen target architecture derived from an architecture blueprint and the physical model, if it were to be used, is possible, and thus, furnishing a proof that the two views are, in fact, equivalent and interchangeable.

At minimum, an architecture blueprint *must* consist of an abstract representation of a processor and a communication infrastructure, the latter of which may further consist of one or more I/O subsystem abstracts. Given the current trend in platform technologies, an architecture blueprint that represents a family of processors or multi-processors, rather than a traditional uni-processor, will not be uncommon. A processor *may* contain one or more internal storage elements and/or have external storage elements as an additional AB requirement.

An architecture blueprint is an integral part of the proposed approach; it exists mandatorily for every PO in the library. Because there always exist at least one processing unit and one I/O component for every PO, it could be further deduced that if a blueprint exists, the POmm/components that are used to construct the concrete platform architecture must also exist.

Axiom 4.7 Let P be a platform object and $S_{AB} \in P$ be a set of the POmm/ components that can be instantiated into the blueprint to implement the concrete platform architecture. Then $exist(P) \rightarrow (S_{AB} \neq \varnothing)$.

4.2.1.8 Platform Object Logical Interface (POlif)

By definition (Definition 4.8), POlif is a collective term used to describe *POmm/tools* and *POmm/components*. POlif modules are the core database of the LPO. Like others, POlif modules are self-descriptive. When implemented using XML, the POlif shall carry its own schema document to differentiate itself from the auxiliary domain of the register files. They shall also permit keyword description of themselves to aid search engines.

A POmm/tool is a logical interface module to the corresponding tool that may physically exist anywhere in the physical LPO space. It contains essential information about itself, especially its identity, and resource locations, that could be configured during the installation process.

A POmm/component, on the other hand, carries a much heavier load of data than a POmm/tool does. In addition to the information about its identity and resource locations, it contains information regarding its characteristics, UML representation, and possibly HW-dependent software routines, as well as tool associations. For each POmm/component's characteristic, information about *name*, *type*, *value*, and *unit* shall always be recorded.

4.2.1.9 Resource Locator

All resources in the LPO should be specified using one common format. It is recommended that the LPO follow the Universal Resource Identifier (URI) format [89] when specifying resource locations in XML documents—the obvious reason for it being the compatibility with the Internet standard. The most common form of the URI is the Universal Resource Locator (URL).

4.2.1.10 Platform Object Manager (POM)

A POM is a software managing tool whose *main* task is to provide for the developer an interface to the member modules of the same PO it belongs to. A POM supports functionalities similar to those of the Facade object [68] which provides a unified interface to a set of interfaces in a subsystem. Like the Facade object, a POM *should* be able to delegate the design, either parts or whole, to appropriate POmm modules. If all members of the LPO adhere to the same tag semantics, a POM can extend its services to encompass all LPO members, including the LPO register file, and all PO register files. Let P be a platform object that contains a set of POmm/tools, P_T. And let $T_{POM} \in P$ be a POM.

Axiom 4.8 $\quad exist(P) \rightarrow (exist(T_{POM}) \wedge (T_{POM} \in P_T))$.

POM's basic operations involve *extracting and processing XML-based data*. To extract data, a POM *may* simply make use of existing XPath engines as convenient tools. These tools typically take an XPath expression as the argument, and often provide the programming interfaces to the popular programming languages like C++ and Java [74 75, 76]. Because of the facility and functionalities offered by XML and XPath, respectively, data embedded in a POmm module are readily accessible so far as their semantics are clearly understood. A POM can construct, as well as reconstruct, any XML tree from the associative schema document. A path expression for each element and attribute can then be acquired simply by traversing the XML tree.

The other primary POM operation is processing the extracted data. In order for the system developer to make use of the available data to the fullest, a POM *should* also behave as a user interface that can perform such tasks as displaying data, and relevant information in a user-friendly format, gathering input information for POmm/tools, searching and locating POmm modules, etc. At the minimum, a POM *should* support the following operations:

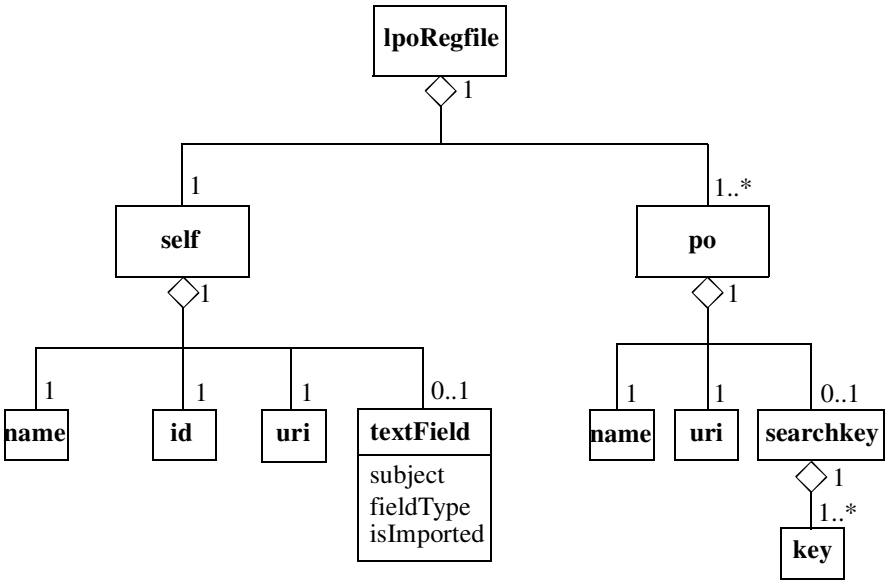


Figure 4.2: Hierarchical structure of the *lpoRegfile.dtd*

- Searching, and fetching LPO modules, e.g. POlif, the architecture blueprint
- Displaying, and formatting data, e.g. displaying Help pages, listing available POmm/components. XSL/XSLT [71] may be of great use for this purpose.
- Providing links, and allowing user selections so that such tasks as accessing remote content, activating POmm/tools, etc., may be implemented. XML applications such as XLink [58] and XPointer [59] could be useful.
- Easing POmm/tool usage. A POM *may* (1) prepare a batch file, (2) provide step-by-step instructions on how to run a POmm/tool on a particular set of inputs, and/or (3) create tool menus that link tool and inputs together, and that permit tool activation. A POM *may* also define a unified environment that allows POmm/tools to interact, e.g. the OMG's CORBA IDL [53] that permits the compliant objects to communicate through an object broker.

4.2.2 Implementation

As the *tree* view structure of XML documents are easier to follow, it will be adopted as a means to describe the implementation of the LPO. Then, the mapping of a tree structure to an equivalent schema document is quite simple and intuitive.

UML's Class diagrams are used to model the tree structure. The root element of an XML tree resides solely at the topmost hierarchical level. Child elements that branch out of their parent are connected to the parent through the UML Aggregation, with the hollow diamond attached to the parent. An order of child elements are significant and is mapped from left to right onto the schema document. Leaf nodes in the tree structure represent the XML terminal nodes that contain strings of type #PCDATA. UML attributes map into required XML attributes; whereas, UML multiplicity becomes the equivalents in XML. The actual DTD documents that implement the LPO are included in Appendix D.

4.2.2.1 LPO Register File

Figure 4.2 depicts the hierarchical structure of the LPO register file (*lpoRegfile.dtd*). Due to dynamic nature of resources, a fail-safe principle of redundancy is exercised to ensure consistency and reliability. Consequently, multiple elements may exist solely to identify a single resource. A description of each element is listed below:

Name	Type	Multiplicity	Description
lpoRegfile	Root	1	Signifies existence of LPO
self	Element	1	Self identification
po	Element	1..*	Link to platform objects
searchkey	Element	0..1	Relevant keywords that can identify self
name	Leaf	1	Self ID by name
id	Leaf	1	Self ID by special identification

Name	Type	Multiplicity	Description
uri	Leaf	1	Self ID by location
textField	Leaf	0..1	Knowledge-based information
key	Leaf	1..*	Keyword string

Attributes are listed as follows:

Name	Type	Base	Description
subject	CDATA	textField	Subject of information
fieldType	CDATA	textField	Expected data format
isImported	Enumeration: "yes" or "no"	textField	Specifies if the content contains link to an imported document

4.2.2.2 PO Register File

Figure 4.3 depicts the hierarchical structure of the PO register file (*poRegfile.dtd*). Same XML element names share syntax and semantics; thus, only descriptions of new elements are presented.

Name	Type	Multiplicity	Description
poRegfile	Root	1	Signifies existence of PO
pom	Element	1	Link to POM
blueprint	Element	1..*	Link to architecture blueprint(s)
poSchema	Element	1	Link to a PO schema document
polif	Element	1..*	Link to POmm
poID	Leaf	1	Reference to the PO to which the register file belongs

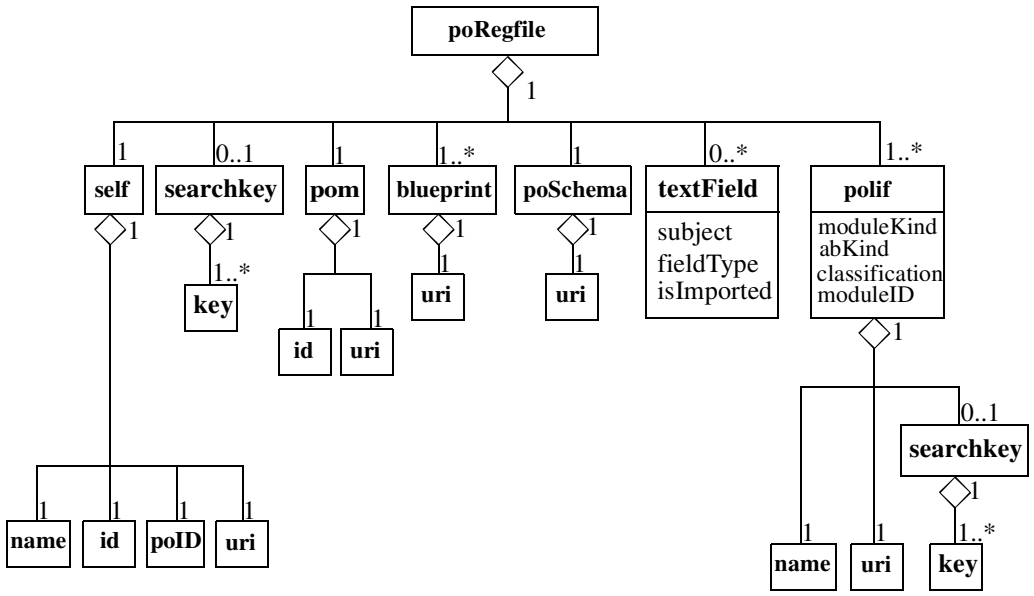


Figure 4.3: Hierarchical structure of the *poRegfile.dtd*

Attributes are listed as follows:

Name	Type	Base	Description
moduleKind	Enumeration: {“component”, “tool”}	polif	Classifies itself to be either *component* or *tool*
abKind	Enumeration: {“pru”, “iu”, “diu”, “ifu”, “mu”, “clock”, “timer”}	polif	Classifies itself to be one of the blueprint types (see Section 5.6.1 for detail).
classification	Enumeration: *PO-dependent*	polif	User-defined category of the module
moduleID	CDATA	polif	Reference to a POmm via ID. Provides a security measure to *name* and *uri* references.

4.2.2.3 POlif

As evident by prior discussions, the POlif constitutes the core of a LPO database. POlif modules, *POmm/tools* and *POmm/components*, contain information necessary for characterizing themselves to be used with the platform-centric SoC design method. To promote scalability, each POlif module is associated with a unique XML document, which is defined by a schema document illustrated as the tree structure in Figures 4.4 - 4.8.

In Figure 4.4, most elements are re-used. Those that need be defined are:

Name	Type	Multiplicity	Description
polif	Root	1	Signifies existence of a POmm
selfURI	Element	1	Possible locations that it may reside
physicalURI	Element	1	Possible locations that the corresponding physical module may reside
installerURI	Element	0..1	Link to an installer
uninstallerURI	Element	0..1	Link to an uninstaller
componentDomain	Element	0..1	Compartment for information about POmm/component. It is not used if a POlif is of type *tool*.
associatedTools	Element	0..1	Specifies possible association between a POmm/component and a POmm/tool(s)
uml	Element	1	Specifies UML representation of the module
functions	Element	0..*	Supplies information, if there is any, about hardware-dependent software routines
characteristics	Element	0..1	Contains *databook* information of the module

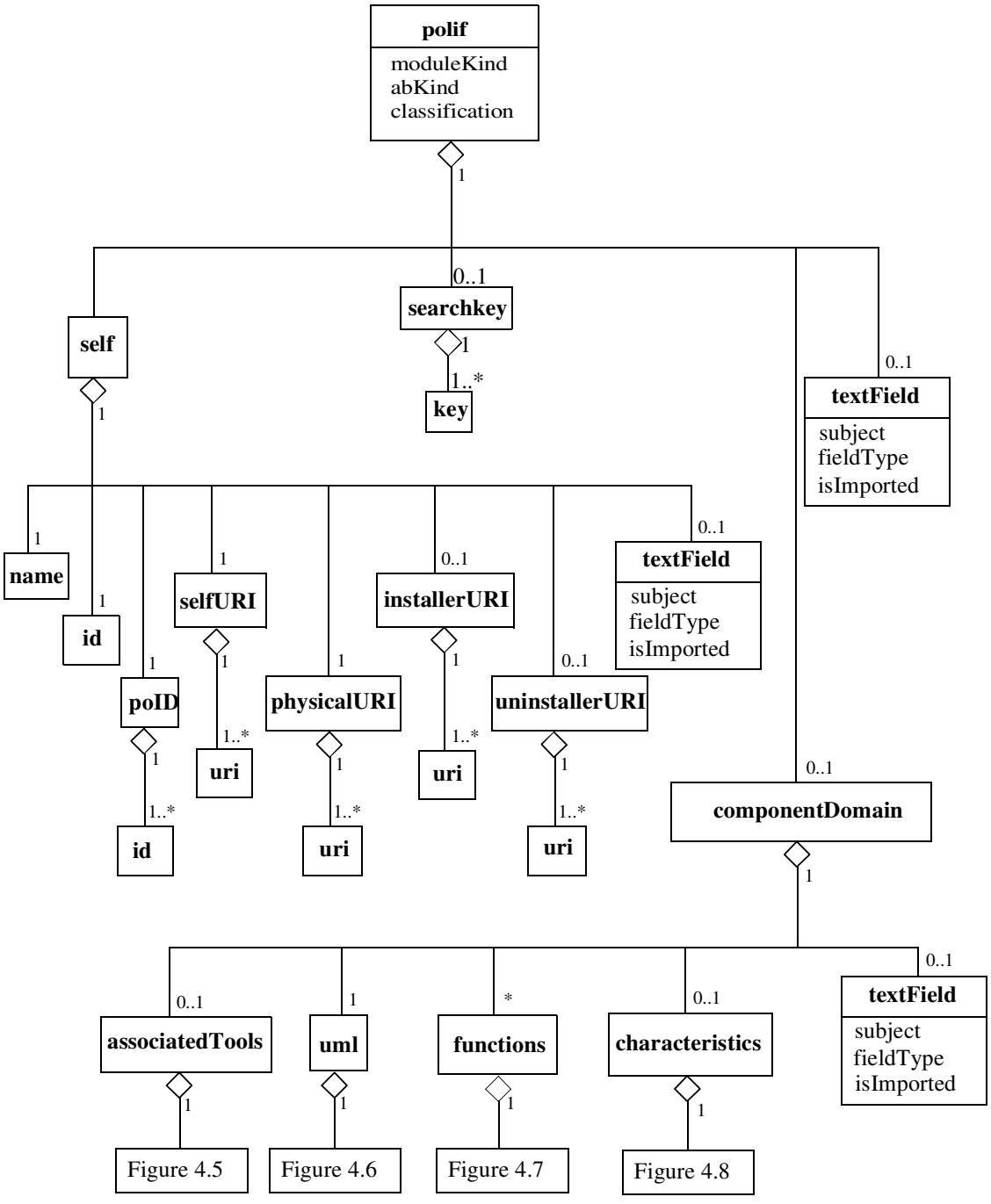


Figure 4.4: Hierarchical structure of the *polif.dtd*

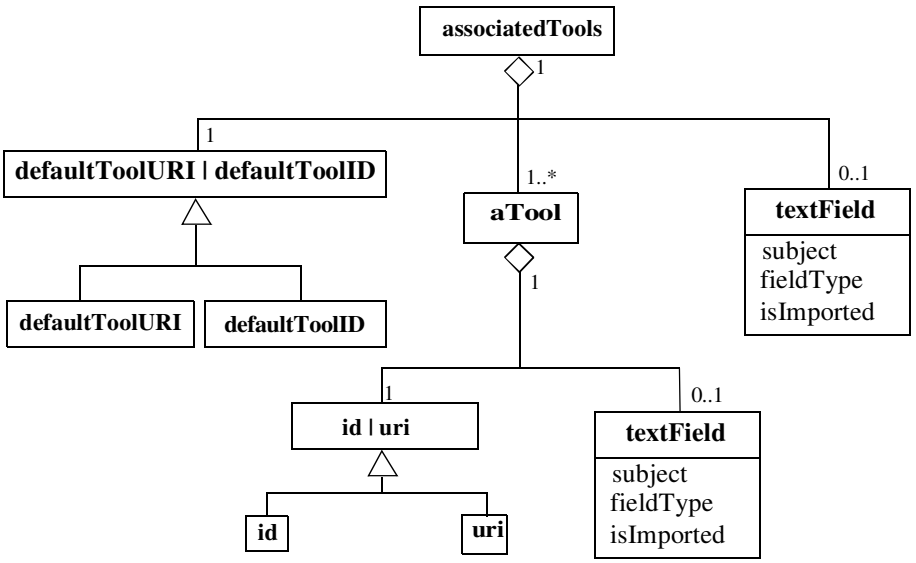


Figure 4.5: Detailed hierarchical structure of the associatedTools element.

Figure 4.5 depicts a detailed structure of the associatedTools element. This tag element only supports a simple association between a POmm/component and one or more POmm/tools. It is expected that, when given the POmm/component identity in the LPO, the associated POmm/tool possesses the knowledge on how to process it. Subelements that need to be defined for the associatedTools elements are:

Name	Type	Multiplicity	Description
defaultToolURI	Leaf	1	Default tool by URI. Cannot co-exist with defaultToolID.
defaultToolID	Leaf	1	Default tool by ID. Cannot co-exist with defaultToolURI.
aTool	Element	1..*	Identity of each associated tool

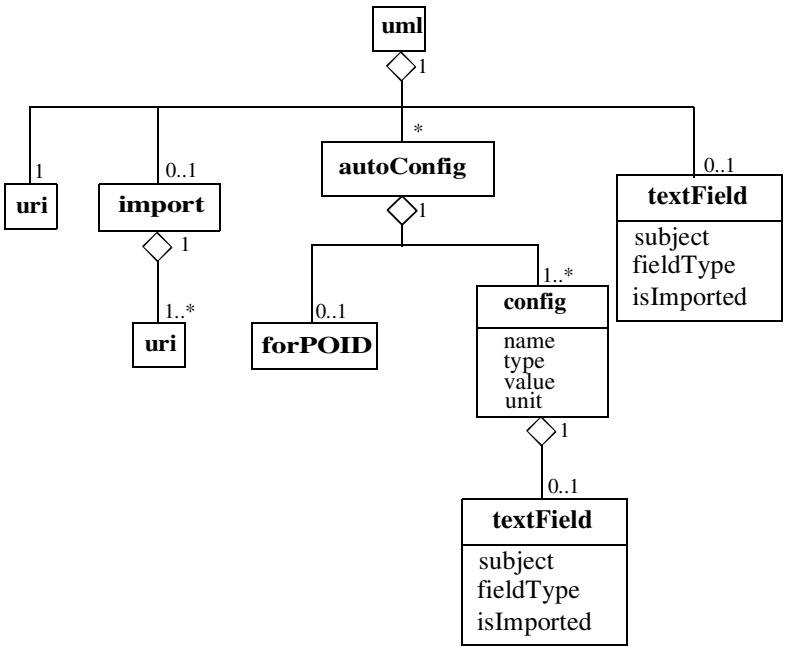


Figure 4.6: Detailed hierarchical structure of the uml element.

Figure 4.6 shows the detailed structure of the uml element. Subelements that have not yet been defined are described below.

Name	Type	Multiplicity	Description
import	Element	0..1	Reuse mechanism that allows UML packages to be imported
autoConfig	Element	0..*	Compartment that holds pre-configured values for UML parameters
config	Element	1..*	Pre-configured UML parameter values
forPOID	Leaf	0..1	ID of platform object that these pre-configured values are applicable for

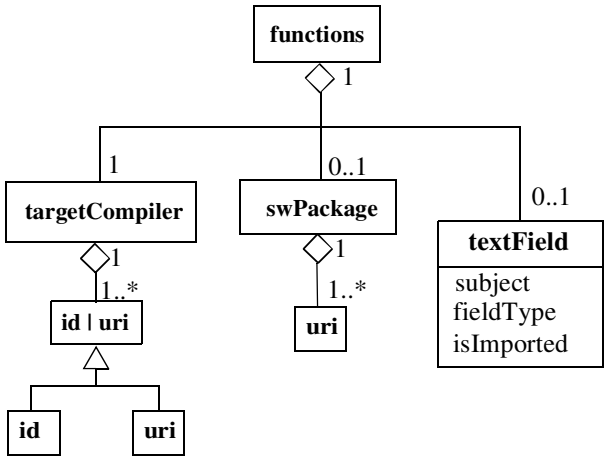


Figure 4.7: Detailed hierarchical structure of the functions element.

The config subelement also have attributes associated with it. They are:

Name	Type	Base	Description
name	Enumeration: *PO-dependent*	config	Name of config data
type	Enumeration: *PO-dependent*	config	Predefined type of config data
value	CDATA	config	Value of config data
unit	Enumeration: *PO-dependent*	config	Unit of config data

The construct of the config element, as well as the preDefined, and userDefined characteristics to be discussed later, adapts the PROPERTY pattern [90] in such a way that a number of such tags that can be cataloged in an XML document can vary without any changes to the schema document. This dissertation expects that POlif providers would eventually agree upon a comprehensive set of predefined types and units, as well as UML parameters and component characteristics. Then fairly standard *enumeration* types can be defined in the schema document.

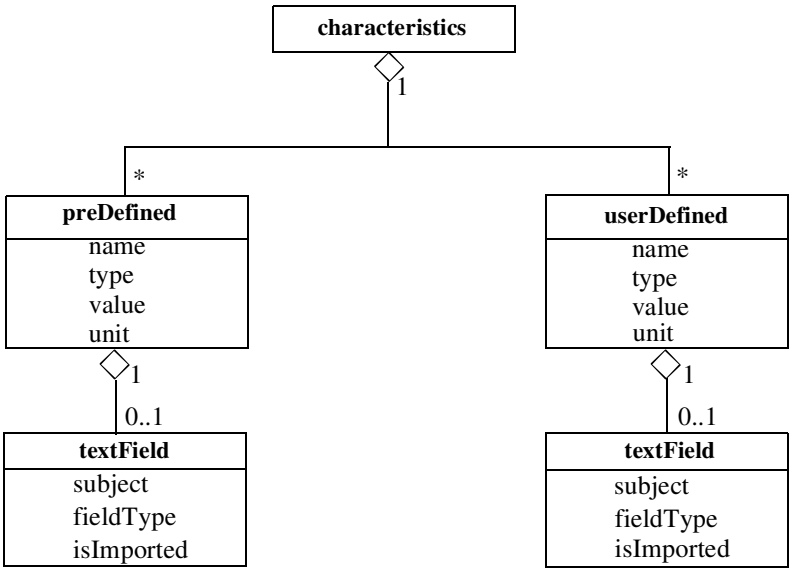


Figure 4.8: Detailed hierarchical structure of the characteristics element.

Figures 4.7 and 4.8 depict the tree structures of the functions and characteristics elements, respectively. The functions element handles information pertinent to hardware-dependent software routines. Within it, the target compiler and locations of included files are specified.

The following completes the element descriptions of the POlif schema document.

Name	Type	Multiplicity	Description
targetCompiler	Element	1	Expected target compiler
swPackage	Element	0..1	Reference to the software
preDefined	Element	0..*	Pre-defined component characteristics
userDefined	Element	0..*	User-defined component characteristics

Permissible attributes are defined as follows:

Name	Type	Base	Description
name	Enumeration: *PO-dependent*	preDefined	Name of preDefined characteristics
name	CDATA	userDefined	Name of userDefined characteristics
type	Enumeration: *PO-dependent*	preDefined, userDefined	preDefined and userDefined type of characteristics
value	CDATA	preDefined, userDefined	characteristics value
unit	Enumeration: *PO-dependent*	preDefined, userDefined	Unit of characteristics

Chapter 5

UML Profile for Codesign Modeling Framework

Using UML, this chapter describes the framework, that, together with the framework in the UML profile for schedulability, performance and time specification, or succinctly the UML real-time profile [29], constitutes the core concept for developing real-time embedded SoC systems in the platform-centric design environment. This framework, called the Codesign Modeling Framework, builds upon the real-time foundation provided by the UML real-time profile; it exists as a supplemental package, and not as a replacement. The UML real-time profile is now in the final phase before OMG standardization.

The UML real-time profile offers a facility for modeling and analyzing real-time applications. Such a facility proves adequate for most software development processes. However, in the codesign environment where hardware and software developments often take place simultaneously, the profile becomes less useful—it is less capable of coping with the hardware development, let alone the complexity of the codesign environment where heterogeneous development processes intermingle systematically.

The UML profile for Codesign Modeling Framework, is aimed at mending such issues. The profile adds to the UML real-time profile the frameworks for modeling exceptions (EMprofile), interrupts (IMprofile), synthesizable HDL (SHDLprofile), as well as an architecture blueprint (ABprofile) that is used as a template to construct the target architecture. The chapter presents details for each profile individually, starting with the utility

package (PCUprofile) that provides generic utility extensions for the framework, followed by the EMprofile, IMprofile, SHDLprofile, and ABprofile profiles, respectively.

5.1 Codesign Modeling Framework in Principle

The Codesign Modeling Framework contains a collection of codesign-oriented, real-time profiles whose intent is to enhance the proposed platform-centric SoC design approach. It works in conjunction with the UML real-time profile so as to supplement it with codesign modeling capability. The objectives of the framework detail as follows:

- Permit heterogeneous modeling of hardware and software in the same unified design environment,
- Support modeling and elaborating of an architecture blueprint that results in the target architecture,
- Enable one-to-one mapping of UML to synthesizable hardware description language (HDL),
- Together with the UML real-time profile, provide a standard means for representing LPO tools and components, thus, promoting reuse, and
- Enhance design for reliability by including frameworks for exception and interrupt modeling.

Figure 5.1 depicts the structure of the Codesign Modeling Framework, as well as the relationships among participated packages and actors. In modeling a system for the platform-centric method, the *developer* derives the target architecture (*TargetArchitecture* package) from an architecture blueprint (*ArchitectureBlueprint* package) supplied by the platform provider, i.e. the *LPO module provider*, and utilizes the derived architecture as the hardware reference for developing software applications (*MyModel* package). Marked by codesign characteristics, the processes of selecting the target architecture and developing software applications can be performed in sequence, in parallel, or iteratively.

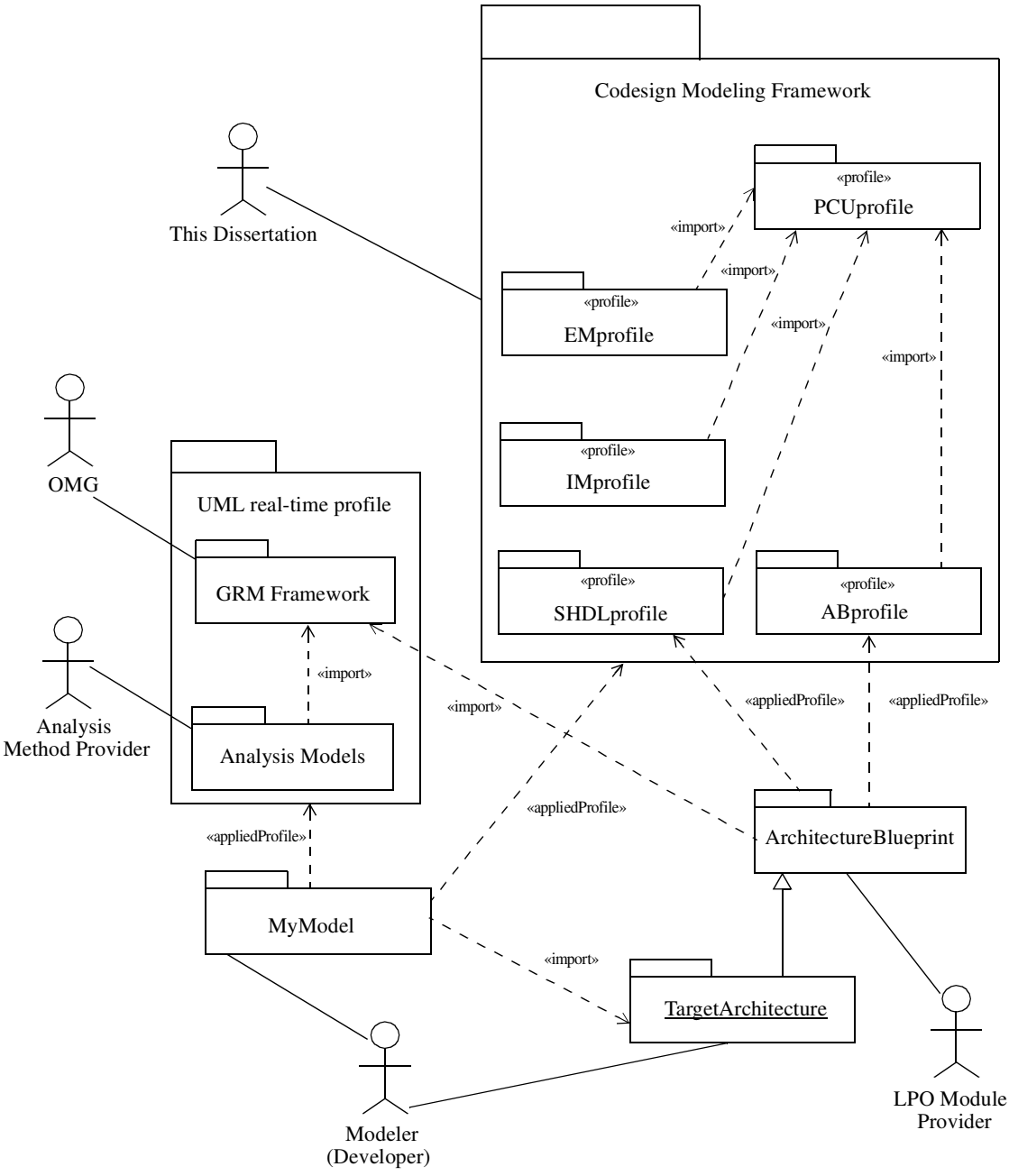


Figure 5.1: Structure of the UML Profile for Codesign Modeling. Also shown are anticipated relationships among participated packages and actors.

In presenting each profile in the Codesign Modeling Framework, this dissertation adopts the two-viewpoint presentation scheme employed in the UML real-time profile specification [29]. The first is the *domain viewpoint* that "captures, in a generic way, the common structural and behavioral concepts" that characterize each profile. The other is the *UML viewpoint*, which is "a specification of how the elements of the domain model are realized in UML." The UML viewpoint identifies the required UML extensions, i.e. stereotypes, tagged values and constraints, and groups them in a profile package. Unlike the domain model, however, the UML viewpoint represents the concrete realization rather than the abstract concepts. As such, these extensions do not necessarily map one-for-one with the domain model.

It shall be noted, however, that all of the extensions to be presented are light-weighted, which means they require no fundamental change to UML.

5.2 Platform-Centric Utility (PCUprofile)

This section introduces generic utility extensions that serve various purposes to enhance the robustness of the proposed approach. These utility concepts are often disjoint, and are expected to be used by most models and profiles within the platform-centric environment.

5.2.1 Domain Viewpoint

5.2.1.1 Main Function Designation

Employing the Codesign Modeling Framework, UML models are likely to be mapped into more than one language, e.g. C# for application and synthesizable Verilog for hardware implementation. Automatically determining proper main functions in such models can be complicated for each programming language has its own way of representing the function. For example, C/C++/Java use the keyword main, and Pascal uses program, while such programming languages as Ada/VHDL require explicit designation at link time, and, thus, any procedure/entity can be main.

Such complexity can be alleviated by explicitly designating the main function on a proper method in the model, and leaving the language-specific main construct to the code generator. This approach results in the model being more uniform and readable.

5.2.1.2 Link from UML to LPO

In the platform-centric environment, the Codesign Modeling Framework uses and reuses LPO resources, whose existences and availability are identified through the use of POM's services. However, to reduce communication overhead during LPO module import, only the resource models represented in UML should be utilized. To later retrieve relevant information, such models must maintain appropriate links to their parent modules in the LPO.

5.2.1.3 Package Processing Instruction

The developer uses LPO components, i.e. POmm/components, for developing the system by means of UML packages. Not all LPO components, however, have hard characteristics. POmm/components, e.g. software library or legacy code, may need to be modified and/or compiled as part of the development process. In some cases, the UML package may just come raw and the appropriate processing steps can only be determined when the configuration of the hardware platform is known. Therefore, a way for specifying default package processing instruction and/or user-defined instruction is preferable.

5.2.1.4 Code Insertion

Although the Codesign Modeling Framework does allow homomorphic mapping of UML to code up to a certain degree, it has no intent to attain absolute formalism. The efficiency in developing systems of great complexity like today's real-time embedded systems relies also on the principle of design with reuse (DwR) [107] and good analysis tools, both of which do not mesh well with the formalism concept. For analysis, UML already is a great tool. For reuse, it relies on the notion of package.

Code reuse in UML is often tool-dependent. Code is grouped by a UML tool into a library and modeled using UML package. Another means to achieve code reuse in UML is by inserting pieces of code into the model and associating them with the desired methods. Then a code generator can produce full functional source code from the model.

While some better tools support an elaborated code insertion scheme, some do not and only dwell on the mechanism suggested by the UML specification—placing texts of code on a UML note. This simple scheme works fine when local variables can be declared within the method body delimiters like in the languages such as C++ and Java. However, it becomes awkward and more complicated when the declarations have to be done outside of the method body delimiters like in Ada and VHDL. Good tools will still be able to handle it, nonetheless, with more effort. To make code insertion in the Codesign Modeling Framework as general and as uniform to many programming languages as possible, it is recommended that, for each method to be given a piece of code, it designates a declaration area and a body of method area apart from each other, such that a code generator can process the inserted piece of code with little effort and no ambiguity.

5.2.1.5 Non-design Variables

Many times in a course of the development process, the developer will want to use non-design variables for various specific purposes not pertinent to the actual development of the system. An obvious example of non-design variables includes constraint variables that capture non-functional system characteristics such as power dissipation, environmental requirements, and rigid form factors. These variables are predominantly used during the validation process.

Because some profiles and/or stereotypes, e.g. the «SHDLarch», perceive regular design variables as conveying further implicit information, it is required that the tools that will interpret these variables be able to differentiate them from any non-design variable that may be placed in the same context.

5.2.2 UML Viewpoint

In defining UML extensions, i.e. stereotypes, tagged values and/or constraints, for the PCUprofile package, the prefix PCU is always attached to the names to differentiate them from similar or same names in other profiles. This is a standard practice observed in the UML profile for schedulability, performance and time specification, and is exercised here to attain the same clarity effect.

5.2.2.1 Mapping Utility Domain Concepts into UML Equivalents

The main function concept maps to the «PCUmain» stereotype attached to an operation (method), and has no tagged value associated with it.

When a UML model (class or object) is derived from a LPO module, it is denoted with the «PCUlpoMember» stereotype. Then, the UML to LPO link concept is reified using either the tagged value PCUuri or PCUid or both. Either one of these tags is enough to identify the corresponding POmm/component in a LPO. Nonetheless, both of them are redundantly furnished for reliability.

The package processing instruction concept maps to the «PCUrun» stereotype on a package. The stereotype has two tagged values, namely PCUrunline, and PCUrunfile, associated with it. The PCUrunline tag specifies a command line to be run against the package. The PCUrunfile tag indicates that processing instructions can be found in the specified file.

The concept of code insertion and reuse maps into the stereotypes «PCUcode», «PCUdeclare», and «PCUcodeBody». A Component can be stereotyped «PCUcode» to indicate that it is a file. The «PCUcode» stereotype defines one tagged value, PCUfileUri that specifies the location of the file. The «PCUdeclare» stereotype permits texts of parameter declaration to be inserted; whereas, «PCUcodeBody» treats the whole textual context as a body of the method.

5.2.2.2 UML Extensions

To minimize the possibility of conflict with other profiles, all extensions in this package are PCU-prefixed. The presentations are of tabular format, as suggested by the UML specification guide [24]. For stereotype tables, the fields include a stereotype name, base class and an associated tagged value. When no tag is defined, it is denoted by --None--. Tag tables include the name, type, multiplicity and domain concept fields.

«PCUattribute»

This stereotype provides a utility to designate design variables, and is particularly useful when used together with the «PCUauxAttr» (see Section 5.2.1.5).

Stereotype	Base Class	Tags
«PCUattribute»	Attribute	--None--

«PCUauxAttr»

This stereotype specifies an auxiliary attribute corresponding to the concept of non-design variables as discussed in Section 5.2.1.5.

Stereotype	Base Class	Tags
«PCUauxAttr»	Attribute	--None--

«PCUcode»

This stereotype provides a file insertion mechanism (see Section 5.2.1.4).

Stereotype	Base Class	Tags
«PCUcode»	Component	PCUfileUri

The following tag is defined:

Tag Name	Tag Type	Multiplicity	Domain Concept
PCUfileUri	String specifying the URI of the file	0..1	See Section 5.2.1.4

«PCUcodeBody»

«PCUdeclare»

The «PCUdeclare» and «PCUcodeBody» represent the declaration, and the body of the code to be inserted into the model, respectively (see Section 5.2.1.4).

Stereotype	Base Class	Tags
«PCUcodeBody»	Note	--None--
«PCUdeclare»	Note	--None--

«PCUconfigList»

This utility stereotype permits configuration attributes to be grouped together separately from their parent class.

Stereotype	Base Class	Tags
«PCUconfigList»	Class	--None--

«PCUlpoMember»

This stereotype specifies an affiliation of a UML model to the LPO. The intent is to furnish a tracing mechanism that will allow relevant information stored in the LPO to be accessible via the UML model itself.

Stereotype	Base Class	Tags
«PCUlpoMember»	Class Object	PCUuri PCUid

The defined tags are:

Tag Name	Tag Type	Multiplicity	Domain Concept
PCUuri	String specifying the URI of the module	0..1	See Section 5.2.1.2
PCUid	String specifying the ID of the module	0..1	See Section 5.2.1.2

«PCUmain»

When adorned on an operation, this stereotype designates the operation to be a main function.

Stereotype	Base Class	Tags
«PCUmain»	Method	--None--

«PCUrun»

The stereotype represents the concept of package processing instruction (see Section 5.2.1.3).

Stereotype	Base Class	Tags
«PCUrun»	Package	PCUrunline PCUrunfile

The tags are defined by:

Tag Name	Tag Type	Multiplicity	Domain Concept
PCUrunline	String specifying the command line	0..1	See Section 5.2.1.3
PCUrunfile	String specifying the file that need to be processed	0..1	See Section 5.2.1.3

The following constraints are defined for this stereotype:

- If the «PCUrun» stereotype is used, at least one of the tags must be used.
- Although it seems redundant, using both tags at the same time is permissible.

«PCUuseConfig»

This stereotype binds a «PCUconfigList» class comprising configuration attributes to their parent class.

Stereotype	Base Class	Tags
«PCUuseConfig»	Dependency	--None--

5.3 Exception Modeling (EMprofile)

In many critical real-time systems, dependability is so vital that a failure is unacceptable; a means to detect errors and faults must be exercised so as to prevent unexpected failures from occurring. Exception provides such a means for system developers, and is a subject of this section. The UML specification [24] does offer a suggestion on how the exception facility should be modeled per se. Nonetheless, it comes, not surprisingly, as a general modeling tip for OO programming. This section expands and explores the language horizon, and devises a simple framework that could be used to model exception handling mechanisms in the platform-centric environment.

5.3.1 Domain Viewpoint

Unless indicating otherwise, the discussions in this section and Section 5.4.1, *Interrupt Domain Viewpoint*, are based primarily on a comprehensive survey on the subjects of real-time systems and programming languages by Burns and Wellings [108].

Exception handling facilities render a means for containing and handling error situations in a programming language. Older programming languages, such as C and RTL/2, have no explicit support for exception handling mechanisms; they rely, instead, on implicit programming techniques such as checking for an unusual return value, and/or programming with a non-local goto. Although more recent programming languages, e.g. Ada, Java, often provide explicit support for exception handling facilities, the models adopted by these languages still vary: (1) they may or may not allow an exception to be explicitly represented; (2) an exception may or may not propagate beyond the expected scope of its handler; and (3) parameters may or may not be passed along with the raised exception.

5.3.1.1 Representation of Exceptions

An exception can be detected either by environment or by application, and can be raised synchronously or asynchronously. Most mainstream programming languages are of

sequential characteristics, and their exception handling facilities support only synchronous notifications—leaving asynchronous notifications that are mostly employed in concurrent programming to be manually handled by programmers.

Environment-detectable exceptions, on the other hand, often come pre-defined by programming languages, while application-related exceptions normally are user-defined. In a case where an explicit declaration of both exception types is required, they tend to have the same supertype, e.g. a Throwable class in Java, or an exception keyword in Ada. Otherwise, a type is only pre-defined for environment-detectable exceptions, and an application can throw any type at all as an exception without pre-declaration. An example of this model is a pre-defined exception class in C++.

From a code generator's viewpoint, these constructs for representing exceptions, though diverse, can be produced automatically given that the code generator is language-aware, and it is capable of identifying exceptions in the model.

5.3.1.2 Exception Handler Domain

Depending on the context of computation, an exception may be associated with more than one handler. As a result, a domain, or a region of computation, must be assigned to each handler to prevent them from clashing on each other when an exception occurs. A domain is normally associated with a block, subprogram, or a statement. The majority of mainstream real-time programming languages, e.g. Ada, Java, C++, uses a block to specify a domain for a handler.

5.3.1.3 Exception Propagation

When an error event causes an exception to be raised, and there is no handler for it in the enclosing domain, most mainstream real-time programming languages allow such an exception to propagate to the next outer-level enclosing domain. This propagation can continue on until the exception is handled or the program is terminated when no handler for it is found.

5.3.1.4 Parameter Passing

When a programming language permits an exception to be represented as an object, it is normally possible that parameters may be passed along with the exception notification by means of the object attributes.

5.3.1.5 Post-handling Actions

After an exception is raised and handled, the handler may either return the control to its invoker and the computation resumes, or it may terminate the program altogether. The termination model is what most exception facilities adopt, and is the only model considered in this dissertation.

5.3.1.6 Usage Model

The usage models adopted by most mainstream real-time programming languages follow predominantly the *throw/try/catch* structure. The throw action permits an exception to be raised within an enclosing domain established by the try block. The catch block, then, traps raised exceptions from the associated try block and allows appropriate handlers to be invoked. A specially designated handler is often allowed as a safety measure for the exceptions overlooked by the primary try/catch blocks in order for the program to never fail undeterministically.

It has been shown by Costello and Truta [109], as well as in the work by Lee [110], that C macros can be used to mimic the throw/try/catch structure, making this popular usage model even more uniform among major programming languages.

5.3.2 UML Viewpoint

In defining UML extensions, i.e. stereotypes, tagged values and/or constraints, for the EMprofile package, the prefix EM is always attached to the names to differentiate them from similar or same names in other profiles.

5.3.2.1 Mapping Exception Domain Concepts into UML Equivalents

To cope with a wide variety of exception modeling characteristics, the profile attempts to model a complete set of relevant information. An exception is generically represented as a *signal class* adorned with the «EMexception» stereotype, which is a generalization of the standard «signal» stereotype. This stereotype contains no tagged value, and primarily serves to indicate a special requirement for code mapping (see Section 5.3.1.1).

Where the target programming language permits, parameters may be passed by means of class attributes (see Section 5.3.1.4). The exception handler domain concept (see Section 5.3.1.2) maps to the scope imposed by a State Machine diagram containing the try and catch states (see Section 5.3.1.6). The termination model concept (see Section 5.3.1.5) maps to a state transition from the handler state to the *final* state. The exception propagation concept (see Section 5.3.1.3) is viewed as a propagation of the exception signal from the current enclosing State diagram to the next outer-level enclosing State diagram.

As per the usage model (see Section 5.3.1.6), the throw action maps to the «EMthrow» stereotype that defines the EMthrowType tagged value. The EMthrowType tag lists exception types that can be thrown by an «EMthrowMethod» object. The try and catch blocks map to the stereotypes «EMtry», and «EMcatch» in the State Machine diagram, respectively. The handler that is specially designated to trap all other exceptions that are raised without being caught by the «EMcatch» stereotype is represented by a state adorned with the «EMcatchAll» stereotype (see Section 5.3.1.6).

5.3.2.2 UML Extensions

To avoid any possible duplicate and ambiguity, all extensions defined in this profile are prefixed with EM.

«EMbind»

This stereotype binds the exception to its *throw* and *handler* class, thus effectively modeling the exception mechanism at a higher-level of abstraction (see Section 5.3.1.6).

Stereotype	Base Class	Tags
«EMbind»	Dependency	--None--

«EMcatch»

This stereotype models the catch structure concept as discussed in Section 5.3.1.6.

Stereotype	Base Class	Tags
«EMcatch»	SimpleState CompositeState	--None--

«EMcatchAll»

When used after the «EMcatch» block, this stereotype allows the exceptions that are raised but not caught by the catch block to be trapped. In this context, its function resembles the finally and others clauses in Java and Ada, respectively. However, when used alone, it will catches all exceptions, and thus, can be translated to catch(...) in C++.

Stereotype	Base Class	Tags
«EMcatchAll»	SimpleState CompositeState	--None--

«EMexception»

This stereotype models the exception representation concept (see Section 5.3.1.1).

Stereotype	Base Class	Tags
«EMexception»	Signal	--None--

«EMhandler»

This stereotype models the exception handling concept (see Section 5.3.1.1).

Stereotype	Base Class	Tags
«EMhandler»	Method	--None--

«EMthrowMethod»

This stereotype models the throw statement part of the throw/try/catch structure (see Section 5.3.1.6).

Stereotype	Base Class	Tags
«EMthrowMethod»	Method	EMthrowType

It defines the following tagged value:

Tag Name	Tag Type	Multiplicity	Domain Concept
EMthrowType	TVL List of throwable exception types, for example (‘rErr’,’wErr’, ‘rwErr’)	0..1	Throw statement (see Section 5.3.1.6)

«EMtry»

It represents the try structure concept as discussed in Section 5.3.1.6.

Stereotype	Base Class	Tags
«EMtry»	SimpleState CompositeState	--None--

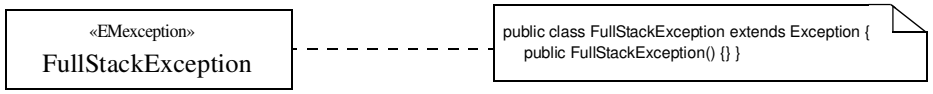
5.3.2.3 Example

The UML usage model for the exception profile utilizes a State diagram to model the *throw/try/catch* structure, and attaches it to a method. This method can be nested and called from within another State diagram, resulting in a exception propagation hierarchy.

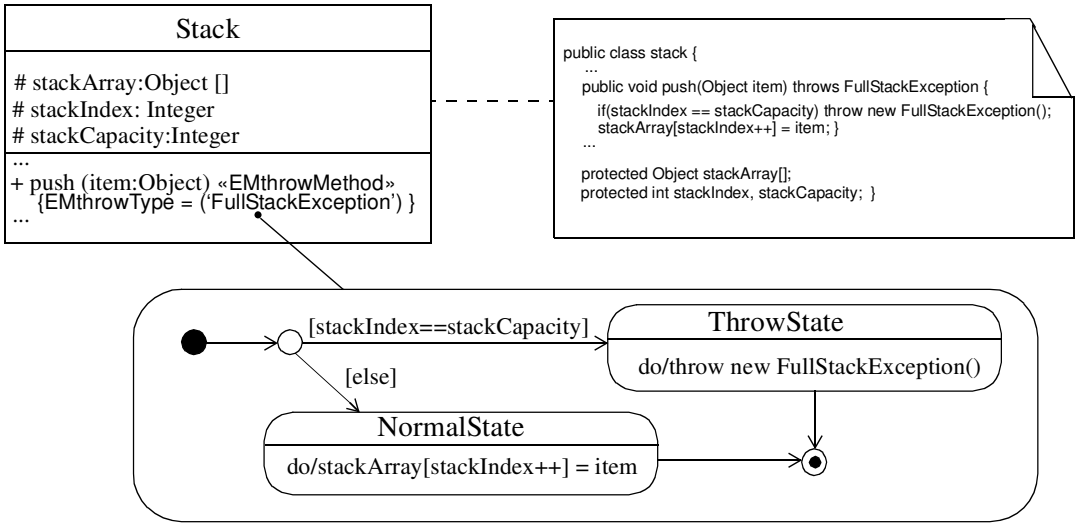
State Machine diagrams in Figure 5.2 demonstrate the UML usage model of the exception modeling profile. Figure 5.2 (a) shows the UML representation of an exception, along with corresponding Java code excerpted from Burns and Wellings [108]. It is to note in this figure that, even though, the exception possesses no parameter, i.e. class attribute, using and passing parameters along with the exception notification is perfectly doable. The State Machine diagram in Figure 5.2 (b) illustrates the modeling of a throw statement. The try/catch blocks in Figure 5.2 (c) are illustrated as simple states. However, they are applicable to composite states as well. The corresponding Java code for the try/catch states is also included in the figure.

5.4 Interrupt Modeling (IMprofile)

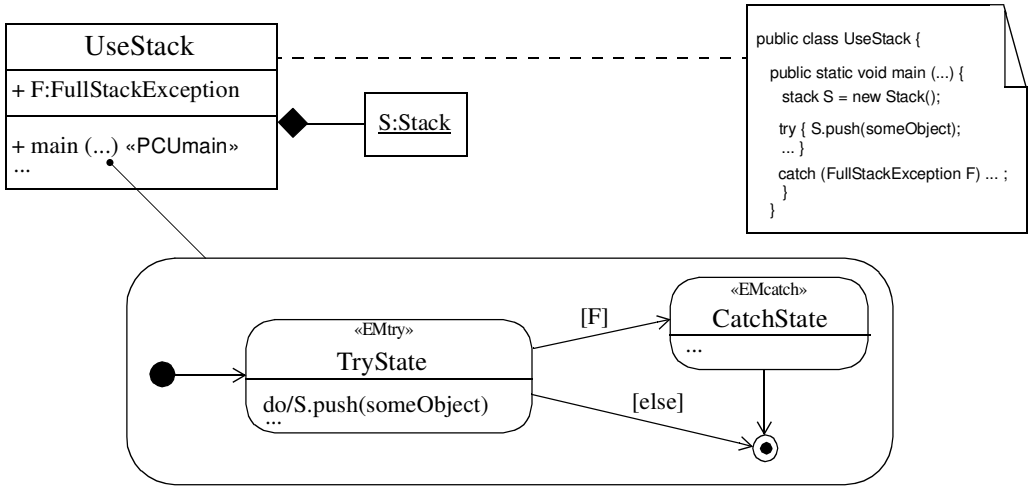
Developers of real-time embedded systems so often requires low-level interrupt service programming to implement applications such as device drivers and controllers. The proposed platform-centric SoC design method expects that the developer would be able to mostly avoid such tedious programming by resorting to available hardware-dependent software packages that accompany LPO hardware components. Nonetheless, given the sheer size and diversity of real-time embedded system characteristics, the availability of these packages is far from being a cure-all remedy. There will always exist the times that no suitable packages can be found, and the interrupt programming becomes inevitable, resulting in an increased complexity. The IMprofile package attempts to provide for the developer an interrupt modeling framework that can help ease the complexity incurred by the interrupt programming process, and that is independent of any programming language in particular.



(a)



(b)



(c)

Figure 5.2: Example of UML Exception Modeling Using «EMprofile»

5.4.1 Domain Viewpoint

The actual implementation of interrupt service facilities differs from one programming language to another. However, the underlying requirements for these languages remain the same. They are: *device register representation and manipulation*, *device encapsulation*, and *interrupt handler*. Subsequent discussions are based predominantly on Chapter 15 of the book by Burns and Wellings [108], that details the technical survey on low-level programming.

5.4.1.1 Interrupt Representation and Characteristics

Although many representations of interrupts are possible [108], in principle, they can be viewed simply as a specialized signal. These interrupt signals can be assigned priority levels. They can also be associated with unique IDs that permit the interrupt handlers to take proper actions when interrupt events occur. Certain interrupts in Ada, clock interrupt for example, are reserved and have no user-defined handlers associated with them. Reserved interrupts are handled through the run-time support system of the language.

5.4.1.2 Device Register Representation

Each device supported on a hardware platform has as many different types of register as are necessary for its operation [108]. A device register is memory-mapped, and accessible through a memory address. Depending on the hardware platform configurations, a device register may (1) either be oriented most-significant-bit first (descending order, big endian), or least-significant-bit first (ascending order, little endian), and (2) be aligned to a specified number of bits, e.g. 8-bit (byte-aligned), 16-bit.

In practice, a device register is divided up into several fields. Each of these fields contains information that is necessary for a correct operation of the device, e.g. control and status data. An accessibility control can be specified such that each individual field can independently set a permission for *read-only* (r), *write-only* (w), or both *read-write* (rw) operations.

To successfully model an interrupt service facility, an expressive way to represent, and manipulate these device registers is a prerequisite. The requirements are: (R1) it shall be able to expressively represent device registers at the bit level, and (R2) it shall include a facility that provides support for bitwise operations. These requirements will be discussed in terms of VHDL as follows.

The requirement R1 involves the support for BIT and BITVECTOR data types—a facility that is extensively supported in most HDL languages. BIT is a scalar type, and has the values '1' and '0', representing logical '1' and '0', respectively. BITVECTOR is then defined as an array of data type BIT. A BITVECTOR can be of a specified *size*, *range*, and *order*—e.g. a descending BIT array of size 8 starting from bit 7 down to 0.

Access to data of type BITVECTOR shall be allowed both on an individual-index basis, or a range-of-index basis. The following bitwise operators shall be supported (requirement R2):

- Logical operators: and or nand nor xor xnor not
- Relational operators: = /= < <= > >=
- Shift operators: sll srl sla sra rol ror
- Concatenating operator: &

When applying logical and relational operators to BITVECTOR operands, the operation is carried out bit-by-bit, and matching position-to-position, until a decisive outcome results. The sll and srl are logical shift left and right. The sla and sra are arithmetic shift left and right, while the rol and ror are left and right rotation, respectively. These operators take the left operand to be a data of type BITVECTOR, and the right operand is an integer indicating a number of positions to shift. The logical shifts fill the vacated bits with logical '0's, while the arithmetic left and right shifts fill the vacated bits with the right-most and the left-most bits, respectively. The sole adding operator takes two operands, either BIT or BITVECTOR, concatenates them, and returns a BITVECTOR as the output.

Table 5.1: Demonstrative use of some bitwise operations

Operations	Results	Operations	Results
not “101”	“010”	“011” xor “101”	“110”
“101” > “111”	FALSE	“010” & “00010”	“01000010”
“10100110” sll 2	“10011000”	“10100110” srl 3	“00010100”
“10100110” sla 2	“10011000”	“10100110” sla 3	“11110100”
“10100110” rol 2	“10011010”	“10100110” ror 3	“11010100”

Assume that all operands are of type BITVECTOR, Table 5.1 demonstrates the use of these operators.

5.4.1.3 Interrupt Handler

An interrupt handler is a software managing routine that executes appropriate actions in response to an interrupt event. Each interrupt signal is bound to a certain handler such that, when an interrupt event occurs, the correct handler becomes active.

5.4.1.4 Device Encapsulation

Interrupt service programming often involves low-level hardware operations that is machine-dependent, and is not portable in general. For a software systems, “it is advisable to encapsulate all the machine-dependent code into units which are clearly identifiable so that separation of portable and non-portable sections are achieved [108].” Examples of such units are classes and packages in Java, protected type facilities in Ada, and a file in C.

5.4.2 UML Viewpoint

In defining UML extensions, i.e. stereotypes, tagged values and/or constraints, for the IMprofile package, the prefix IM is always attached to the names to differentiate them from similar or same names in other profiles.

5.4.2.1 Mapping Interrupt Domain Concepts into UML Equivalents

The interrupt model is attained by stereotyping a Class with the «IMinterrupt» stereotype, where the following tagged values are defined: IMpriority, IMisReserved, and IMid. The interrupt ID maps to IMid, the priority to IMpriority, and a reserved interrupt is specified by the IMisReserved tag (see Section 5.4.1.1).

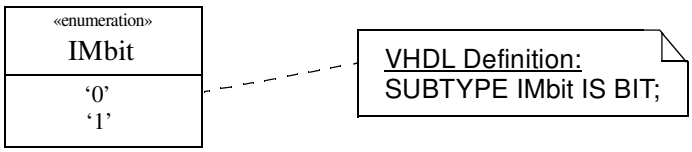
The interrupt handler concept maps to the «IMhandler» stereotype attached to a method, where it can be bound to an «IMinterrupt» using the «IMbind» stereotype. A class containing the «IMhandler» method represents the device interface block that encapsulates the device-dependent operations together in one place (see Section 5.4.1.4). This Class must be stereotyped with the «IMdeviceIF» stereotype.

5.4.2.2 Mapping Data Type into UML Equivalents

The BIT and BITVECTOR data types map to the IMbit Enumeration, and the «IMbitVector» stereotype, respectively. Their definitions, usages, and constraints are described in detail as follows.

IMbit

It is an enumeration whose permissible values, '0' and '1', represent the logical '0' and '1', respectively.



Device Register Representation

To represent device registers in UML, the «IMbitVector» and «IMbitField» stereotypes are used in conjunction (see also Section 5.4.2.3). The «IMbitVector» stereotype is adorned on a Class to designate it as a container of a BITVECTOR data. A BITVECTOR

data is represented as an attribute (or attributes) with the «IMbitField» stereotype attached to it. The type expression of the BITVECTOR data follows the format below that satisfactorily expresses size and range of the data (see Section 5.4.1.2):

field_name '[' *from_bit* ':' *to_bit* ']' ':' **IMbit** '[' *size* ']' *«IMbitField»*

In addition, when multiple bit fields of the same characteristics are declared in succession, only one «IMbitField» stereotype may be used prior to the first bit field declaration to imply a declaration block. The «IMbitField» block terminates where it encounters another stereotype, or where it reaches the end of the compartment.

A typical control and status register for the computer has the following structure [108], whose equivalent UML representation is shown in Figure 5.3.

```
Bits   15 - 12    : Errors      -- Errors
       11         : Busy        -- Busy
       10 - 8     : Unit        -- Unit select
       7          : Done        -- Done/Ready
       6          : Ienable     -- Interrupt enable
       5 - 3      :             -- Reserved
       2 - 1      : Dfun        -- Device function
       0          : Denable     -- Device enable
```

Also, by using «IMbitVector» and «IMbitField» stereotypes to represent a device register. it implies that:

- If all bit fields can be read from, there must exist a getRegValue() method which returns the register value that results from concatenating all bit fields together. This function is defined in the «IMbitVector» class as:

  + getRegValue: IMbit[IMvectorSize]

- Similarly, if all bit fields can be written to, there must exist a setRegValue() method that takes an IMbit array of size IMvectorSize as the input, and assigns its value to the corresponding bit fields. This function is defined in the «IMbitVector» class as:

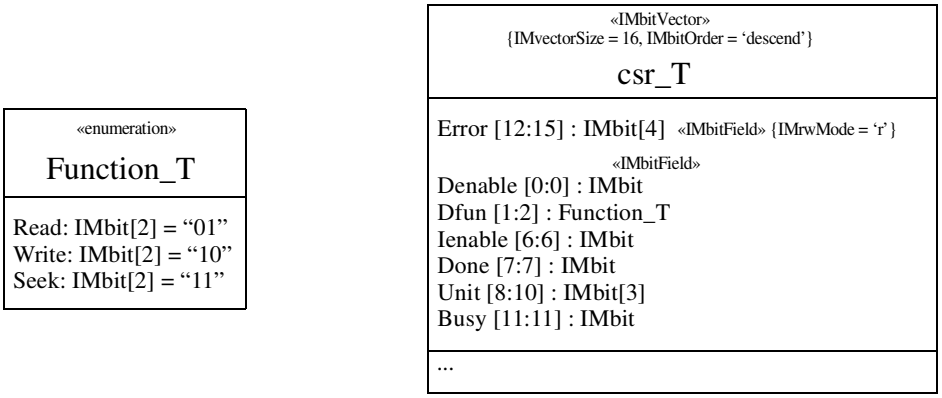


Figure 5.3: Example of the UML representation of a control and status register

+ setRegValue(v: IMbit[IMvectorSize])

- There always exists the getSize*BitfieldName*() method that return the length of the bit field indicated by the *BitfieldName*. This method is defined in the «IMbitVector» class as:

  + getSize*BitfieldName*: Integer

- There always exist the set*BitfieldName*() and get*BitfieldName*() methods associated with each bit field, unless it is marked as read-only (IMrwMode = 'r'), or write-only (IMrwMode = 'w', see Section 5.4.1.2). In such a case, only the proper method is implemented. For example, assume that the Errors bit field is read-only while the Ienable can be read from or written to, then the following methods are defined:

  + getSizeErrors: Integer
  + getErrors: IMbit[4]
  + getSizeIenable: Integer
  + getIenable: IMbit
  + setIenable(v: IMbit)

5.4.2.3 Mapping Operators into UML Equivalents

In mapping the operators described in Section 5.4.1.2 into UML equivalents, a utility class IMoppak is defined as a grouping mechanism. The operators placed inside the IMoppak class become known globally. Let IMbit_T be a super type of types IMbit and IMbit[]. Then, the operators belonging to the IMoppak class are defined as follows (see Table 5.2).

5.4.2.4 UML Extensions

To avoid any possible duplicate and ambiguity, all extensions defined in this profile are prefixed with IM.

Table 5.2: Definition of IMoppak operators

Op	IMoppak Operators	Op	IMoppak Operators
and	IMand(v1: IMbit_T, v2: IMbit_T) : IMbit_T	**or**	IMor(v1: IMbit_T, v2: IMbit_T) : IMbit_T
nand	IMnand(v1: IMbit_T, v2: IMbit_T) : IMbit_T	**nor**	IMnor(v1: IMbit_T, v2: IMbit_T) : IMbit_T
xor	IMxor(v1: IMbit_T, v2: IMbit_T) : IMbit_T	**xnor**	IMxnor(v1: IMbit_T, v2: IMbit_T) : IMbit_T
not	IMnot(v: IMbit_T) : IMbit_T	**=**	IMisEqual(v1: IMbit_T, v2: IMbit_T) : Boolean
/=	IMisNotEqual(v1: IMbit_T, v2: IMbit_T) : Boolean	**>**	IMisGreater(v1: IMbit_T, v2: IMbit_T) : Boolean
>=	IMisGreaterEqual(v1: IMbit_T, v2: IMbit_T) : Boolean	**<**	IMisLess(v1: IMbit_T, v2: IMbit_T) : Boolean
<=	IMisLessEqual(v1: IMbit_T, v2: IMbit_T) : Boolean	**sll**	IMsll(v: IMbit_T, k: Integer) : IMbit_T
srl	IMsrl(v: IMbit_T, k: Integer) : IMbit_T	**sla**	IMsla(v: IMbit_T, k: Integer) :IMbit_T
sra	IMsra(v: IMbit_T, k: Integer) : IMbit_T	**rol**	IMrol(v: IMbit_T, k: Integer) : IMbit_T
ror	IMror(v: IMbit_T, k: Integer) : IMbit_T	**&**	IMcat(v1: IMbit_T, v2: IMbit_T) : IMbit_T

«IMbind»

The stereotype models the explicit binding between an interrupt and its handler (see Section 5.4.1.3). The UML representation shows as an attachment of the «IMbind» stereotype on a dependency between «IMdeviceIF» and «IMinterrupt» classes.

Stereotype	Base Class	Tags
«IMbind»	Dependency	--None--

«IMbitField»

The stereotype represents an individual bit field within a device register (see Section 5.4.1.2, see also Section 5.4.2.2).

Stereotype	Base Class	Tags
«IMbitField»	Attribute	IMrwMode

The tag defined for it is:

Tag Name	Tag Type	Multiplicity	Domain Concept
IMrwMode	Enumeration: ('r', 'w', 'rw') Default value: 'rw'	0..1	Accessibility Mode of Register's Bit Field (see Section 5.4.1.2)

«IMbitVector»

This stereotype represents the BITVECTOR data type as discussed in Section 5.4.1.2 (see also Section 5.4.2.2). Just like the BITVECTOR data type, «IMbitVector», together with «IMbitField», are used to model the device register concept in UML.

Stereotype	Base Class	Tags
«IMbitVector»	Class Object	IMaddress IMalignment IMbitOrder IMvectorSize

The definition of each tagged value is presented as follows:

Tag Name	Tag Type	Multiplicity	Domain Concept
IMaddress	Integer	0..1	Address location of the register device (see Section 5.4.1.2)
IMalignment	Integer	0..1	Number of bits in Bit-alignment (see Section 5.4.1.2)
IMbitOrder	Enumeration: (‘ascend’, ‘descend’)	0..1	Bit orientation (see Section 5.4.1.2)
IMvectorSize	Integer	1	Number of bits in a device register

«IMdeviceIF»

This stereotype represents the device encapsulation concept as described in Section 5.4.1.4.

Stereotype	Base Class	Tags
«IMdeviceIF»	Class Object	--None--

«IMhandler»

This stereotype represents the device handler concept as discussed in Section 5.4.1.3.

Stereotype	Base Class	Tags
«IMhandler»	Method	--None--

The following constraint is defined for this stereotype:

- A «IMhandler» method must reside in the «IMdeviceIF» class.

«IMinterrupt»

This stereotype models the interrupt concept (see Section 5.4.1.1).

Stereotype	Base Class	Tags
«IMinterrupt»	Class Object	IMid IMisReserved IMpriority

The following tagged values are defined:

Tag Name	Tag Type	Multiplicity	Domain Concept
IMid	String	0..1	Interrupt ID (see Section 5.4.1.1)
IMisReserved	Boolean	0..1	Reserved interrupt (see Section 5.4.1.1)
IMpriority	Integer	0..1	Interrupt priority (see Section 5.4.1.1)

5.4.2.5 Usage Model Framework

Figure 5.4 delineates a possible usage model framework for the Interrupt Modeling Profile (IMprofile) package. In the figure, *InterruptInterface* and *AnInterrupt* represents a design template that the developer can quickly generalize for the modeling of device-specific operations. This template, which specifies relevant tagged values, and the device-specific run-time library are supplied by the LPO component provider.

The device encapsulation block, *MyInterruptInterface*, provides an interface facility between the software system and the device. Its behavior can be described using State or Sequence diagrams, or code insertion (*Behavior1*). Within its region of computation, defined are instances of device registers, as well as the interrupt handler whose functionality is described by the *Behavior2* module. The «IMbind» stereotype on a dependency between *MyInterruptInterface* and *MyInterrupt* classes practically binds the handler method in *MyInterruptInterface* to the interrupt represented by the class *MyInterrupt*. Using this framework, a simple homomorphic mapping from the interrupt model to code becomes possible as shown in Table 5.3.

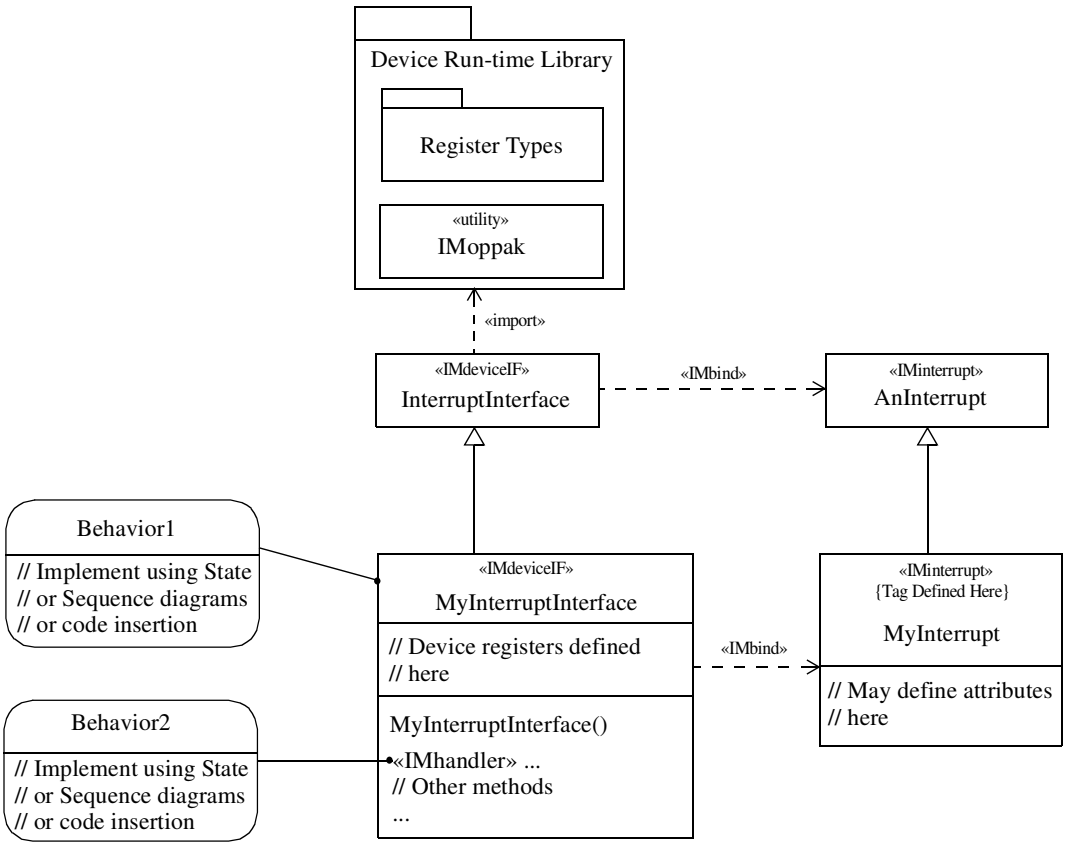


Figure 5.4: Usage Model Framework for the Interrupt Modeling Profile

Table 5.3: Interrupt model to code mapping

Model Element	Real-Time Java [108, 111]	Ada [108]
«IMdeviceIF»	Class	Protected procedure. If the model element is a template class, a parameter can be passed to the constructor, and thus, the protected procedure.
«IMpriority»	Class PriorityParameters	pragma Interrupt_Priority(IMpriority)
«IMhandler»	Class AsynchEventHandler	Procedure
«IMbind»	MyInterrupt.addHandler(IMhandler) MyInterrupt.bindTo(IMid)	pragma Attach_Handler(IMhandler, IMid)
«IMinterrupt»	AsynchEvent MyInterrupt	Ignored. However, it signifies the requirements for various Ada interrupt packages. Moreover, tagged values are used.
Registers	RawMemoryAccess	var
Methods in IMdeviceIF	methods	entry procedures
Behavior1	Either ignored, or becomes a controller method in parent class.	Initialization procedure for the protected procedure.
Behavior2	method body	procedure body

5.5 Synthesizable HDL Modeling (SHDLprofile)

Thus far, this chapter has established a bridging mechanism (Section 5.2) that permits information from the LPO to be retrieved and used in the UML context that encompasses system modeling, analyzing, and implementing. Thereafter Sections 5.3 and 5.4, introduce the Exception and Interrupt Modeling profiles that, together with the upcoming UML real-time profile, provide a comprehensive support for the development of platform-based real-time applications. This section embarks on the issues of hardware implementation, and

presents the Synthesizable HDL Modeling (SHDLprofile) package that enables the use of UML to descriptively model hardware for synthesis. With the SHDLprofile package, both hardware and software development can take place in one unified environment, rendering the development task less complicated.

5.5.1 Domain Viewpoint

In the Electronic Design and Automation (EDA) realm, Hardware Description Languages (HDL) are used to describe hardware functionalities, verify functional correctness, and synthesize the code for subsequent fabrications. The synthesis capability of today's mainstream HDL languages, i.e. Verilog and VHDL, has advanced so much from a few years back that it becomes very commonplace and almost indispensable in the development of any hardware system. In implementing the SHDLprofile package, only the synthesizable subset of HDL syntax and semantics is considered.

HDL languages differ from traditional software-oriented programming languages in many aspects. Of all the discrepancies, it likely is the concepts of signals, time, and concurrency that renders the dissimilitude quite notable. Such concepts require specialized data types, data values and language constructs that make the UML modeling of HDL appears extraneously awkward, let alone the fact that UML is fully object-oriented, while most, if not all, HDL languages are not. Even with the current real-time-oriented UML profile for Schedulability, Performance and Time Specification, UML is still deemed inadequately expressive for the purpose of describing hardware.

The Synthesizable HDL profile, tailored specifically for the proposed approach, is aimed to augment UML with the capability to expressively describe hardware by means of UML models, such that unambiguous UML-to-HDL mapping can be realized to aid the design and implementation of hardware systems. The profile, however, does not attempt to strictly formalize UML for formalism tends to hinder the analysis capability which is so essential for the proposed approach. Instead, the profile relaxes the strict formalism to allow code insertion where appropriate to expedite the development process.

HDL synthesizability is considerably dependent on the interpretation by synthesis tools; there really is no perennial guarantee for interoperability. As such, this section relies on several sources, books, tutorials, papers, and manuals alike, to define the common working subset of synthesizable HDL that is likely to yield the same synthesis results. As per VHDL, the IEEE 1076.6 standard for VHDL Register Transfer Level (RTL) Synthesis [112] serves as the ultimate reference. On the other hand, the interoperability standard for synthesizable Verilog is still work in progress (see http://www.vhdl.org/vi/vlog-synth/). Every effort is made to ensure the best possible interoperability of synthesizable Verilog in this dissertation. The basis of subsequent discussions related to VHDL and Verilog comes predominantly from [113], [114], [115].

Little known use of UML modeling for hardware design has been documented. Much attention has been directed to formalizing UML object and dynamic models so as to automate the code generation process. Björklund and Lilius [116] have demonstrated that automatic generation of optimized synthesizable VHDL code from UML State diagrams could be achieved. In McUmber and Cheng [117], a method to formalize UML object and dynamic models that allows in-model VHDL simulation is reported. Nonetheless, none of these researches provides UML facilities comprehensive enough to tackle the issue of UML modeling for hardware design, which shall (1) allow hardware systems to be described structurally and behaviorally for proper analysis, and (2) include comprehensive facilities for modeling specialized HDL data types, data objects, operations, and language constructs. Rather, the profile bears some conceptual resemblance to the executable UML concept described in Mellor [118], but is more complete as an aid for hardware design and is specifically customized to work well in the platform-centric design environment. No concrete work regarding the UML modeling of Verilog HDL has been found. Such passive research activities in this field can probably be attributed to the almost non-OO nature of Verilog that may prove less attractive compared to the object-based VHDL. It is expected, though, that research activities will increase as Verilog-2005 rolls out [119] and UML 2.0 becomes mature.

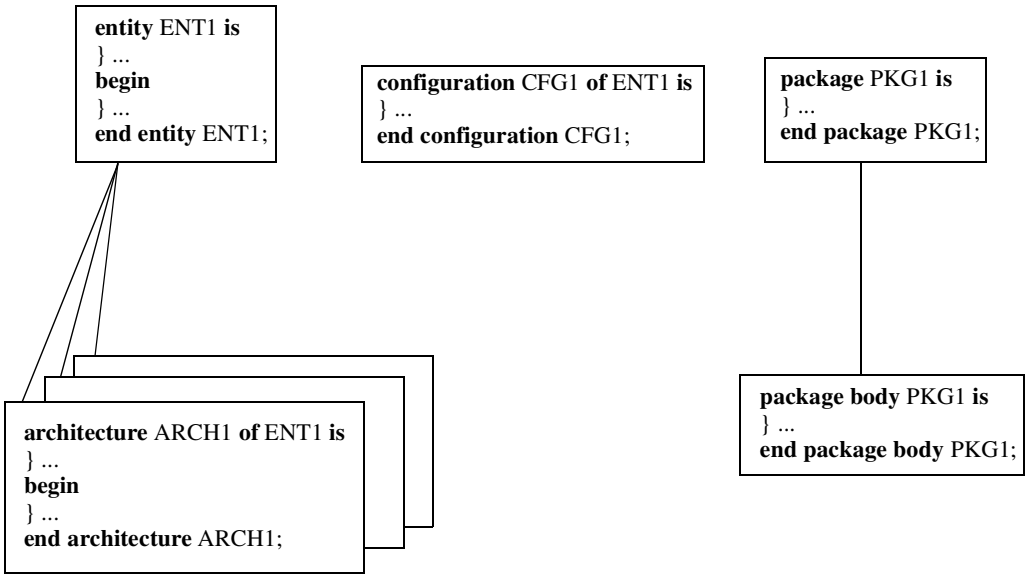


Figure 5.5: VHDL Design Units

5.5.1.1 HDL Design Entities

The VHDL design entities are more modular and more complicated per se when compared with the Verilog counterparts. As such, the discussion herein will proceed solely in terms of VHDL.

VHDL specifies five different design units: entity, architecture, configuration, package and package body, as illustrated in Figure 5.5 [113]. An entity/architecture pair, known collectively as a *design entity*, is the main construct for describing a hardware component; it is analogous to Verilog's module. The entity provides the port information of a particular design entity, while the architecture provides the functional body description. The configuration functions as a binding mechanism that associates an entity to a particular architecture. Finally, the package and package body hold common design data that can be used by a design entity. Their Verilog equivalent is the include construct.

5.5.1.2 Data Types, Data Objects, and Operations

Unlike the IMprofile package that explicitly defines bit-oriented data types, and operations (see Sections 5.4.2.2 and 5.4.2.3) in order to mitigate the difficulty in modeling interrupt service routines, the SHDLprofile package, instead, adopts the *data types*, *data objects*, and *operations* of the target language.

Whereas VHDL defines a number of data types, Verilog only defines one and it never has to be explicitly specified in any declarative statement. A data object is a mechanism adopted by both VHDL and Verilog to pass data from one point to another. Examples of a data object in VHDL are signal and variable, while examples in Verilog are wire, reg (register), and parameter. A code generator is expected to be able to understand, syntactically and semantically, the language-specific declarations and descriptions in the model.

5.5.1.3 Code Structure

The HDL code structure typically comprises declarative statements, sequential statements, and concurrent statements.

Like most other programming languages, the declarative statements declare objects for use in other parts of the design unit. In VHDL, these statements are always located before the begin clause in a package body, architecture, process, procedure, or function statement. Verilog, on the other hand, has no dedicated declarative region.

Statements after the begin clause in the process statement (VHDL), and inside the always statement (Verilog) are sequential. So are those in the procedure (VHDL), task (Verilog) and function (VHDL and Verilog) statement.

As per concurrent statements, the supported set for VHDL/Verilog encompasses process/always statements, signal/continuous assignments, and VHDL procedure calls. In order to execute concurrently, VHDL signal assignments and procedure calls must not be nested inside a process statement.

5.5.1.4 Behavioral Description

Sequence of HDL statements, that may execute sequentially or concurrently within a design unit, constitute a behavior of that design unit. In VHDL, the following behavioral constructs are synthesizable: process, wait on, wait until, procedure, function, if clause, case clause, for loop, generate, next, and exit. In Verilog, the list includes: always, @( ), task, function, if clause, case clause, and for loop. The collaboration among design units is achieved by means of port and signal (or reg in Verilog).

It is often the case in VHDL that only the wait until statement is permissible for synthesis. In addition, the process and procedure body can only contain at most one wait statement, usually as the first statement in the sequence. In the case of the task statement in Verilog, it can contain no wait equivalent, i.e. @( ), at all. Such a diverse usage of wait statements can result in a modeling complication. As such, it is often a good practice to exclude all synchronous wait statements from the body of subprograms and/or process/ always statement, and use VHDL's *sensitivity list* or Verilog's @( ) statement, instead.

A concurrent statement, i.e. the process statement, and the statement in the architecture body that do not belong to any process, never terminates. Once it executes to its entirety, it starts over from the beginning. This concept is to be referred to as an *endless continuation*.

A generate statement in VHDL furnishes a mechanism to render a description of regular concurrent statements compact. A for-generate can be used to replicate such statements a predetermined number of times; whereas, an if-generate stipulates the replication of such statements by means of conditional statements. The generate statements are expanded back to the normal, explicit descriptions during compilation time.

Parametrized design is also possible in both VHDL and Verilog, making use of the generic clause, and parameter overloading, respectively. Reuse is attained through the VHDL package design unit, and the Verilog include clause.

Coding style is also significant since the synthesis tool often infers from it to make synthesis-related decisions. For example, using an if statement can infer a priority encoder to be synthesized, while a case statement inferring a mux.

5.5.2 UML Viewpoint

In defining UML extensions, i.e. stereotypes, tagged values and/or constraints, for the SHDLprofile package, the prefix SHDL is always attached to the names to differentiate them from similar or same names in other profiles.

Due to the object-based nature of VHDL, it is more straightforward to present the mapping using VHDL concepts and constructs. Then, where appropriate, related information on the corresponding Verilog counterparts ensue. The presentation is organized by first considering in Section 5.5.2.1 the mapping of the design entity collaboration concept, followed by the mapping of the generic HDL structure and behavioral constructs in Sections 5.5.2.2 and 5.5.2.3, respectively. Thereafter, Section 5.5.2.4 details the UML extensions for the SHDLprofile package. The presentation concludes in Section 5.5.2.5 with a demonstrative example on using this profile.

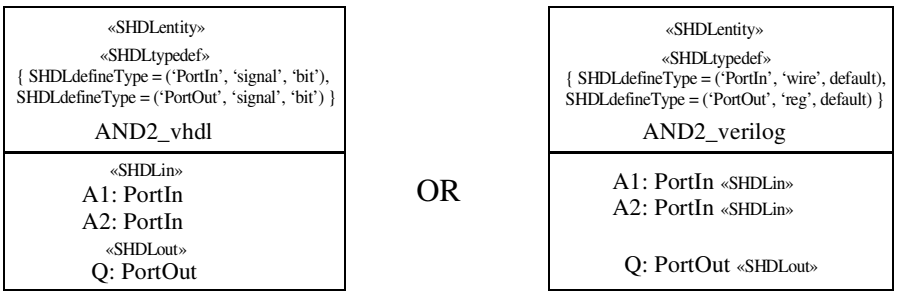
5.5.2.1 Mapping Design Entity Collaboration Mechanisms into UML Equivalents

The collaboration among design entities involves the port and signal concepts—how to define and represent them in UML, and how ports from different design units are bound together to establish a communication channel by means of the signal data object (reg in Verilog).

A port declaration maps to an attribute declaration in an entity class, with one of the «SHDLin», «SHDLout», «SHDLinout» stereotypes attached to the attribute to indicate the port direction. Each port takes on one of the user-defined types that specifies the HDL-dependent data type and data object type. These types are described in the «SHDLtypedef» tag section. Though not explicitly specified, syntactically all VHDL ports are data objects of type signal. Verilog only defines one base data type, which never has to be declared (see

Section 5.5.1.2). In Verilog, I/O ports are viewed as wire. Verilog inputs should be declared a wire, while the outputs can either be declared a wire or a reg.

As an example, an AND2 ports may be represented as follows:



5.5.2.2 Mapping Generic HDL Structures into UML Equivalents

The design entity concept maps to the «SHDLmodule» stereotype that embellishes an *abstract* class (see Section 5.5.1.1). This stereotype represents a synthesizable HDL domain, and can be iteratively nested. Its presence denotes the existence of the *entity/ architecture* pairs, or other «SHDLmodule» classes. The stereotype defines no tagged value.

The entity and architecture concepts map into a class with the «SHDLentity» and «SHDLarch» stereotype, respectively. In Verilog, the «SHDLentity» stereotype represents the header area of the module, while, the «SHDLarch» stereotype represents the module body. The «SHDLentity» class shall contain no method definition. In fact, it must contain only port definitions that define the design entity interface. The VHDL configuration concept maps to the «SHDLbind» stereotype that binds a pair of «SHDLentity»/«SHDLarch» classes together.

The VHDL generic clause (parameter overloading in Verilog) maps to a «SHDLentity» *template* class. Parameters in the template window represent the generic parameters in VHDL.

As opposed to the «SHDLentity» that specifies the communication interface for the «SHDLmodule», the «SHDLarch» describes the behavior for it. The VHDL architecture name maps to the name of the «SHDLarch», while architecture-scoped signal declarations map to attribute declarations that are stereotyped by «SHDLdataObject».

Within the architecture body, instantiation of other entities is possible. In VHDL, such a process involves (1) declaring the design entity to instantiate, and (2) instantiating the design entity, and binding the desirable ports and signals together via the port map statement. The design entity concept maps to a user-defined data object, and thus, could be declared in the attribute list compartment. The actual instantiation of the design entity maps to a *composition* relationship from the «SHDLarch» *class* to the «SHDLentity» *object*. port map statements map to port value assignments in the «SHDLentity» object.

The process (Verilog's always) statement maps to an object stereotyped with «SHDLprocess». This object can be implemented by State Machine diagrams, or by inserting source code for it. The relationship between the «SHDLarch» class and the «SHDLprocess» object is defined by a composition association (see Figure 5.7).

The concurrent (Verilog's assign) statements that reside outside the process block in the architecture body, and that do not involve entity instantiations. map to a container object stereotyped with the «SHDLparBlock» stereotype. By treating these statements as having a container object associated with them, they can be annotated with QoS properties and analyzed just like any other objects, if need be. Then there are two ways to associate the unmapped concurrent statements with the «SHDLparBlock» object. The first method is to use code insertion defined in Section 5.2.2.2, i.e. «PCUdeclare», and «PCUcodeBody». Secondly, concurrent State Machine diagrams can be employed. As an example, consider the VHDL code that negates the values of din and assigns the new value to a signal dout:

```
dout <= not din;                -- concurrent statement
```

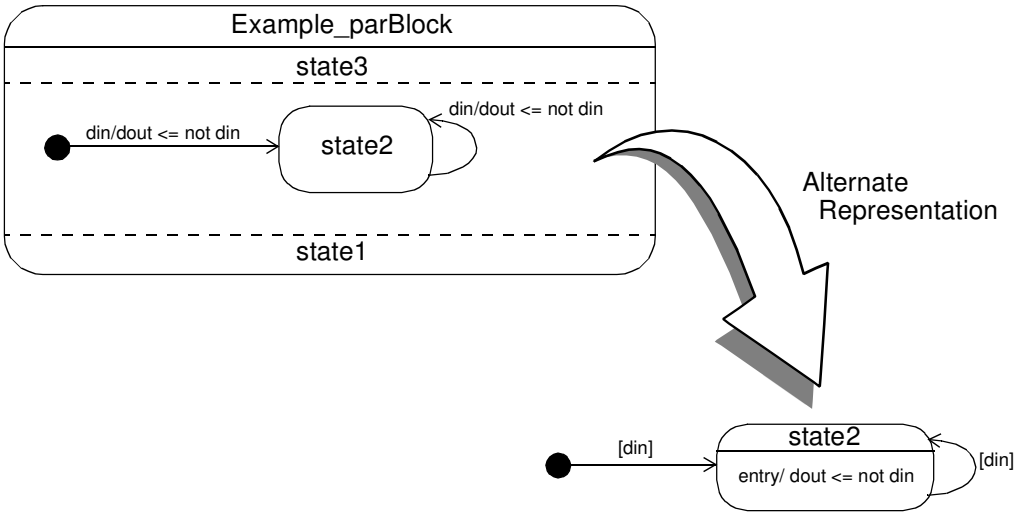


Figure 5.6: Concurrent state representation of dout <= not din

The State Machine diagram for this expression is portrayed in Figure 5.6. It has no final state and is perennially active. Every time an event occurs on din, the above statement is evaluated and dout has a new value assigned.

A generate statement maps to a class being stereotyped with «SHDLgenerate» (see Section 5.5.1.4). The usage model for the «SHDLgenerate» stereotype is portrayed in Figure 5.7. As seen in the figure, an «SHDLgenerate» class may contain one or more instances of either the «SHDLparBlock» or «SHDLentity» object. The for statement is inferred by the SHDLgenFor tag owned by the «SHDLgenerate» stereotype. Conditional statements for the if-generate map to constraints on the Composition.

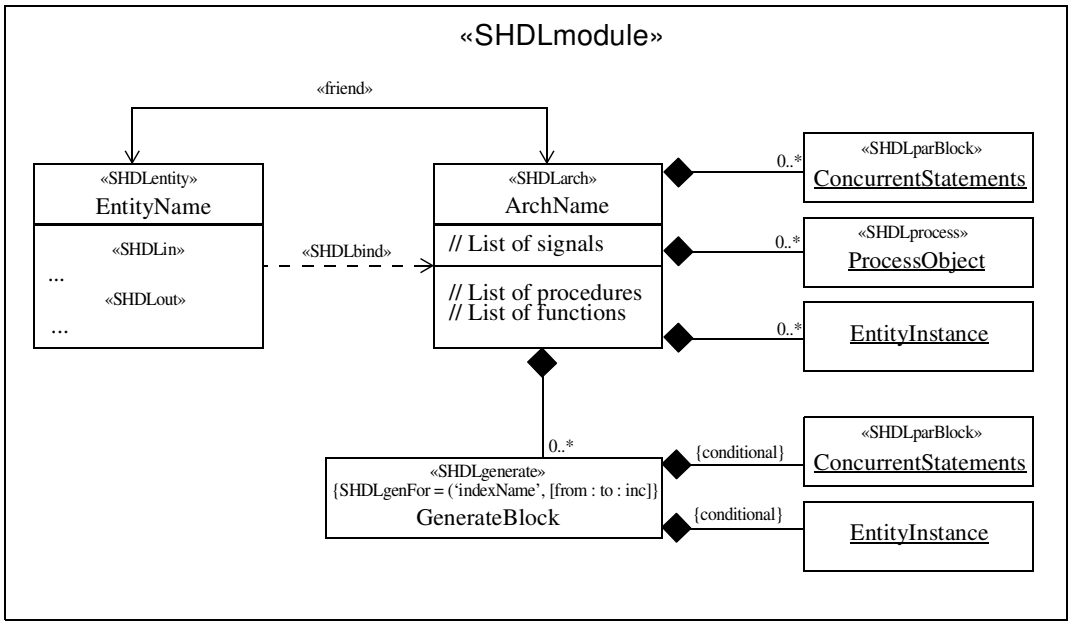


Figure 5.7: Summary of the relationships among entities in the «SHDLmodule»

Procedure and function declarations map to method declarations in the method compartment of the «SHDLarch» class. All methods must be visible *publicly* (+). A method that has no return value infers a procedure (VHDL) or a task (Verilog), whereas a method that has a return value infers a function (VHDL and Verilog). Procedure and/or function parameters must be of type SHDLdataType. Similar to the concurrent statement, its implementation entails the use of code insertion (see Section 5.2.2.2), and/or State Machine diagrams (see Section 5.5.2.3 for details). Figure 5.7 summarizes the relationships among relevant entities within the «SHDLentity»/«SHDLarch» pair.

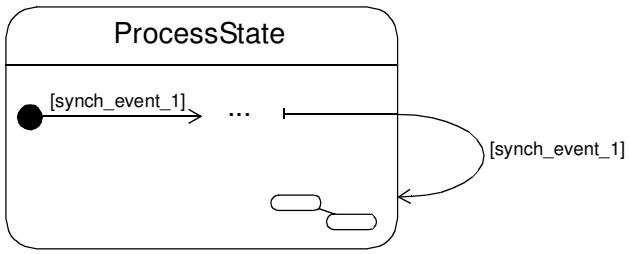
The package and package body, where global variables and functional facilities are defined, maps to a class stereotyped with «SHDLpackage». The stereotype has no tagged value. Then the global variables map to attributes in the attribute list compartment; whereas, procedures and functions map to methods in the method compartment. See the discussion above for details about the method inferences of procedures and functions.

5.5.2.3 Mapping Synthesizable HDL Behaviors into UML Equivalents

State Machine diagrams are the UML model of choice for representing behavioral HDL statements. To implement the «SHDLmodule», sequential state models can be attached to the «SHDLprocess» object, and an entity instance. A concurrent state model can be associated with the concurrent statement object («SHDLparBlock»).

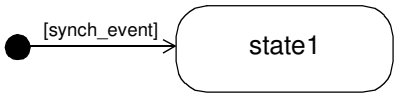
The endless continuation concept (see Section 5.5.1.4) maps to a self-iterating state, where there is no *final* state, and the state always transitions back to itself—pending on the same guard condition as described by the event guard on the transition from the start state to the first state. The state model depicted in Figure 5.6 is one such example.

The process statement maps to a self-iterating composite state as depicted below. Due to its endless continuation characteristic, the composite state contains no final state, and the last substate emanates a transition out of the boundary of the composite state before looping back with the same event guard as the one modeled from the sensitivity list.



In VHDL, it is required for this profile that if a wait statement is to be utilized in a process and/or procedure, it must be the very first statement and the only wait statement in the process/procedure body. It is recommended that a sensitivity list be used with the process statement, in place of the wait statement. Verilog's task, and VHDL/Verilog's function do not support the synchronization concept.

Then, the wait statement in process/procedure, and/or the sensitivity list concept in the process statement map to the *event guard* on the transition from the *start* state to the first state, and the synchronization signals map to the *synch_event* conditions, as shown below.



The following illustrate the mapping of HDL behavioral constructs to the UML equivalents. These constructs include: transition on wait conditions, if clause, case statements, for loop, VHDL's next and exit statements, and while loop.

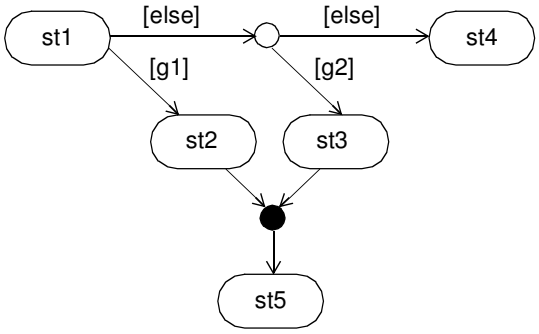
Transition on wait conditions

VHDL **Verilog**

If clause

Order control over the if sequence statements sometimes is significant, and can be modeled as shown below using a *dynamic choice path*.

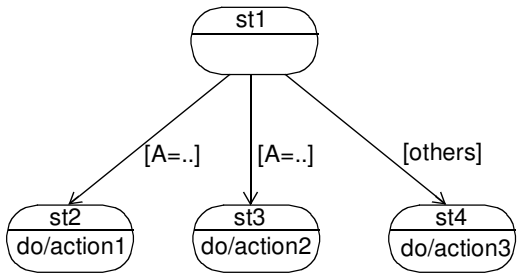
```
-- If statements
--
st1;
if g1 then
   st2;
else if g2 then
   st3;
else
   st4;
end if;
st5;
```



case statement

The standard UML modeling of a case statement is in better agreement than the if counterpart. Generally, the case statement is modeled as illustrated below.

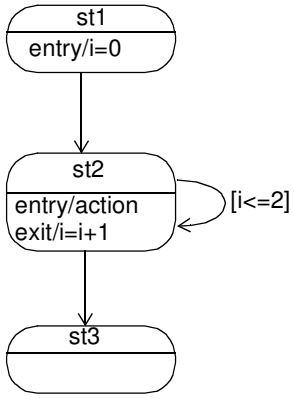
```
-- VHDL
-- st1
case A is
  when ...  => action1;   -- st2
  when ...  => action2;   -- st3
  others    => action3;   -- st4
end case;


// Verilog
// st1
case(A)
  ... : action1;        // st2
  ... : action2;        // st3
  default: action3;     // st4
endcase;
```



for loop

The for loop model presented here follows that of Rational Rose™ [102] very closely.

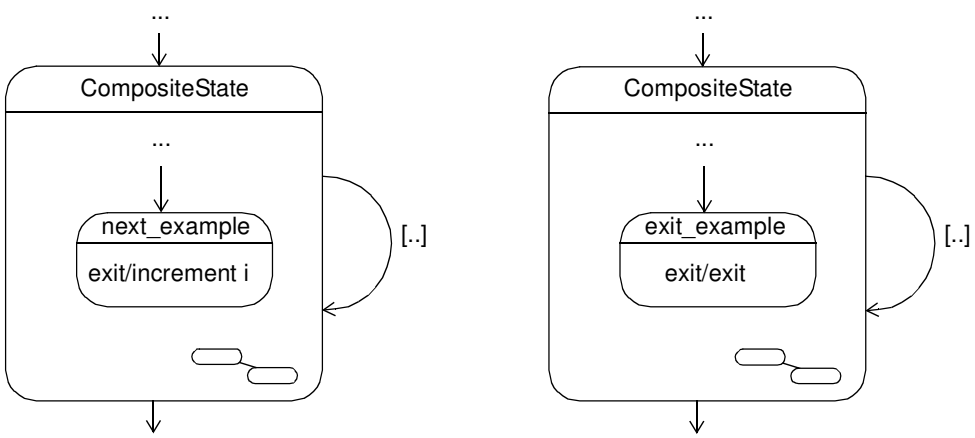
```
-- VHDL
--
for i in 0 to 2 loop
   action;
end loop;

// Verilog
//
for(i=0; i<=2; i=i+1)
  begin
    action;
  end
```



VHDL's next and exit statement

The next statement in a loop maps to a state with only one action: exit/increment loop index. Similarly, the exit statement maps to a state with the action: exit/exit.

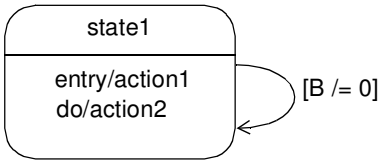
next Statement **exit Statement**



while loop

The while loop, due to its unrestrained nature, is supported by only few synthesis tools (yet, with certain set of rules to follow). However, it merits to be included here for completeness.

```
-- VHDL
--
while B /= 0 loop
  action1;
  action2;
end loop;
```



5.5.2.4 UML Extensions

To avoid any possible duplicate and ambiguity, all extensions defined in this profile are prefixed with SHDL.

Port Direction («SHDLin», «SHDLout», «SHDLinout»)

These stereotypes help specify port directions for attributes of the «SHDLentity» (see Section 5.5.2.1).

Stereotype	Base Class	Tags
«SHDLin»	Attribute	--None--
«SHDLout»	Attribute	--None--
«SHDLinout»	Attribute	--None--

The following constraint is defined for these stereotypes:

- Holder of the Attribute base class must be stereotyped «SHDLentity».

«SHDLarch»

This stereotype represents the architecture concept of the entity/architecture pair presented in Sections 5.5.1.1 and 5.5.2.2.

Stereotype	Base Class	Tags
«SHDLarch»	Class	--None--

SHDLattrType

The SHDLattrType allows a tuple of string values that specifies the HDL language-dependent data type and data object type to be mapped to a user-defined attribute type. Its value is described using the Tag Value Language (TVL) as defined in the UML real-time profile specification [29]. To represent the syntax, this dissertation follows the standard BNF notational conventions, where:

- A string between double quotes (“) represents a literal,

- A token in angular brackets (< >) is a non-terminal,
- A token enclosed in square brackets ([<element>]) implies an optional element of an expression,
- A token followed by an asterisk (<element>*) implies an open-ended number of repetitions of that element,
- A vertical bar indicates a choice of substitutions.

The TVL uses parentheses to identify arrays, commas to separate elements of arrays, and single quotes for string literals.

The SHDLattrType is defined as follows:

<shdlAttrTypeStr> ::= (<attrTypeName> , <dataObjStr> , <dataTypeStr>)
<attrTypeName> ::= <String>
<dataObjStr> ::= <String> | “default”
<dataTypeStr> := <String> | “default”

«SHDLbind»

This stereotype represents the binding of the «SHDLentity» and «SHDLarch» classes (see Section 5.5.2.2).

Stereotype	Base Class	Tags
«SHDLbind»	Dependency	--None--

«SHDLentity»

The stereotype represents the entity concept as described in Sections 5.5.1.1 and 5.5.2.2.

Stereotype	Base Class	Tags
«SHDLentity»	Class Object	--None--

SHDLforInfoType

Similar to the SHDLattrType, the SHDLforInfoType is described using the TVL language. It permits necessary information for executing a for-loop, i.e. index parameter name, range, and incremental step, to be captured through the SHDLgenFor tag (see «SHDLgenerate» below).

The SHDLforInfoType is defined as follows:

<shdlForInfoTypeStr> ::= (<indexNameStr> , <indexRange>)
<indexNameStr> ::= <String>
<indexRange> ::= "[" <from> ":" <to> ":" <incStep> "]"
<from> ::= <Integer>
<to> ::= <Integer>
<incStep> ::= <Integer>

«SHDLgenerate»

This stereotype represents the modeling of VHDL's generate block (see Section 5.5.2.2).

Stereotype	Base Class	Tags
«SHDLgenerate»	Class	SHDLgenFor

The stereotype defines one tagged value, which is:

Tag Name	Tag Type	Multiplicity	Domain Concept
SHDLgenFor	SHDLforInfoType	1	generate Statement, see Section 5.5.2.2

«SHDLmodule»

This stereotype represents the design entity concept as described in Sections 5.5.1.1 and 5.5.2.2.

Stereotype	Base Class	Tags
«SHDLmodule»	Class	--None--

«SHDLparBlock»

This stereotype represents the grouping of concurrent statements that exist within the architecture body, but outside any process (see Section 5.5.2.2).

Stereotype	Base Class	Tags
«SHDLparBlock»	Object	--None--

«SHDLprocess»

This stereotype represents the existence of a process in the architecture body (see Sections 5.5.2.2 and 5.5.2.3).

Stereotype	Base Class	Tags
«SHDLprocess»	Object	SHDLsensitive SHDLsensitive_pos SHDLsensitive_neg

It defines the following tagged values:

Tag Name	Tag Type	Multiplicity	Domain Concept
SHDLsensitive	TVL List of signals that a process is sensitive to changes, e.g. (‘clear’, ‘reset’)	0..1	Sensitivity List (see Section 5.5.2.2 and 5.5.2.3)
SHDLsensitive_pos	TVL List of signals whose positive edge a process is sensitive to	0..1	Sensitivity List (see Section 5.5.2.2 and 5.5.2.3)
SHDLsensitive_neg	TVL List of signals whose negative edge a process is sensitive to	0..1	Sensitivity List (see Section 5.5.2.2 and 5.5.2.3)

«SHDLtypedef»

This utility stereotype enables the HDL-specific data type and data object type associated with a particular data value to be captured using a user-defined attribute type (see Section 5.5.2.1). The «SHDLtypedef» defines one tag, namely, SHDLdefineType.

Stereotype	Base Class	Tags
«SHDLtypedef»	Class Object Note	SHDLdefineType

The SHDLdefineType tag is defined as follows:

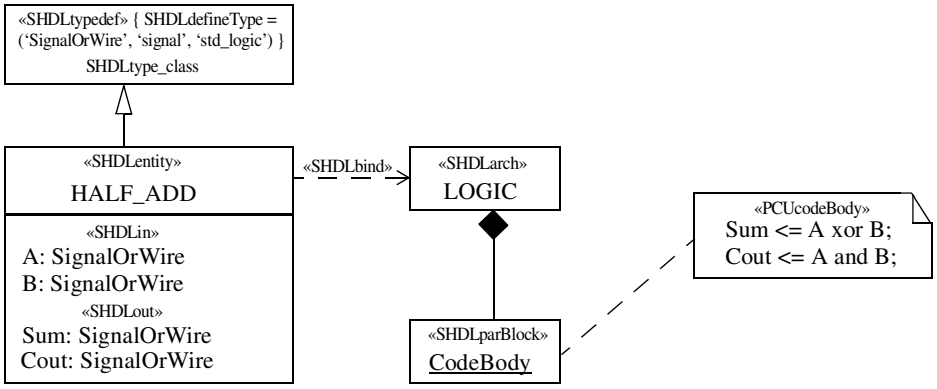
Tag Name	Tag Type	Multiplicity	Domain Concept
SHDLdefineType	SHDLattrType	1..*	HDL-specific data type and data object type, see Section 5.5.2.1

5.5.2.5 Example Usage

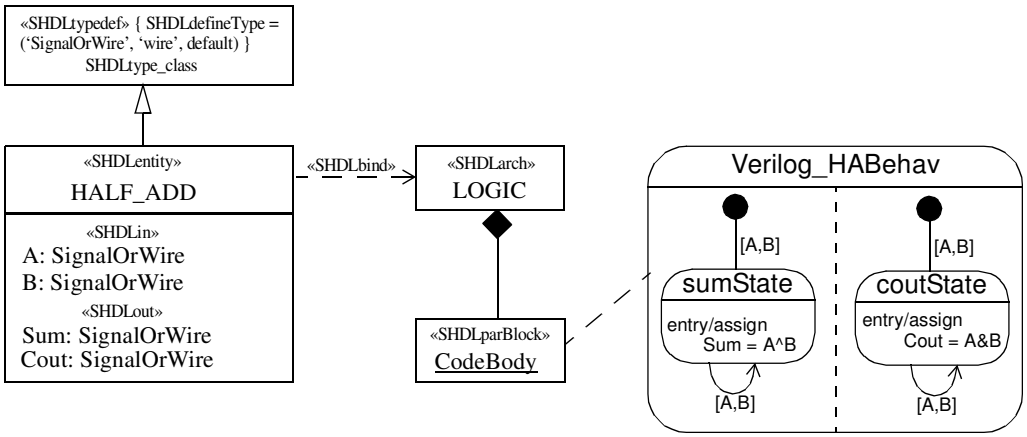
To demonstrate the usage model for the SHDLprofile package, specifically the framework depicted in Figure 5.7, an implementation of a six-bit-add-two-bit adder modified from Smith [113] is presented.

The example uses the structural style of hardware modeling to eventually realize a six-bit-add-two-bit adder. A one-bit half adder is first described that is utilized later to construct a one-bit full adder. Then by properly connecting full adder instances together, the example is able to attain the structural description of a six-bit-add-two-bit adder as desired. Figure 5.8 portrays the UML models of the half adder targeted for (a) VHDL and (b) Verilog; different implementation styles are used for demonstrative purposes.

Although not explicitly displayed in the figure, properly package importation shall be strictly exercised. Figure 5.9 depicts the implementation model of a full adder, using the half adder entities in Figure 5.8. Figure 5.10 shows how the desired six-bit-add-two-bit adder is implemented for the targeted VHDL. Because of its regular structure, the design takes advantage of the generate statement. The model for Verilog should be fairly easy to acquire from the VHDL model, with only one discrepancy: All six FA instances must be explicitly instantiated in Verilog for it does not have a generate equivalent. Specific tools may choose to support the generate model for Verilog; it is purely tool-dependent.



(a) VHDL Model



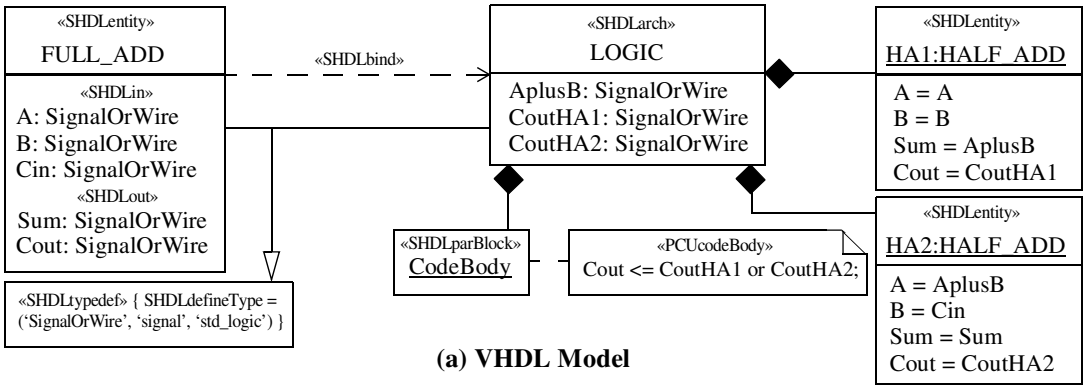
(b) Verilog Model

```
-- VHDL
-- Library usage not shown
--
entity HALF_ADD is
  port(A, B: in std_logic; Sum, Cout: out std_logic);
end entity HALF_ADD;

architecture LOGIC of HALF_ADD is
begin
   Sum <= A xor B;
   Cout <= A and B;
end architecture LOGIC;
```

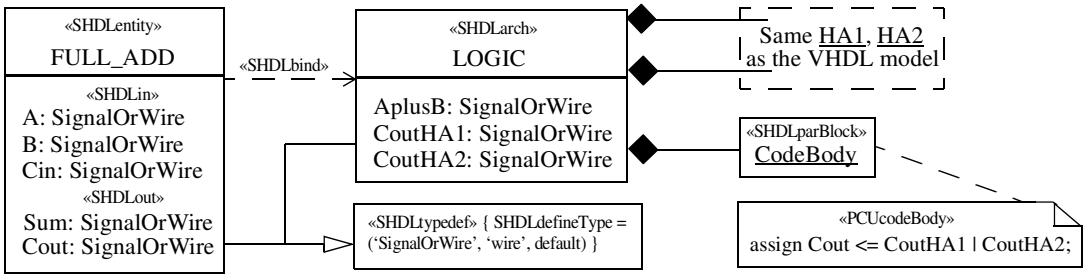
```
// Verilog
//
module HALF_ADD(A, B, Sum,Cout);
   input A, B;
   output Sum, Cout;

   assign Sum = A ^ B;
   assign Cout = A & B;

endmodule
```

(c) Source code

Figure 5.8: A half adder implementation in (a) VHDL, and (b) Verilog. The corresponding source code in VHDL and Verilog is shown in (c).



(a) VHDL Model



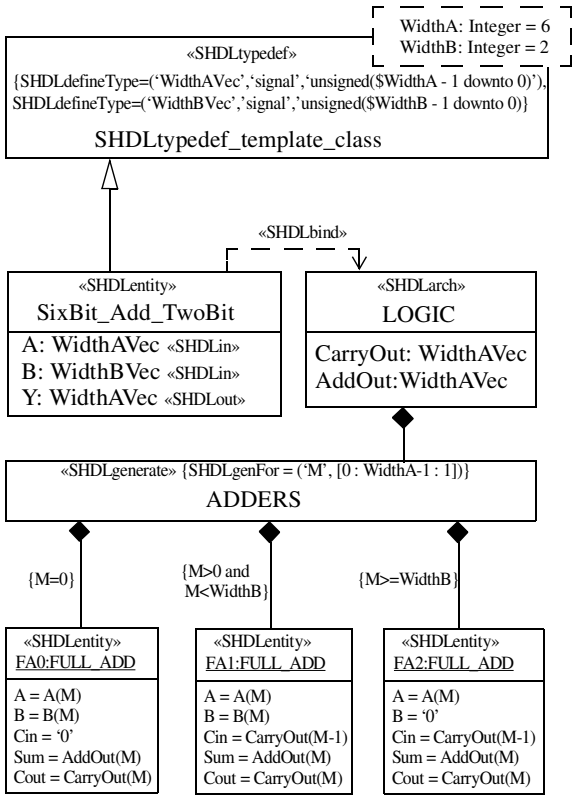
(b) Verilog Model

```
-- VHDL
--
entity FULL_ADD is
port (A, B, Cin: in std_logic;
       Sum, Cout: out std_logic);
end entity FULL_ADD;

architecture LOGIC of FULL_ADD is
  component HALF_ADD
    port (A, B: in std_logic;
          Sum, Cout: out std_logic);
    end component;
  signal AplusB, CoutHA1, CoutHA2: std_logic;
begin
  HA1: HALF_ADD port map (A=>A, B=>B,
         Sum=>AplusB, Cout=>CoutHA1);
  HA2: HALF_ADD port map (A=>AplusB,
         B=>Cin, Sum=>Sum, Cout=>CoutHA2);
  Cout <= CoutHA1 or CoutHA2;
end architecture LOGIC;
```

```
// Verilog
//
module FULL_ADD (A, B, Cin, Sum, Cout)
    input A, B, Cin;
    output Sum, Cout;

    wire AplusB, CoutHA1, CoutHA2;

  HALF_ADD HA1(.A(A), .B(B), .Sum(AplusB),
                .Cout(CoutHA1));
  HALF_ADD HA2(.A(AplusB), .B(Cin),
                .Sum(Sum), .Cout(CoutHA2));

  assign Cout = CoutHA1 | CoutHA2;

endmodule
```

(c) Source code

Figure 5.9: A full adder implementation in (a) VHDL, and (b) Verilog. The corresponding source code in VHDL and Verilog is shown in (c).



(a) VHDL-targeted six-bit-add-two-bit adder

```
entity SIXBIT_ADD_TWOBIT is
  generic(WidthA: integer := 6;
          WidthB: integer := 2);
  port(A: in unsigned(WidthA-1 downto 0);
       B: in unsigned(WidthB-1 downto 0);
       Y: out unsigned(WidthA-1 downto 0));
end entity SIXBIT_ADD_TWOBIT;

architecture LOGIC of SIXBIT_ADD_TWOBIT is
  component FULL_ADD
    port(A, B, Cin: in std_logic;
         Sum, Cout: out std_logic);
  end component;
  signal CarryOut: unsigned(WidthA-1 downto 0);
  signal AddOut: unsigned(WidthA-1 downto 0);
begin
ADDERS: block
  begin
    for M in 0 to WidthA-1 generate
      if(M=0) generate
        FA0: FULL_ADD port map (A=>A(M),
          B=>B(M), Cin=>'0', Sum=>AddOut(M),
          Cout=>CarryOut(M));
      end generate;
      if(M>0 and M<WidthB) generate
        -- Port Map
      end generate;
      if(M>=WidthB) generate
        -- Port Map
      end generate;
end block ADDERS;
end architecture LOGIC;
```

(b) Source code

Figure 5.10: A six-bit-add-two-bit adder implementation in VHDL (a), and the corresponding source code (b).

5.6 Architecture Blueprint Modeling (ABprofile)

The UML profile for Codesign Modeling Framework comprises several subprofiles, as described in prior sections. The PCUprofile supplies for the other subprofiles the common facilities, as well as the essential link to the LPO. The EMprofile and IMprofile packages provide for a more convenient real-time application development by allowing exception handling routines, and interrupt service routines, respectively, to be modeled at a higher level of abstraction. On the hardware side of the Codesign Modeling Framework, the SHDLprofile addresses the need for UML modeling of synthesizable HDL languages, that, when applied in tandem with the other subprofiles in the framework, renders a uniform hardware and software environment for the development of platform-centric SoC systems.

The Architecture Blueprint Modeling profile (ABprofile), on the other hand, deals with the UML representation of platform architectures that renders easy configuration and/or derivation of the desired target architecture.

5.6.1 Domain Viewpoint

A system platform can involve predesigned and precharacterized hardware, middleware, and software components (see Section 2.1, *The Platform Concept*, for further details). A combination of these components can result in myriad possible target architectures. As such, it is only appropriate that the platform architecture be represented abstractly in such a way that subsequent instantiations of platform-compatible components can conveniently occur that yield the target architecture as the desirable product. The abstract representation of platform architectures is specifically referred to in this dissertation as an *architecture blueprint*, or simply *blueprint* (see Chapter 4: Definition 4.7, and Section 4.2.1.7).

In principle, the abstract architecture specified by an architecture blueprint aids the configuration/derivation of the target architecture by furnishing an architectural template for the system developer. A blueprint-specific tool or tools are expected to always accompany the blueprint to assist in the process of configuring/deriving the target architecture.

5.6.1.1 Modeling the Blueprint for Configuring/Deriving the Target Architecture

UML allows a hardware platform to be modeled as a Node. By associating components, applications and/or middleware with a Node, the developer can acquire the following:

- Explicitly portray relationships among software and hardware components,
- Draw pre-characterized Node information to be used for such tasks as performance analysis, and schedulability analysis.

However, in the hardware-software codesign environment, it often requires more than just representing hardware platforms and deploying software components in order to be useful. Routine codesign tasks such as mapping HDL code to a programmable logic device or instantiating a core onto the target architecture cannot be satisfactorily handled by means of standard UML notations. All these testaments mandate that a more comprehensive model shall be implemented to satisfy beyond the normal confines of software engineering. Now by associating a certain design with an architectural blueprint (AB) type, the developer and tool shall be able to interpret the appropriate relationship between them, some of which are shown in Table 5.4.

The benefits of utilizing the AB types in the proposed approach can arguably be twofold: (1) as previously mentioned, implicative interpretation of the relationship between design and AB entities can be deduced, and (2) it furnishes coarse-grained categories in an architecture blueprint that can be used as search keywords within the LPO domain. Below is the list of these AB types:

- *Programmable/Reprogrammable Unit (PRU)*. The category encompasses blueprint entities that can be programmed and/or reprogrammed *on the field*. Typically, an entity in this class provides a quick system prototype for the developer. Examples are Field Programmable Gate Array (FPGA), Programmable Logic Device (PLD), and Electronically Erasable Programmable ROM (EEPROM).

- *Implementable Unit (IU)*. Affiliated blueprint entities in this class bear some similarities to those in the PRU category. They are designed by the developer, but must be sent for fabrication and packaging after the design and verification. Because of this distinct nature, the IU belongs in its own class apart from the PRU. Examples of AB entities in this category is the Application-Specific IC (ASIC), and standard cell devices.
- *Drop-In Unit (DIU)*. The defining characteristic of AB entities in this class is that they have been pre-developed, i.e. all characteristics can be precisely acquired. Unless it is a processor, this type of entity has no direct interactions with the design and requires an interface unit (IFU), e.g. device driver and controller, to manage the communications. Example of AB entities in this category are peripheral devices, processors and coprocessors.

Table 5.4: Semantic inferences of the relationships between design components and physical hardware

Design	Blueprint Type	Inference
HDL, EDIF netlist, Data	Programmable/Reprogrammable Unit, e.g. FPGA	Configuration file for a PRU unit is expected
HDL, EDIF netlist	Implementable Unit, e.g. ASIC.	Design must be fabricated
--None--	Drop-in Unit, e.g. peripheral devices, DSP	Hardware entities already in full-development; may require interface module for communication
General design component	Drop-in Unit: a processor	Design is deployed in a processor
Data	Memory Unit (MU)	Data resides in memory
Application; LPO component	Interface Unit (IFU)	Device driver, controller exists; Pin connections information exists.
General design component	Clock	Clock utilization
General design component	Timer	Timer utilization

- *Interface Unit (IFU)*. The entities in this class provides a communication means for the design and other AB entities. Examples of the IFU entities are PCI bus, Parallel I/O, and Firewire.
- *Memory Unit (MU)*. The AB entities of this type represent the storage element class. Semantically, the MU is a subtype of the PRU; it exists primarily for the search purpose. Examples are RAM and ROM.
- *Clock*. The Clock type is a subtype of the DIU type. Nonetheless, because of its perennial presence in real-time systems, it merits to be distinguished from the rest of the class. When present, the «RTclock» stereotype in the UML real-time profile [29] shall refer to it.
- *Timer*. Like Clock, the Timer type is a subtype of the DIU and it corresponds to the «RTtimer» concept in the UML real-time profile [29].

5.6.2 UML Viewpoint

In defining UML extensions, i.e. stereotypes, tagged values and/or constraints, for the ABprofile package, the prefix AB is always attached to the names to differentiate them from similar or same names in other profiles.

5.6.2.1 Mapping Blueprint Domain Concepts into UML Equivalents

Physical hardware maps to a class that is stereotyped with «ABnode». A design unit, describing hardware implementation and software application alike, maps to a blueprint component represented by an «ABcomponent»-adorned class.

The «ABnode» stereotype comprises one tag: ABisKindOf. This tag permits the type of the «ABnode» class to be documented (see Section 5.6.1.1). The mapping of the «ABcomponent» class onto the «ABnode» class is specified by the «ABmap» stereotype adorned on a Dependency. Similarly, the «ABdeploy» stereotype is used to indicate the mapping of software application to the «ABnode».

5.6.2.2 Mapping the Blueprint Model Instance into the Physical Model

To prove the validity of using the AB model to configure/derive the target architecture in place of the physical model, the mapping that demonstrates the one-on-one relationship between the target architecture and the physical models is shown in Table 5.5.

From Table 5.5, it can be seen that the homomorphic mapping is possible with a couple of exceptions: there are no corresponding semantics in the physical model for the «ABprogram» and «ABbecome» stereotypes. This comes as an evidence for the inadequacy of the physical model as a modeling facility for the hardware-software codesign environment.

Table 5.5: Mapping of the AB's target architecture model into the physical model

Target Architecture Model	Physical Model	Target Architecture Model	Physical Model
'clock' Object	clock Node	'timer' Object	timer Node
Object	physical HW	parametrized Object	configured HW
Class Type	Node Type	Class Instance (owner)	Node Instance
Link	Communication	Class Instance (owned)	Component Instance
Dependency	Dependency	Composition	Composition
«ABdeploy»	«deploy»	«ABprogram»	--None--
		«ABbecome»	--None--

5.6.2.3 UML Extensions

To avoid any possible duplicate and ambiguity, all extensions defined in this profile are prefixed with AB.

«ABbecome»

This stereotype represents the relationship between a design unit and a hardware entity of type IU (see Section 5.6.1.1). It indicates that the design unit is a synthesizable HDL description of the hardware entity.

Stereotype	Base Class	Tags
«ABbecome»	Dependency	--None--

The following constraint is defined for this stereotype:

- The client element at the tail of the arrow must be of IU type.

«ABcomponent»

This stereotype models a design unit that may indiscriminately represents application software, or a hardware component (see Section 5.6.1.1).

Stereotype	Base Class	Tags
«ABcomponent»	Class Object	--None--

«ABdeploy»

This stereotype infers that a design unit can be supported by a hardware entity. Normally, it signifies the residency of a design unit in the hardware. It is the ABprofile equivalent of the standard «deploy» stereotype.

Stereotype	Base Class	Tags
«ABdeploy»	Dependency	--None--

«ABnode»

This stereotype represents hardware as view by the architecture blueprint (see Section 5.6.1.1).

Stereotype	Base Class	Tags
«ABnode»	Class Object	ABisKindOf

The tag for this stereotype is defined as follows:

Tag Name	Tag Type	Multiplicity	Domain Concept
ABisKindOf	Enumeration: (‘pru’, ‘iu’, ‘diu’, ‘ifu’, ‘mu’, ‘clock’, ‘timer’)	0..1	AB Type (see Section 5.6.1.1)

«ABprogram»

This stereotype represents the relationship between a design unit and a hardware entity of type PRU, where the design unit is used to program the hardware entity (see Section 5.6.1.1).

Stereotype	Base Class	Tags
«ABprogram»	Dependency	--None--

The following constraint is defined for this stereotype:

- The client element at the tail of the arrow must be of PRU type.

5.7 UML to SystemC Mapping

In Section 2.4, the collaborative usage model of the platform-centric and the SystemC approaches was presented, and potential advantages were discussed. This section focuses instead on the mapping between the two models to demonstrate that models coherence can be maintained, and that the mapping process can be automated.

Table 5.6: Mapping of the SystemC constructs to the platform-centric UML models (SW and HW perspectives)

SystemC	UML (SW Perspective)	UML (HW Perspective)
SC_MODULE	«CRConcurrent»	«SHDLentity»
SC_CTOR	«CRMain»	«SHDLarch»
SC_METHOD	method	method
SC_THREAD	«CRAction» {isAtomic=true}	«SHDLprocess»
SC_CTHREAD	«CRAction» {isAtomic=true} *Note: With presence of clock*	«SHDLprocess» {SHDLsensitive=('clock')}
sc_main()	«CRAction» sc_main	«SHDLentity» sc_main

Because SystemC is entirely based on an OO language, i.e. C++, mapping to and from UML is quite natural and intuitive. However, of particular interest is how specialized SystemC macros and functions map to the UML constructs. Table 5.6 summarizes this mapping from the software and hardware points of view. The fact that the proposed approach is software-biased probably would make the software modeling scheme the more attractive between the two. This SW-viewed UML/SystemC model utilizes the RTconcurrencyModeling package defined in the UML Real-Time Profile [29] as the basis for capturing SystemC specialized constructs. On the other hand, the hardware approach could be useful for migrating existing UML/HDL models to the UML/SystemC. In such a case, an intelligent tool would be capable of understanding the SystemC constructs from the SHDLprofile-based UML models, resulting in minimal changes required of the UML models. It also eases the task of replacing HDL code with the corresponding SystemC.

As an example, consider an implementation of a D flipflop with an asynchronous reset, whose VHDL description is presented below. Figure 5.11 delineates the SW-viewed UML/SystemC model of such a flipflop, while Figure 5.12 depicts the model as perceived from the HW viewpoint.

(Use RTconcurrencyModeling, PCUprofile)

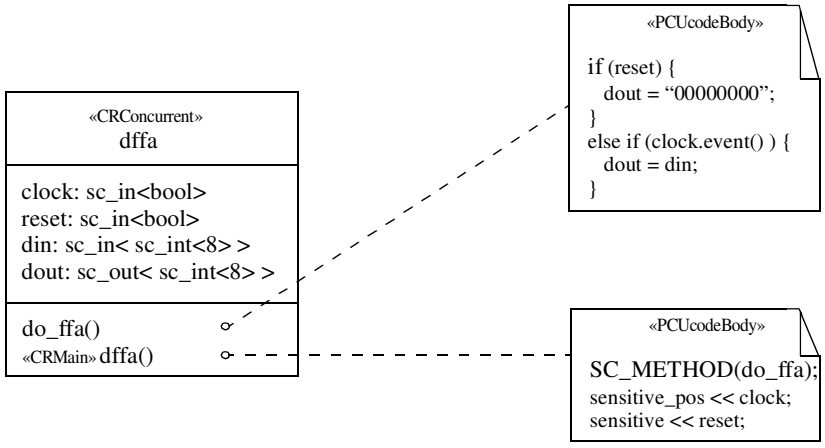


Figure 5.11: A SW-viewed UML/SystemC model of an 8-bit D-F/F

```
-- VHDL implementation of an 8-bit D-F/F
-- dffa.vhd

entity dffa is
    port( clock :   in std_logic;
          reset :   in std_logic;
          din   :   in std_logic_vector(7 downto 0);
          dout  :   out std_logic_vector(7 downto 0) );
end dffa;

architecture rtl of dffa is
begin
    process(reset, clock)
    begin
        if reset = ‘1’ then
            dout <= “00000000”;
        elsif clock’event and clock = ‘1’ then
            dout <= din;
        end if;
    end process;
end rtl;
```

(Use SHDLprofile, PCUprofile)

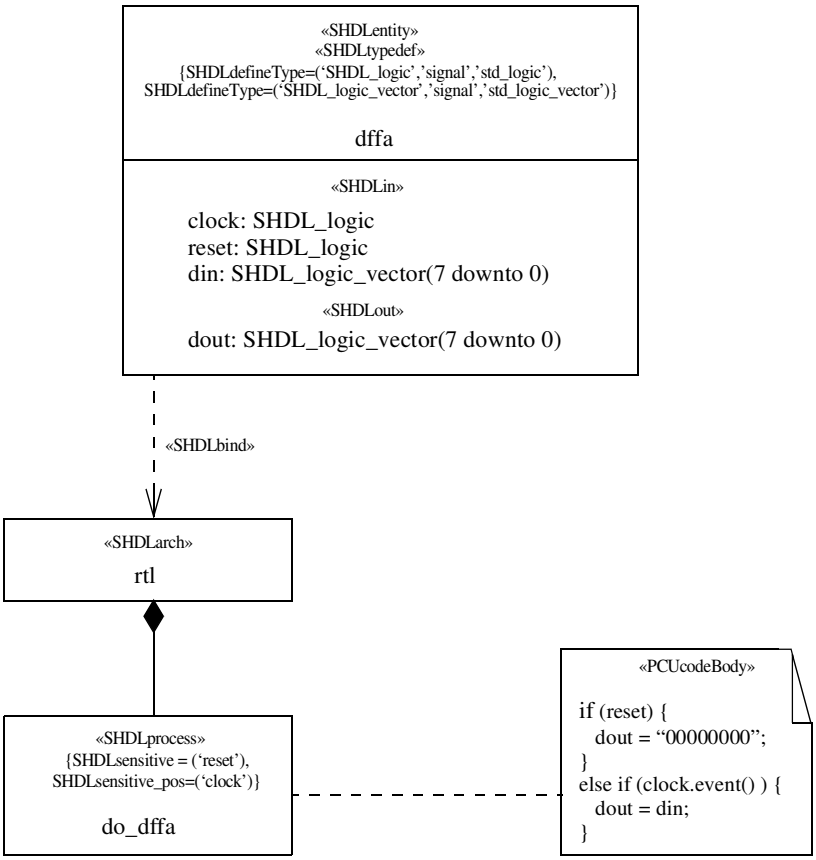


Figure 5.12: A HW-viewed UML/SystemC model of an 8-bit D-F/F

Chapter 6

Application Case Study: A Simplified Digital Camera System

This chapter demonstrates the robustness of the proposed platform-centric SoC design approach by using it to develop a simplified digital camera system. A digital camera is a complex system comprising both mechanistic and electronic components—rendering it very well-suited as an application case study for this dissertation. The chapter starts by giving an overview of typical digital camera operations, as well as relevant mechanistic and electronic components. Thereafter a set of general requirements is given, and the development process begins in the manner prescribed by the proposed approach (see Chapter 2). After detailing the tasks involved in each main step of the platform-centric development process flow, i.e. the *platform-independent, platform-analysis, platform-dependent*, and *system derivation process* steps, the chapter concludes by comparing cost-effectiveness of the proposed approach with that of the SpecC approach [20] using COCOMO II.2000 [19].

6.1 Digital Camera System

Comparing to a traditional film camera, a digital camera operates very much on the same principle, although minute operational details differ considerably. It has by and large the same user interface as that of a film camera, but with additional options only attainable through digital photography. A digital camera captures and stores images in digital format on a storage device. Figure 6.1 depicts a block diagram of a typical digital camera system.

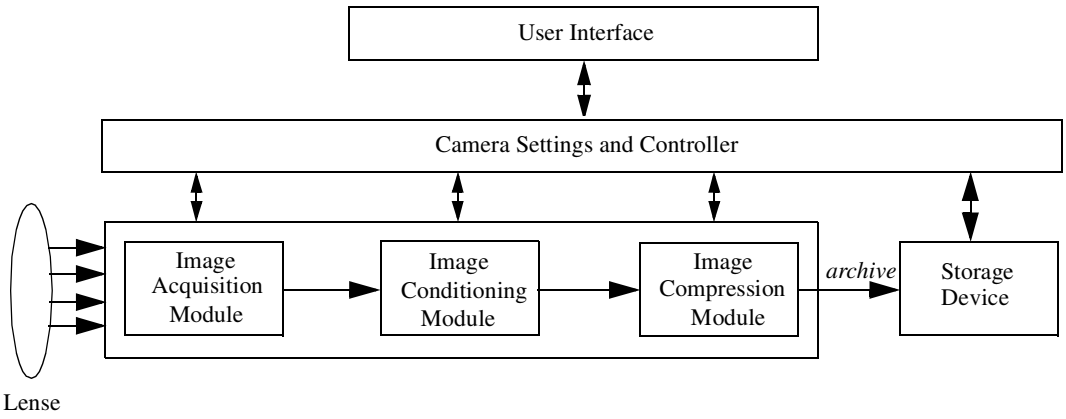


Figure 6.1: Block diagram of a typical digital camera system.

6.1.1 Image Acquisition Module

Just like a traditional film camera, light reflecting off the scene or subject is directed by the lense onto the image acquisition module, which normally comprises an image sensor, and/or an analog to digital converter (ADC). The duration and amount of light exposure is regulated by a shutter and an aperture, respectively.

An image sensor constitutes the core for any digital camera system. It is based predominantly on either the Charge Coupled Device (CCD) or Complementary Metal Oxide Semiconductor (CMOS) technology (see Table 6.1 for comparison). Within these sensors, photocells are arranged in rows and columns, and become electronically charged when exposed to light. "This charge can then be converted to an 8-bit value where 0 represents no exposure while 255 represents very intense exposure of the cells to light [120]." Some of the columns of the photocell array are covered with a black strip of paint, and are used for zero-bias adjustments, i.e. *white balancing*, of all the cells in the array. Because the CMOS imager is inherently noisy, an ADC is often integrated to help prevent further image quality deterioration [121].

Table 6.1: Comparison of the CMOS and CCD image sensors [121, 122]

Criteria	CMOS image sensor	CCD image sensor
Cost-effectiveness	More cost-effective	Less cost-effective
Image Quality	Less superior (more susceptible to noise)	More superior (less susceptible to noise)
Pixels Access/ Download	Direct access possible (similar to memory access)	No selective read. Must serially flush off the whole image
Integrated ADC	Yes	No

Normal digital camera operations commence with the process of determining proper settings for the scene or subject to be photographed. Such tasks typically involve adjusting the focus, setting image quality, measuring and gathering shooting parameters, and selecting an appropriate shutter duration and aperture opening (*f-stop* in photographic term). Once the required parameters are set and the shutter button is pressed, the following sequence of operations typically ensues [121]:

- The shutter is closed; the sensor becomes temporarily inactive, and is instantly flushed off all residual charges. This step is to prepare the sensor to capture a new image.
- Depending on the camera and the settings, the residual charges that are flushed off the sensor may be analyzed to acquire the proper settings for automatic point-and-shoot operations. Or if a LCD viewfinder is present, it will display the flushed image on the screen.
- The sensor becomes active and, at the same time, the shutter opens, exposing the sensor to light—charging it as a result. The shutter remains open for the specified exposure duration, before closing again. The image can now be captured and streamed off to the Image Conditioning module.

- The shutter re-opens, and the camera is ready to take another picture.

In many digital cameras, the time it takes for the sensor to flush, as well as to read and set the shooting control parameters is often non-trivial—ranging from as little as 60 milliseconds to as long as 1.5 seconds [121]. This so-called shutter lag can be improved by utilizing a larger buffer memory, and/or a faster processor. Higher communication bandwidth can also speed up the lag. In some CMOS image sensors where rows and columns can be selectively read, the shutter lag tends to improve as well.

6.1.2 Image Conditioning Module

The functionality of this module chiefly concerns color processing and image enhancing tasks aimed to render a visually better image for the user. The actual algorithms for achieving such a result vary from one camera to another, depending on the imposed design criteria.

When the Image Conditioning module receives the digital data representation of the image from the Image Acquisition module, specifically, the ADC, it first determines whether or not demosaicing is required. Most digital cameras today carry only one image sensor, instead of multiple image sensors for multiple color components. As such, to be able to acquire a full resolution image from a single image sensor, the color filter array (CFA) architecture is utilized to assign a color component to each photocell [121]. The CFA-imposed digital data from this sensor can later be demosaiced with respect to the CFA architecture to derive the full resolution image. An example of the CFA architecture is the popular Bayer pattern.

Once a full resolution image is acquired, color processing and image enhancing tasks can begin. These tasks involve, for example, color balance and saturation settings, white balancing, noise removing, as well as other image effects and enhancement such as sharpness enhancing, red-eye eliminating, or sepia coloring.

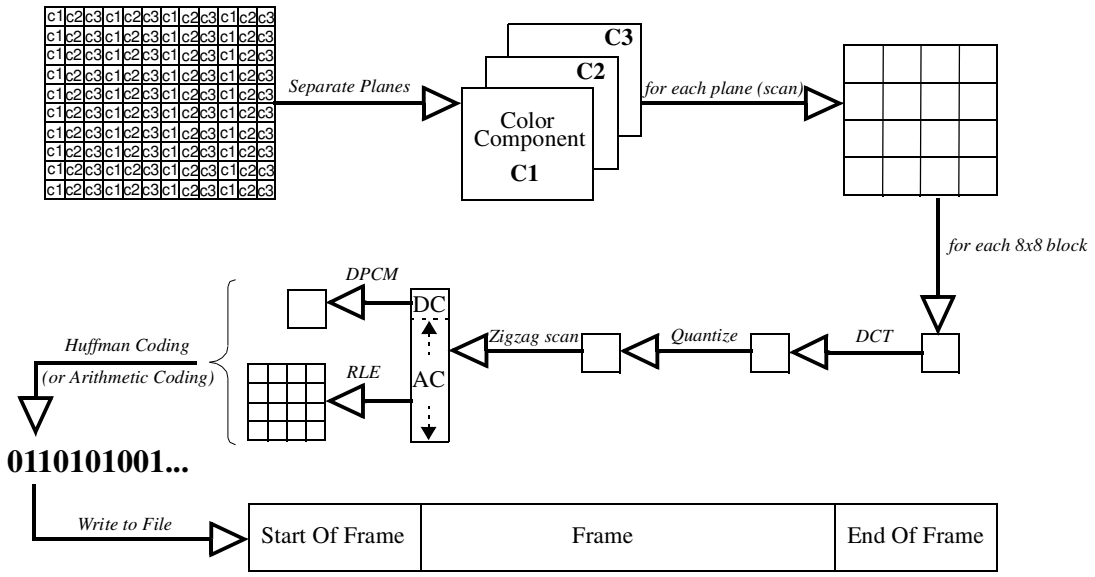


Figure 6.2: Block diagram of the baseline JPEG encoder

6.1.3 Image Compression Module

To make optimal use of the storage device as well as to expedite upload/download time, the preprocessed image from the Image Conditioning module may further be compressed before writing onto the media. Most digital cameras let the user choose the image quality settings that determine (1) the compression quality, and (2) the image dimension. Often, this module is also responsible for generating image thumbnails.

Arguably, the most popular image compression algorithm used by digital cameras today is the JPEG standard. JPEG works well with continuous-toned images, e.g. natural scenes or photograph pictures. It defines both lossy and lossless algorithms, as well as the embodiment of minimum requirements, called *Baseline JPEG* or simply just *JPEG*, that guarantees portability across different decoder implementations. The baseline JPEG is lossy; it, nonetheless, permits different compression quality settings to be specified—for example, a value in the range of 0 to 100 where 0 means most compressed, worst quality,

and 100 least compressed, best quality. Figure 6.2 illustrates the baseline JPEG encoder block diagram.

In Figure 6.2, the JPEG encoder takes the color components representing the image as its inputs, and divides each of them into non-overlapping blocks of 8x8 pixels. For each block, JPEG transforms the spatial data into frequency domain using the Discrete Cosine Transform (DCT). Then in the quantization step, it rounds these frequency data (DC and AC components) to the closest pre-defined values in the quantization tables, attempting to minimize the number of total bits—hereby, resulting in a smaller image size. As the human vision is less sensitive to high-frequency components, larger gains in compression ratio may further be achieved by allowing larger errors to occur and/or eliminating some high-frequency components altogether. The quantized data are then entropy-coded using Differential Pulse Code Modulation (DPCM) and Huffman encoding for DC components, and Run-Length encoding (RLE) and Huffman encoding for AC components. Finally a JPEG file can be produced using the predefined file format.

6.2 Digital Camera System Requirements

For the case study, a functional prototype of a digital camera is developed that must meet the following initial requirements. As before, requirement levels are indicated using the keywords described in the guideline furnished by the Internet Engineering Task Force (IETF)'s RTF2119 [63], as shown in Table 4.1.

6.2.1 Functional Requirements

Functional requirements describe system behaviors. For the case study, the following functional requirements are imposed.

6.2.1.1 General Operational Requirements

The image acquisition and conditioning modules, illustrated in Figure 6.1, are assumed to be implemented by another team. This simplified digital camera shall comprise only the

user interface, the JPEG encoder module (see Figure 6.2), and the archiving step that writes the compressed image onto a media. It must provide means to upload images onto a PC. It is also responsible for implementing the camera control logics.

6.2.1.2 User Interface's Operational Requirements

This simplified digital camera system has a pre-set shutter speed, as well as fixed aperture and focus settings. However, it permits the user to select image quality (*normal* or *good*), as well as a single shot or 2-shot burst mode. The camera status should be appropriately displayed.

Stored images can be removed via the digital camera's user interface. An *image upload* operation causes all stored images to be transferred to a PC, without deleting the images on the media; it is analogous to the *file copy* operation.

6.2.1.3 Input and Output

The inputs are the color components of the image; how these inputs are read shall be determined by the actual implementation of the JPEG encoder. The eventual output is a compressed JPEG file or files (in a 2-shot burst mode) written onto a media.

6.2.2 Non-functional Requirements

Non-functional requirements are requirements that do not concern system behaviors; they encompass such metrics as:

- Performance, e.g. time required to process an image,
- Size, e.g. number of logic elements, and/or size of embedded software,
- Power, e.g. measure of average power consumption,
- Energy, e.g. battery lifetime.

Vahid and Givargis [120] describes non-functional requirements as consisting of constrained metrics, and optimization metrics. Constrained metrics are values that must not violate a specified threshold(s); whereas, optimization metrics refer to certain design goals that aim to improve the system. This section describes the non-functional requirements for the case study.

6.2.2.1 Operating Time Constraint

To be useful, the time used to take and store one image shall not exceed 1 second. The time is measured from the moment the shutter is pressed to the moment the camera is ready to take another picture.

Although faster operating time is normally desirable, it must also be thoroughly justified.

6.2.2.2 Heat Dissipation and Energy Requirement

Even though the prototype is implemented on the NiOS development board that relies on an AC-to-DC adaptor as the power source, the actual product will likely use batteries. Therefore, it is desirable to minimize the power consumption as much as possible, so as to prolong battery life. It is required, however, that no cooling fan shall be used.

6.2.2.3 Hardware Platform Requirements

A functional prototype of the simplified digital camera system being developed for the case study shall be deployed on the Altera NiOS soft core embedded processor platform targeted for the EP20K200EFC484-2X programmable logic device (PLD). Compatible platform components may be utilized that are available in the LPO, such that the resultant product closely emulates the actual digital camera operating environment. Descriptive summary of the NiOS processor and platform, as well as the EP20K200EFC484-2X PLD device is presented as follows (see their respective datasheets from Altera [81] for more details).

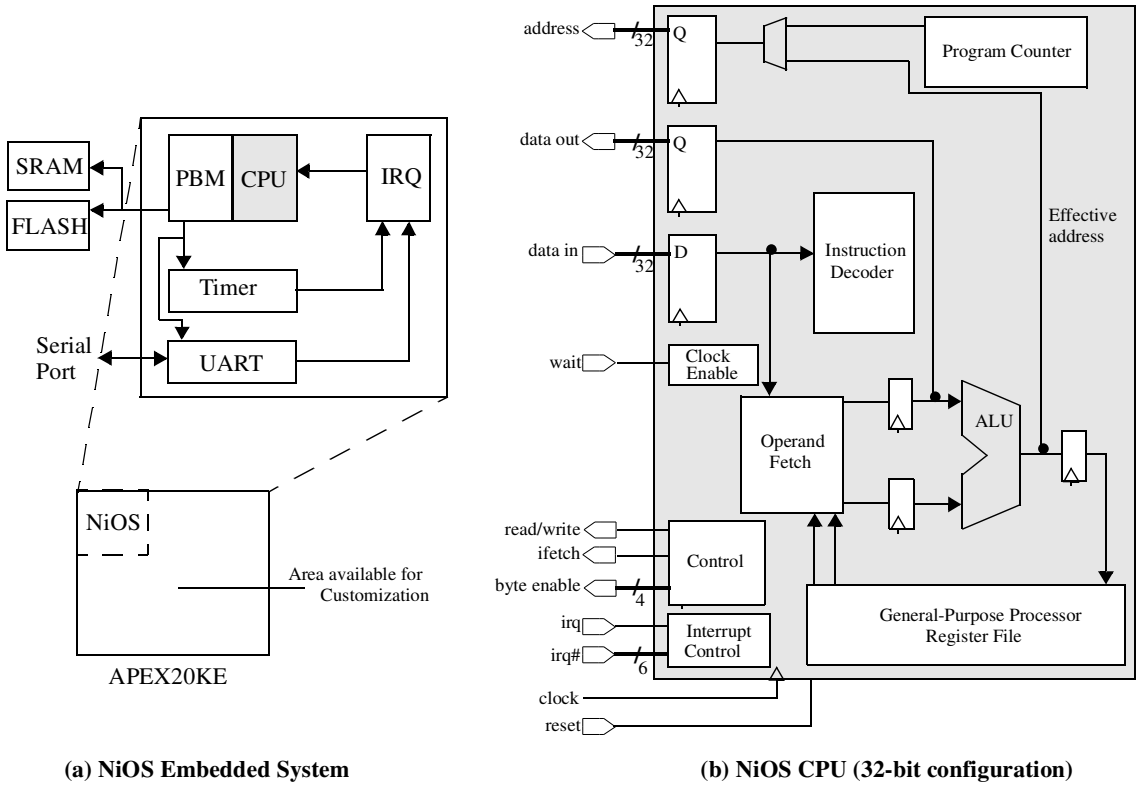


Figure 6.3: The NiOS embedded processor [81]

NiOS Embedded Processor

The NiOS embedded processor is a general-purpose, five-stage pipelined RISC soft processor core in which instructions run in a single clock cycle. The NiOS CPU can be configured for a wide range of applications; its 16-bit instruction set is targeted for embedded applications in particular. The NiOS supports a fully-synchronous address/data bus interface. In addition, it features a configurable 16-bit or 32-bit data path, 64 vectored interrupts, 1-to-31-bit single-clock shift operations, as well as 128 to 512 general-purpose registers. Two hardware-assisted multiplications are available that permit a single bit per cycle multiplication (MSTEP) or a fast integer multiplication (MUL). Instruction set extensions can be realized via the custom instruction feature. Figure 6.3 depicts the 32-bit NiOS processor.

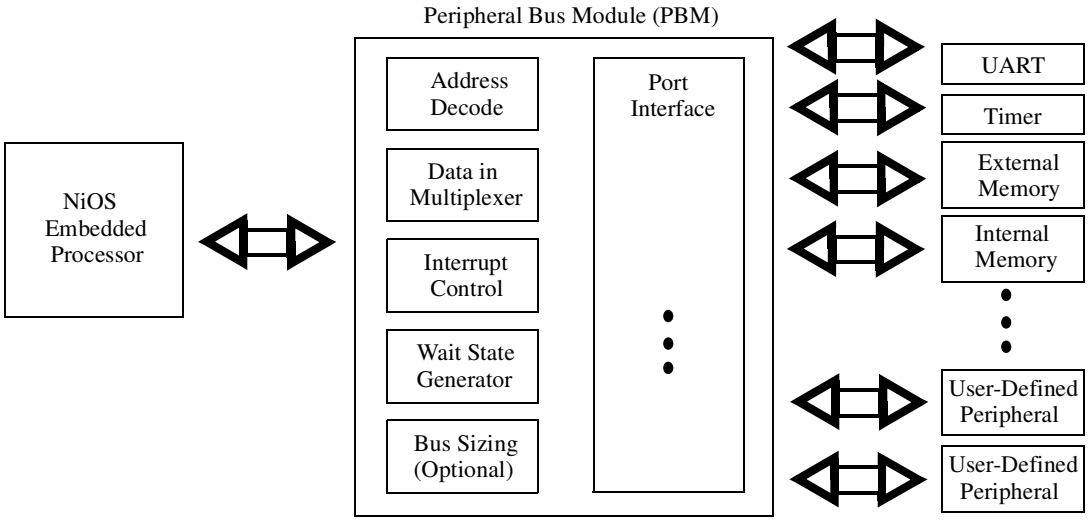


Figure 6.4: NiOS platform, showing communication between the NiOS processor core and its peripherals [81]

NiOS Platform

The NiOS embedded processor provides for customizable, on-chip peripherals that attributes to a convenient configuration of the NiOS platform. Once properly configured, the developer can have the peripheral bus module (PBM) for such peripherals as timer, SRAM, FLASH, universal asynchronous receiver/transmitter (UART), and parallel I/O (PIO) automatically generated. The following PBM features are fully customizable:

- Base address and address span
- Data width
- Read/Write access restrictions (read-only, write-only, read-write)
- Wait states, and
- IRQ signal/priority.

Figure 6.4 depicts the NiOS platform that portrays the communication between the NiOS embedded processor and its peripherals.

EP20K200EFC484-2X PLD Device

The EP20K200EFC484-2X belongs in the Altera APEX20KE device group within the APEX20K family. Like all of its sibling devices, the APEX20KE is built upon the MultiCore architecture consisting of logic array blocks (LABs), whose basic units are the logic elements (LEs). Each LAB comprises 10 LEs; 16 LABs can be combined to form a new hierarchical structure called MegaLAB. In addition to these 16 LABs, a MegaLAB contains an advanced embedded structure called an embedded system block (ESB).

The ESB is the heart of the MultiCore architecture. Each ESB contains 2048 programmable bits that can be configured as:

- product-term logic, which is superior for control logic functions such as address decoding and state machines,
- LUT-based logic, or
- three types of memory: dual-port RAM, read-only memory (ROM), or content-addressable memory (CAM) which allows the address to be identified from a data input.

Specifically, the EP20K200EFC484-2X device contains 52 ESBs/8320 LEs, with the maximum RAM bits (ESB bits) of 106496. It is housed in the 484-pin fine line, ball grid array (BGA) package; the maximum of 376 I/O pins are available for use by the system developer. The device allows up to two phase lock loop (PLL) implementations, and supports four voltage interfaces: 5 V, 3.3 V, 2.5 V, and 1.8 V.

The NiOS embedded processor core is optimized for the APEX20K programmable logic devices. When programmed on the EP20K200EFC484-2X, the 32-bit configuration uses approximately 20% of the resources (approximately 13% in the 16-bit configuration), and can execute up to 50 MIPS with the fastest permissible clock speed of 50 MHz.

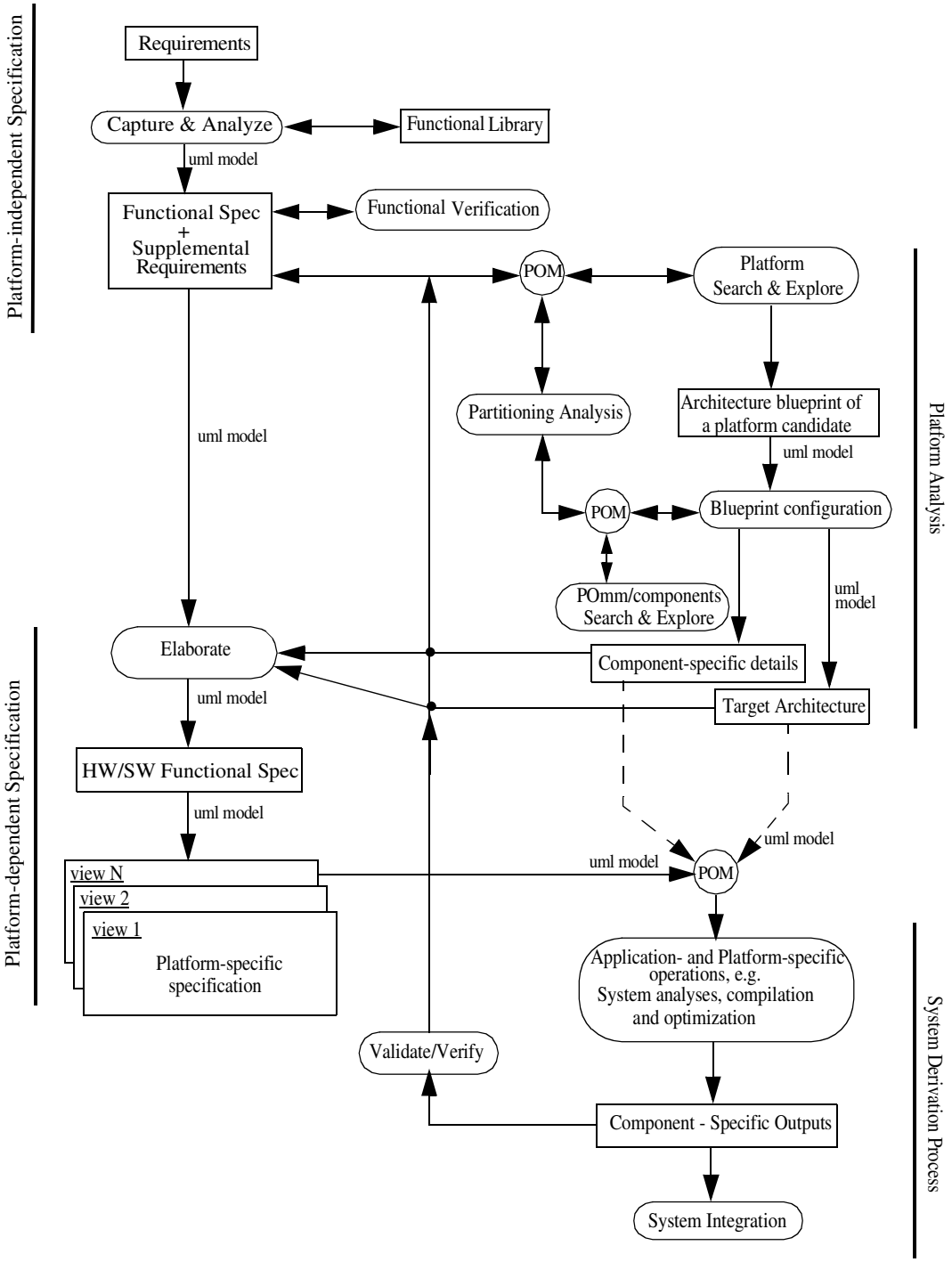


Figure 6.5: The platform-centric SoC method design flow

By using the platform-centric approach to develop the simplified digital camera described above, the required tasks follow the prescribed stages that are treated in detail in Chapter 2 and also illustrated in Figure 6.5. The remainder of this chapter presents specific work details for each stage, before concluding with the cost comparison against the SpecC methodology [20].

6.3 Platform-Independent Specification

This stage of the proposed platform-centric approach involves deriving the functional specification of the system that is still independent of any platform specifics. Due to this nature, proven techniques in software engineering are applicable to be used by the system developer. Some well-known UML processes include the Rose™ Unified Process [102], the UML/Catalysis as described by Graham [123], and the Rapid Object-Oriented Process for Embedded Systems (ROPES) covered in detail in the UML book by Douglass [28].

Much like the object-oriented programming techniques that specify functional interfaces without committing prematurely to any specific implementation, the platform-centric SoC design method imposes no specific UML process on the derivation of this specification. As a result, the proposed approach is flexible, and can readily adapt to the diverse requirements of the complex development process of the real-time embedded SoC systems. To derive the platform-independent specification in this stage, the dissertation employs the modified version of the Requirement Specification Process as detailed in Maciaszek [36]. This Use Case driven technique commences with the task of requirement determination that clarifies the initial requirement document, shedding more light to it from the developer's point of view. It then enters the elaboration loop from either the Use Case analysis or Class analysis entry points (or both, in parallel), and iteratively refining, and deriving the desired specification using various UML techniques. The resultant UML model comprises the static system structure captured by the Class diagram, the system behavior represented by such a diagram as Sequence or Collaboration, and the system state as described by the State diagram. Figure 6.6 depicts this process flow.

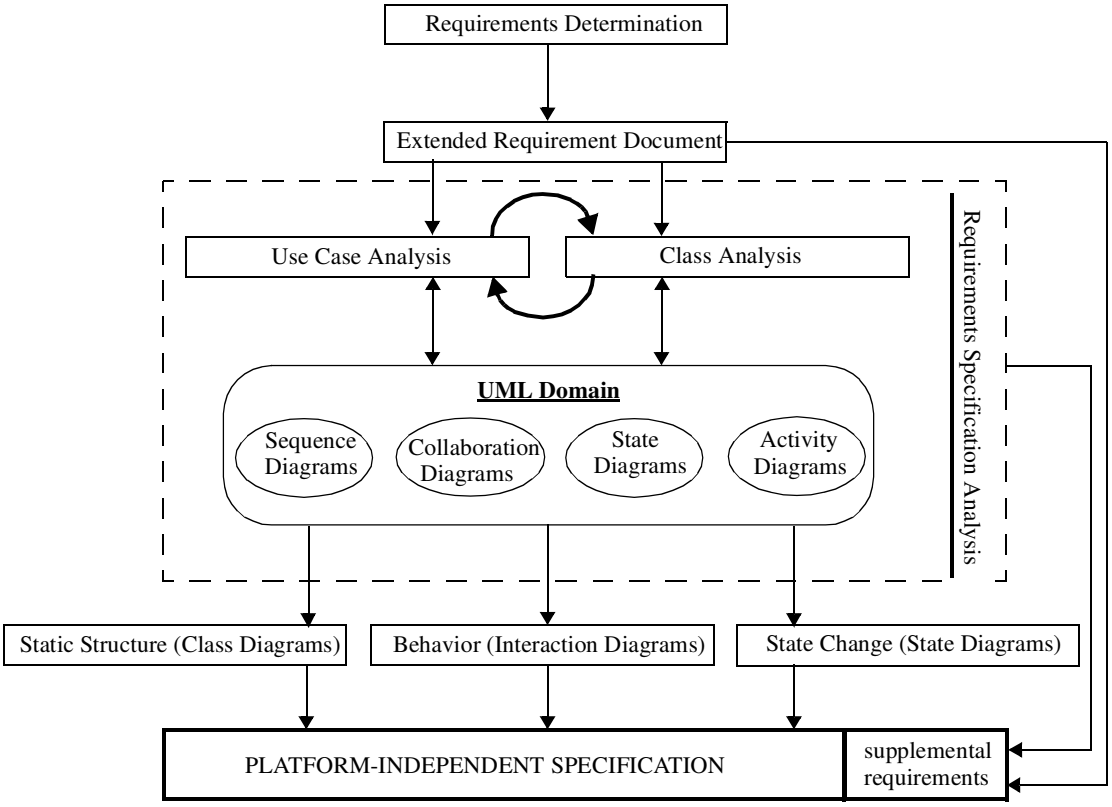


Figure 6.6: Platform-Independent Specification Process Flow

The requirements determination task often requires multi-lateral efforts involving many subtasks that will elicit, negotiate, validate, manage, and model the customer's requirements (see [36], [123] for details). The requirements document resulted from these subtasks is subsequently analyzed that eventually will yield the platform-independent specification.

It is quite normal in practice that the initial requirements are ambiguous and/or incomplete. The requirements statement presented in Section 6.2 decidedly provides insufficient information for the system developer to proceed further into the development process, and the developer need to go through the requirements determination process (preferably together with the customers). Since the requirements determination task is

somewhat beyond the scope of this dissertation, its detailed procedures will be omitted. The resultant extended requirements document, which is the product of reworking the requirements statement in Section 6.2, appears as follows:

Digital Camera's Extended Requirements

(**R1**) The user powers on the camera. The camera displays a ready message and also an image count. A LED light illuminates. The camera is ready to take a picture.

(**R2**) The user presses the shutter. The camera takes a picture and stores the image in the memory. During the operation, the LED light goes off and comes back on when finished. The image count on display gets incremented. The camera displays a ready message.

(**R3**) The user enters the menu mode by pressing the menu button. The user can browse the available options that include: image quality setting, shot mode, upload, and delete.

(**R4**) The user uses the select button to show and to change the current setting for each available option. For the image quality setting, the user can choose between "good" or "normal", with "normal" as the default. For the shot mode, the user can select either single-shot mode (default) or two-shot burst mode. The upload and delete settings can be either "yes" or "no" (default).

(**R5**) The user uses the done button to commit to the settings and brings the menu view up one level iteratively until exiting the menu mode altogether, where the camera is ready to take a picture.

(**R6**) The user uses the done button to activate the upload operation. When upload operation is requested, the camera transmits all images stored in the memory to the PC, where the interface software writes the JPEG files from these data. No image is deleted upon completion. When finished, the status is displayed with no change in the image count and the camera is back in the menu mode.

(**R7**) The user uses the done button to activate the delete operation where the camera removes all images from the memory. The image count on display is reset to zero, and the camera is back in the menu mode.

6.3.1 Use Case Analysis

Depending on the entry point that the extended requirements document enters the Requirements Specification Analysis domain, the Use Case analysis may either be the kick-start or the follow-up activity of the string of requirements specification refinement tasks that proceed sequentially and iteratively. It is also possible, and arguably beneficial, that the two analysis processes be performed simultaneously so as to impart one's analytical strength onto the other, and *ergo*, becoming more effective in deriving the platform-independent specification.

The Use Case diagram models the system as seen looking in from the outside, as such, making it attractive for modeling the requirements document which generally describes the *user's* viewpoint towards system functionalities. In UML terms, a Use Case diagram comprises one or more *use cases*, each of which captures a system behavior that is outwardly visible to an *actor* or *actors*, and that readily responds to external events caused by the actor. A Use Case diagram is a graphical representation of use cases and actors and how they interact.

Use cases can be derived from the identification of tasks of the actor [36]. From the digital camera's extended requirements, two actors are identified, namely, a *user*, and a *PC*, and seven use cases are manifest in the requirements statement. Figure 6.7 portrays the Use Case diagram of the digital camera system that is derived directly from the extended requirements. An extension point in a use case merely allows a functionality of another use case to be summoned at a specified location.

From this *initial* Use Case diagram, each individual use case, and its respective actor, gets further elaborated to produce a more detailed requirements document which narrows the focus down from the system to the use case level. Details and formats of this derivative requirements document vary from one organization to another, but normally include information such as pre-/post-conditions, and the main and alternative flow, as demonstrated below for the *Take a picture* use case.

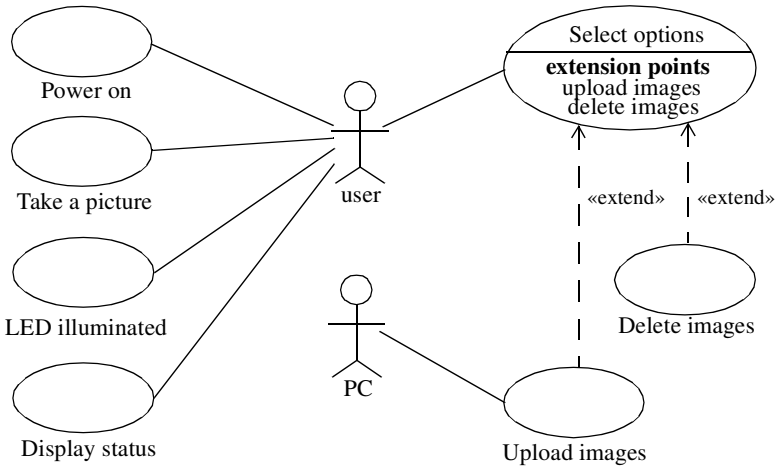


Figure 6.7: Initial Use Case diagram as derived directly from the digital camera's extended requirements

Use Case Document 1: Take a picture

Use Case: Take a picture

Description: This use case allows the user to take a picture using the digital camera

Actor: User

Pre-conditions: (PRE-1) The camera powers on without error.

(PRE-2) It is in a ready state.

Main flow: (MR-1) The use case begins when the shutter is pressed. The LED light goes off.

(MR-2) The camera system reads in the input color components. The shot-mode setting determines the number of times these color components need to be read per one press-shutter operation.

(MR-3) The camera system compresses the input color components using the baseline JPEG algorithm. The compression quality parameter is determined by the "GOOD" or "NORMAL" setting from the setting menu.

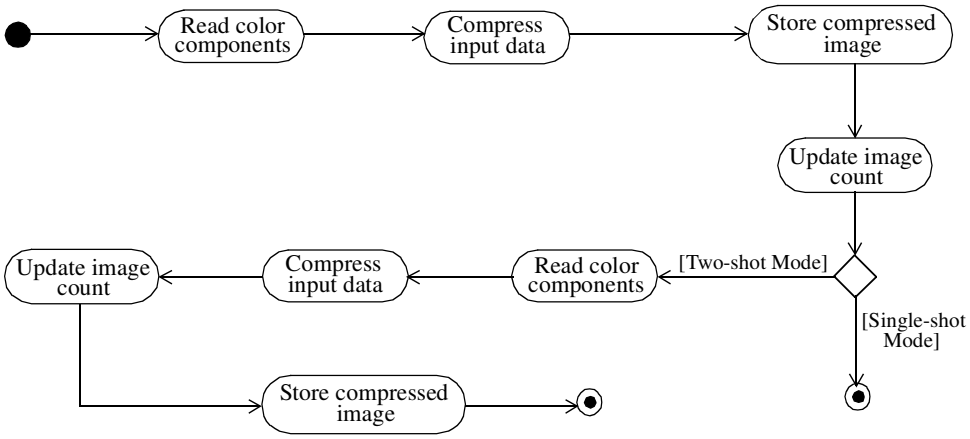


Figure 6.8: The *Take a picture* Activity diagram

(MR-4) The camera system stores the compressed image (or images) in the memory. It also increments and displays an image count. The LED light comes back on.

Alternative Flow: No Alternative flow.

Post-conditions: (POST-1) The compressed image is stored in the memory so that it can later be retrieved/deleted.

(POST-2) The number of stored images is updated.

(POST-3) The system state is unchanged.

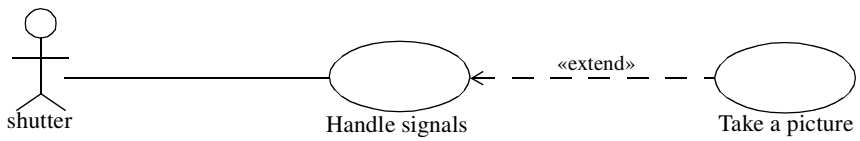
(POST-4) The shutter comes back to its non-pressed state and ready to be pressed again.

This *Take a picture* Use Case document can further be analyzed and captured using the Activity diagram in order to delineate the logical action states of the main and the alternative flow—the task which could prove helpful in eliciting more requirements. The resultant Activity diagram could also be used as a reference for the Class Analysis task. From the *Take a picture* requirements document, the following action states are educed: Read color components, Compress input data, Store compressed image, and Update image count. Figure 6.8 portrays the *Take a picture* Activity diagram.

By iteratively performing such analyses for all the initial use cases, additional use cases manifest that can contribute to a more profound insight about the system. In so doing, the developer may proceed:

- depth-first, until no additional requirements can be derived for a particular use case, at which point the developer moves on to a new use case, or
- breadth-first, where the use cases are discovered layer by layer, or
- alternately between Use Case and Class analyses, treating the two analysis tasks as concurrent processes that are supportive of one another.

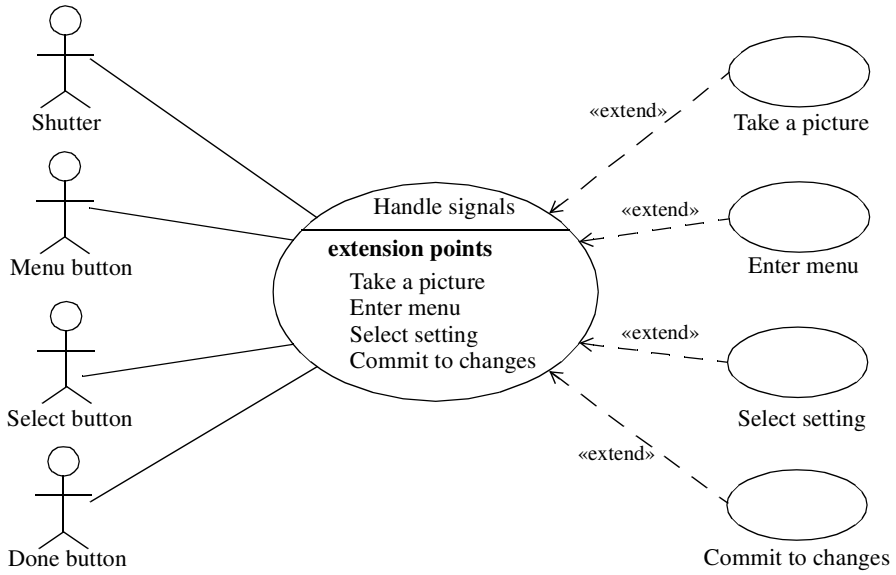
When developing an embedded system within the platform-centric (or generally, *codesign*) environment, the Use Case diagram that captures the extended requirements typically will reveal most of the peripheral components to the developer. The subsequent Use Case analysis on a peripheral-related use case tends to uncover a communication interface between software and hardware components in the system. Consider the Use Case analysis of the *Take a picture* requirements document. It is apparent from the flow statements that the only actor for this use case model is the *shutter*. Then, by looking at the system from the actor's (shutter) point of view, it is easy to conceive that none of the activities being described in the requirements document can possibly be done directly by the actor. A new use case is required that handles the stimulus generated by the actor (see MR-1), and that behaves as a proxy to the other activities in the requirements document, as depicted below. The **Use Case Document 2** that follows details the requirements for the *Handle signals* use case.



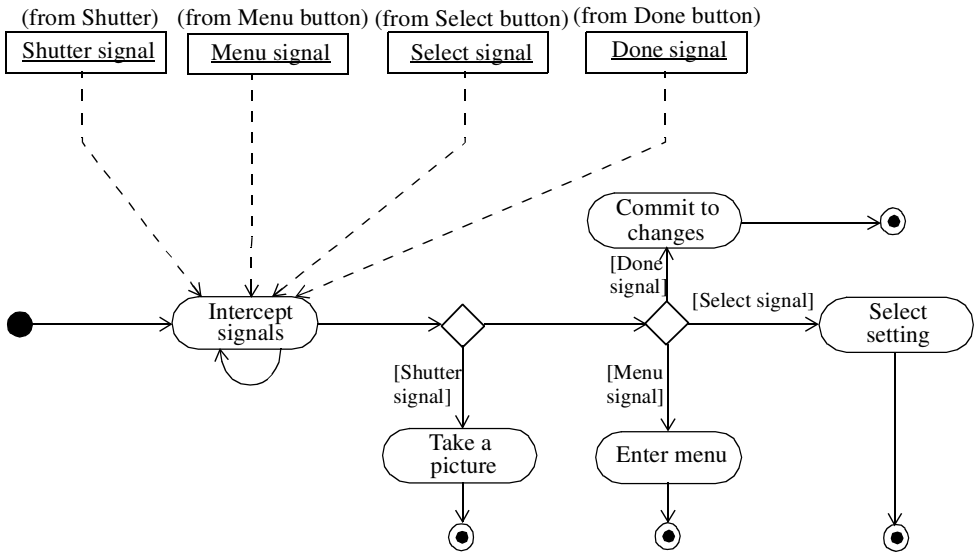
Use Case Document 2: Handle signals

Use Case:	Handle signals
Description:	This use case intercepts the shutter signal and activates the system to take a picture
Actor:	Shutter
Pre-conditions:	(PRE-1) The camera system is ready to accept a signal from the actor. (PRE-2) The actor is ready to send a signal (PRE-3) All pre-conditions in the *Take a picture* use case apply, i.e. the power is on safely and the system is in the ready state.
Main flow:	(MR-1) The use case begins when the shutter is pressed. (MR-2) A signal is sent to the camera system. (MR-3) The system intercepts the signal, examines it. (MR-4) [At the extension point] the system begins taking a picture once it is certain that the signal comes from the shutter.
Alternative Flow:	(AR-1) If the signal does not come from the shutter, appropriate extended use case is called.
Post-conditions:	(POST-1) The compressed image is stored in the memory so that it can later be retrieved/deleted. (POST-2) The number of stored images is updated. (POST-3) The system state is unchanged. (POST-4) The shutter comes back to its non-pressed state and ready to be pressed again.

It is also evident from the Alternative Flow statement AR-1 that there exist other use cases that interact with the Handle signals use case as well. As the matter of fact, upon completion of the Use Case analysis, the Use Case diagram, and the respective Activity diagram will appear as depicted in Figure 6.9.



(a) The *Handle signals* Use Case diagram



(b) The *Handle signals* Activity diagram

Figure 6.9: Derived from the *Handle signals* requirements document, (a) the eventual Use Case diagram, and (b) the Activity diagram.

6.3.2 Class Analysis

Although this dissertation presents the Class analysis task after the Use Case analysis, such an order is not necessarily true in practice. Depending on the practicality of how the requirements document enters the Requirements Specification domain, the developer may perform the Class analysis first, or choose to run both analysis tasks simultaneously.

A handful of techniques exist in books, papers and the Internet, that can contribute to the Class analysis process. Douglass [98], in particular, treats this ever-mystical topic comprehensively and in great detail. A few other useful books dealing with this subject matter include, but not limited to, those by Graham [123], Douglass [28], Maciaszek [36], and Fowler [25].

Where the Use Case analysis derives the platform-independent specification as it is outwardly visible to the actors, the Class analysis task does just the opposite: It attempts to derive the specification as it is seen internally. The Class analysis task presented herein employs the Noun analysis technique to extract candidate classes from the requirements document. UML diagrams, especially the Sequence and the Use Case diagrams, and Class analysis techniques such as the Abbott's textual analysis [124] and the Class-Responsibility-Collaboration (CRC) Cards technique, which has been developed by Beck and Cunningham (see http://c2.com/doc/oopsla89/paper.html for the original paper, and/or a book by Wirfs-Brock, Wilkerson, and Wiener [125] for further details) can be utilized to help identify class attributes and methods, as well as relationships among the candidate classes.

The guideline to the Abbott's textual analysis, as presented in [123] is summarized in Table 6.2. The Noun analysis simply is a subset of this technique that concerns only with improper nouns that infer candidate classes. By grouping candidate classes with respect to some criteria, the developer can break down the problem into smaller chunks in a divide-and-conquer manner, which can be handled more easily and effectively when also applying other techniques to help identify class attributes and methods, as well as potential relationships among candidate classes.

6.3.2.1 Noun Analysis/Textual Analysis

The noun analysis refers to the process of selecting *improper nouns* from the requirements document that can potentially be characterized as a class in the UML model. Performing such a task as part of the iterative analysis process to derive the platform-independent specification, this dissertation analyzes the nouns in the extended requirements document, before focusing on each individual use case, and proceeding in a depth-first manner that eventually produces a complete use-case-centric Class diagram as the result. The system's Class diagram is derived by merging all of the use-case-centric Class diagrams together, performing additional analyses only where necessary during the merge. In the extended requirements and Take a picture requirements, the candidate nouns are underscored.

Table 6.2: Guidelines for the Abbott's textual analysis.

Part of Speech	UML Model Component	Example
proper noun	instance	the EP20K200EFC484-2X PLD device
improper noun	class/type/role	push-button switch
doing verb	operation	start
being verb	classification	is a
having verb	composition	has a
stative verb	invariance condition	own
modal verb	data semantics, pre-condition, post-condition, or invariance condition	must be
adjective	attribute value or class	good, normal
adjectival phrase	association, operation	back in the menu mode
transitive verb	operation	enter
intransitive verb	exception or event	goes off, commit to

In order to traverse depth-first, consider again the *Take a picture* requirements. By grouping the candidate nouns from this document and the extended requirements, basing the grouping criteria on the *Take a picture* use case, the following candidate classes are identified:

- camera system
- LED light
- color components
- display
- [baseline] JPEG
- shutter
- compressed image
- memory
- [take a picture] operation
- user

By concentrating on these nouns (classes), and doing the Abbott's textual analysis on them within the confines of the *Take a picture* use case, additional class characteristics are acquired as shown below:

Text	**Functions**	**Potentially owned by**	**Potentially associated with**
image count	attribute	camera system; memory	--N/A--
image quality setting	attribute	display; camera system; operation; JPEG	--N/A--
shot mode	attribute	display; camera system; operation; JPEG	--N/A--
ready message	attribute	display	--N/A--
store	operation	memory	compressed image
press	operation	shutter	user
increment	operation	memory; camera system	operation
read (input)	operation	color components	operation; JPEG
read shot mode setting	operation	display; camera system; operation; JPEG	JPEG
read image quality setting	operation	display; camera system; operation; JPEG	JPEG
display	operation	display	camera system; memory; operation
on/off	operation	LED light	camera system; operation
take a picture	operation	operation	--All Classes--

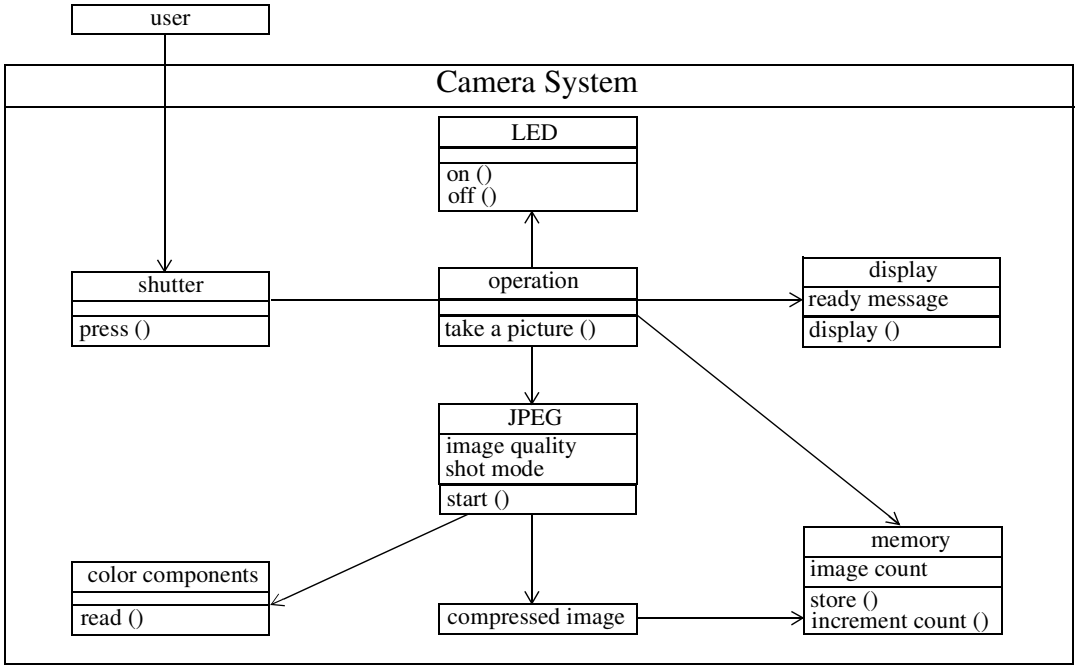


Figure 6.10: Preliminary use-case-centric Class diagram derived from the Noun Analysis/Textual Analysis

Although not evident from the textual analysis, it is fairly intuitive to perceive the *camera system* class as maintaining an *ownership* relationship with all other classes in the system—hence, inferring *composition* in the Class model. Figure 6.10 depicts the early development of the Class diagram as derived from the Noun analysis/Abbott's textual analysis above.

6.3.2.2 Code Reuse

Advanced knowledge of the NiOS platform utilization requirement gives the developer a certain degree of prescience that could impact the decision making process. Surfacing almost naturally is the choice of using the C programming language and/or NiOS macro code to develop the application software, as well as the opportunity to reuse the public-domain JPEG library (in C) to expedite the development process.

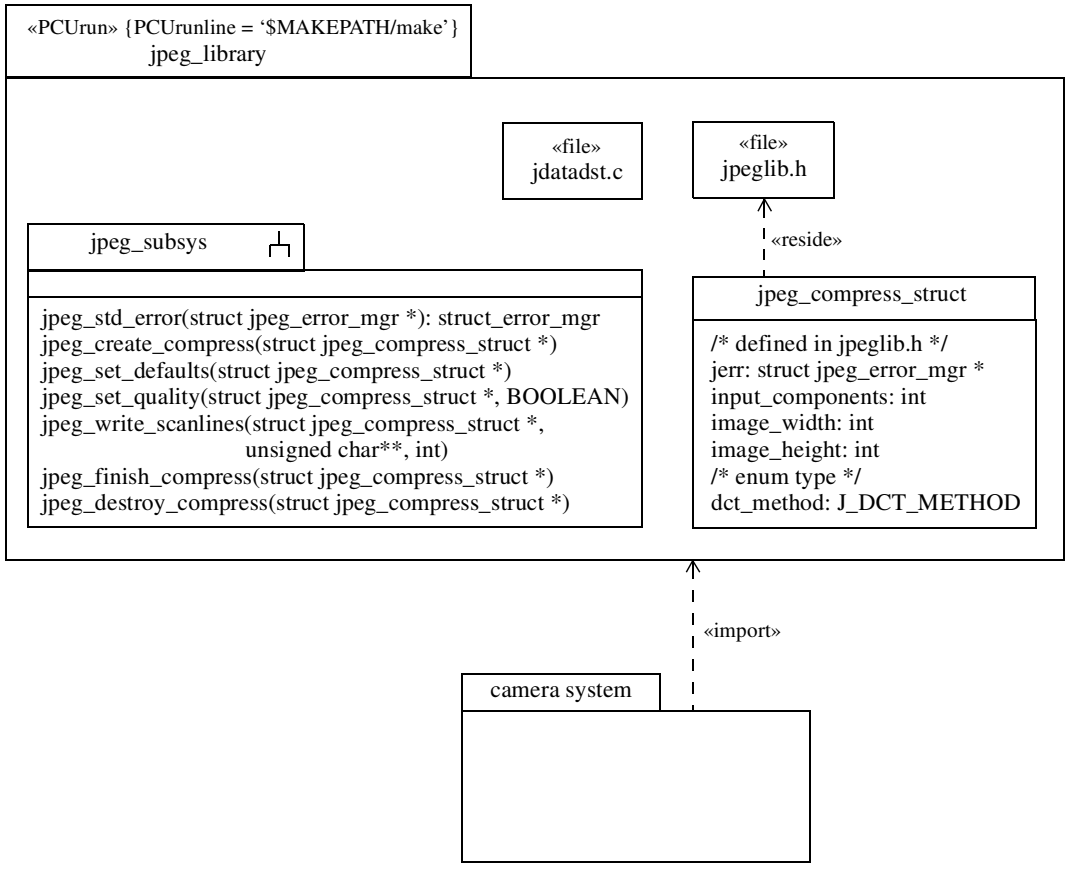


Figure 6.11: UML-encapsulated JPEG library. The figure shows the subsystem package that provides a functional interface to the required library functions.

The developer who is familiar with the JPEG standard would likely know also that its success has come substantially from the courtesy of the Independent JPEG Group (IJG) for their effort in distributing free and highly portable C-code implementation of the algorithm. The platform-centric approach envisages the C JPEG library as legacy software that could be encapsulated using the UML notations as shown in Figure 6.11. The embellished «PCUrun» stereotype specifies the Makefile to be executed—permitting the inclusion of the library at the modifiable source code level rather than at the stationary archive level (e.g. *jpeglib.a*).

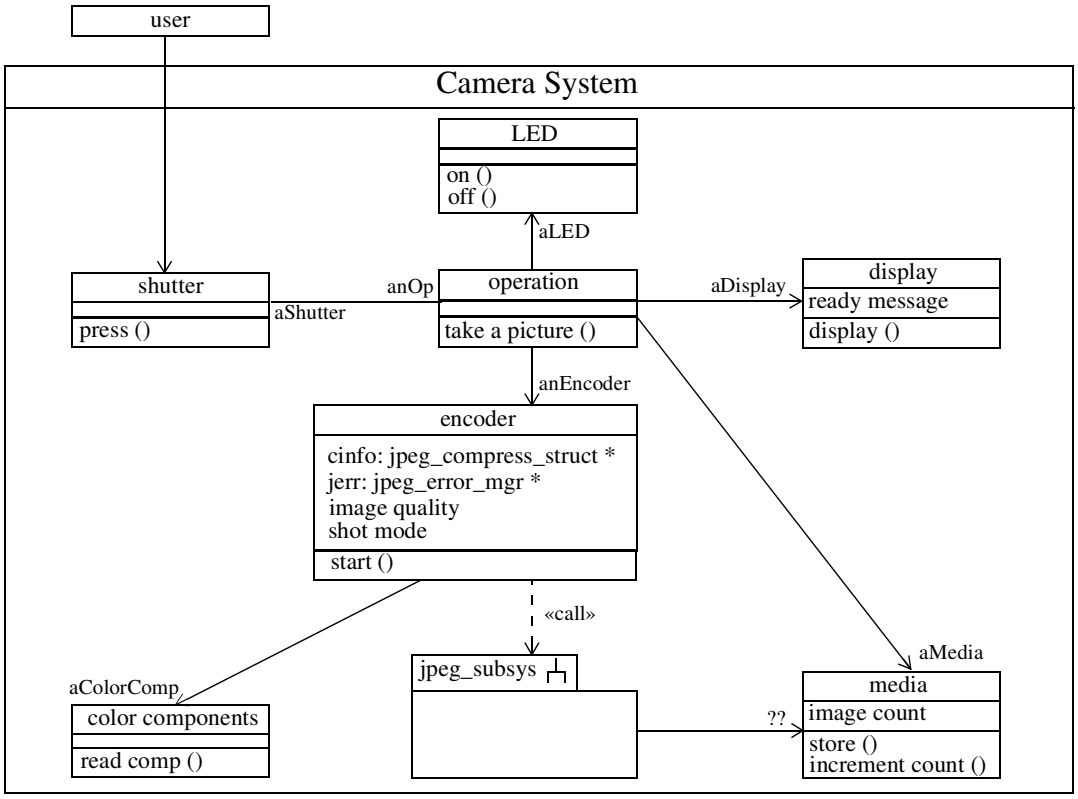


Figure 6.12: UseCase-centric Class diagram utilizing the IJG's JPEG library package

The IJG's JPEG library imposes further requirements that affect the format and choice of the input color components, as well as the structure of the compression engine that includes the image quality attribute. The compressed image class identified earlier can be implemented by the C structure jpeg_compress_struct. The memory class, renamed media to reflect the non-volatile nature of the memory, will be used from either the library or the camera system packages. The image quality attribute corresponds to the compression quality concept and can be set by calling the jpeg_set_quality() function. Figure 6.12 depicts the Class diagram, which presently includes the services from the JPEG library package. In the figure, the JPEG class is also renamed encoder to reflect the fact that it only copes with the forward operation of the JPEG standard.

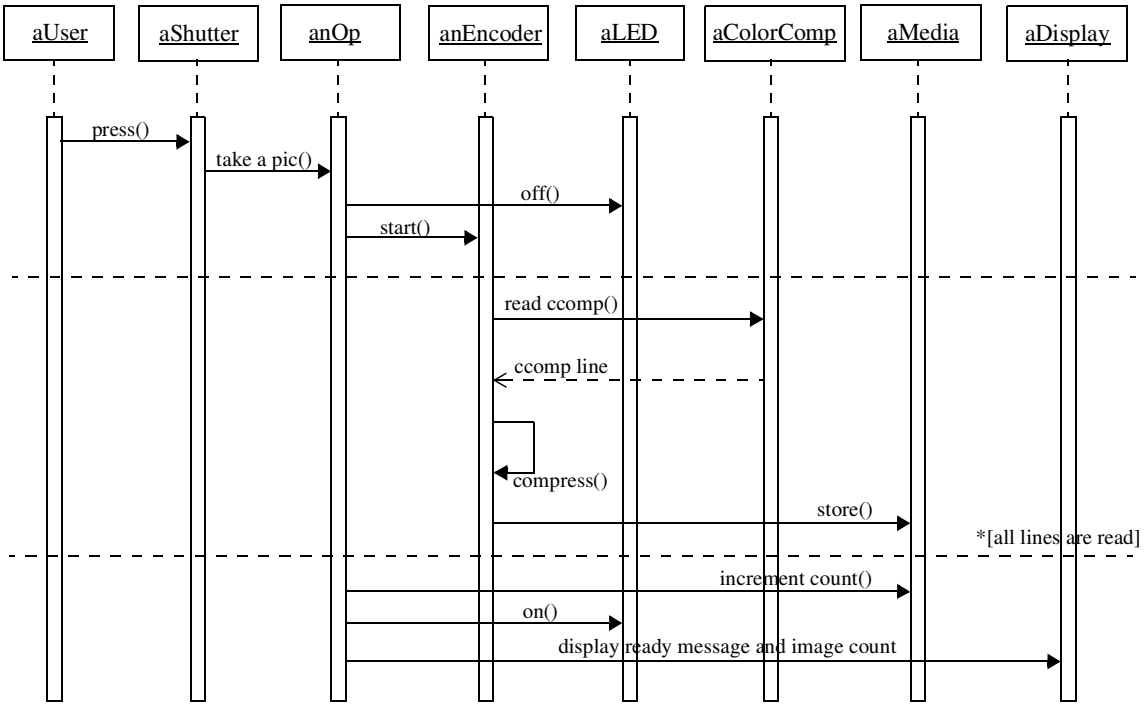


Figure 6.13: Sequence diagram describing the main scenario for Figure 6.12.

By identifying roles and responsibilities of the classes in Figure 6.12, and then modeling their interactions using the interaction diagrams, that comprise the Sequence and the Collaboration model, the developer can potentially elicit additional information which could manifest itself as class attributes or operations, or spawn new requirements that merit their own Use Case and/or Class analysis.

The Sequence diagram in Figure 6.13 portrays a scenario that describes the principal collaboration of the classes in Figure 6.12. As deducibly evident from the figure, an awkward relationship between the shutter and the operation class necessitates a closer scrutiny. The thorough analysis of their relationship would likely result in an identification of an interface class that is perennially active, waiting for the shutter signal, and that is also responsible for furnishing a proper response to the intercepted signal—the idea resembling the interrupt handling mechanism concept as depicted in Figure 6.9.

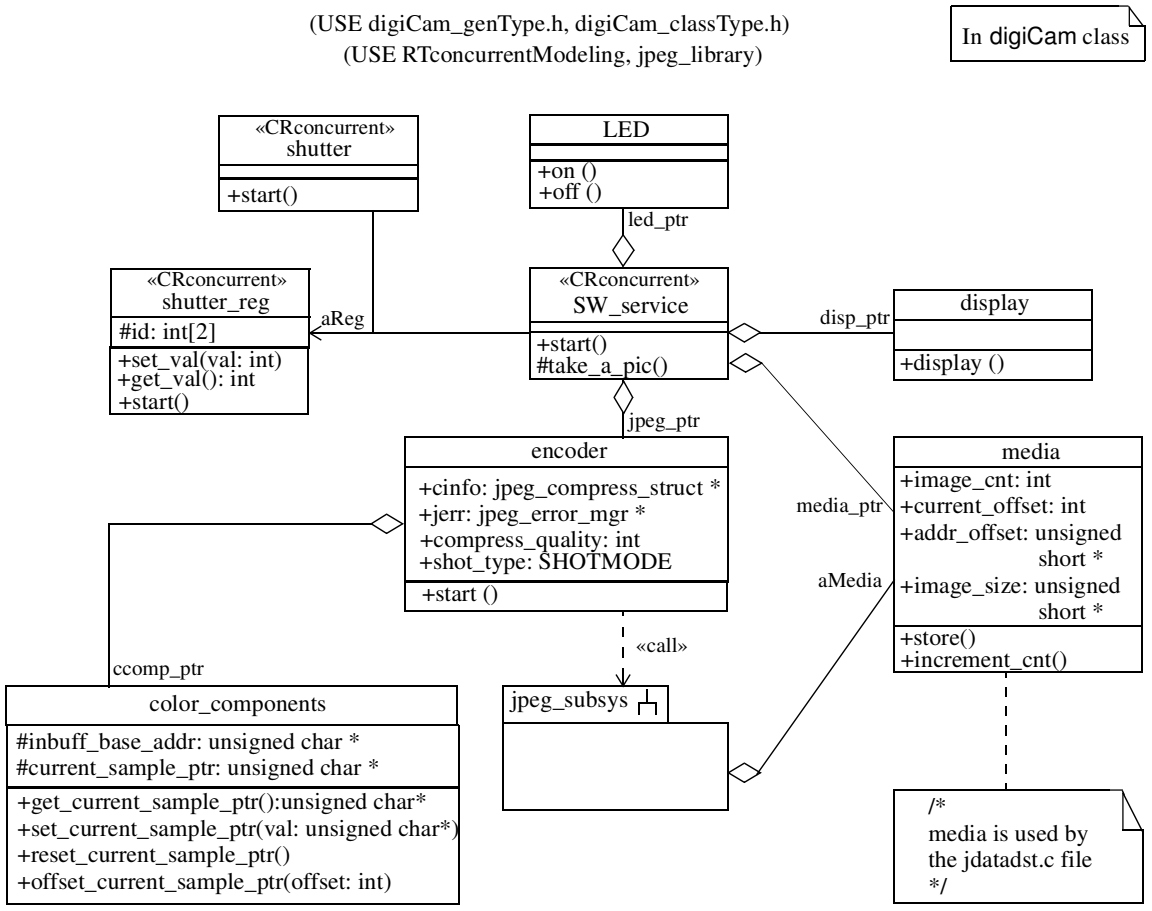


Figure 6.14: Detailed Class diagram for the *Take a picture* use case

The read comp (), which returns an input scan line (ccomp line), the compress (), and the store () functions all come from the library, and must be renamed accordingly. The IJG's JPEG library also already specifies for the developer the scan-line format. Information such as the need to keep track of the stored image locations and the current memory location can become more obvious via the diagrammatic representation. Likewise, the behaviors that may raise doubt, e.g. "Ought the increment count () operation to be called by the operation class or by the media class?", or "Does the media class actually play two distinct roles?", tend to become more distinguishable, graphically, as well. Figure 6.14 illustrates the detailed resultant Class diagram of the *Take a picture* use case.

A few notable changes are evident in Figure 6.14. The operation class is more properly renamed SW_service (SW for *switch*) and the camera system class now becomes the digiCam class that possesses the main function. The digiCam employs the concurrency modeling facility (the RTconcurrentModeling package [29]) to capture the concurrent characteristics of the shutter, shutter_reg, and SW_service classes. The press() function is replaced with the start() function that internally simulates the *press* action as coming from the standard input, i.e. a keyboard. The Sequence diagram in Figure 6.15 delineates the detailed behaviors of these concurrent classes, as seen by a controller class that is not shown here in the figure. The C-like pseudocode of the start() functions appears below.

Pseudocode of the start() function

The shutter_reg class

```
void start():
    pipe(id);
```

The shutter class

```
void start():
    if(fork == 0):
        for(;;):
            scanf(shutter_signal_id);
            aReg.set_val(shutter_signal_id);
```

The SW_service class

```
void start():
    for(;;):
        aReg.get_val();
        if (shutter_signal_id)
            take_a_pic();
```

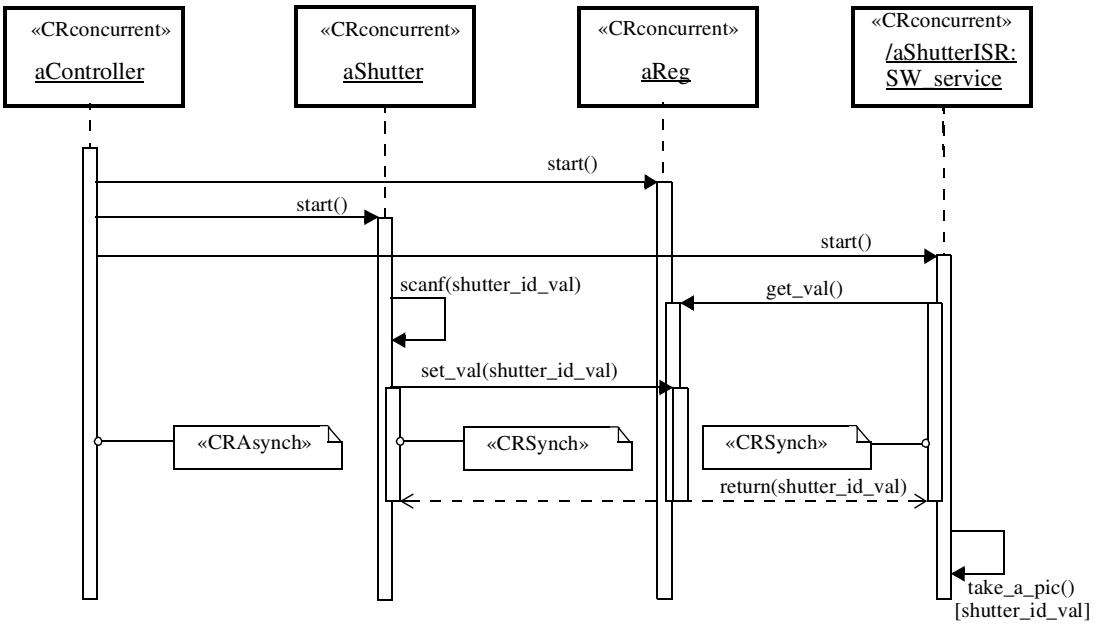


Figure 6.15: Sequence diagram describing the action that leads to an activation of the take_a_pic() function.

6.3.3 Concluding Remarks

To derive the platform-independent specification for the simplified digital camera system, this dissertation performs the Use Case and Class analyses in a cyclical manner, resulting in an iterative refinement process that yields the desirable specification as the outcome.

As priorly demonstrated, the adopted requirements analysis approach commences by identifying the primary use cases and classes from the initial requirements document. Then, for each primary use case, it determines the relevant classes, including the primary classes already identified, before proceeding with the Use Case and Class analyses, in a depth-first manner, until the final UseCase-centric Class diagram results. The platform-independent specification is derived by merging all such Class diagrams together, while performing further analyses as necessary.

Figure 6.16 depicts the space-optimal Class diagram that, along with the Sequence and the State diagrams, as well as the supplemental requirements document, constitutes the platform-independent specification. The detailed specifications document can be found in Appendix E.

Although not shown explicitly, all the classes in Figure 6.16 belong to the digiCam class (via the *composition*). The SW_shutter, SW_menu, SW_select, and SW_done are the generalized classes of the SW_pushButton, all of which share the same device register (SW_reg) that allows the SW_service class to intercept the incoming *press* signal, and call the appropriate service routine. The display class is also renamed mssg_service to be more specific about its characteristics. Another notable addition is the utilization of the EMprofile package (see Chapter 5, Section 5.3 for details), which is discovered during the *Power-up* Use Case analysis. For simplicity, this camera only checks the system readiness by querying the existence of all anticipated peripherals—the operation that is performed by the peripherals_checkup class during power-up.

Mindful readers might notice that the platform-independent specification presented herein perceives the system development process as encompassing both hardware and software, permitting the hardware components and architecture to be *configured* as well as *designed*. This is to contrast with the current UML modeling practice where hardware is often regarded as being external to the development process and the hardware components and architecture can only be *configured* to model the relationships between hardware the software system under development. The specification is also reflective of the anticipated hardware/software system structure and behaviors, the benefits of which could be twofold:

- It provides for the developer the precise functional specification of the hardware components expected to be acquired, configured or designed for the system, and
- It eases the task of transitioning from the platform-independent specification to the HW/SW specific platform-dependent specification, as shall be seen later in this chapter.

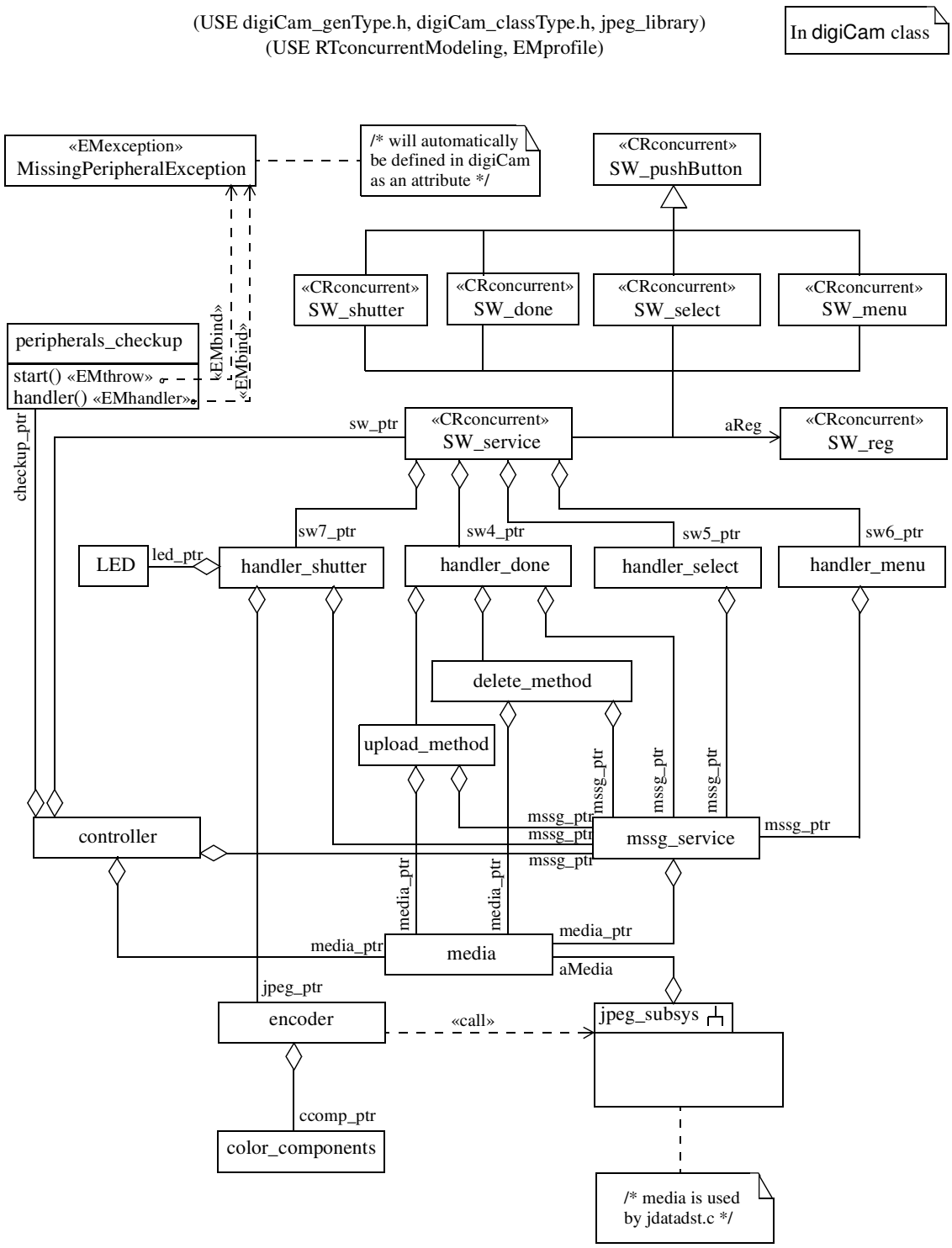


Figure 6.16: Detail-minimal Class diagram for the digital camera system

Thus far this section has covered a lot of ground demonstrating, for the first stage of the proposed design flow as depicted in Figure 6.5, how the platform-centric SoC design method utilizes the cross-disciplinary techniques in Software and Requirements engineering to derive the platform-independent specification from the initial requirements document. To conclude, this dissertation presents the supplemental requirements document as follows.

Digital Camera's Supplemental Requirements

(**NF-R1**) Let *timeA* be the time when the shutter is pressed, and *timeB* be the time when the camera is ready to take another picture. Then it is required that when taking a picture of NORMAL quality:

timeB - *timeA* $\leq$ 1 second,

for at least 95 percent of the time.

Note: The number of input samples, and effectively the image sensor capacity, is still unknown and must be determined.

(**NF-R2**) Faster is better.

(**NF-R3**) The final product will operate on battery, as such it should operate with as little power consumption as practically possible.

(**NF-R4**) No fan allowed.

(**NF-R5**) Must use the NiOS soft core, and the EP20K200EFC484-2X APEX20KE device

(**NF-R6**) To be competitive, a throughput of 1 Mega-samples/second, corresponding to a 640x480-pixel, color image sensor, is expected.

(**NF-R7**) [*From the Take a picture Use Case*] To be operational, the power must be on and remains on, and all required peripherals must exist and functional, i.e.

```
context digiCam:
    inv: (self.current_state = STATE_READY or
          self.current_state = STATE_MENU) implies
          self.power_state = ON
    pre:  self.current_state <> STATE_ERROR
```

6.4 Platform Analysis

The principal task in this stage involves identifying the platform from within the LPO that is best suited the requirements represented by the platform-independent specification. Such a task is non-trivial. In Figure 6.5, this task is illustrated as an iterative process involving the search and exploration of available platforms in the LPO, as well as the hardware/software partitioning analysis of the platform-independent specification per each candidate platform. In most hardware/software SoC design approaches, where there exists no explicit support for the derivation process of the platform-independent specification, the equivalent of this stage is where the process flow commences, whose input is the requirements specification[1] which has been derived elsewhere.

The *search and explore* task requires the developer to interact extensively with the LPO via the Platform-Object Manager (POM) interface software. A simple POM, like the one shown in Figure 6.17, relies on XSL/XSLT to format the LPO's XML database and presents it to the developer in a familiar HTML format. The *Summary* section provides the developer with quick cues as per the characteristics of each platform members (POmm/components and POmm/tools). While the *Search* operation permits non-LPO searches, an operation such as the XPath-based *Query/Filter* helps the developer zero in on a certain aspect of the LPO database, making it useful for the task at this stage. To simulate the work environment anticipated by the POM, the DTD schema documents and the XML register files have been placed in different machines, accessible only through the Internet. The information applicable to all POmm/components and POmm/tools is amassed from the actual Nios components and tools, furnished by their respective providers. Due to time restriction, only a simple POM is developed for the case study. It is envisaged, however, that a more sophisticated version of the POM software, possibly the one that integrates itself seamlessly into the platform-centric UML CASE tool, would contribute favorably to the efficiency of the proposed approach.

1. Requirements specification is a generic term referring to the same concept as the platform-independent specification, which is a platform-centric jargon.

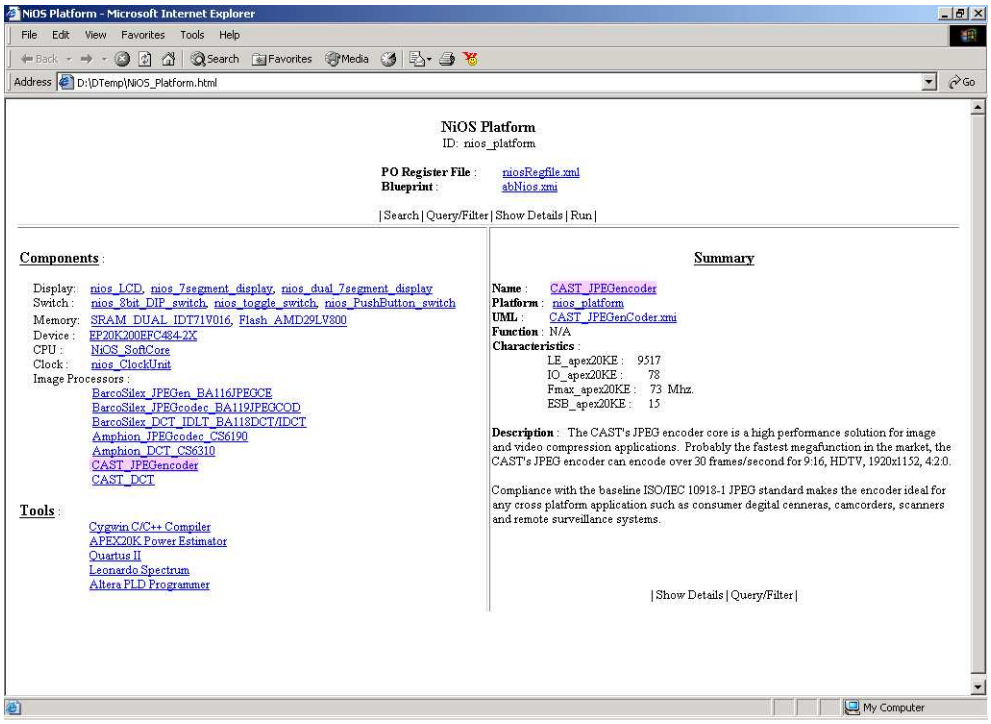


Figure 6.17: A simple POM interface window portraying the NiOS platform, with a short summary on the JPEG Encoder module provided by CAST Inc. (http://www.cast-inc.com)

For this particular case study, actual work is curtailed owing largely to the specific requirements for the NiOS platform and the EP20K200EFC484-2X PLD device. In a more typical scenario, the developer would have to explore the available choices of platform, and assess and compare the feasibility of using each one of them for the problem at hand. This process can be automated using CAD tools. It can also be achieved manually, or by utilizing both approaches together. The use of platforms helps to allay the difficulty of such a task by placing well-defined constraints, i.e. the platforms and their respective components, onto the design space—contracting the sheer size of it as a result. Anyhow the developer's experiences still contribute considerably to the success of this task. Figure 6.18 depicts the architecture blueprint of the Altera's NiOS platform system, where the constraints and enumeration types are presented in Figure 6.19.

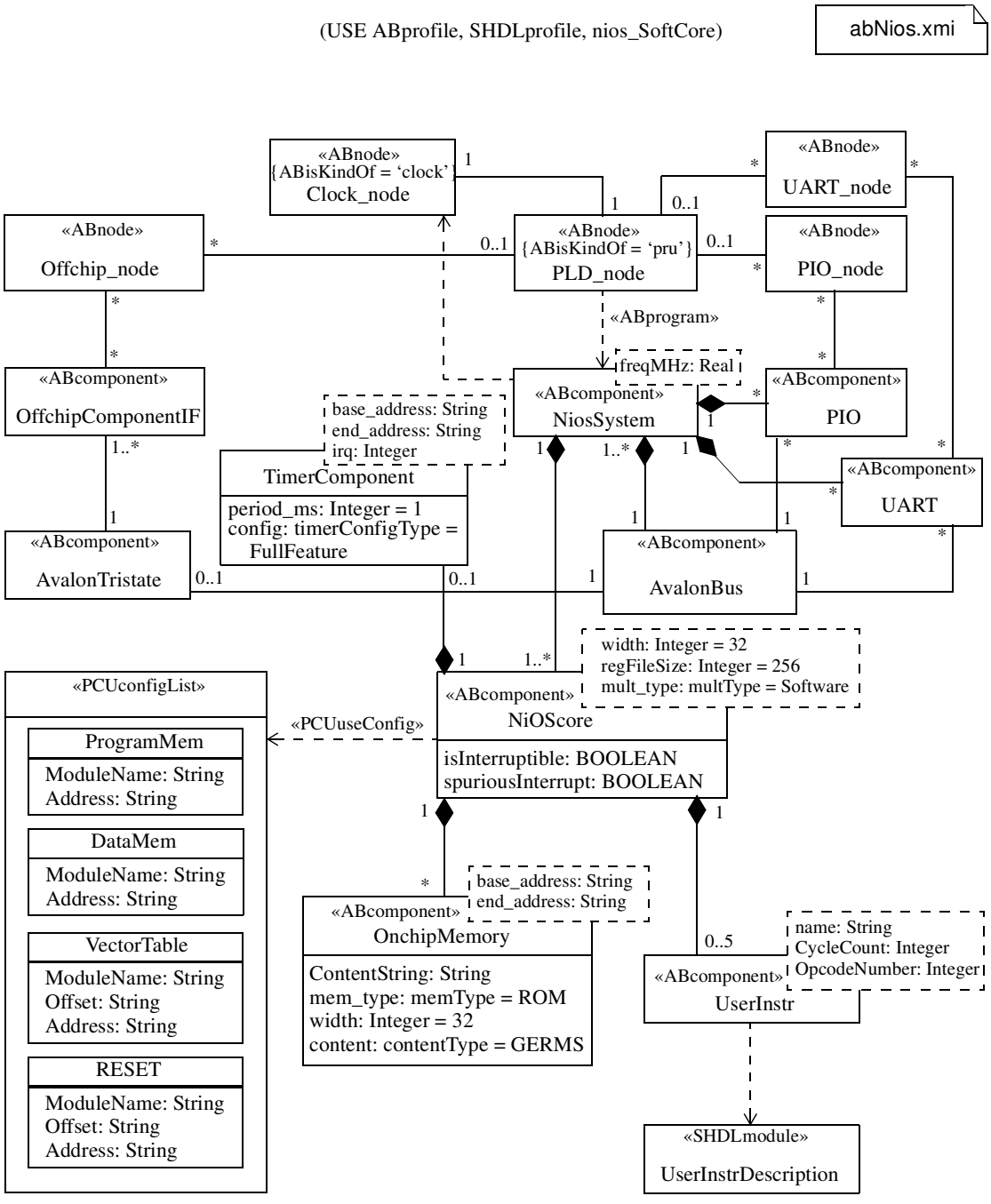


Figure 6.18: The Architecture Blueprint of the NiOS platform (*abNios.xmi*), depicting the platform structure

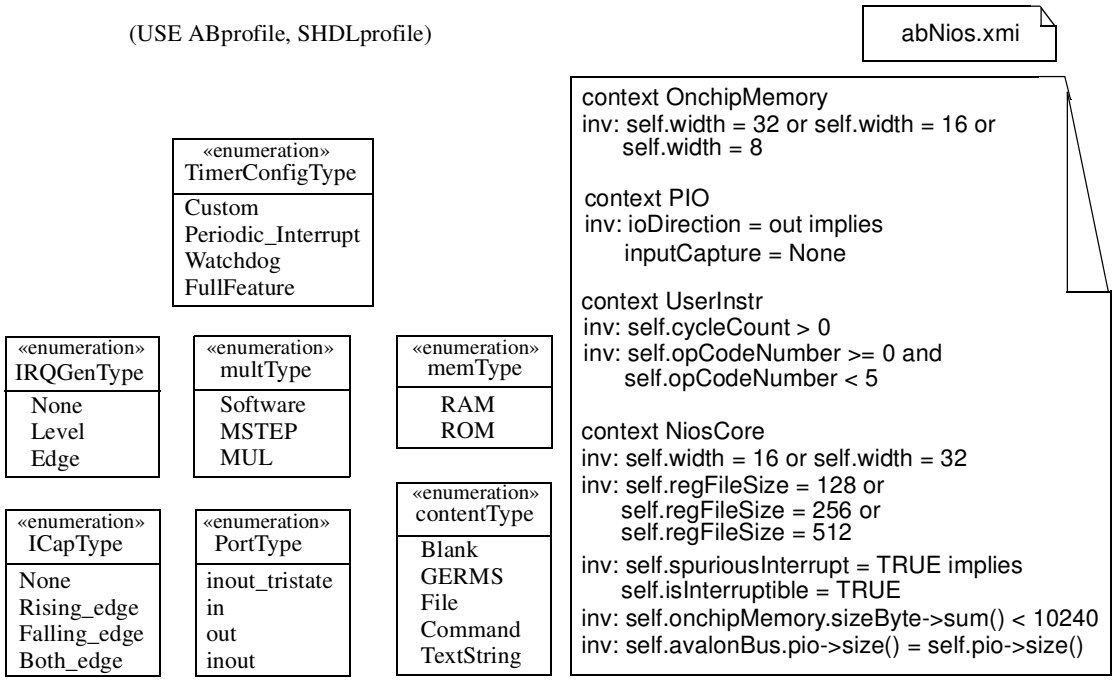


Figure 6.19: The Architecture Blueprint of the NiOS platform (*abNios.xmi*), depicting a partial list of constraints and enumerated types

6.4.1 Automated Architecture Selection and System Partition

Algorithms exist that can assist the system developer in partitioning the system as well as in selecting the target architecture. Axelsson provides a comprehensive overview on a number of such algorithms in his thesis [23], the detailed treatment on some of which can be found in [91, 92, 93, 94].

Architecture selection and system partition algorithms utilize pre-characterized metrics from the platform-independent specification and from the candidate hardware components to determine the feasible target architecture and to partition the specification into hardware and software domains that can be mapped onto the architecture. Such an automated task is very computationally expensive, and asserts no guarantee that the optimal solution will ever be attained, especially where the design space is large and complex. Such shortcomings afflict the usefulness of the automated approach tremendously.

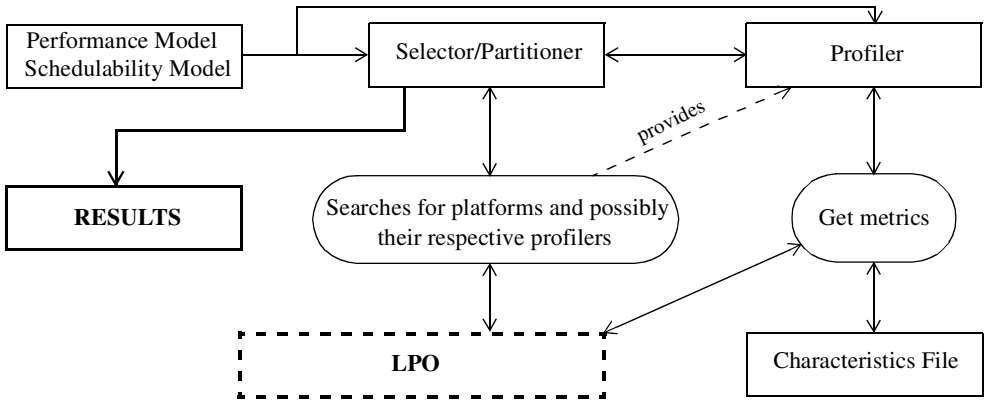


Figure 6.20: Generic usage model for the automated architecture selection and /or system partition algorithms

The platform-centric SoC design method specifies a general usage model for the automated architecture selector and/or system partitioner as illustrated in Figure 6.20. In the figure, the selector/partitioner feeds off the performance and/or schedulability model (see UML Real-Time Profile for details [29]) that identifies the desired system objectives for the algorithms. If profiling has to be done on the model, the selector/partitioner either searches the LPO and calls the applicable profiler for each platform or, in a rare case, runs the model against its own profiler to acquire the metrics necessary for the algorithms. Common profiling parameters for candidate platform components, such as throughput or execution time, may be read off the characteristics section of its XML data file, and/or obtained from a dedicated characteristics file maintained elsewhere, possibly as part of the customer support by the selector/partitioner vendors. Once all the required metrics are determined, the selector/partitioner begins its long voyage that usually only ends when certain criteria are met, rather than when the optimal solution is found. Figure 6.21 depicts the performance model for the scenario where the camera is used to take a picture under a *Single-Shot, Normal Quality* setting. The figure shows how the 1 frame per second requirement, Requirement (NF-R1) in the Supplemental Requirements Document, can be captured. For simplicity, this Sequence model for the JPEG encoding algorithm is not detail-accurate, especially with respect to the IJG implementation.

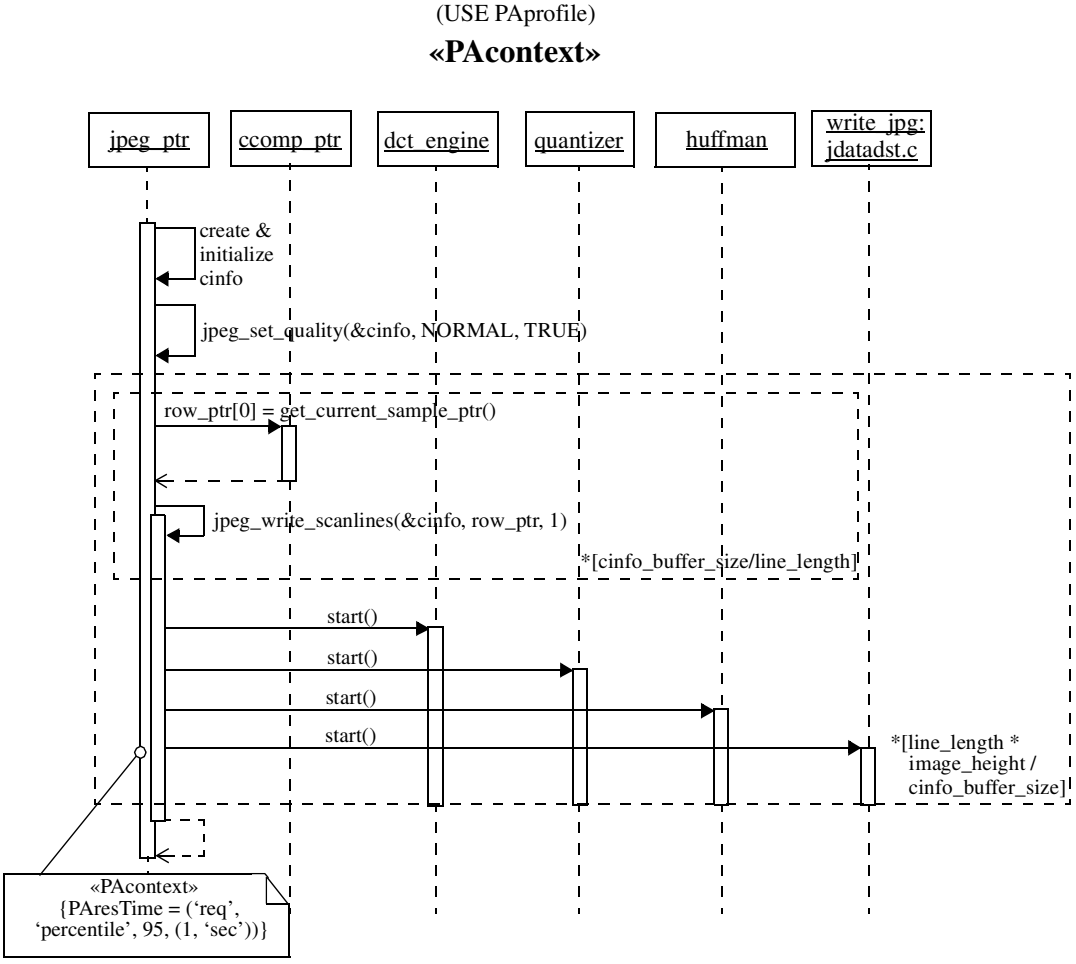


Figure 6.21: Performance model as specified in the UML Real-Time Profile. The figure describes Requirement NF-R1 from the Supplemental Requirements Document.

6.4.2 Manual Approach to Selecting Target Architecture

In optimization-based computations found in many different research areas, especially the Operations Research, the initial value plays a crucial role in determining how quickly the algorithm converges. In a platform-based codesign problem, the platform represents a pre-determined, and configurable initial values set for a specific problem space. It permits rapid convergence or, *divergence*, if no solution can be found; whereas, the pre-targeted problem space associated with it determines the best-attainable solution for the problem.

For this case study, the inclusion of the Altera's NiOS platform has already been mandated. To get preliminary insight as per how the NiOS system should be configured, the developer may perform a simple profiling on the platform-independent specification, specifically the tasks related to the *Take a picture* use case for it is where the timing requirement is placed. Table 6.3 shows the JPEG profiling results (modified from *http://www.ececs.uc.edu/~ddel/projects/dss/asap/node2.html*); while, Chen, et. al. [126] reports similar figures for both Intel PIII 650 MHz, and NiOS 33 MHz (with Software Multiply) platforms. They also present the profiling results of the data handling stage, that consumes approximately 1.72%, and 1.22% of the total execution time (encoding) for the PIII and NiOS platforms, respectively.

Table 6.3: Software profiling data on the JPEG algorithm

JPEG Stages	Percentage of Overall Run Time					
	Pic. 1	**Pic. 2**	**Pic. 3**	**Pic. 4**	**Pic. 5**	**Pic. 6**
FDCT	77.91	78.85	78.85	77.71	78.12	78.29
Quantize	1.62	1.58	1.55	1.62	1.63	1.58
ZigZag	0.34	0.34	0.34	0.34	0.34	0.34
Huffman Encode	14.01	12.71	12.98	13.9	14.1	13.13

It is evident from Table 6.3 and the previous discussion that the FDCT stage is the most computationally expensive. A closer look at the IJG's JPEG library, which is now a part of the platform-independent specification, reveals the support for Fixed- or Floating-point computations, as well as the JPEG algorithms by Arai, Agui & Nakajima (AA&N, see a book by Pennebaker and Mitchell [128] for details), and by Loeffler, Ligtenberg & Moschytz (LL&M) [127]. The AA&N algorithm runs faster (80 multiplies, 464 adds per an 8x8 2D FDCT), but it is less precise when utilizing the Fixed-point computation; whereas the LL&M is more precise, doing 192 multiplies, and 512 adds for the same 8x8 2D FDCT operation. Table 6.4 details possible options for configuring and partitioning the NiOS platform and the platform-independent specification, based on the information acquired from the specification (the JPEG library), and the LPO. Table 6.5 presents the profiles for the NiOS-native multipliers, namely, Software, MSTEP, and MUL, where the MSTEP is a serial multiplier, while the MUL is the parallel implementation.

Table 6.4: System configuration options. By committing to a combination of these options, the developer acquires the target architecture, while simultaneously partitioning the platform-independent specification.

Category	Available Options
Algorithms	AA&N, LL&M
Computational Precision	Fixed-point, Floating-point
Hardware Accelerator	HW Encoder, HW FDCT, HW Multiply, None
Multiply	Software, MSTEP, MUL, Floating-point MUL (implemented using the custom-instruction)
NiOS CPU	16-bit, 32-bit

Table 6.5: Characteristics of the NiOS-native Multipliers

Multiply Type	Logic Elements	Clock cycles (16-bit CPU)	Clock cycles (32-bit CPU)
Software	0	80	250
MSTEP	~20	18	80
MUL	~400	2	16

The NiOS system compilation process typically takes about 45-90 minutes on a 256MByte RAM, 600MHz PIII notebook to generate the target system for the developer, making it fairly convenient to compile and re-compile if any of the configurations has to be adjusted. The following decisions, based on the specification and the profiling results, are made for the digital camera system:

- *32-bit CPU*: The 16-bit option might be less effective. The input data are at least 8-bit in size. After a series of additions and multiplications, it is possible that the internal data are going to be larger than 16 bits. Moreover, the 16-bit representation of the internal data might cause the precision error problem to worsen.
- *Hardware Multiply*: It is less costly compared with other hardware accelerators, making it a good starting point. Indeed, a quick look at the hardware encoders as displayed by the POM would unveil that they will not fit in the required EP20K200EFC484-2X PLD device, thus, eliminating them altogether. The *MUL* option is chosen, to accommodate the multiply-intensive computation of the FDCT (see Table 6.5).
- *Fixed-point*: It is faster than the floating-point counterpart. In addition, there exists no substantial gain in using the floating-point over the fixed-point precision.
- *LL&M*: Because the fixed-point precision is configured, the LL&M algorithm is employed to lessen the effect of the precision error.

- 4 *Push-button switches* (ID: nios_pushButton_switch), one apiece for the Shutter, Select, Menu, and Done operation.
- A LED (ID: nios_LED)
- RS232/UART (ID: nios_UART) as a communication means for uploading the images onto the PC.
- Instead of one display as previously specified, a more visible *dual seven-segment display* unit (ID: nios_dual_7segment_display) is used to show the number of stored images, while a *LCD display* (ID: nios_LCD) is employed to display text messages.
- 33.33 MHZ clock generator and distributor unit (ID: nios_clockUnit)
- *Flash memory* (ID: Flash_AMD29LVB00) for non-volatile memory, and a *SRAM* (ID: SRAM_DUAL_IDT71V016) for the executable during prototyping.

It is to note that the decision regarding the peripheral components is made in accordance with their behavioral models described in the platform-independent specification (Figure 6.16). In practice, right-first-time decisions rarely occur, and adjustments as well as fine-grained calibrations are almost unavoidable. The proposed approach helps make the execution of these tedious tasks more tolerable and efficient.

To produce the target architecture, the developer imports the architecture blueprint, and the relevant UML models of the selected peripherals into a common package, and instantiates and configures them as necessary—creating concrete objects from the imported classes. Then, the developer links these objects together in a fashion prescribed by the architecture blueprint, effectively, creating the communications, and completing the instantiation process of the target architecture. Figure 6.22 shows the UML Class diagram representing the NiOS-compliant LCD, which can be retrieved by the POM and used by the developer to configure the target architecture. Figure 6.23 then delineates the resultant target architecture for this particular system configuration.

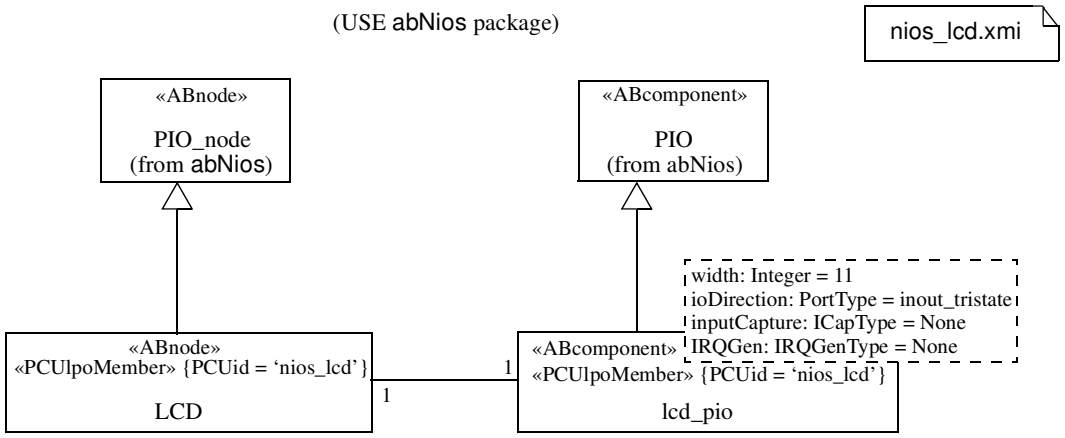


Figure 6.22: UML representation of the LCD. This UML package is accessible through the <uml> tag from within the XML file that describes the LCD (a POmm/component).

6.5 Platform-Dependent Specification and System Derivation Process

As opposed to the first two stages, the platform-dependent specification and the system derivation process do not include any iterative sub-process; however, they do traverse back and forth *mostly* between themselves in an iterative refinement manner that involves (1) the derivation of the platform-dependent specification based on the results from the preceding stages, (2) the activation of the appropriate POmm/tools in the system derivation process whose input is the platform-dependent specification which is being portrayed as different model views in Figure 6.5, and (3) the analysis of the results to further refine the specification, resulting in a loop-back, until all requirements are met. As also indicated in Figure 6.5, if all the requirements could not be attained successfully by refining the platform-dependent specification alone, more fundamental changes may be imperative and the platform-centric process flow leaves the bipartite iteration and goes back to either the platform-independent stage or the platform-analysis stage to make an appropriate modification, before proceeding again into the prescribed development process flow.

(USE abNios, nios_lcd, nios_led, nios_clockUnit, nios_UART, nios_pushButton_switch, ...)

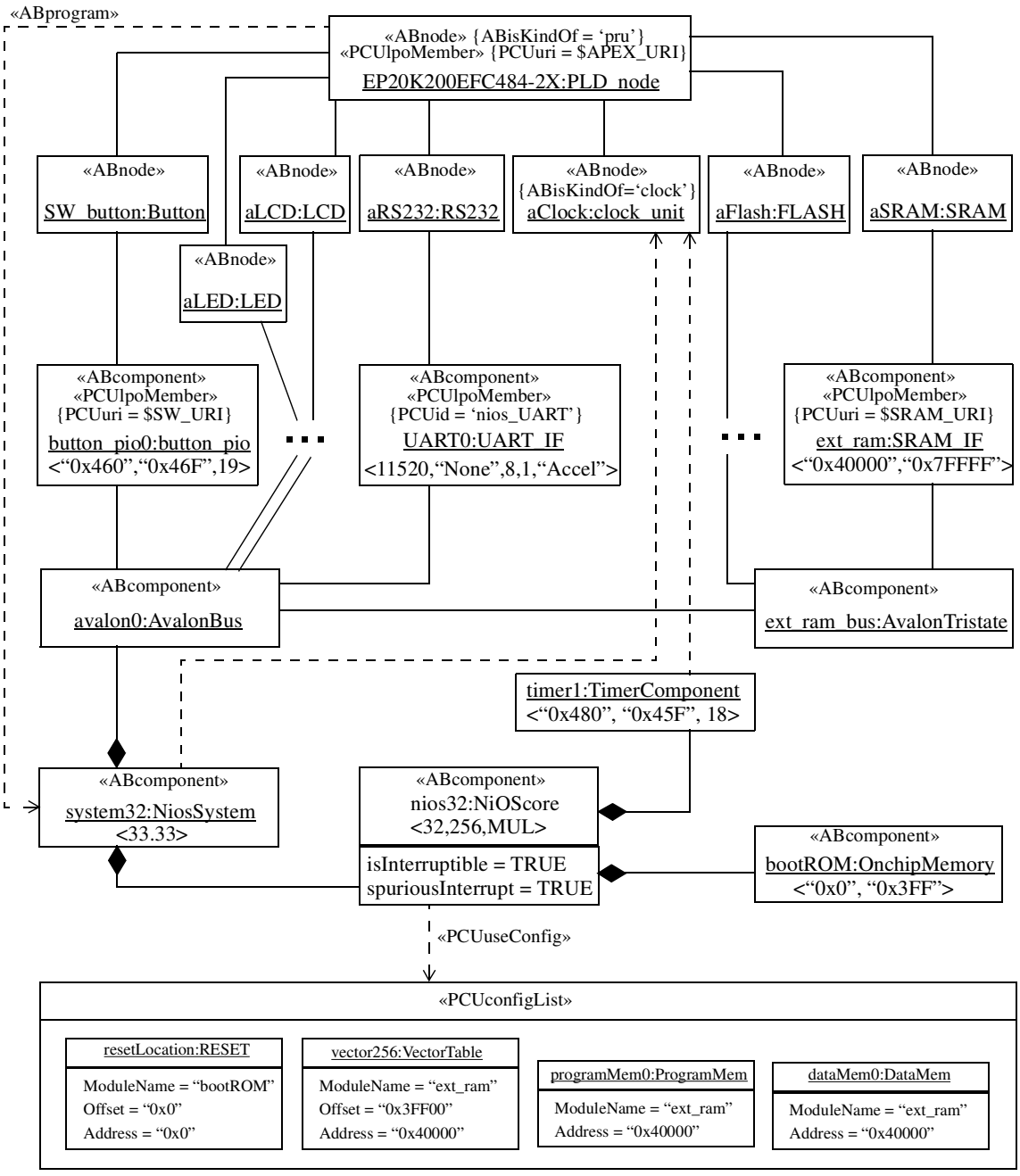


Figure 6.23: The UML description of the candidate target architecture as derived from the blueprint and the associated POmm/components

6.5.1 Peripheral Interface Routines and Quartus II

In traditional hardware/software codesign approaches, peripheral-interface routines, e.g device drivers, are normally written by the developer, using low-level macro code specific to the chosen platform. The utilization of the platform-centric SoC design method, or the *platform-based* approach in general, enables such software routines to be prepared in advance instead by the peripheral and/or platform providers and handed to the developer as a high-level run-time library package that could save the developer time and effort over low-level programming. The exact details as per how such a run-time library package is supported and distributed depends largely on how the platform and the peripheral providers agree on the collaboration framework. For the NiOS platform employed herein, the C run-time library for the relevant peripherals is supported by Altera (the platform provider). The developer accesses the library routines by including the machine-generated *nios.h* file with the application source code.

The NiOS run-time library is created as a byproduct of the NiOS system generation by the Quartus II software (visit *http://www.altera.com* for details). At the time of its installation, Quartus II registers itself, i.e. updating the *poRegfile*, as a POmm/tool that belongs to the NiOS platform. To conform to the platform-centric approach, Quartus II would ideally be able to extract required information right out of the target architecture (Figure 6.23) and produce the run-time library package as well as the PLD-configuration files, in the SRAM object file (*sof*) and Intel's hexadecimal output file (*hexout*) formats. However, to modify this commercial software to be totally compliant with the proposed approach is beyond the means of this thesis. As a workaround, an intermediate program could be written that analyzes the target architecture in the XMI format, extracts the required information, and generates the Peripherals Template File (*ptf*), which can then be input to Quartus II. After a successful run, the developer uses the NiOS system library to complete the platform-dependent specification as depicted in Figure 6.25; the report file (*rpt*) to perform the power estimation (Figure 6.24); and the *sof* or *hexout* file to program the EP20K200EFC484-2X device using the PLD programmer tool listed in Figure 6.17.

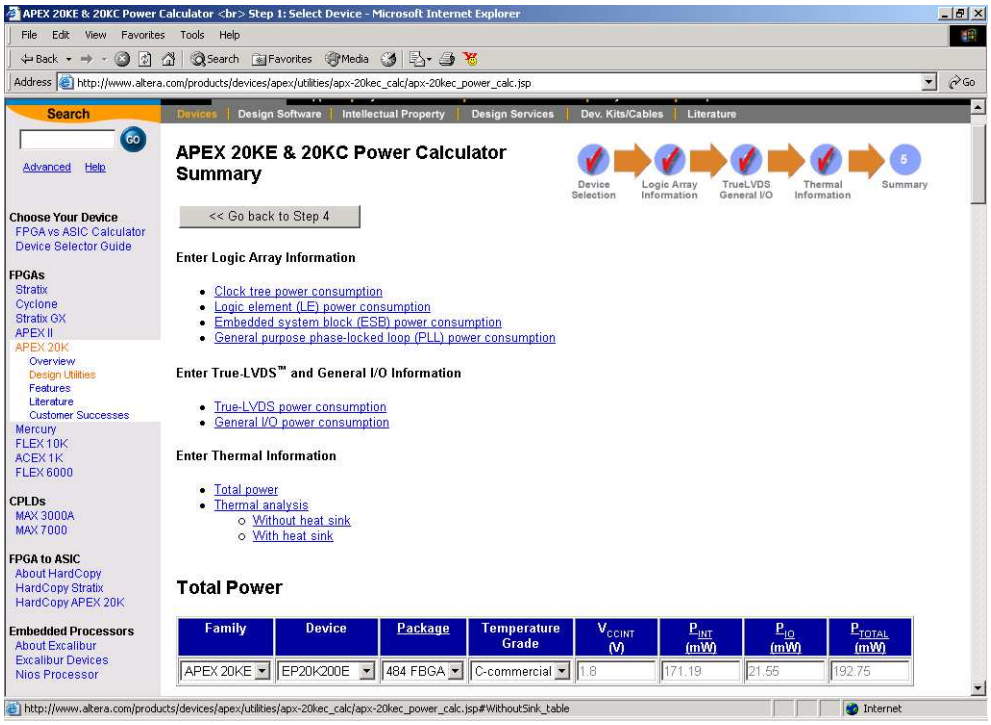


Figure 6.24: The EP20K200EFC484-2X PLD device power calculator provided as a Web application by Altera

The EP20K200EFC484-2X APEX20KE power calculator, which is supported by Altera and depicted in Figure 6.24, shows the estimated power of 192.75 mW—a fairly reasonable value commercially. Although more optimization effort may ensue to further reduce the power consumption to prolong battery life, this figure already meets the hard requirement that no fan is allowed. The input parameter values for this estimation are listed in Appendix A. The results obtained from this estimator provide a preliminary insight as per the power consumption characteristics of the system under development. More accurate results will manifest later during the test phase.

Table 6.6: The mapping of peripheral-related classes from the platform-independent specification to the platform-dependent specification

Platform-Independent	Platform-Dependent
mssg_service	mssg_service, but it behaves as a unified interface to two new classes, sevenseg_IF and lcd_IF
LED	led_IF
SW_pushButton and its generalized classes	IMprofile facility that specifies the interrupt (button_interrupt), the interrupt service routine (SW_service.start()) and the binding mechanism («IMbind»)
color_components' input buffer (an array representation)	FLASH memory @ 0x104000
media's table of content: array of image locations array of image sizes	 FLASH memory @ 0x100000 FLASH memory @ 0x100020
media's effective storage space (an array representation)	FLASH memory @ 0x120000

6.5.2 Transitioning to the Platform-Dependent Specification

As the name implies, the platform-dependent specification infers the structural and behavioral description of a software system to be deployed on a known platform. The developer derives this specification from the platform-independent document and the target architecture model, that are presented in Figures 6.16 and 6.23, respectively.

Table 6.6 shows the mapping of the peripheral-related classes from the platform-independent specification to its platform-specific counterpart. As evident from the table, the mapping is fairly minimal and systematic, where the peripheral-related classes are replaced by their respective interface classes and the array pointers are adjusted to reflect

the chosen addresses[1]. Such relative ease can be attributed to their explicit modeling in the platform-independent specification that permits them to be treated as objects, and become more structural and reusable (see also Section 6.3.3). To emphasize the point, consider the UART which is never explicitly modeled. Without the benefit of class reuse, the developer would have to modify every single occurrence of the UART-related operations—a process which can be tedious and error-prone. In this case, the UART is only used within the upload_method class, and involves only the overloaded printf() function that can output the printed string to the GERMS monitor via the RS232 PC serial port. Figure 6.25 depicts the detail-suppressed class model of the platform-dependent specification. Its detailed description can be found in Appendix E.

6.5.3 Deriving the System

The sheer complexity of today's SoC systems development mandates that automated tools be an integral part of the design process. The presence of such tools in a well-integrated environment can enhance the prescient insight of the developer, resulting in an effective decision making process that can expedite the system development as a whole.

As previously mentioned, the system derivation process is an iterative process involving different tools for different model views that represents different aspects of the system under development. To further drive the refinement process of the digital camera system development, the developer can re-target the profiling of the JPEG compression task against the preliminary system architecture so as to expose additional expectable, architecture-dependent system characteristics. Table 6.7 tabulates the execution times of the JPEG encoder, along with its FDCT sub-module's, that result from the profiling against the compression quality values of 65, 75, 90 and 99. The profiling input is a 227x149 RGB raw color components of size 33.03 KBytes, and the target architecture is the one illustrated in Figure 6.23.

1. The NiOS system utilizes a memory-mapped architecture.

(USE digiCam_genType.h, digiCam_classType.h, jpeg_library, EMprofile, IMprofile)

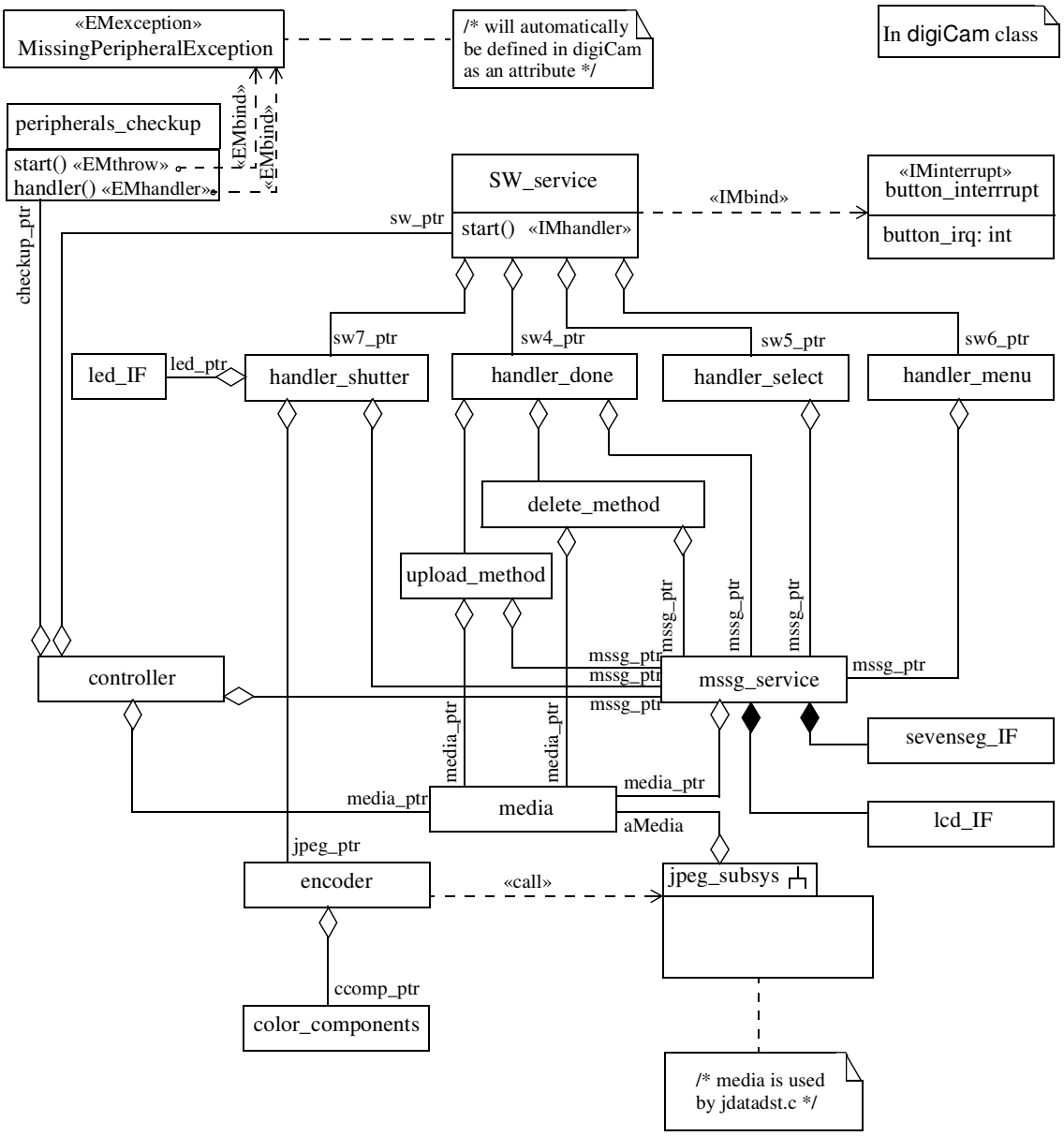


Figure 6.25: Detail-minimal platform-specific Class diagram describing the digital camera system

Table 6.7: Profiling results on timing characteristics of the JPEG compression (LL&M algorithm) of the 227x149 RGB color input components with respect to different compression quality values

Compression Quality	Encoder	FDCT (851 blocks)	Non-FDCT	JPEG Output Size
65	861.53 msec	203.97 msec	657.56 msec	4.75 KBytes
75	885.9 msec	204 msec	681.9 msec	5.02 KBytes
90	1095 msec	203.98 msec	891.02 msec	10.0 KBytes
99	2037 msec	203.98 msec	1833.02 msec	27.7 KBytes

By perceptive inspection, no drastic difference among the four images is found. Consequently the developer specifies the compression quality values, and makes the requirements adjustment. The NORMAL image quality is therefore defined to correspond to the compression quality of 65, and 90 is defined for the GOOD image quality.

A simple linear projection can be performed that could roughly estimate the worst-case execution time for the NORMAL and GOOD image compression on the required 640x480 RGB input components (Requirement NF-R6). By using the input sample size of 14400 (640x480x3) with the timing characteristics as shown in Table 6.8 that come directly from Table 6.7, the developer obtains the estimated worst-case execution times of 9.216 and 11.232 seconds for the NORMAL and GOOD setting, respectively. Both of these profiling results are greater than the 1 second requirement. Hence, the developer performs additional profiling on different settings and architectures, the results of which are shown in Table 6.8. Notice that the hardware encoder option is not included because it does not fit into the EP20K200EFC484-2X device, whose usable resource is 8320 LEs, while all of the encoder implementations (from *Amphion*, *Barco Silex*, and *CAST*) require the LEs in excess of this threshold.

Table 6.8: Profiling results on timing characteristics of the JPEG compression of the required 640x480 RGB color input components subject to different configurations

Timing Characteristics (msec/ 2D 8x8 block) **non-FDCT** (NORMAL)**:** 0.40, **non-FDCT** (GOOD)**:** 0.54, **DCT** (AA&N)**:** 0.19, **DCT** (LL&M)**:** 0.24					
Algorithms	**HW Accelerators**	**Settings**	**Total (seconds)**	**non-FDCT (seconds)**	**FDCT (seconds)**
LL&M	MUL	NORMAL	9.216	5.760	3.456
		GOOD	11.232	7.776	3.456
	HW FDCT (with DMA)	NORMAL	5.760	5.760	Negligible
		GOOD	7.776	7.776	Negligible
AA&N	MUL	NORMAL	8.496	5.760	2.736
		GOOD	10.512	7.776	2.736
	HW FDCT (with DMA)	NORMAL	5.760	5.760	Negligible
		GOOD	7.776	7.776	Negligible

The profiling results in Table 6.8 suggest that (1) there exists no feasible solution to the problem, i.e. given the current requirements specification and the availability of the POmm/components, the developer cannot achieve the targeted digital camera system without the specification violation, and (2) a constraint relaxation is needed that could result in the modification of the requirements specification. Indeed it is quite ambitious to implement a JPEG encoder on a 33 MHz, moderate density PLD device that can compress a 640x480 RGB image in 1 second. The following detail some plausible scenarios assuming that the decision is made in favor of relaxing the constraints.

- Another PLD device is acquired that is more powerful, such that it can house a complete hardware implementation of the JPEG encoder,

- The requirement reduces the number of input samples, and effectively, leaves the competitive digital camera market (maybe, entering the cellular phone market instead?). This decision would also affect the criteria for choosing an image sensor for the digital camera system.

By employing a more powerful, higher density PLD device, the developer could eradicate the bottleneck caused by slow clock speed, and high-volume data transfer. The pursuit of this option would likely yield the feasible digital camera system. However, it would incur additional cost of improving a PLD device, and developing/acquiring the hardware JPEG encoder.

On the other hand the developer could compute a more suitable input image size based on the data in Table 6.8. Because of the explicit 1 fps requirement on the NORMAL setting operation, and also because of the use of the fixed-point arithmetic for the JPEG engine, the data from the LL&M, NORMAL settings are used in the calculation. Through a simple linear interpolation of the digital camera performance where the system with the FDCT hardware accelerator runs at 0.4 msec/block, and that with the hardware multiply (MUL) at 0.64 msec/block, the developer attains the estimated input sample size of 53312, and 33344 for the hardware FDCT and MUL multiply, respectively. These numbers can be translated into the image dimension of approximately 168x320, and 168x200 pixels, both of which are much smaller than the 640x480 tentative specification.

6.5.4 Concluding Remarks

Thus far, this dissertation has presented, in detail in a pedagogical manner, the development process for the simplified digital camera system based on the proposed platform-centric SoC design method. It has demonstrated the strength of the proposed approach as an analysis, modeling, design, and documentation means that could expedite the complex SoC system development process via a well-integrated design environment that embraces the exploitation of platforms, UML and XML technologies.

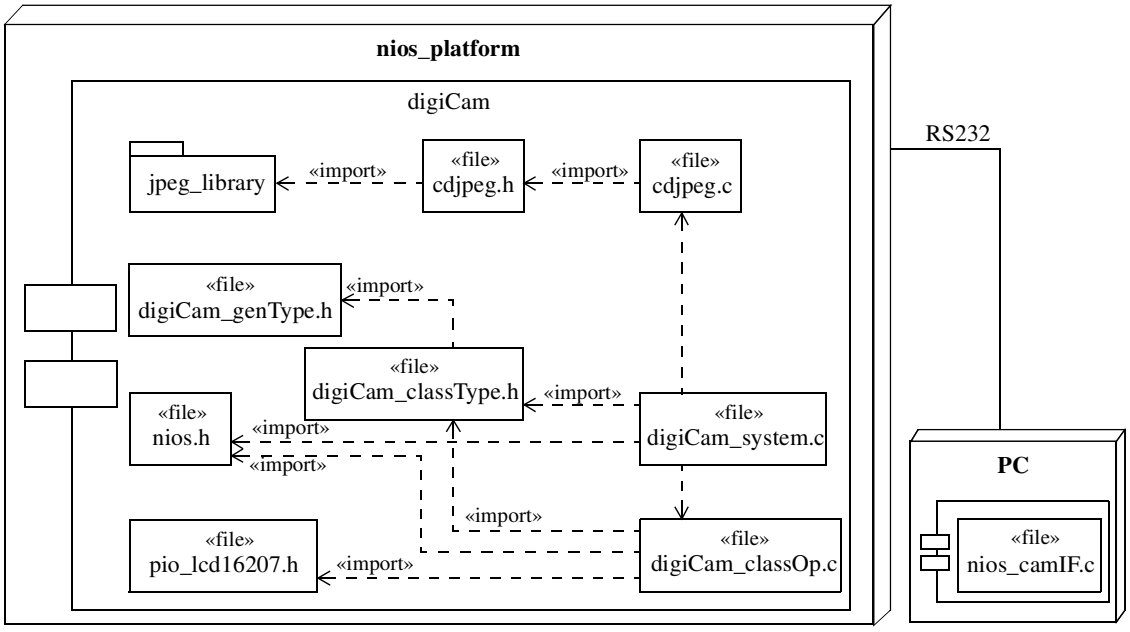


Figure 6.26: The source files hierarchy for the digital camera system

Where the task description ended in the previous section, the developer could have resumed by deciding on the available options and proceeded until the final system would result that meets all the specified requirements. Since all the steps involved have already been discussed, the dissertation halts here to avoid redundancy. The next section presents the implementation results, and compares the cost-effectiveness of the proposed approach against that of the SpecC methodology.

6.6 Implementation Results

The eventual implementation of this digital camera system conforms to the platform-dependent specification as discussed in Section 6.4 (see also Figures 6.23, and 6.25), with the exception that the input image dimension has effectively been scaled down to 160x160, instead of 640x480 as required by the early specification (see Section 6.5.3 for details). Figure 6.26 depicts the source files hierarchy of the implementation.

6.6.1 Resultant Timing and Compression Characteristics

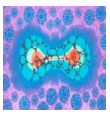
C is one of the most popular programming languages currently used to develop real-time embedded applications. To demonstrate that C and UML can work well together through their object-oriented discrepancy, all source files are presently implemented in C that mimics the *look and feel* of an OO programming language. Such a *structural* approach incurs an estimate 85% increase in code size compared with the *traditional* approach, but produces a much better correlation between the UML model and its corresponding C code.

Table 6.9 tabulates the timing and compression characteristics for different input images, all of which are represented by the 160x160 RGB components of size 25KBytes. As evident from the results, this fully-functional digital camera system prototype performs reasonably well, exhibiting only a single timing violation (*sample06.jpg*), while all natural scenes yield the timing results under the 1 second mark. Further extensive investigation will determine if this system actually fulfills the 95%, 1 fps requirement as dictated in the specification.

6.6.2 Research Evaluation

This section introduces the notion of *cost* as a metric for evaluating the robustness of the proposed approach. The cost modeling technique estimates the system development cost by taking into account a number of factors, from human to technology to the complexity of the project itself, that affect the development process. Such a technique could be very accurate when carefully calibrated to a specific problem, yet it normally yields good results, even without much calibration, when used to compare cost-effectiveness of two or more development costs. In the discussion to follow, the COCOMO II.2000 [19] cost modeling technique is utilized that comparatively evaluates the cost-effectiveness of the proposed approach against the SpecC methodology. At present SpecC is undergoing a standardization process to become a standard language and interchange format for system specification design (see *http://www.specc.org*). Thus, it constitutes an eligible benchmark for an evaluation of this research.

Table 6.9: Timing and compression characteristics data from different input images

Input Image Characteristics: 160x160 RGB, 25 KBytes				
Input Images	**Settings**	**Total (seconds)**	**JPEG file size (KBytes)**	**% Compressed**
sample01 	NORMAL GOOD	0.515 0.922	2.70 8.44	89.2 66.24
sample02 	NORMAL GOOD	0.523 0.957	2.99 9.83	88.04 60.68
sample03 	NORMAL GOOD	0.545 0.987	3.78 10.9	84.88 56.4
sample04 	NORMAL GOOD	0.722 1.213	5.59 14.8	77.64 40.8
sample05 	NORMAL GOOD	0.948 1.678	9.18 23.1	63.28 7.6
sample06 	NORMAL GOOD	1.028 1.841	11.5 24.7	54 1.2
sample07 	NORMAL GOOD	0.743 1.457	6.45 19	74.2 24

Typically, a development cost of an embedded system equals a sum of costs incurred by software and hardware, plus any cost or revenue adjustment amounted from missing or meeting the window of opportunity when the product enters the market. For an evaluation of this research, however, only the software cost is used in the calculation—a decision attributable to the fact that the SpecC and the proposed approaches permit the developer to work at too high a level of design abstraction for the current hardware cost modeling technique to be an effective efficiency indicator. On the other hand, the window of opportunity adjustment, which is computed by using the time-to-market cost estimation model [6, 21, 22], is directly proportional to the results from the software cost estimation, and hence, can be ignored when comparing cost-effectiveness of the two approaches.

6.6.2.1 Software Cost Modeling

Software cost estimators primarily consist of a core or nominal effort equation that relates the labor effort for developing software to the size of the software system. This nominal effort equation represents the cost of developing a software system under ideal conditions. To get a more realistic view of such software cost, effort adjustment factors are applied to the nominal estimate to adjust for organization and project-specific economic factors. For example, the COCOMO II [19] model uses the adjustment factors as shown in Table 6.10 to make the nominal estimate more realistic.

Although many of these tools employs very different parametric cost estimating relationships (CERs), they all make similar claims about how the design can affect the software development cost. In all cases, historical data show that increased development cost and time can occur as a result of squeezing more and more functionality in smaller and smaller space and time intervals, incurring more design efforts and overall cost as a result. REVIC [12], COCOMO II [19], and PRICE-S [18], for example, assume that resource requirements of less than 50% capacity have no cost impact, but as the utilization approaches 100% the cost impact becomes extreme as depicted in Figure 6.27 [16].

Table 6.10: COCOMO II.2000 effort multipliers (EMs).

Cost Drivers	Ratings
Product Factors	
EM_1: Required software reliability (RELY)	0.82 to 1.26
EM_2: Database size (DATA)	0.90 to 1.28
EM_3: Product complexity (CPLX)	0.73 to 1.74
EM_4: Developed for reusability (RUSE)	0.95 to 1.24
EM_5: Documentation match to life-cycle needs (DOCU)	0.81 to 1.23
Platform Factors	
EM_6: Execution-time constraint (TIME)	1.00 to 1.63
EM_7: Main-storage constraint (STOR)	1.00 to 1.46
EM_8: Platform volatility (PVOL)	1.00 to 1.30
Personnel Factors	
EM_9: Analyst capability (ACAP)	0.71 to 1.42
EM_{10}: Programmer capability (PCAP)	0.76 to 1.34
EM_{11}: Personnel continuity (PCON)	0.81 to 1.29
EM_{12}: Applications experience (APEX)	0.81 to 1.22
EM_{13}: Platform experience (PLEX)	0.85 to 1.19
EM_{14}: Language and tool experience (LTEX)	0.84 to 1.20
Project Factors	
EM_{15}: Use of software tools (TOOL)	0.78 to 1.17
EM_{16}: Multisite development (SITE)	0.80 to 1.22
EM_{17}: Required development schedule (SCED)	1.00 to 1.43

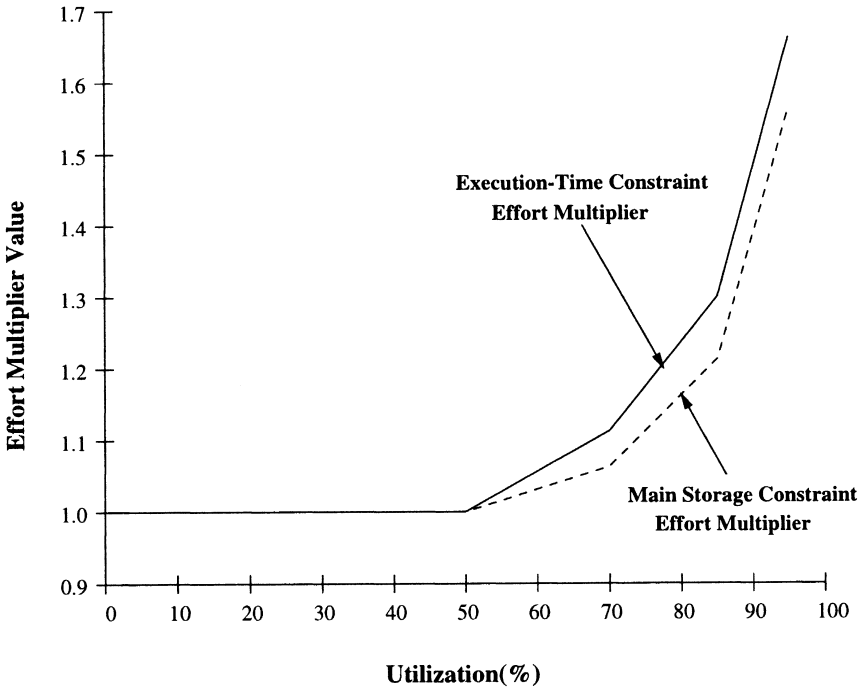


Figure 6.27: Execution time and main storage constraint effort multipliers vs. resource utilization.

6.6.2.2 Cost Comparison

The COCOMO II.2000 parametric cost model [19] is used to evaluate the robustness of this research against the SpecC methodology based on cost-effectiveness of the two approaches. The COCOMO II.2000 model estimates the system development effort using the equation of the form:

$$C_E = A(KSLOC)^E \prod_{i=1}^{17} EM_i \qquad \text{(EQ 6.1)}$$

where the unit of the effort equation (C_E) is in person-months, i.e. the amount of time one person spends working on the system development project for one month. The coefficient A is a productivity constant which captures the effects on effort with projects of increasing

size. *E*, an aggregation of five *scale factors* (*SF*), accounts for the relative economies of scale (values less than 1.0) or diseconomies of scale (values greater than 1.0) encountered for projects of different sizes. Its value can be computed according to the equation below:

$$E = B + 0.01 \sum_{j=1}^{5} SF_j \quad \text{(EQ 6.2)}$$

where the constant $B = 0.91$.

The effort multipliers EM_i model the effect of personnel, computer, product, and project attributes on software development cost. Table 6.10 gives a brief description of the COCOMO II.2000 effort multipliers.

However, the primary input to the COCOMO II.2000 cost estimator is the software size estimate, *KSLOC*. *KSLOC* denotes the number of source lines of code (thousands), which include application code, OS kernel services, control and diagnostics, and support software (see Appendix F for details). Software size estimates comprise two parts: the number of new source lines of code (*KNSLOC*), and the number of adapted source lines of code (*KASLOC*)—both are represented as a numerical multiplication of one thousand. This evaluation uses the following COCOMO II.2000 model to calculate the total number of source lines of code:

$$KSLOC = KNSLOC + (KRSLOC \times AAM_R) + (KASLOC \times AAM_A) \quad \text{(EQ 6.3)}$$

where AAM_R and AAM_A has the general form:

$$AAM = \frac{AA + [(0.4(DM) + 0.3(CM) + 0.3(IM)) \times (1 + 0.02(SU)(UNFM))]}{100} \quad \text{(EQ 6.4)}$$

The symbols in the equations are defined in Table 6.11.

Table 6.11: Software Sizing Model Symbol Definitions

Symbols	Description
KNSLOC	Size of component expressed in thousands of new source lines of code
KRSLOC	Size of the reused software component expressed in thousands of adapted source lines of code
KASLOC	Size of the adapted software component expressed in thousands of adapted source lines of code
AAM^a	Adaptation adjustment modifier.
AA	Degree of assessment and assimilation
DM	Percentage of design modified
CM	Percentage of code modified
IM	Percentage of integration and test modified
SU	Software understanding penalty
UNFM	Software unfamiliarity

a. In Eq. 6.3, the subscript *R* denotes *Reused*, while *A* denotes *Adapted*.

Before determining the number of effective source lines of code (*KSLOC*) resulted from employing the SpecC approach to develop the specified digital camera system, this dissertation recounts the principal tasks as prescribed by Figure 6.28 as follows (see also Section 2.3):

- *Specification phase*. SpecC derives the system specification during this phase of operation. As opposed to the proposed approach, it specifies no analysis process as per how the specification might be derived from the initial requirements. The resultant specification comprises the JPEG encoder, which can be obtained from *http://www.ics.uci.edu/~specc/*, as well as the behavioral specification of the digital camera interface.

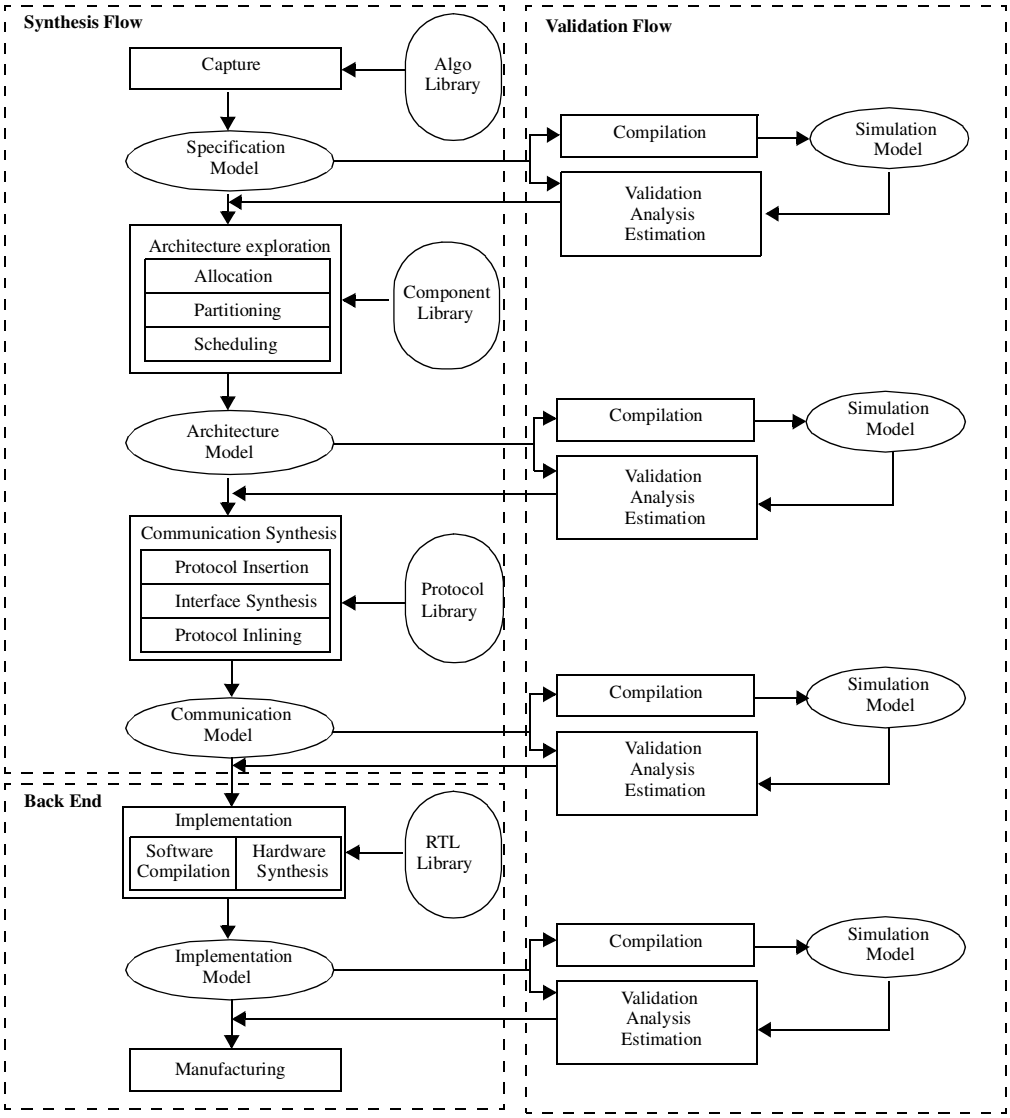


Figure 6.28: The SpecC methodology process flow.

- *Architecture phase*. As shown in Figure 6.28, this phase involves the architecture selection task and the system partition task. SpecC exploits its own profiling tool to extract the metrics necessary to perform these tasks from the NiOS-based hardware components, assumed to be available for the developer. At the end of this phase, a more refined specification results that is an equivalent of the platform-dependent specification in the platform-centric SoC design method.
- *Communication phase*. During this phase, the communication channels between hardware and software are modeled, elaborated and implemented. The result is a refined architecture model with all the abstract communication channels being resolved and synthesized. At the end of this phase, SpecC produces a component netlist in VHDL and software I/O instructions in C as its output
- *Implementation phase*. The developer uses the Cygwin C/C++ compiler to compile the C code. The hardware portion of the model, including the interfaces between hardware and software, is synthesized using Leonardo Spectrum. This design phase generates the implementation model which consists of object codes executing on the NiOS processor and a gate-level netlist of the hardware components which must then be fed to Quartus II for I/O pin connections and the *sof* file generation.

As identifiable from the SpecC tasks description above, the non-trivial source code that contributes to the *KSLOC* count come from the SpecC JPEG encoder specification model, the camera interface's behavioral description, and the NiOS specific software I/O instructions[1]. Because SpecC is an OO superset of C, the source lines for the interface behavioral description is estimated to be equal to the *traditional* C implementation of the same specification, which was written for the platform-centric development process for fast verification purpose. On the other hand, the lines count for the NiOS specific software I/O instructions is taken directly form the Quartus II generated run-time library. Table 6.12 summarizes the relevant numbers of source lines of code for both approaches.

1. The hardware glue logics are assumed to be small, and thus, negligible.

Table 6.12: Summary of the source lines of code applicable to the SpecC and platform-centric approaches

Relevant *KSLOC*	SpecC		Platform-Centric	
	Size (thousand)	**Type**	**Size (thousand)**	**Type**
Structural C implementation of digiCam	n/a	n/a	0.936	*KNSLOC*
Traditional C implementation of digiCam[a]	0.567	*KNSLOC*	n/a	n/a
jdatadst.c	n/a	n/a	0.121	*KASLOC*
IJG JPEG library	n/a	n/a	6.345	*KRSLOC*
SpecC JPEG encoder[a]	0.986	*KASLOC*	n/a	n/a
Software I/O routines	1.8	*KNSLOC*	1.8	*KRSLOC*

a. Represents the *best-case* estimate of the actual SpecC implementation.

Derived from Table 6.12, and based on Equations 6.3 and 6.4 , the effective source lines of code for the SpecC, and the proposed approach are 2.46, and 1.13 *KSLOC*. Then, by applying Equations 6.1 and 6.2, the estimated incurring costs for the development of the digital camera system using the SpecC and the platform-centric SoC design method are 4.0, and 1.3 person-months, respectively. Please refer to Appendix A for the parameter values used in the calculations in this section.

6.6.3 Concluding Remarks

Costliness-wise, the proposed approach benefits tremendously from the use of a platform, and the IJG's JPEG library, where they attribute to a smaller *KSLOC* value, regardless of the total number of source lines which is much larger than that of the SpecC (9.202K to 3.353K). Such results serve to further attest the role of reuse as indeed being indispensable at all phases of the complex system development process.

Chapter 7

Conclusions

7.1 Thesis Contributions

It has been shown in this thesis that, in an era where the complexity of developing SoC systems constantly increases, and the technology-to-market time dwindles in response to market dynamics and competitiveness, system developers can benefit from the integrated use of platforms, OO analysis techniques, and the ubiquity of the Internet technology, with very impressive scheduling and cost-saving results. In summary, this thesis has presented a novel platform-centric SoC design method that improves cost and technology-to-market time for the development of complex systems, while also effectively enhancing design space exploration. The novel contributions of this research include the following:

- Key technologies, specifically, *platforms*, *UML*, and *XML*, were identified, and seamlessly integrated that contribute to the development of a flexible, and robust system design process which is favorably applicable to a multitude of complex SoC system requirements.
- A robust design approach, i.e. the proposed platform-centric SoC design method, was developed that, in addition to the previously identified technologies, fosters reuse of both UML models (abstract platform components) and non-UML models (IJG's JPEG library), as well as the use and reuse of knowledge through the WWW technologies. In addition, it allows the internal processes to vary that could be more fitting to the chosen platform.

- A unified, visual representation of the system under development was achieved through the UML profile for Codesign Modeling Framework (CMF), that is based on UML 1.5 [24] and the UML Real-Time profile [29], and that can be utilized within the proposed platform-centric environment for modeling, design, analysis, synthesis, implementation and documentation purposes. The CMF profile allows all aspects of the SoC system development process, right from the initial requirements, to be described using one common language for better efficiency.
- The specification of an XML database, known as the Library of Platform Objects (LPO) was described in detail, along with its anticipated usage and behaviors. The LPO could span the whole Internet space, and could be distributed—paving a way for a discernible possibility that the system design community might converge on very few standard platforms such that the task of populating platform components and tools could be performed in a standard way, not limited to any one individual or organization, and these components and tools could enter and exit the LPO freely so long as they are Internet-accessible.
- The use of UML to assist in the development of a complex hardware/software codesign system, such as a digital camera, that involves real-time characteristics was demonstrated. It was also shown that such a UML application efficiently empowered an incorporation of OO analysis techniques, as well as enhanced design reuse, resulting in an overall improvement of the platform-centric SoC design approach.

7.2 Publications and Awards

The following publications have resulted from this research:

- C. Arpnikanondt, V. Madisetti, “Constraint-Based Codesign (CBC) of Embedded Systems: The UML Approach,” *Yamacraw Technical Report*, Georgia Institute of Technology, 1999.

- C. Arpnikanondt, V. Madisetti, *A Platform-Centric Codesign Approach for SoC Systems Development*, Book Proposal to Kluwer Academic Publishers, 2004.

The following award has resulted from this research:

- US Army Research Lab (ARL) Advanced Sensors Consortium Research Excellence Award, February 1999, College Park, Md.

7.3 Future Directions

This thesis is the product of integrating technologies in three different disciplines, namely *platforms*, *UML*, and *XML*, so as to form a robust system development approach that is applicable for use with a multitude of complex system requirements today and tomorrow. Each one of these technologies has gained steady research interest in both the industry and academia, and would likely continue to contribute to the proposed platform-centric approach. Nonetheless, to reap full benefits proffered by the approach it would mandate a consensus of support among all involved parties—the system houses, IC manufacturers, and tools and components providers. In the end, it is ultimately the concerted collaboration that matters heftily towards success.

Much exciting research readily exists in this novel platform-centric design approach, where three state-of-the-art technologies convolute. This thesis envisages *reuse* as the principal driving force that contributes to an efficient system development process. A number of researches that address this issue can be conceivable on many different levels and from many different perspectives. Research areas such as Retargetable Compilation, Mixed-Language Programming, and Legacy Software Reengineering all deserve profound research efforts for they make the proposed approach more flexibly and more accessibly reusable.

A tool-integrated environment for the POM represents another intriguing research area within this novel approach. Since the POM relates closely with XML and the Internet

technologies, how such research disciplines as Network Programming and Distributed Computing can enhance the POM capability would be worth an investigation. A UML-capable, UML-RT/CMF enabled POM would also be ideal for the approach.

In addition, this thesis assumes that platforms are pre-built, and readily available to the system developer. The actual development of a platform, however, is an extremely complex process as evidenced by Sabbagh [96], and other related work [70, 129]. More research is needed in this area to reduce the platform development cost, while efficiently producing a system platform that is well-suited to as many applications as possible.

Appendix A

Cost and Power Estimate Parameter Values

This appendix comprises two main sections. Section A.1 lists the parameter values used to estimate the development cost for the SpecC and the platform-centric (PC) approaches, while Section A.2 deals with the input parameter values used in the power consumption estimation of the digital camera system.

A.1 COCOMO II.2000 Cost Parameters

Tables A.1-A.6 provide detailed development cost parameter values for the SpecC and platform-centric approaches. Specifically, Table A.1 lists the *new*, *reused*, and *adapted* number of source lines of code, i.e. *KNSLOC*, *KRSLOC*, and *KASLOC*, respectively, for both methods. Table A.2 lists the input parameter values used to convert the reused source lines of code to the equivalent effective source lines of code (*KSLOC*); whereas Tables A.3 and A.4 list the values for the adapted source lines conversion for the platform-centric and the SpecC. Tables A.5 and A.6 summarize the scale factor (*SF*) ratings, and the effort multiplier (*EM*) ratings for the two approaches, respectively.

Since this thesis largely concerns with the relative efficiency of the two methods, it compares the actual figures from the proposed approach against the best-case estimated values from the SpecC approach, the results of which were discussed in Section 6.6. These estimated values were obtained from various sources as specified in Table A.1.

Table A.1: Summary of the source lines of code applicable to the SpecC and platform-centric approaches

Relevant *KSLOC*	SpecC		Platform-Centric	
	Size (thousand)	Type	Size (thousand)	Type
Structural C implementation of digiCam	n/a	n/a	0.936	*KNSLOC*
Traditional C implementation of digiCam[a]	0.567	*KNSLOC*	n/a	n/a
jdatadst.c	n/a	n/a	0.121	*KASLOC*
IJG JPEG library	n/a	n/a	6.345	*KRSLOC*
SpecC JPEG encoder[b]	0.986	*KASLOC*	n/a	n/a
Software I/O routines[c]	1.8	*KNSLOC*	1.8	*KRSLOC*

a. Obtained from a byproduct of this thesis
b. Obtained from the work by Dr. Gajski's team at *http://www.ics.uci.edu/~specc/*
c. Obtained from the run-time library generated by Quartus II

Table A.2: PC input parameter values for *KRSLOC* to *KSLOC* conversion

Parameters	Situation	Rating
Assessment & Assimilation (*AA*)	Basic module search & documentation	2
Programmer Unfamiliarity (*UNFM*)	Mostly familiar	0.2
Software Understanding (*SU*)	Reused	n/a
Percent Design Modified (*DM*)	Reused	0
Percent Code Modified (*CM*)	Reused	0
Percent Integration Required (*IM*)	Reused, very small	1%

Table A.3: PC input parameter values for *KASLOC* to *KSLOC* conversion

Parameters	Situation	Rating
Assessment & Assimilation (*AA*)	Basic module search & documentation	2
Programmer Unfamiliarity (*UNFM*)	Mostly familiar	0.2
Software Understanding (*SU*)	Structural code, well descriptive	Very High: 10
Percent Design Modified (*DM*)	Very small	1%
Percent Code Modified (*CM*)	Small	5%
Percent Integration Required (*IM*)	Small	5%

Table A.4: SpecC input parameter values for *KASLOC* to *KSLOC* conversion

Parameters	Situation	Rating
Assessment & Assimilation (*AA*)	Basic module search & documentation	2
Programmer Unfamiliarity (*UNFM*)	Completely familiar	0
Software Understanding (*SU*)	Structural code, well descriptive	Very High: 10
Percent Design Modified (*DM*)	Very small	1%
Percent Code Modified (*CM*)	Small to moderate	20%
Percent Integration Required (*IM*)	Small	5%

Table A.5: SpecC/PC scale factor (*SF*) values with B = 0.91

Scale Factors	Situation	Rating
SF_1: Precedentedness (*PREC*)	Considerable experience working with software	High: 2.48
SF_2: Development Flexibility (*FLEX*)	Considerable need for complying to requirements	High: 2.03
SF_3: Architecture/Risk Resolution (*RESL*)	Some critical risk identification & tool support	Nominal: 4.24
SF_4: Team Cohesion (*TEAM*)	Basic consistency of objectives and culture	Nominal: 3.29
SF_5: Process Maturity (*PMAT*)	CMM Level 2	Nominal: 4.68

Table A.6: SpecC and PC effort multiplier (*EM*) values with A = 2.94

Effort Multipliers	Situation	SpecC Rating	PC Rating
EM_1: Reliability (*RELY*)	Low, easily recoverable losses	Low: 0.92	Low: 0.92
EM_2: Database Size (*DATA*)	Testing DB/Program SLOC < 10	Low: 0.9	Low: 0.9
EM_3: Product Complexity (*CPLX*)	SpecC: Low-level coding; interrupt PC: Component reuse	High: 1.17	Low: 0.87
EM_4: Developed for Reuse (*RUSE*)	Across-program reusability	High: 1.07	High: 1.07
EM_5: Required Documentation (*DOCU*)	Right-sized to life-cycle needs	Nominal: 1.0	Nominal: 1.0
EM_6: Exe Time Constraint (*TIME*)	< 50% use of available exe. time	Nominal: 1.0	Nominal: 1.0
EM_7: Main Storage Constraint (*STOR*)	SpecC: < 50% usage PC: 85% usage	Nominal: 1.0	Very High: 1.46
EM_8: Platform Volatility (*PVOL*)	1 year up	Low: 0.87	Low: 0.87
EM_9: Analyst Capability (*ACAP*)	SpecC: High, little tool support PC: High, with tool support	High: 0.85	Very High: 0.71
EM_{10}: Programmer Capability (*PCAP*)	Nominal at 55th percentile	Nominal: 1.0	Nominal: 1.0
EM_{11}: Personnel Continuity (*PCON*)	Nominal at 12% turnover per year	Nominal: 1.0	Nominal: 1.0
EM_{12}: Application Experience (*APEX*)	1 year	Nominal: 1.0	Nominal: 1.0
EM_{13}: Platform Experience (*PLEX*)	1 year	Nominal: 1.0	Nominal: 1.0
EM_{14}: Language & Tool Experience (*LTEX*)	1 year	Nominal: 1.0	Nominal: 1.0
EM_{15}: Use of Software Tool (*TOOL*)	strong, mature, well-integrated with processes	Very High: 0.78	Very High: 0.78
EM_{16}: Multisite Development (*SITE*)	Fully collocated	Extra High: 0.80	Extra High: 0.80
EM_{17}: Required Development Schedule (*SCED*)	Nominal	Nominal: 1.0	Nominal: 1.0

A.2 Power Consumption Input Parameters

Altera provides a power estimator utility for the APEX20KE device family through its web site. This tool was modeled into the LPO as one of the NiOS's POmm/tools; it was used to estimate the power consumption of the digital camera system under development. The input parameter values for the estimator were retrieved from the NiOS system report file (*.rpt*) which was generated automatically by Quartus II after the completion of the NiOS system building process. Table A.7 summarizes these parameter values.

Table A.7: Input parameter values for Altera's APEX20KE PLD device

Parameters	Values
Device	EP20K200E
Package	484FBGA, Commercial Grade
V_{CCINT}	1.8 V
$I_{CCINT,standby}$	10 mA
f_{max}	33.33 MHz
Number of flip-flops	1429
Total LE	3526
Total LE wit carry chain	37
Average LE toggle percentage	12.5
ESB output turbo ON	111
ESB output turbo OFF	0
Average ESB output toggle percentage	12.5
Number of OUT and INOUT pins	95
Average output toggle percentage	12.5
Average capacitance load	10 pF
I/O standard	3.3 LVTTL/LVCMOS

Appendix B

Codesign Modeling Framework Stereotypes and Tags Listing

This appendix provides a quick reference to all the UML stereotypes and tag values defined in this dissertation (see Chapter 5). Section B.1 lists the stereotypes, and Section B.2 the defined tags, in alphabetical order.

B.1 Stereotypes Listing

Stereotypes	Base Class	Tags	Description
«ABbecome»	Dependency	--None--	Relates design unit to hardware entity of type IU
«ABcomponent»	Class Object	--None--	A POmm/component concept
«ABdeploy»	Dependency	--None--	Signifies residency of a design unit in hardware, equivalent to UML «deploy»
«ABnode»	Class Object	ABisKindOf	Hardware as perceived by an architecture blueprint
«ABprogram»	Dependency	--None--	Relates design unit to hardware entity of type PRU
«EMbind»	Dependency	--None--	Binds the exception, the throw method and the handler together
«EMcatch»	SimpleState CompositeState	--None--	Exception: *catch*

Stereotypes	Base Class	Tags	Description
«EMcatchAll»	SimpleState CompositeState	--None--	Exception: indiscriminate *catch*
«EMexception»	Signal	--None--	Represents an exception
«EMhandler»	Method	--None--	Represents an exception handler method
«EMthrowMethod»	Method	EMthrowType	Exception: *throw*
«EMtry»	SimpleState CompositeState	--None--	Exception: *try*
«IMbind»	Dependency	--None--	Binds an interrupt and its handler
«IMbitField»	Attribute	IMrwMode	Represents an individual bit field within a device register
«IMbitVector»	Class Object	IMaddress IMalignment IMbitOrder IMvectorSize	A collection of «IMbitField» elements. It corresponds to a BITVECTOR type in VHDL.
«IMdeviceIF»	Class Object	--None--	Represents the device encapsulation concept
«IMhandler»	Method	--None--	Represents an interrupt handler method
«IMinterrupt»	Class Object	IMid IMisReserved IMpriority	Represents an interrupt
«PCUattribute»	Attribute	--None--	Designates design variables
«PCUauxAttr»	Attribute	--None--	Designates non-design variables
«PCUcode»	Component	PCUfileUri	A file insertion mechanism
«PCUcodeBody»	Note	--None--	Represents body of code to be inserted
«PCUdeclare»	Note	--None--	Represents the declaration part of code to be inserted
«PCUconfigList»	Class	--None--	Permits configuration attributes to be grouped together separately from their parent class
«PCUlpoMember»	Class Object	PCUuri PCUid	Provides link from UML model to the LPO library
«PCUmain»	Method	--None--	Specifies the main function
«PCUrun»	Package	PCUrunline PCUrunfile	A *package* processing instruction

Stereotypes	Base Class	Tags	Description
«PCUuseConfig»	Dependency	--None--	Binds the «PCUconfigList» class to the parent class
«SHDLarch»	Class	--None--	Represents architecture in the VHDL's entity/architecture pair
«SHDLbind»	Dependency	--None--	Represents the binding of the entity/ architecture pair
«SHDLentity»	Class Object	--None--	Represents entity in the VHDL's entity/ architecture pair
«SHDLgenerate»	Class	SHDLgenFor	Models VHDL's generate block
«SHDLin»	Attribute	--None--	Input port
«SHDLinout»	Attribute	--None--	Bidirectional port
«SHDLmodule»	Class	--None--	HDL design module
«SHDLout»	Attribute	--None--	Output port
«SHDLparBlock»	Object	--None--	A grouping of concurrent statements
«SHDLprocess»	Object	--None--	Represents the existence of a process in an architecture body
«SHDLtypedef»	Class Object Note	SHDLdefine-Type	Allows user-defined types to be defined that represents both HDL's data type and data object

B.2 Tags Listing

Tag Name	Base Stereotype	Type	Multiplicity
ABisKindOf	«ABnode»	Enumeration: (‘pru’, ‘iu’, ‘diu’, ‘ifu’, ‘mu’, ‘clock’, ‘timer’)	0..1
EMthrowType	«EMthrowMethod»	TVL List of throwable exception types, for example (‘rErr’,’wErr’, ‘rwErr’)	0..1
IMaddress	«IMbitVector»	Integer	0..1
IMalignment	«IMbitVector»	Integer	0..1
IMbitOrder	«IMbitVector»	Enumeration: (‘ascend’, ‘descend’)	0..1
IMid	«IMinterrupt»	String	0..1
IMisReserved	«IMinterrupt»	Boolean	0..1
IMpriority	«IMinterrupt»	Integer	0..1
IMrwMode	«IMbitField»	Enumeration: (‘r’, ‘w’, ‘rw’)	0..1
IMvectorSize	«IMbitVector»	Integer	1
PCUfileUri	«PCUcode»	String	0..1
PCUid	«PCUlpoMember»	String	0..1
PCUrunfile	«PCUrun»	String	0..1
PCUrunline	«PCUrun»	String	0..1
PCUuri	«PCUlpoMember»	String	0..1
SHDLdefineType	«SHDLtypedef»	SHDLattrType	1..*
SHDLgenFor	«SHDLgenerate»	SHDLforInfoType	1
SHDLsensitive	«SHDLprocess»	TVL List	0..1
SHDLsensitive_neg	«SHDLprocess»	TVL List	0..1
SHDLsensitive_pos	«SHDLprocess»	TVL List	0..1

Appendix C

The LPO Tags Listing

In this appendix, a brief summary of the XML tags' semantics that are defined in Chapter 4 of this dissertation is presented as a quick reference, in alphabetical order. Section C.1 summarizes the tags definition; whereas, Section C.2 compiles relevant attributes.

C.1 LPO Tags Semantics

Name	Type	Multiplicity	DTD File	Description
associatedTools	Element	0..1	polif	Specifies possible association between a POmm/component and a POmm/tool(s)
aTool	Element	1..*	polif	Identity of each associated tool
autoConfig	Element	0..*	polif	Compartment that holds pre-configured values for UML parameters
blueprint	Element	1..*	poRegfile	Link to architecture blueprint(s)
characteristics	Element	0..1	polif	Contains *databook* information of the module
componentDomain	Element	0..1	polif	Compartment for information about POmm/component. It is not used if a POlif is of type *tool*.
config	Element	1..*	polif	Pre-configured UML parameter values
defaultToolID	Leaf	1	polif	Default tool by ID. Cannot co-exist with defaultToolURI.
defaultToolURI	Leaf	1	polif	Default tool by URI. Cannot co-exist with defaultToolID.

Name	Type	Multiplicity	DTD File	Description
forPOID	Leaf	0..1	polif	ID of platform object that these pre-configured values are applicable for
functions	Element	0..*	polif	Supplies information, if there is any, about hardware-dependent software routines
id	Leaf	1	lpoRegfile	Self ID by special identification
import	Element	0..1	polif	Reuse mechanism that allows UML packages to be imported
installerURI	Element	0..1	polif	Link to an installer
key	Leaf	1..*	lpoRegfile	Keyword string
lpoRegfile	Root	1	lpoRegfile	Signifies existence of LPO
name	Leaf	1	lpoRegfile	Self ID by name
physicalURI	Element	1	polif	Possible locations that the corresponding physical module may reside
po	Element	1..*	lpoRegfile	Link to platform objects
poID	Leaf	1	poRegfile	Reference to the PO to which the register file belongs
polif	Element	1..*	poRegfile	Link to POmm
polif	Root	1	polif	Signifies existence of a POmm
pom	Element	1	poRegfile	Link to POM
poRegfile	Root	1	poRegfile	Signifies existence of PO
poSchema	Element	1	poRegfile	Link to a PO schema document
preDefined	Element	0..*	polif	Pre-defined component characteristics
searchkey	Element	0..1	lpoRegfile	Relevant keywords that can identify self
self	Element	1	lpoRegfile	Self identification
selfURI	Element	1	polif	Possible locations that it may reside
swPackage	Element	0..1	polif	Reference to the software
targetCompiler	Element	1	polif	Expected target compiler
textField	Leaf	0..1	lpoRegfile	Knowledge-based information
uml	Element	1	polif	Specifies UML representation of the module

Name	Type	Multiplicity	DTD File	Description
uninstallerURI	Element	0..1	polif	Link to an uninstaller
uri	Leaf	1	lpoRegfile	Self ID by location
userDefined	Element	0..*	polif	User-defined component characteristics

C.2 LPO Attributes Listing

Name	Type	Base	Description
abKind	Enumeration: {“pru”, “iu”, “diu”, “ifu”, “mu”, “clock”, “timer”}	polif	Classifies itself to be one of the blueprint types (see Section 5.6.1 for detail).
classification	Enumeration: *PO-dependent*	polif	User-defined category of the module
fieldType	CDATA	textField	Expected data format
isImported	Enumeration: “yes” or “no”	textField	Specifies if the content contains link to an imported document
moduleID	CDATA	polif	Reference to a POmm via ID. Serves as a backdrop to *name* and *uri* references.
moduleKind	Enumeration: {“component”, “tool”}	polif	Classifies itself to be either *component* or *tool*
name	Enumeration: *PO-dependent*	config	Name of config data
name	Enumeration: *PO-dependent*	preDefined	Name of preDefined characteristics
name	CDATA	userDefined	Name of userDefined characteristics
subject	CDATA	textField	Subject of information
type	Enumeration: *PO-dependent*	config	Predefined type of config data
type	Enumeration: *PO-dependent*	preDefined, userDefined	Predefined and userdefined type of characteristics

Name	Type	Base	Description
unit	Enumeration: *PO-dependent*	config	Unit of config data
unit	Enumeration: *PO-dependent*	preDefined, userDefined	Unit of characteristics
value	CDATA	config	Value of config data
value	CDATA	preDefined, userDefined	Characteristics value

Appendix D

DTD Files Listing

This appendix presents the implementation of the relevant LPO schema documents in the Document Type Definitions (DTD) format. It begins with an implementation of the *lpoRegfile.dtd* in Section D.1, followed by the *poRegfile.dtd* in Section D.2. Thereafter, an implementation of the *polif.dtd* ensues in Section D.3. All verification tasks were done using MSXML 4.0, downloadable from *http://www.microsoft.com/downloads/*.

D.1 lpoRegfile.dtd

```
1 <!ENTITY fileDescription "
2 !-- ------------------------------------------------------------ --
3 !--                                                              --
4 !-- File name:        lpoRegfile.dtd                             --
5 !-- Author:           Chonlameth Arpnikanondt                    --
6 !-- Last revised:     09/06/03                                   --
7 !-- Description:      This dtd file is intended to serve as a    --
8 !--                   simple example for demonstrating what the  --
9 !--                   dtd file for each PO module may look like  --
10 !--                  within the platform instance. It represents --
11 !--                  just one of many possible implementations  --
12 !--                  of the recommendations for constructing the --
13 !--                  LPO as presented in my thesis.             --
14 !--                                                             --
15 !-- ------------------------------------------------------------ --
16
17 ">
18
19 <!ELEMENT lpoRegfile (self, po+)>
20
21 <!-- Start with self here -->
```

```
22 <!ELEMENT self (name, id, uri, textField?)>
23 <!ELEMENT name (#PCDATA)>
24 <!ELEMENT textField (#PCDATA)>
25 <!ELEMENT id (#PCDATA)>
26 <!ELEMENT uri (#PCDATA)>
27 <!ELEMENT key (#PCDATA)>
28
29 <!-- Main Entry... consisting of toolModule and componentModule -->
30 <!ELEMENT po (name, uri, searchkey?)>
31 <!ELEMENT searchkey (key+)>
32
33 <!-- Attribute declaration -->
34 <!-- This DTD uses a MIME type for the fieldType attribute -->
35 <!ATTLIST textField subject CDATA #IMPLIED>
36 <!ATTLIST textField fieldType CDATA "text/plain">
37 <!ATTLIST textField isImported (yes | no) "no">
38
```

D.2 poRegfile.dtd

```
1 <!ENTITY fileDescription "
2 !-- ------------------------------------------------------------ --
3 !--                                                              --
4 !-- File name:        poRegfile.dtd                              --
5 !-- Author:           Chonlameth Arpnikanondt                    --
6 !-- Last revised:     09/06/03                                   --
7 !-- Description:      This dtd file is intended to serve as a    --
8 !--                   simple example for demonstrating what the  --
9 !--                   dtd file for each PO module may look like  --
10 !--                  within the platform instance. It represents --
11 !--                  just one of many possible implementations  --
12 !--                  of the recommendations for constructing the --
13 !--                  LPO as presented in my thesis.             --
14 !--                                                             --
15 !-- ------------------------------------------------------------ --
16
17 ">
18
19 <!-- Define abstract type for each module -->
20 <!ENTITY % typeFile SYSTEM "file:///C:/Data/Thesis/XML/Test/niosClassification.
txt">
```

```
<!ENTITY % classificationList "(%typeFile;)">


<!ELEMENT poRegfile (self, searchkey?, pom, blueprint+, poSchema, textField*, polif+)>

<!-- Start with self here -->
<!ELEMENT self (name, id, poID, uri)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT poID (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT uri (#PCDATA)>

<!-- the POM, PO Manager -->
<!ELEMENT pom (id, uri)>
<!-- Architecture blueprint -->
<!ELEMENT blueprint (uri)>
<!-- Location of the PO schema file -->
<!ELEMENT poSchema (uri)>
<!-- Information about the PO -->
<!ELEMENT textField (#PCDATA)>

<!-- Main Entry... consisting of toolModule and componentModule -->
<!ELEMENT polif (name, uri, searchkey?)>
<!ELEMENT searchkey (key+)>
<!ELEMENT key (#PCDATA)>

<!-- Attribute declaration -->
<!-- This DTD uses a MIME type for the fieldType attribute -->
<!ATTLIST textField subject CDATA #IMPLIED>
<!ATTLIST textField fieldType CDATA "text/plain">
<!ATTLIST textField isImported (yes | no) "no">
<!ATTLIST polif moduleKind (component | tool) #REQUIRED>
<!ATTLIST polif abKind (pru | iu | diu | ifu | mu | clock | timer) #IMPLIED>
<!ATTLIST polif classification %classificationList; #IMPLIED>
<!-- moduleID should match an id as defined in each corresponding xml file -->
<!ATTLIST polif moduleID ID #REQUIRED>

```

D.3 polif.dtd

```
1 <!ENTITY fileDescription "
2 !-- ------------------------------------------------------------ --
3 !--                                                              --
4 !-- File name:          polif.dtd                                --
5 !-- Author:             Chonlameth Arpnikanondt                  --
6 !-- Last revised:       09/06/03                                 --
7 !-- Description:        This dtd file is intended to serve as a  --
8 !--                     simple example for demonstrating what the --
9 !--                     dtd file for each PO module may look like --
10 !--                    within the platform instance. It represents --
11 !--                    just one of many possible implementations --
12 !--                    of the recommendations for constructing the --
13 !--                    LPO as presented in my thesis.           --
14 !-- Note:               Choice of ELEMENT and ATTLIST for certain --
15 !--                    parameters is reached by considering if it --
16 !--                    should be treated as object value or object --
17 !--                    property. This places all IDs and URIs in --
18 !--                    ELEMENT self to be regarded as ELEMENT.  --
19 !--                                                             --
20 !-- ------------------------------------------------------------ --
21
22 !-- Namespace is ignored by default for it does not mesh well --
23 !-- with DTD. When XML-Schema is stable enough, one may want --
24 !-- to convert this DTD to a schema and make full use of the --
25 !-- namespace. --
26 ">
27
28 <!-- Define abstract type for each module -->
29 <!ENTITY % typeFile SYSTEM "file:///C:/Data/Thesis/XML/Test/niosClassification.
txt" >
30 <!ENTITY % classificationList "(%typeFile;)">
31
32 <!-- Define sample UML parameters -->
33 <!ENTITY % umlParam "(     Width |
34                            BaseAddr |
35                            EndAddr |
36                            Irq)">
37
38 <!-- Define component values characteristics -->
39 <!ENTITY % valueName "(    execute_time |
40                            access_time |
```

```
41                                  delay_time |
42                                  latency |
43                                  setup_time |
44                                  width |
45                                  length |
46                                  power_dissipation |
47                                  frequency |
48                                  period |
49                                  16b_acc_le |
50                                  16b_acc_eab |
51                                  16b_acc_performance)">
52
53 <!ENTITY % valueType "(     int |
54                                  signed_int |
55                                  unsigned_int |
56                                  float |
57                                  double |
58                                  long |
59                                  longlong)">
60
61 <!ENTITY % timeUnit "       fs |
62                                  ps |
63                                  ns |
64                                  us |
65                                  ms |
66                                  sec |
67                                  min |
68                                  hr">
69
70 <!ENTITY % lengthUnit "    fm |
71                                  pm |
72                                  nm |
73                                  um |
74                                  mil |
75                                  mm |
76                                  cm |
77                                  in">
78
79 <!ENTITY % freqUnit "MHz |
80 GHz">
81
82 <!ENTITY % valueUnit "(%timeUnit; | %lengthUnit; | %freqUnit; | none)">
83
```

```
84
85 <!ELEMENT polif (self, searchkey?, componentDomain?, textField?)>
86 <!ELEMENT self (     name,
87                      id,
88                      poID,
89                      selfURI,
90                      physicalURI?,
91                      installerURI?,
92                      uninstallerURI?,
93                      textField?)>
94 <!ELEMENT name (#PCDATA)>
95 <!ELEMENT poID (id)+>
96 <!ELEMENT selfURI (uri)+>
97 <!ELEMENT physicalURI (uri)+>
98 <!ELEMENT installerURI (uri)+>
99 <!ELEMENT uninstallerURI (uri)+>
100 <!ELEMENT textField (#PCDATA)>
101 <!ELEMENT id (#PCDATA)>
102 <!ELEMENT uri (#PCDATA)>
103 <!ELEMENT searchkey (key+)>
104 <!ELEMENT key (#PCDATA)>
105
106 <!-- Start declaration for children of self -->
107
108 <!-- the componentDomain declaration -->
109 <!ELEMENT componentDomain (     associatedTools?,
110                                 uml,
111                                 functions*,
112                                 characteristics?,
113                                 textField?)>
114
115 <!-- There can be more than one tool. -->
116 <!ELEMENT associatedTools ((    defaultToolURI |
117                                 defaultToolID),
118                                 aTool+,
119                                 textField?)>
120 <!ELEMENT defaultToolURI (#PCDATA)>
121 <!ELEMENT defaultToolID (#PCDATA)>
122 <!-- Each tool can be associated with more than one possible tool call -->
123 <!ELEMENT aTool ((uri | id),
124                      textField?)>
125
126 <!-- This part concerns UML and its characteristics -->
```

```
<!ELEMENT uml (   uri,
                  import?,
                  autoConfig*,
                  textField?)>
<!ELEMENT import (uri)+>
<!ELEMENT autoConfig (forPOID?, config+)>
<!ELEMENT forPOID (#PCDATA)>
<!ELEMENT config (textField?)>

<!-- This part handles hardware-dependent functions -->
<!ELEMENT functions (targetCompiler, swPackage?, textField?)>
<!ELEMENT targetCompiler (id | uri)+>
<!ELEMENT swPackage (uri)+>

<!-- This part deals with characteristics, both predefined and user-defined -->
<!ELEMENT characteristics (preDefined*, userDefined*)>
<!ELEMENT preDefined (textField?)>
<!ELEMENT userDefined (textField?)>

<!-- Attribute declaration -->
<!ATTLIST config name %umlParam; #REQUIRED
          type %valueType; #REQUIRED
          value CDATA #REQUIRED
          unit %valueUnit; #REQUIRED>
<!ATTLIST  preDefined name %valueName; #REQUIRED
          type %valueType; #REQUIRED
          value CDATA #REQUIRED
          unit %valueUnit; #REQUIRED>
<!ATTLIST  userDefined name CDATA #REQUIRED
          type %valueType; #REQUIRED
          value CDATA #REQUIRED
          unit %valueUnit; #REQUIRED>

<!-- This DTD uses a MIME-like extension for the fieldType attribute -->
<!ATTLIST textField subject CDATA #IMPLIED>
<!ATTLIST textField fieldType CDATA "text/plain">
<!ATTLIST textField isImported (yes | no) "no">
<!ATTLIST polif moduleKind (component | tool) #REQUIRED>
<!ATTLIST polif abKind (pru | iu | diu | ifu | mu | clock | timer) #IMPLIED>
<!ATTLIST polif classification %classificationList; #IMPLIED>

```

Appendix E

Digital Camera Specification

This appendix details the Class diagrams for the platform-independent and platform-dependent specifications of the simplified digital camera system presented in Chapter 6 of this dissertation. The Class diagram for the platform-independent specification is shown first in Figure E.1. Figure E.2 then delineates the platform-dependent Class model. The attributes and methods of each class in the diagrams are summarized in Section E.1; whereas Section E.2 presents the implementation details.

E.1 Attributes and Methods

Due to space limitation, attributes and methods of the relevant classes in Figures E.1 and E.2 are summarized here for better clarity. These attributes and methods correspond to those in Figures E.1 and E.2, and not those in the source code that may differ due to class relationships and an impact of language specifics. Unless stated otherwise, all class descriptions are applicable to both figures.

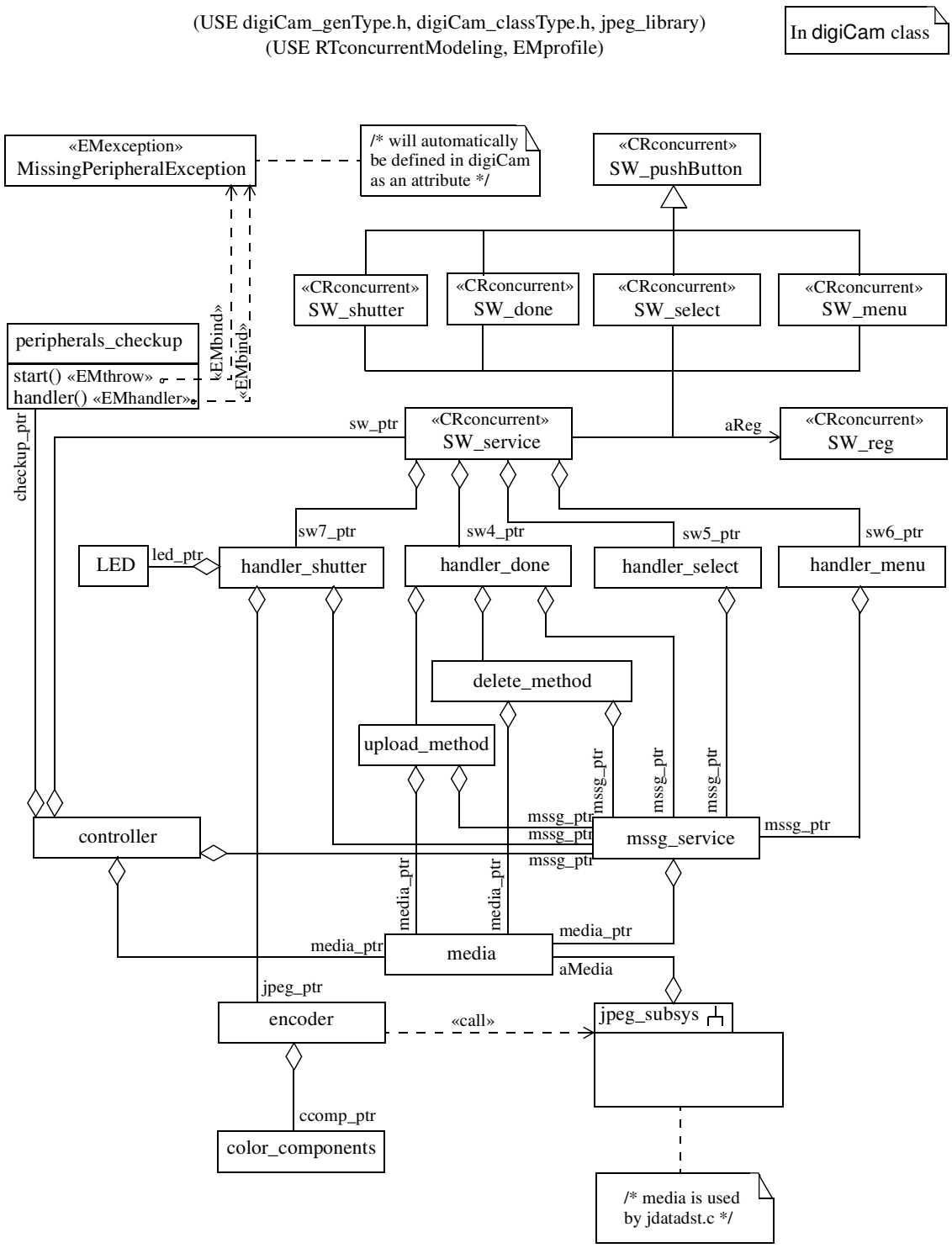


Figure E.1: Platform-independent Class diagram for the digital camera system

(USE digiCam_genType.h, digiCam_classType.h, jpeg_library, EMprofile, IMprofile)

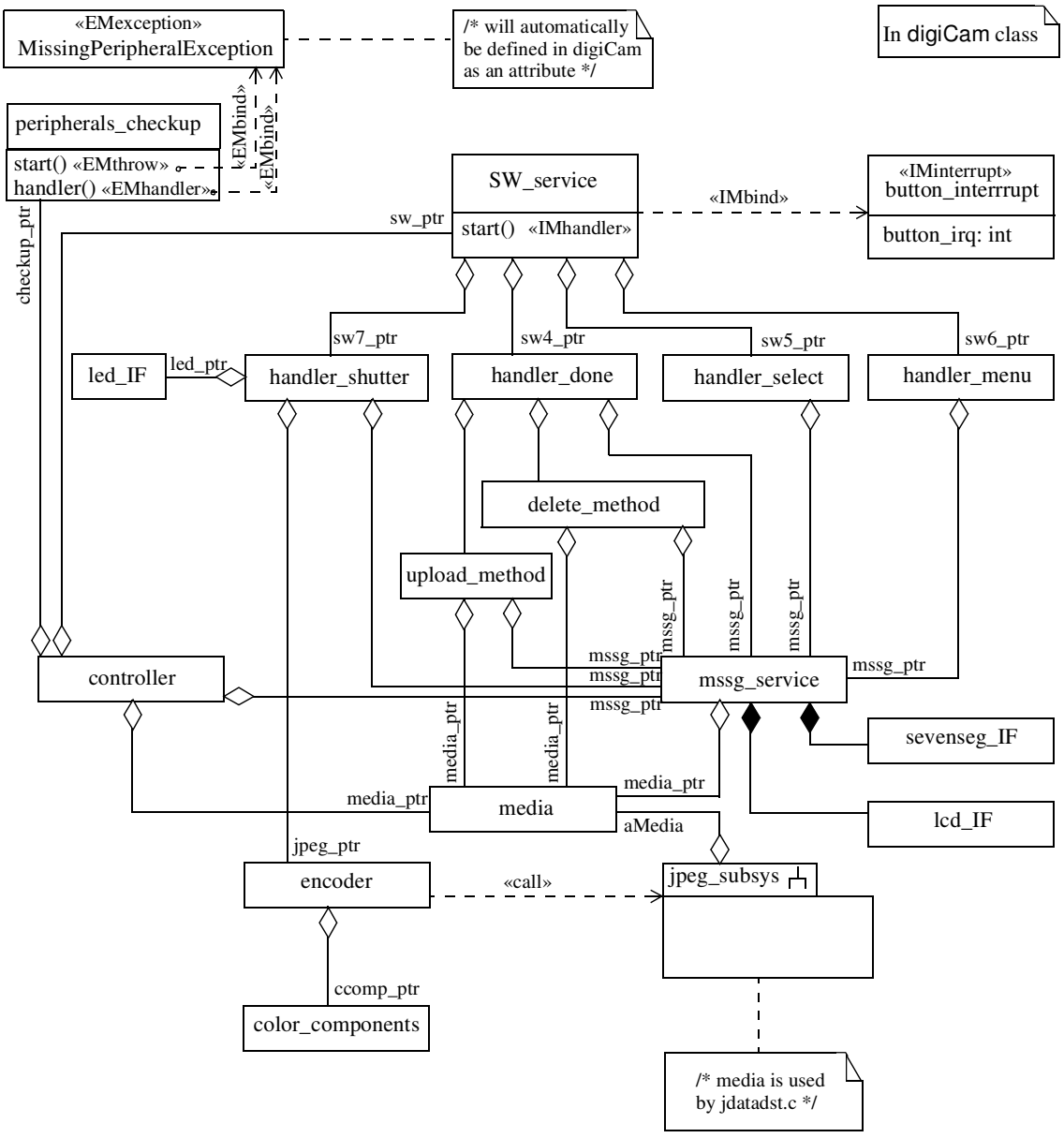


Figure E.2: Detail-minimal platform-specific Class diagram describing the digital camera system

buttonInterrupt

Attributes	Methods
+button_irq: int	/* none */

color_components

Attributes	Methods
+inbuff_base_addr: unsigned char * #current_sample_ptr: unsigned char *	+get_current_sample_ptr():unsigned char * +set_current_sample_ptr(val: unsigned char *) +reset_current_sample_ptr() +offset_current_sample_ptr(offset: int)

controller

Attributes	Methods
/* none */	+start() -wait_on_interrupt()

delete_method

Attributes	Methods
/* none */	+start()

encoder

Attributes	Methods
+compress_quality: int +shot_type: SHOTMODE +cinfo: struct jpeg_compress_struct +jerr: struct jpeg_error_mgr	+start()

handler_done

Attributes	Methods
/* none */	+start()

handler_menu

Attributes	Methods
/* none */	+start()

handler_select

Attributes	Methods
/* none */	+start()

handler_shutter

Attributes	Methods
/* none */	+start()

lcd_IF

Attributes	Methods
/* none */	+showTxt(txt: char *) +config()

LED

Attributes	Methods
/* none */	+on() +off()

led_IF

Attributes	Methods
/* none */	+on() +off()

media

Attributes	Methods
+image_cnt: int +current_offset: int +addr_offset: unsigned short * +image_size: unsigned short *	/* none */

mssg_service

Attributes	Methods
+message_txt: MESSAGE_ID	+display() /* following are applicable to Figure E.1 only */ +write_lcd_txt(str: char *) +write_sevenseg_num(num: int)

peripherals_checkup

Attributes	Methods
/* none */	+start() «EMthrowMethod» +handler() «EMhandler»

sevenseg_IF

Attributes	Methods
/* none */	+showNum(num: int)

shutter_reg (Figure E.1 Only)

Attributes	Methods
#id: int[2]	+start() +get_val(): int +set_val(id: int)

SW_pushButton (Figure E.1 Only)

Attributes	Methods
#id: int	+start() +set_id(val: int)

SW_service

Attributes	Methods
/* following applied to Figure E.2 */ #sw_pio: np_pio *	+start() /* the following applied to Figure E.2 only */ +clear_pio() +prepare_pio() +setup_isr()

upload_method

Attributes	Methods
/* none */	+start()

E.2 Implementation Details

This section lists the C implementation of the platform-dependent specification of the digital camera system. To achieve a better correlation between the source code and the UML diagram, the translation process adheres to the following rules:

- The C *struct* is used to represent the *class* concept.
- Visibility could be seen in the Class diagram, but effectively disappears in the code.
- Methods are represented as function pointers in the *struct*.
- To enable the *struct* method to access its struct-scope data and operations, the *struct* itself is passed automatically, *by reference*, into the method implementation.
- The code generator also automatically inserts the init() function into each *struct* (class). This function is responsible for binding the *struct* method (function pointer) to its implementation.

This structural C code implementation leads to an estimate 85% increase in code size compared with a more traditional approach.

digiCam_system.c

```
1 /*
2 !-- ------------------------------------------------------------ --
3 !--                                                              --
4 !-- File name:          digiCam_system.c                         --
5 !-- Author:             Chonlameth Arpnikanondt                  --
6 !-- Last revised:       10/10/03                                 --
7 !-- Description:        This is the main system file for         --
8 !--                     the digital camera system.               --
9 !--                                                              --
10 !--                    This file is mapped from the UML digiCam --
11 !--                    package, where the required digiCam software --
12 !--                    and hardware components reside. The main --
13 !--                    function is chiefly responsible for      --
14 !--                    initializing these components, and call  --
15 !--                    the system's controller.                 --
16 !--                                                             --
17 !-- ------------------------------------------------------------ --
18 */
19
20 #include <stdio.h>
21 #include "cdjpeg.h"
22 #include "nios.h"
23 #include "pio_lcd16207.h"
24 #include "digiCam_classType.h"
25
26 /* digiCam system attributes */
27 // main state
28 DIGICAM_STATE current_state = STATE_READY;
29 // menu
30 MENU_STATE current_menu_state = STATE_MENU_ENTRY;
31 MENU_STATE next_menu_state = STATE_MENU_ENTRY;
32 // image quality
33 IMGQUAL_STATE current_imgqual_state = STATE_IMGQUAL_ENTRY;
34 IMGQUAL_STATE next_imgqual_state = STATE_IMGQUAL_ENTRY;
35 // shot mode
36 SHOTMODE_STATE current_shotmode_state = STATE_SHOTMODE_ENTRY;
37 SHOTMODE_STATE next_shotmode_state = STATE_SHOTMODE_ENTRY;
38 // upload
39 UPLOAD_STATE current_upload_state = STATE_UPLOAD_ENTRY;
40 UPLOAD_STATE next_upload_state = STATE_UPLOAD_ENTRY;
41 // delete
42 DELETE_STATE current_delete_state = STATE_DELETE_ENTRY;
43 DELETE_STATE next_delete_state = STATE_DELETE_ENTRY;
44
45 int backto_Main = FALSE;
```

```
46 int in_Menu = FALSE;
47 int in_Select = FALSE;
48 int in_Delete = FALSE;
49
50 SHOTMODE SHOT_DEFAULT = ONESHOT;
51 MESSAGE_ID TXT_DEFAULT = READY_TXT;
52 int QUAL_DEFAULT = QUAL_NORMAL;
53
54 /* UML classes that are defined as digiCam attributes */
55 struct led_IF aLED;
56 struct media aMedia;
57 struct lcd_IF aLCD;
58 struct sevenseg_IF aSevenseg;
59 struct mssg_service aMssgService;
60 struct color_components aColorComp;
61 struct peripherals_checkup aSysTest;
62 struct encoder anEncoder;
63 struct upload_method anUploadOp;
64 struct delete_method aDeleteOp;
65 struct handler_shutter aShutterIsr;
66 struct handler_menu aMenuIsr;
67 struct handler_select aSelectIsr;
68 struct handler_done aDoneIsr;
69 struct button_interrupt aButtonIrq;
70 struct SW_service aButtonIsr;
71 struct controller aController;
72
73 /* digiCam methods */
74
75 void reset_next_states(int isBackToMain)
76 {
77        // Reset all relevant next states to ENTRY state
78        next_imgqual_state = STATE_IMGQUAL_ENTRY;
79        next_shotmode_state = STATE_SHOTMODE_ENTRY;
80        next_upload_state = STATE_UPLOAD_ENTRY;
81        next_delete_state = STATE_DELETE_ENTRY;
82
83        if(isBackToMain)
84        next_menu_state = STATE_MENU_ENTRY;
85 }
86
87 void configure_digiCam_system()
88 {
89        /* initial input buffer address */
90        char* inbuff_addr = ((char *) nasys_main_flash + IN_BUFFER_OFFSET);
91        /* Button pio address */
```

```
92      np_pio* pio_addr = na_button_pio;
93      /* Button irq number */
94      int button_irq = na_button_pio_irq;
95
96      /* bind init */
97      (aMedia.init) = (& media_init);
98      (aLCD.init) = (& lcd_init);
99      (aSevenseg.init) = (& sevenseg_init);
100     (aMssgService.init) = (& mssg_service_init);
101     (aColorComp.init) = (& color_components_init);
102     (aSysTest.init) = (& peripherals_checkup_init);
103     (anEncoder.init) = (& encoder_init);
104     (aShutterIsr.init) = (& handler_shutter_init);
105     (anUploadOp.init) = (& upload_method_init);
106     (aDeleteOp.init) = (& delete_method_init);
107     (aDoneIsr.init) = (& handler_done_init);
108     (aSelectIsr.init) = (& handler_select_init);
109     (aMenuIsr.init) = (& handler_menu_init);
110     (aButtonIsr.init) = (& SW_service_init);
111     (aController.init) = (& controller_init);
112
113     /* then call init to configure the digiCam system */
114     /* to simulate digiCam op, aMedia will be called later */
115     // (aMedia.init)(&aMedia);
116     (aLCD.init)(&aLCD);
117     (aSevenseg.init)(&aSevenseg);
118     (aMssgService.init)(&aMssgService, &aMedia, TXT_DEFAULT);
119     (aColorComp.init)(&aColorComp, (unsigned char*) inbuff_addr);
120     (aSysTest.init)(&aSysTest);
121     (anEncoder.init)(&anEncoder, &aColorComp, QUAL_DEFAULT, SHOT_DEFAULT);
122     (aShutterIsr.init)(&aShutterIsr, &aMssgService, &aLED, &anEncoder);
123     (anUploadOp.init)(&anUploadOp, &aMedia, &aMssgService);
124     (aDeleteOp.init)(&aDeleteOp, &aMedia, &aMssgService);
125     (aDoneIsr.init)(&aDoneIsr, &anEncoder, &aDeleteOp, &anUploadOp, &aMssgService);
126     (aSelectIsr.init)(&aSelectIsr, &aMssgService);
127     (aMenuIsr.init)(&aMenuIsr, &aMssgService);
128     (aButtonIsr.init)(&aButtonIsr, &aShutterIsr, &aMenuIsr, &aSelectIsr, &aDoneIsr,
129     &aButtonIrq, pio_addr, button_irq);
130     (aController.init)(&aController, &aSysTest, &aMedia, &aMssgService, &aButtonIsr);
131 }
132
133 int main()
134 {
135     configure_digiCam_system();
136     (aController.start)(&aController);
137     return 0; }
```

digiCam_genType.h

```
1 /*
2 !-- ------------------------------------------------------------ --
3 !--                                                              --
4 !-- File name:          digiCam_genType.h                        --
5 !-- Author:             Chonlameth Arpnikanondt                  --
6 !-- Last revised:       10/10/03                                 --
7 !-- Description:        This file contains various definitions   --
8 !--                     and pre-defined parameter values required --
9 !--                     for a successful operation of the digiCam --
10 !--                    system.                                  --
11 !--                                                             --
12 !-- ------------------------------------------------------------ --
13 */
14
15 #ifndef DIGICAM_GENTYPE_H
16 #define DIGICAM_GENTYPE_H
17
18
19 /* Needed for platform-independent specs */
20
21 /* Boolean types */
22 #define TRUE 1
23 #define FALSE 0
24
25 /* input image related */
26 #define IMG_HEIGHT 160 /* Test with 149 */
27 #define IMG_WIDTH 160 /* Test with 227 */
28 #define LINE_LENGTH 480 /* IMG_WIDTH * COLORCOMP_DEFAULT */
29 #define COLORCOMP_DEFAULT 3
30
31 /* buffer and memory related */
32 /* only needed for the platform-independent specs */
33 // #define INPUT_BUFFER_SIZE 0x20000 /* 64K buffer */
34 // #define MEDIA_BUFFER_SIZE 0x80000 /* 250K buffer */
35
36 #define UPLOAD_MARK -99
37 #define QUAL_NORMAL 65
38 #define QUAL_GOOD 90
39 #define MAX_IMG_NUM 16
40 #define EVEN_BIT 0x00000001
41
42 /* State variables */
```

```
43 typedef enum{         STATE_ENTRY,
44                       STATE_READY,
45                       STATE_MENU,
46                       STATE_ERROR} DIGICAM_STATE;
47 typedef enum{         STATE_MENU_ENTRY,
48                       STATE_MENU_IMGQUAL,
49                       STATE_MENU_SHOTMODE,
50                       STATE_MENU_UPLOAD,
51                       STATE_MENU_DELETE} MENU_STATE;
52 typedef enum{         STATE_IMGQUAL_ENTRY,
53                       STATE_IMGQUAL_NORMAL,
54                       STATE_IMGQUAL_GOOD} IMGQUAL_STATE;
55 typedef enum{         STATE_SHOTMODE_ENTRY,
56                       STATE_SHOTMODE_ONESHOT,
57                       STATE_SHOTMODE_TWOSHOT} SHOTMODE_STATE;
58 typedef enum{         STATE_UPLOAD_ENTRY,
59                       STATE_UPLOAD_NO,
60                       STATE_UPLOAD_YES} UPLOAD_STATE;
61 typedef enum{         STATE_DELETE_ENTRY,
62                       STATE_DELETE_NO,
63                       STATE_DELETE_YES} DELETE_STATE;
64
65 /* message identifiers */
66 typedef enum{         READY_TXT,
67                       MENU_ENTRY_TXT,
68                       IMGQUAL_TXT,
69                       SHOTMODE_TXT,
70                       UPLOAD_TXT,
71                       DELETE_TXT,
72                       IMGQUAL_NORMAL_TXT,
73                       IMGQUAL_GOOD_TXT,
74                       SHOTMODE_ONESHOT_TXT,
75                       SHOTMODE_TWOSHOT_TXT,
76                       UPLOAD_YES_TXT,
77                       UPLOAD_NO_TXT,
78                       DELETE_YES_TXT,
79                       DELETE_NO_TXT} MESSAGE_ID;
80
81 /* shotmode option */
82 typedef enum{         ONESHOT,
83                       TWOSHOT} SHOTMODE;
84
85 /* ON/OFF status */
```

```
86 /* Only for platform-independent specs */
87 // typedef enum{OFF, ON} ONOFF_STATUS;
88
89
90 /* Following are needed for the platform-dependent specs */
91
92 /* Switches ... all switch-related parameters begin with SW */
93 #define SW4 4094 /* value captured when pressed */
94 #define SW5 4093
95 #define SW6 4091
96 #define SW7 4087
97
98 /* delay loops */
99 #define LONG_LOOP 6666000
100 #define SHORT_LOOP 666600
101
102 /* address offsets */
103 #define IMG_ADDR_OFFSET 0x0
104 #define IMG_SIZE_OFFSET 0x20
105 #define IMG_DATA_OFFSET 0x20000
106 #define IN_BUFFER_OFFSET 0x4000
107 #define FLASH_SECTOR_SIZE 0x10000
108
109 /* exception signal */
110 typedef int digiCam_exception;
111 typedef digiCam_exception MissingPeripheralException;
112
113 #endif // DIGICAM_GENTYPE_H
114
```

digiCam_classType.h

```
1 /*
2 !-- ------------------------------------------------------------ --
3 !--                                                              --
4 !-- File name:          digiCam_classType.h                      --
5 !-- Author:             Chonlameth Arpnikanondt                  --
6 !-- Last revised:       10/10/03                                 --
7 !-- Description:        This file contains class definitions     --
8 !--                     as specified in the UML document. In C,  --
9 !--                     these classes are implemented as structures. --
10 !--                                                             --
11 !-- ------------------------------------------------------------ --
12 */
13
14 #ifndef DIGICAM_CLASSTYPE_H
15 #define DIGICAM_CLASSTYPE_H
16
17 #include "digiCam_genType.h"
18
19 struct led_IF {
20        // attributes
21        //
22        // methods
23        void (*on) ();
24        void (*off) ();
25 };
26
27 struct media {
28        // attributes
29        int image_cnt;
30        int current_offset;
31        unsigned short *addr_offset;
32        unsigned short *image_size;
33        // methods
34        void (*init) (struct media *self);
35 };
36
37 struct lcd_IF {
38        // attributes
39        //
40        // methods
41        void (*showTxt)(char* txt);
42        void (*config)();
```

```
43        void (*init)(struct lcd_IF *self);
44 };
45
46 struct sevenseg_IF {
47        // attributes
48        //
49        // methods
50        void (*showNum) (int num);
51        void (*init)(struct sevenseg_IF *self);
52 };
53
54 struct mssg_service {
55        // attributes
56        MESSAGE_ID message_txt;
57        struct lcd_IF lcd;
58        struct sevenseg_IF sevenseg;
59        struct media *media_ptr;
60        // methods
61        void (*display) (struct mssg_service * self);
62        void (*write_lcd_txt) (struct mssg_service * self, char* str);
63        void (*write_sevenseg_num) (struct mssg_service * self, int num);
64        void (*init) (struct mssg_service * self,
65                  struct media *media_ptr,
66                  MESSAGE_ID txt);
67 };
68
69 struct color_components {
70        // attributes
71        unsigned char *inbuff_base_addr;
72        unsigned char *current_sample_ptr;
73        // methods
74        unsigned char* (*get_current_sample_ptr) (struct color_components* self);
75        void (*set_current_sample_ptr) (struct color_components* self, unsigned char* val);
76        void (*reset_current_sample_ptr) (struct color_components* self);
77        void (*offset_current_sample_ptr) (struct color_components* self, int offset);
78        void (*init)(struct color_components *self, unsigned char* iv);
79 };
80
81 struct peripherals_checkup {
82        // attributes
83        //
84        // methods
85        void (*start) (struct peripherals_checkup* self);
```

```
86      MissingPeripheralException (*checkup) ();
87      void (*handler) ();
88      void (*init)(struct peripherals_checkup *self);
89 };
90
91 /* Jpeg encoder */
92 struct encoder {
93      // attributes
94      int compress_quality;
95      SHOTMODE shot_type;
96      struct jpeg_compress_struct cinfo;
97      struct jpeg_error_mgr jerr;
98      struct color_components* ccomp_ptr;
99      // methods
100     void (*start)(struct encoder *self);
101     void (*init)(struct encoder *self,
102     struct color_components *ccomp_ptr,
103     int qual_iv, SHOTMODE shottype_iv);
104 };
105
106 struct handler_shutter {
107     // attributes
108     struct mssg_service * mssg_ptr;
109     struct led_IF * led_ptr;
110     struct encoder * jpeg_ptr;
111     // methods
112     void (*start)(struct handler_shutter *self);
113     void (*init)(struct handler_shutter *self,
114     struct mssg_service *mssg_ptr,
115     struct led_IF *led_ptr,
116     struct encoder *jpeg_ptr);
117 };
118
119 /* upload */
120 struct upload_method {
121     // attributes
122     struct media * media_ptr;
123     struct mssg_service * mssg_ptr;
124     // methods
125     void (*start)(struct upload_method *self);
126     void (*init)(struct upload_method *self,
127             struct media *media_ptr,
128             struct mssg_service *mssg_ptr);
```

```
129 };
130
131 /* delete */
132 struct delete_method {
133       // attributes
134       struct media * media_ptr;
135       struct mssg_service * mssg_ptr;
136       // methods
137       void (*start)(struct delete_method *self);
138       void (*init)(struct delete_method *self,
139                   struct media *media_ptr,
140                   struct mssg_service *mssg_ptr);
141 };
142
143 /* handler done can be defined now */
144 struct handler_done {
145       // attributes
146       struct encoder * jpeg_ptr;
147       struct delete_method * del_ptr;
148       struct upload_method * send_ptr;
149       struct mssg_service *mssg_ptr;
150       // methods
151       void (*start)(struct handler_done *self);
152       void (*init)(struct handler_done *self,
153                   struct encoder *jpeg_ptr,
154                   struct delete_method *del_ptr,
155                   struct upload_method *send_ptr,
156                   struct mssg_service *mssg_ptr);
157 };
158
159 /* handler select */
160 struct handler_select {
161       // attributes
162       struct mssg_service *mssg_ptr;
163       // methods
164       void (*start)(struct handler_select *self);
165       void (*init)(struct handler_select *self, struct mssg_service *mssg_ptr);
166 };
167
168 /* handler menu */
169 struct handler_menu {
170       // attributes
171       struct mssg_service *mssg_ptr;
```

```
172     // methods
173     void (*start)(struct handler_menu *self);
174     void (*init)(struct handler_menu *self, struct mssg_service *mssg_ptr);
175 };
176
177 /* interrupt service routine class */
178 struct button_interrupt {
179     // attributes
180     int button_irq;
181     // methods
182 };
183
184 struct SW_service {
185     // attributes
186     // int current_id; // will use the nios routine instead
187     np_pio *sw_pio;
188     struct handler_shutter * sw7_ptr;
189     struct handler_menu * sw6_ptr;
190     struct handler_select * sw5_ptr;
191     struct handler_done * sw4_ptr;
192     struct button_interrupt * button_ptr;
193     // methods
194     // void (*set_current_id)(struct SW_service *self, int val); // not needed anymore
195     void (*clear_pio)(struct SW_service *self);
196     void (*prepare_pio)(struct SW_service *self);
197     void (*setup_isr)(struct SW_service *self);
198     void (*start)(int self_ptr);
199     void (*init)(struct SW_service *self,
200             struct handler_shutter *sw7_ptr,
201             struct handler_menu *sw6_ptr,
202             struct handler_select *sw5_ptr,
203             struct handler_done *sw4_ptr,
204             struct button_interrupt *button_ptr,
205             np_pio* pio_addr, int button_irq);
206 };
207
208 /* controller class */
209 struct controller {
210     // attributes
211     struct peripherals_checkup *checkup_ptr;
212     struct media *media_ptr;
213     struct mssg_service *mssg_ptr;
214     struct SW_service *sw_ptr;
```

```
215     // methods
216     void (*start)(struct controller *self);
217     void (*wait_on_interrupt)(struct controller *self);
218     void (*init)(struct controller *self,
219                 struct peripherals_checkup *checkup_ptr,
220                 struct media *media_ptr,
221                 struct mssg_service *mssg_ptr,
222                 struct SW_service *sw_ptr);
223 };
224
225 /* function prototypes */
226
227 /* Media struct initialization */
228 extern void media_init(struct media *self);
229
230 /* LCD display */
231 extern void lcd_init(struct lcd_IF *self);
232
233 /* Seven-segment display */
234 extern void sevenseg_init(struct sevenseg_IF *self);
235
236 /* mssg_service interface */
237 extern void mssg_service_init(struct mssg_service * self,
238                 struct media * media_ptr,
239                 MESSAGE_ID txt);
240
241 /* color_components standard get and set methods */
242 extern void color_components_init(struct color_components *self, unsigned char* iv);
243
244 /* peripherals checkup */
245 extern void peripherals_checkup_init(struct peripherals_checkup *self);
246
247 /* Jpeg encoder */
248 extern void encoder_init(struct encoder *self,
249                 struct color_components *ccomp_ptr,
250                 int qual_iv, SHOTMODE shottype_iv);
251
252 /* shutter isr service */
253 extern void handler_shutter_init(struct handler_shutter *self,
254                 struct mssg_service *mssg_ptr,
255                 struct led_IF *led_ptr,
256                 struct encoder *jpeg_ptr);
257
```

```
258 /* upload operation */
259 extern void upload_method_init(struct upload_method *self,
260                 struct media *media_ptr,
261                 struct mssg_service *mssg_ptr);
262
263 /* delete-all operation */
264 extern void delete_method_init(struct delete_method *self,
265                 struct media *media_ptr,
266                 struct mssg_service *mssg_ptr);
267
268 /* handler done */
269 extern void handler_done_init(struct handler_done *self,
270                 struct encoder *jpeg_ptr,
271                 struct delete_method *del_ptr,
272                 struct upload_method *send_ptr,
273                 struct mssg_service *mssg_ptr);
274
275 /* handler select service */
276 extern void handler_select_init(struct handler_select *self,
277                 struct mssg_service *mssg_ptr);
278
279 /* menu service */
280 extern void handler_menu_init(struct handler_menu *self,
281                 struct mssg_service *mssg_ptr);
282
283 /* SWITCH services */
284 extern void SW_service_init(struct SW_service *self,
285                 struct handler_shutter *sw7_ptr,
286                 struct handler_menu *sw6_ptr,
287                 struct handler_select *sw5_ptr,
288                 struct handler_done *sw4_ptr,
289                 struct button_interrupt *button_ptr,
290                 np_pio* pio_addr, int button_irq);
291
292 /* begins the controller implementation */
293 extern void controller_init(struct controller *self,
294                 struct peripherals_checkup *checkup_ptr,
295                 struct media *media_ptr,
296                 struct mssg_service *mssg_ptr,
297                 struct SW_service *sw_ptr);
298
299 #endif // DIGICAM_CLASSTYPE_H
300
```

digiCam_classOp.c

```
1 /*
2 !-- ------------------------------------------------------------      --
3 !--                                                                    --
4 !-- File name:          digiCam_classOp.c                            --
5 !-- Author:             Chonlameth Arpnikanondt                      --
6 !-- Last revised:       10/10/03                                     --
7 !-- Description:        This file contains class methods             --
8 !--                     implementation.                              --
9 !--                                                                  --
10 !-- ------------------------------------------------------------     --
11 */
12
13
14 #include <stdio.h>
15 #include "cdjpeg.h"
16 #include "nios.h"
17 #include "pio_lcd16207.h"
18 #include "digiCam_classType.h"
19
20 extern reset_next_states(int isBackToMain);
21
22 /* LED operations */
23 void on()
24 {
25        na_led_pio->np_piodirection = 3; /* set direction: output */
26        na_led_pio->np_piodata = 3; /* both on */
27 }
28
29 void off()
30 {
31        na_led_pio->np_piodirection = 3; /* set direction: output */
32        na_led_pio->np_piodata = 0; /* both off */
33 }
34
35 /* won't be used in platform-dependent */
36 /*
37 void blink()
38 {
39        int i;
40
41        na_led_pio->np_piodirection = 3;
42        for(i=0;i<5;i++) {
```

```
43        na_led_pio->np_piodata = 1;
44        nr_delay(200);
45        na_led_pio->np_piodata = 2;
46        nr_delay(200);
47        }
48 }
49 */
50
51 /* Media struct initialization */
52 void media_init(struct media *self)
53 {
54        int i, cnt, sz_tmp, offset_tmp;
55
56        /* initialize addresses */
57        self->image_cnt = 0;
58        self->current_offset = IMG_DATA_OFFSET;
59        self->addr_offset = (unsigned short *)((char *)nasys_main_flash +
60                                IMG_ADDR_OFFSET);
61        self->image_size = (unsigned short *)((char *)nasys_main_flash +
62                                IMG_SIZE_OFFSET);
63
64        /* Then check memory content so that correct values can be filled in */
65        i = 0;
66        cnt = 0;
67        while(0xFFFF != self->addr_offset[i]) {
68                cnt++;
69                i += 2;
70        }
71        self->image_cnt = cnt;
72        // Fetch values only when there is image in memory
73        // Otherwise, just init
74        if(cnt != 0) {
75                sz_tmp = (self->image_size[2*cnt-1]<<16) | (self->image_size[2*(cnt-1)]);
76                offset_tmp = (self->addr_offset[2*cnt-1]<<16) |
77                (self->addr_offset[2*(cnt-1)]);
78                if( (sz_tmp & EVEN_BIT) == 1) sz_tmp++;
79                self->current_offset = offset_tmp + sz_tmp;
80        }
81 }
82
83 /* LCD display */
84 void showTxt(char * txt)
85 {
```

```
86        nr_pio_lcdwritescreen(txt);
87 }
88
89 void lcd_config()
90 {
91        nr_pio_lcdinit(na_lcd_pio);
92        nr_pio_lcdwritescreen(" ");
93 }
94
95 void lcd_init(struct lcd_IF *self)
96 {
97        (self->showTxt) = (&showTxt);
98        (self->config) = (&lcd_config);
99 }
100
101 /* Seven-segment display */
102 void showNum(int num)
103 {
104       nr_pio_showhex((num / 10) * 16 + num % 10);
105 }
106
107 void sevenseg_init(struct sevenseg_IF *self)
108 {
109       (self->showNum) = (&showNum);
110 }
111
112 /* mssg_service interface */
113
114 void write_lcd_txt(struct mssg_service * self, char* str)
115 {
116       ((self->lcd).showTxt)(str);
117 }
118
119 void write_sevenseg_num(struct mssg_service * self, int num)
120 {
121       ((self->sevenseg).showNum)(num);
122 }
123
124 void display (struct mssg_service * self)
125 {
126       int i;
127       int LESS_LONG_LOOP = LONG_LOOP-500;
128
```

```
129     ((self->sevenseg).showNum)((self->media_ptr)->image_cnt);
130     switch(self->message_txt) {
131         case MENU_ENTRY_TXT:
132             // Menu greeting text
133             ((self->lcd).showTxt)("Menu Mode: SW6: Browse");
134             for(i=0;i<LONG_LOOP;i++);
135             ((self->lcd).showTxt)("SW5: Select SW4: Done");
136             for(i=0;i<LESS_LONG_LOOP;i++);
137             break;
138         case IMGQUAL_TXT:
139             ((self->lcd).showTxt)("Image Quality..");
140             break;
141         case SHOTMODE_TXT:
142             ((self->lcd).showTxt)("Shot Mode..");
143             break;
144         case UPLOAD_TXT:
145             ((self->lcd).showTxt)("UPLOAD..");
146             break;
147         case DELETE_TXT:
148             ((self->lcd).showTxt)("DELETE..");
149             break;
150         case IMGQUAL_NORMAL_TXT:
151             ((self->lcd).showTxt)("Image Quality - Normal");
152             break;
153         case IMGQUAL_GOOD_TXT:
154             ((self->lcd).showTxt)("Image Quality - Good");
155             break;
156         case SHOTMODE_ONESHOT_TXT:
157             ((self->lcd).showTxt)("Single Shot");
158             break;
159         case SHOTMODE_TWOSHOT_TXT:
160             ((self->lcd).showTxt)("Burst - Two-Shot");
161             break;
162         case UPLOAD_YES_TXT:
163             ((self->lcd).showTxt)("UPLOAD - YES");
164             break;
165         case UPLOAD_NO_TXT:
166             ((self->lcd).showTxt)("UPLOAD - NO");
167             break;
168         case DELETE_YES_TXT:
169             ((self->lcd).showTxt)("DELETE - YES");
170             break;
171         case DELETE_NO_TXT:
```

```
172                 ((self->lcd).showTxt)("DELETE - NO");
173                 break;
174           case READY_TXT:
175           default:
176                 // READY_MESSAGE
177                 ((self->lcd).showTxt)("MAIN> SW7: SHOOTSW6: Menu");
178                 for(i=0;i<LESS_LONG_LOOP;i++);
179           }
180           nr_delay(500);
181 }
182
183 void mssg_service_init(struct mssg_service *self,
184                             struct media *media_ptr,
185                             MESSAGE_ID txt)
186 {
187       self->message_txt = txt;
188       (self->display) = (&display);
189       (self->write_lcd_txt) = (&write_lcd_txt);
190       (self->write_sevenseg_num) = (&write_sevenseg_num);
191       ((self->lcd).init) = (&lcd_init);
192       ((self->sevenseg).init) = (&sevenseg_init);
193       self->media_ptr = media_ptr;
194
195       ((self->lcd).init)(& (self->lcd));
196       ((self->sevenseg).init)(& (self->sevenseg));
197       ((self->lcd).config)();
198       ((self->sevenseg).showNum)((self->media_ptr)->image_cnt);
199 }
200
201 /* color_components standard get and set methods */
202 unsigned char* get_current_sample_ptr(struct color_components* self)
203 {
204       return self->current_sample_ptr;
205 }
206
207 void set_current_sample_ptr(struct color_components* self, unsigned char* val)
208 {
209       self->current_sample_ptr = val;
210 }
211
212 void reset_current_sample_ptr(struct color_components* self)
213 {
214       self->current_sample_ptr = self->inbuff_base_addr;
```

```
215 }
216
217 void offset_current_sample_ptr(struct color_components* self, int offset)
218 {
219     self->current_sample_ptr += offset;
220 }
221
222 void color_components_init(struct color_components *self, unsigned char* iv)
223 {
224     self->inbuff_base_addr = iv;
225     self->current_sample_ptr = iv;
226     (self->get_current_sample_ptr) = (& get_current_sample_ptr);
227     (self->set_current_sample_ptr) = (& set_current_sample_ptr);
228     (self->reset_current_sample_ptr) = (& reset_current_sample_ptr);
229     (self->offset_current_sample_ptr) = (& offset_current_sample_ptr);
230 }
231
232 /* peripherals checkup */
233 MissingPeripheralException checkup()
234 {
235     unsigned char peripheral_status = 0x00;
236     MissingPeripheralException err = FALSE;
237
238     #ifdef na_seven_seg_pio
239     peripheral_status = 0x01; // set first bit 0001
240     #endif
241
242     #ifdef na_lcd_pio
243     peripheral_status |= 0x02; // set second bit 0010
244     #endif
245
246     #ifdef na_button_pio
247     peripheral_status |= 0x04; // set third bit 0100
248     #endif
249
250     #ifdef na_ext_flash
251     peripheral_status |= 0x08; // set fourth bit 1000
252     #endif
253
254     #ifdef na_led_pio
255     peripheral_status |= 0x10; // set fifth bit 1000
256     #endif
257
```

```
258     if(peripheral_status != 0x1F) err = TRUE;
259
260     return err;
261 }
262
263 void peripherals_checkup_handler()
264 {
265     int i;
266
267     nr_pio_lcdwritescreen("Device Error!");
268
269     na_led_pio->np_piodirection = 3; // set direction: output
270     na_led_pio->np_piodata = 3; // both off
271     for(i=0;i<5;i++) {
272             na_led_pio->np_piodata = 1; // turns one one
273             nr_delay(200);
274             na_led_pio->np_piodata = 2; // alternate
275             nr_delay(200);
276     }
277     na_led_pio->np_piodata = 3; // both off
278     nr_delay(200);
279
280     exit(0);
281 }
282
283 void peripherals_checkup_start(struct peripherals_checkup* self)
284 {
285     /* try & catch the throw clause C style */
286     if(self->checkup())
287             self->handler();
288     /* get to this point means it's okay... turns on LEDs */
289     na_led_pio->np_piodirection = 3;
290     na_led_pio->np_piodata = 3;
291 }
292
293 void peripherals_checkup_init(struct peripherals_checkup *self)
294 {
295     (self->checkup) = (& checkup);
296     (self->handler) = (& peripherals_checkup_handler);
297     (self->start) = (& peripherals_checkup_start);
298 }
299
300 /* Jpeg encoder */
```

```
301 void encoder_start(struct encoder *self)
302 {
303     JSAMPROW row_ptr[1];
304     int i, cnt;
305
306     ((self->ccomp_ptr)->reset_current_sample_ptr)(self->ccomp_ptr);
307     (self->cinfo).err = jpeg_std_error(& (self->jerr));
308     jpeg_create_compress(& (self->cinfo));
309
310     (self->cinfo).input_components = COLORCOMP_DEFAULT;
311     (self->cinfo).in_color_space = JCS_RGB;
312     (self->cinfo).image_height = IMG_HEIGHT;
313     (self->cinfo).image_width = IMG_WIDTH;
314
315     jpeg_set_defaults(& (self->cinfo));
316
317     (self->cinfo).dct_method = JDCT_ISLOW;
318     jpeg_stdio_dest(& (self->cinfo), stdout);
319     jpeg_set_quality(& (self->cinfo), (self->compress_quality), TRUE);
320
321     jpeg_start_compress(& (self->cinfo), TRUE);
322
323     cnt = 0;
324     while ((self->cinfo).next_scanline < (self->cinfo).image_height) {
325             row_ptr[0] = ((self->ccomp_ptr)->get_current_sample_ptr)(self->ccomp_ptr);
326             (void) jpeg_write_scanlines(& (self->cinfo), row_ptr, 1);
327       ((self->ccomp_ptr)->offset_current_sample_ptr)((self->ccomp_ptr), LINE_LENGTH);
328     }
329
330     jpeg_finish_compress(& (self->cinfo));
331     jpeg_destroy_compress(& (self->cinfo));
332 }
333
334 void encoder_init(struct encoder *self,
335             struct color_components *ccomp_ptr,
336             int qual_iv, SHOTMODE shottype_iv)
337 {
338     self->compress_quality = qual_iv;
339     self->shot_type = shottype_iv;
340     self->ccomp_ptr = ccomp_ptr;
341     (self->start) = (& encoder_start);
342 }
343
```

```
344
345 /* shutter isr service */
346 void handler_shutter_start(struct handler_shutter *self)
347 {
348     extern DIGICAM_STATE current_state;
349
350     switch(current_state) {
351         case STATE_READY:
352             ((self->led_ptr)->off)();
353             ((self->jpeg_ptr)->start)(self->jpeg_ptr);
354             if( (self->jpeg_ptr)->shot_type == TWOSHOT )
355                 ((self->jpeg_ptr)->start)(self->jpeg_ptr);
356             ((self->led_ptr)->on)();
357             ((self->mssg_ptr)->write_sevenseg_num)((self->mssg_ptr),
358             ((self->mssg_ptr)->media_ptr)->image_cnt);
359             break;
360         default:
361             ((self->mssg_ptr)->write_lcd_txt)((self->mssg_ptr), "Camera Not Ready..");
362             nr_delay(500);
363     }
364 }
365
366 void handler_shutter_init(struct handler_shutter *self,
367                           struct mssg_service *mssg_ptr,
368                           struct led_IF *led_ptr,
369                           struct encoder *jpeg_ptr)
370 {
371     self->mssg_ptr = mssg_ptr;
372     self->led_ptr = led_ptr;
373     self->jpeg_ptr = jpeg_ptr;
374     (self->start) = (& handler_shutter_start);
375     ((self->led_ptr)->on) = (& on);
376      ((self->led_ptr)->off) = (& off);
377 }
378
379 /* upload operation */
380 void upload_method_start(struct upload_method *self)
381 {
382     extern UPLOAD_STATE current_upload_state;
383     extern UPLOAD_STATE next_upload_state;
384     unsigned char *c;
385     unsigned int dataaddr;
386     int datasize = 0;
```

```
387     int i, j, k;
388
389     // Reset upload
390     current_upload_state = STATE_UPLOAD_NO;
391     next_upload_state = STATE_UPLOAD_YES;
392     ((self->mssg_ptr)->write_lcd_txt)((self->mssg_ptr), "Please wait. Uploading images...");
393
394     // for all images, transmit...
395     // start off with the marker
396     printf("%d\n", UPLOAD_MARK);
397     for(i=0,k=0; k<(self->media_ptr)->image_cnt; i+=2,k++) {
398             // for each image
399             // 1) get data address
400             dataaddr = (unsigned int) (((self->media_ptr)->addr_offset[i+1] << 16) |
401             ((self->media_ptr)->addr_offset[i]));
402             datasize = (((self->media_ptr)->image_size[i+1] << 16) |
403             ((self->media_ptr)->image_size[i]));
404             c = (unsigned char *) ((char *)nasys_main_flash + dataaddr);
405
406             printf("%d\n", datasize);
407             for(j=0;j<datasize;j++)
408                     printf("0x%x\n", c[j]);
409     }
410     // end mark
411     printf("%d\n", UPLOAD_MARK);
412
413     ((self->mssg_ptr)->write_lcd_txt)((self->mssg_ptr), "Done...");
414     nr_delay(1000);
415 }
416
417 void upload_method_init(struct upload_method *self,
418                         struct media *media_ptr,
419                         struct mssg_service *mssg_ptr)
420 {
421     self->media_ptr = media_ptr;
422     self->mssg_ptr = mssg_ptr;
423     (self->start) = (& upload_method_start);
424 }
425
426
427 /* delete-all operation */
428 void delete_method_start(struct delete_method *self)
429 {
```

```
430     extern DELETE_STATE current_delete_state;
431     extern DELETE_STATE next_delete_state;
432     int current_offset = IMG_DATA_OFFSET;
433     unsigned short * data_addr;
434     int i, image_cnt = (self->media_ptr)->image_cnt;
435
436     // This is a delete all operation
437     current_delete_state = STATE_DELETE_NO;
438     next_delete_state = STATE_DELETE_YES;
439     ((self->mssg_ptr)->write_lcd_txt)((self->mssg_ptr), "Deleting images...");
440
441     // if there is an image...
442     if(image_cnt > 0) {
443             data_addr = (unsigned short *)((char *)nasys_main_flash + current_offset);
444             nr_flash_erase_sector(nasys_main_flash, data_addr);
445             current_offset += FLASH_SECTOR_SIZE;
446             for(;;) {
447                 if(current_offset > (self->media_ptr)->current_offset)
448                     break;
449                 else {
450                     data_addr = (unsigned short *)((char *)nasys_main_flash+current_offset);
451                     nr_flash_erase_sector(nasys_main_flash, data_addr);
452                     current_offset += FLASH_SECTOR_SIZE;
453                 }
454             }
455     }
456
457     // now, image content is gone
458     // prepare to delete table of content @ nasys_main_flash
459     nr_flash_erase_sector(nasys_main_flash, nasys_main_flash);
460     // initialize the media content struct
461     ((self->media_ptr)->init)(self->media_ptr);
462
463     // adjust hex display...
464     for(i=image_cnt-1; i>=0; i--) {
465             ((self->mssg_ptr)->write_sevenseg_num)((self->mssg_ptr), i);
466             nr_delay(1000);
467     }
468     ((self->mssg_ptr)->write_lcd_txt)((self->mssg_ptr), "Done...");
469     nr_delay(1000);
470 }
471
472 void delete_method_init(struct delete_method *self,
```

```
473                 struct media *media_ptr,
474                 struct mssg_service *mssg_ptr)
475 {
476     self->media_ptr = media_ptr;
477     self->mssg_ptr = mssg_ptr;
478     (self->start) = (& delete_method_start);
479 }
480
481
482 /* handler done */
483 void handler_done_start(struct handler_done *self)
484 {
485     extern DIGICAM_STATE current_state;
486     extern IMGQUAL_STATE current_imgqual_state;
487     extern SHOTMODE_STATE current_shotmode_state;
488     extern UPLOAD_STATE current_upload_state;
489     extern DELETE_STATE current_delete_state;
490     extern int backto_Main;
491     extern int in_Menu;
492     extern int in_Select;
493
494     if((in_Menu && (!in_Select)) && (current_state == STATE_MENU)) {
495             // go to Ready
496             current_state = STATE_READY;
497             in_Menu = FALSE;
498             backto_Main = TRUE;
499             (self->mssg_ptr)->message_txt = READY_TXT;
500             // and reset all relevant next states
501             reset_next_states(backto_Main);
502     }
503     // else if coming from Select
504     else if(in_Select && (current_state == STATE_MENU)) {
505     // go back to Menu, leaves Select
506             in_Select = FALSE;
507             in_Menu = TRUE;
508             (self->mssg_ptr)->message_txt = MENU_ENTRY_TXT;
509             // and reset all but the STATE_MENU_ENTRY
510             reset_next_states(backto_Main);
511     }
512
513     // update parameters
514     if(current_imgqual_state == STATE_IMGQUAL_GOOD)
515             (self->jpeg_ptr)->compress_quality = QUAL_GOOD;
```

```
516     else if(current_imgqual_state == STATE_IMGQUAL_NORMAL)
517             (self->jpeg_ptr)->compress_quality = QUAL_NORMAL;
518
519     if(current_shotmode_state == STATE_SHOTMODE_ONESHOT)
520             (self->jpeg_ptr)->shot_type = ONESHOT;
521     else if(current_shotmode_state == STATE_SHOTMODE_TWOSHOT)
522             (self->jpeg_ptr)->shot_type = TWOSHOT;
523
524     if(current_upload_state == STATE_UPLOAD_YES)
525             ((self->send_ptr)->start)(self->send_ptr);
526     if(current_delete_state == STATE_DELETE_YES)
527             ((self->del_ptr)->start)(self->del_ptr);
528 }
529
530 void handler_done_init(struct handler_done *self,
531                         struct encoder *jpeg_ptr,
532                         struct delete_method *del_ptr,
533                         struct upload_method *send_ptr,
534                         struct mssg_service *mssg_ptr)
535 {
536     self->jpeg_ptr = jpeg_ptr;
537     self->del_ptr = del_ptr;
538     self->send_ptr = send_ptr;
539     self->mssg_ptr = mssg_ptr;
540     (self->start) = (& handler_done_start);
541 }
542
543 /* handler select service */
544 void handler_select_start(struct handler_select *self)
545 {
546     extern DIGICAM_STATE current_state;
547     extern MENU_STATE current_menu_state;
548     extern MENU_STATE next_menu_state;
549     extern IMGQUAL_STATE current_imgqual_state;
550     extern IMGQUAL_STATE next_imgqual_state;
551     extern SHOTMODE_STATE current_shotmode_state;
552     extern SHOTMODE_STATE next_shotmode_state;
553     extern UPLOAD_STATE current_upload_state;
554     extern UPLOAD_STATE next_upload_state;
555     extern DELETE_STATE current_delete_state;
556     extern DELETE_STATE next_delete_state;
557     extern int backto_Main;
558     extern int in_Menu;
```

```
559     extern int in_Select;
560
561     IMGQUAL_STATE tmp_imgqual_state;
562     SHOTMODE_STATE tmp_shotmode_state;
563     UPLOAD_STATE tmp_upload_state;
564     DELETE_STATE tmp_delete_state;
565
566     if(current_state == STATE_MENU) {
567
568             in_Select = TRUE;
569             switch(current_menu_state) {
570                 case STATE_MENU_ENTRY:
571                     in_Select = FALSE;
572                     break;
573                 case STATE_MENU_IMGQUAL:
574                     if(next_imgqual_state == STATE_IMGQUAL_ENTRY) {
575                         if(current_imgqual_state == STATE_IMGQUAL_ENTRY ||
576                         current_imgqual_state == STATE_IMGQUAL_NORMAL) {
577                             next_imgqual_state = STATE_IMGQUAL_GOOD;
578                             current_imgqual_state = STATE_IMGQUAL_NORMAL;
579                             (self->mssg_ptr)->message_txt = IMGQUAL_NORMAL_TXT;
580                             }
581                         else if(current_imgqual_state == STATE_IMGQUAL_GOOD) {
582                             next_imgqual_state = STATE_IMGQUAL_NORMAL;
583                             (self->mssg_ptr)->message_txt = IMGQUAL_GOOD_TXT;
584                             }
585                     }
586                     else {
587                         tmp_imgqual_state = current_imgqual_state;
588                         current_imgqual_state = next_imgqual_state;
589                         next_imgqual_state = tmp_imgqual_state;
590                         if(current_imgqual_state == STATE_IMGQUAL_NORMAL)
591                             (self->mssg_ptr)->message_txt = IMGQUAL_NORMAL_TXT;
592                         else
593                             (self->mssg_ptr)->message_txt = IMGQUAL_GOOD_TXT;
594                     }
595                     break;
596
597                 case STATE_MENU_SHOTMODE:
598                     if(next_shotmode_state == STATE_SHOTMODE_ENTRY) {
599                         if(current_shotmode_state == STATE_SHOTMODE_ENTRY ||
600                         current_shotmode_state == STATE_SHOTMODE_ONESHOT) {
601                             next_shotmode_state = STATE_SHOTMODE_TWOSHOT;
```

```
602                         current_shotmode_state = STATE_SHOTMODE_ONESHOT;
603                         (self->mssg_ptr)->message_txt = SHOTMODE_ONESHOT_TXT;
604                     }
605                     else if(current_shotmode_state == STATE_SHOTMODE_TWOSHOT) {
606                         next_shotmode_state = STATE_SHOTMODE_ONESHOT;
607                         (self->mssg_ptr)->message_txt =SHOTMODE_TWOSHOT_TXT;
608                     }
609                 }
610                 else {
611                     tmp_shotmode_state = current_shotmode_state;
612                     current_shotmode_state = next_shotmode_state;
613                     next_shotmode_state = tmp_shotmode_state;
614                     if(current_shotmode_state == STATE_SHOTMODE_ONESHOT)
615                         (self->mssg_ptr)->message_txt = SHOTMODE_ONESHOT_TXT;
616                     else
617                         (self->mssg_ptr)->message_txt = SHOTMODE_TWOSHOT_TXT;
618                 }
619                 break;
620
621             case STATE_MENU_UPLOAD:
622                 if(next_upload_state == STATE_UPLOAD_ENTRY) {
623                     if(current_upload_state == STATE_UPLOAD_ENTRY ||
624                     current_upload_state == STATE_UPLOAD_NO) {
625                         next_upload_state = STATE_UPLOAD_YES;
626                         current_upload_state = STATE_UPLOAD_NO;
627                         (self->mssg_ptr)->message_txt = UPLOAD_NO_TXT;
628                     }
629                     else if(current_upload_state == STATE_UPLOAD_YES) {
630                         next_upload_state = STATE_UPLOAD_NO;
631                         (self->mssg_ptr)->message_txt = UPLOAD_YES_TXT;
632                     }
633                 }
634                 else {
635                     tmp_upload_state = current_upload_state;
636                     current_upload_state = next_upload_state;
637                     next_upload_state = tmp_upload_state;
638                     if(current_upload_state == STATE_UPLOAD_NO)
639                         (self->mssg_ptr)->message_txt = UPLOAD_NO_TXT;
640                     else
641                         (self->mssg_ptr)->message_txt = UPLOAD_YES_TXT;
642                 }
643                 break;
644
```

```
645                     case STATE_MENU_DELETE:
646                         if(next_delete_state == STATE_DELETE_ENTRY) {
647                             if(current_delete_state == STATE_DELETE_ENTRY ||
648                             current_delete_state == STATE_DELETE_NO) {
649                                 next_delete_state = STATE_DELETE_YES;
650                                 current_delete_state = STATE_DELETE_NO;
651                                 (self->mssg_ptr)->message_txt = DELETE_NO_TXT;
652                             }
653                             else if(current_delete_state == STATE_DELETE_YES) {
654                                 next_delete_state = STATE_DELETE_NO;
655                                 (self->mssg_ptr)->message_txt = DELETE_YES_TXT;
656                             }
657                         }
658                         else {
659                             tmp_delete_state = current_delete_state;
660                             current_delete_state = next_delete_state;
661                             next_delete_state = tmp_delete_state;
662                             if(current_delete_state == STATE_DELETE_NO)
663                                 (self->mssg_ptr)->message_txt = DELETE_NO_TXT;
664                             else
665                                 (self->mssg_ptr)->message_txt = DELETE_YES_TXT;
666                         }
667                         break;
668
669                         default:
670                             in_Select = FALSE;
671                             reset_next_states(backto_Main);
672                 } // closes switch
673     } // closes if
674 } // closes select_ISR
675
676 void handler_select_init(struct handler_select *self,
677                         struct mssg_service *mssg_ptr)
678 {
679     self->mssg_ptr = mssg_ptr;
680     (self->start) = (& handler_select_start);
681 }
682
683 /* menu service */
684 void handler_menu_start(struct handler_menu *self)
685 {
686     extern DIGICAM_STATE current_state;
687     extern MENU_STATE current_menu_state;
```

```
688     extern MENU_STATE next_menu_state;
689     extern int backto_Main;
690     extern int in_Menu;
691     extern int in_Select;
692
693     if(!backto_Main) {
694
695             current_state = STATE_MENU;
696             in_Menu = TRUE;
697             in_Select = FALSE; // reset
698             current_menu_state = next_menu_state;
699             if(next_menu_state == STATE_MENU_ENTRY) {
700                     // assign next state
701                     next_menu_state = STATE_MENU_IMGQUAL;
702                     // show menu greeting
703                     (self->mssg_ptr)->message_txt = MENU_ENTRY_TXT;
704             }
705             else {
706                     switch(current_menu_state) {
707                         case STATE_MENU_IMGQUAL:
708                             next_menu_state = STATE_MENU_SHOTMODE;
709                             (self->mssg_ptr)->message_txt = IMGQUAL_TXT;
710                             break;
711                         case STATE_MENU_SHOTMODE:
712                             next_menu_state = STATE_MENU_UPLOAD;
713                             (self->mssg_ptr)->message_txt = SHOTMODE_TXT;
714                             break;
715                         case STATE_MENU_UPLOAD:
716                             next_menu_state = STATE_MENU_DELETE;
717                             (self->mssg_ptr)->message_txt = UPLOAD_TXT;
718                             break;
719                         case STATE_MENU_DELETE:
720                             next_menu_state = STATE_MENU_IMGQUAL;
721                             (self->mssg_ptr)->message_txt = DELETE_TXT;
722                             break;
723                         default:
724                             current_state = STATE_READY;
725                             in_Menu = FALSE;
726                             // reset state variables
727                             reset_next_states(TRUE);
728                     }
729             }
730     }
```

```
731
732     if(backto_Main) {
733             backto_Main = FALSE;
734             current_state = STATE_READY;
735             in_Menu = FALSE;
736             reset_next_states(backto_Main);
737     }
738 }
739
740 void handler_menu_init(struct handler_menu *self,
741                         struct mssg_service *mssg_ptr)
742 {
743     self->mssg_ptr = mssg_ptr;
744     (self->start) = (& handler_menu_start);
745 }
746
747 /* SWITCH services */
748 void clear_pio(struct SW_service *self)
749 {
750     (self->sw_pio)->np_pioedgecapture = 0;
751 }
752
753 void prepare_pio(struct SW_service *self)
754 {
755     (self->sw_pio)->np_piodirection = 0; // all inputs
756     (self->sw_pio)->np_piointerruptmask = 0xFF; // all generate irq's
757 }
758
759 void setup_isr(struct SW_service *self)
760 {
761     nr_installuserisr(((self->button_ptr)->button_irq), self->start, (int) self);
762 }
763
764 void SW_service_start(int self_ptr)
765 {
766     struct SW_service *self;
767
768     self = (struct SW_service*) self_ptr;
769
770     (self->clear_pio)(self);
771     switch((self->sw_pio)->np_piodata) {
772             case SW7:
773                     ((self->sw7_ptr)->start)(self->sw7_ptr);
```

```
774                         break;
775                 case SW6:
776                         ((self->sw6_ptr)->start)(self->sw6_ptr);
777                         break;
778                 case SW5:
779                         ((self->sw5_ptr)->start)(self->sw5_ptr);
780                         break;
781                 case SW4:
782                         ((self->sw4_ptr)->start)(self->sw4_ptr);
783                 }
784     (self->clear_pio)(self);
785     (self->prepare_pio)(self);
786 }
787
788 void SW_service_init(struct SW_service *self,
789                         struct handler_shutter *sw7_ptr,
790                         struct handler_menu *sw6_ptr,
791                         struct handler_select *sw5_ptr,
792                         struct handler_done *sw4_ptr,
793                         struct button_interrupt *button_ptr,
794                         np_pio* pio_addr, int button_irq)
795 {
796     self->sw7_ptr = sw7_ptr;
797     self->sw6_ptr = sw6_ptr;
798     self->sw5_ptr = sw5_ptr;
799     self->sw4_ptr = sw4_ptr;
800     self->button_ptr = button_ptr;
801     self->sw_pio = pio_addr;
802
803     (self->start) = (& SW_service_start);
804     (self->clear_pio) = (& clear_pio);
805     (self->prepare_pio) = (& prepare_pio);
806     (self->setup_isr) = (& setup_isr);
807     ((self->button_ptr)->button_irq) = button_irq;
808 }
809
810
811 /* begins the controller implementation */
812 void wait_on_interrupt(struct controller *self)
813 {
814     for(;;){
815             ((self->mssg_ptr)->display)(self->mssg_ptr);
816     }
```

```
817 }
818
819 void controller_start(struct controller *self)
820 {
821     /* check system readiness */
822     ((self->checkup_ptr)->start)(self->checkup_ptr);
823     /* init media content */
824     ((self->media_ptr)->init)(self->media_ptr);
825     /* install interrupt service routine */
826     ((self->sw_ptr)->setup_isr)(self->sw_ptr);
827     /* clear and prepare button pio */
828     ((self->sw_ptr)->clear_pio)(self->sw_ptr);
829     ((self->sw_ptr)->prepare_pio)(self->sw_ptr);
830
831     /* loop forever waiting for interrupt signals */
832     (self->wait_on_interrupt)(self);
833 }
834
835 void controller_init(struct controller *self,
836                 struct peripherals_checkup *checkup_ptr,
837                 struct media *media_ptr,
838                 struct mssg_service *mssg_ptr,
839                 struct SW_service *sw_ptr)
840 {
841     self->checkup_ptr = checkup_ptr;
842     self->media_ptr = media_ptr;
843     self->mssg_ptr = mssg_ptr;
844     self->sw_ptr = sw_ptr;
845     (self->start) = (& controller_start);
846     (self->wait_on_interrupt) = (& wait_on_interrupt);
847 }
848
```

jdatadst.c (Modified from IJG's JPEG Library)

```
1 /*
2 * jdatadst.c
3 *
4 * Copyright (C) 1994-1996, Thomas G. Lane.
5 * This file is part of the Independent JPEG Group's software.
6 * For conditions of distribution and use, see the accompanying README file.
7 *
8 * This file contains compression data destination routines for the case of
9 * emitting JPEG data to a file (or any stdio stream). While these routines
10 * are sufficient for most applications, some will want to use a different
11 * destination manager.
12 * IMPORTANT: we assume that fwrite() will correctly transcribe an array of
13 * JOCTETs into 8-bit-wide elements on external storage. If char is wider
14 * than 8 bits on your machine, you may need to do some tweaking.
15 */
16
17 /* this is not a core library module, so it doesn't define JPEG_INTERNALS */
18 #include "jinclude.h"
19 #include "jpeglib.h"
20 #include "jerror.h"
21 #include "nios.h"
22
23 #ifndef DIGICAM_CLASSTYPE_H
24 #include "digiCam_classType.h"
25 #endif
26
27 int flash_data_count = 0;
28 int flash_start_image_offset = 0;
29 extern struct media aMedia;
30
31 /* Expanded data destination object for stdio output */
32
33 typedef struct {
34         struct jpeg_destination_mgr pub; /* public fields */
35
36         FILE * outfile; /* target stream */
37         JOCTET * buffer; /* start of buffer */
38 } my_destination_mgr;
39
40 typedef my_destination_mgr * my_dest_ptr;
41
42 #define OUTPUT_BUF_SIZE 4096 /* choose an efficiently fwrite'able size */
```

```
43 #define FLASH_BUF_SIZE 2048 /* need it for data item is of char size */
44
45
46 /*
47 * Initialize destination --- called by jpeg_start_compress
48 * before any data is actually written.
49 */
50
51 METHODDEF(void)
52 init_destination (j_compress_ptr cinfo)
53 {
54         my_dest_ptr dest = (my_dest_ptr) cinfo->dest;
55
56
57         /* Allocate the output buffer --- it will be released when done with image */
58         dest->buffer = (JOCTET *)
59         (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_IMAGE,
60         OUTPUT_BUF_SIZE * SIZEOF(JOCTET));
61
62         dest->pub.next_output_byte = dest->buffer;
63         dest->pub.free_in_buffer = OUTPUT_BUF_SIZE;
64
65         // store image location
66         if (flash_start_image_offset == 0)
67                 flash_start_image_offset = aMedia.current_offset;
68
69 }
70
71
72 /*
73 * Empty the output buffer --- called whenever buffer fills up.
74 *
75 * In typical applications, this should write the entire output buffer
76 * (ignoring the current state of next_output_byte & free_in_buffer),
77 * reset the pointer & count to the start of the buffer, and return TRUE
78 * indicating that the buffer has been dumped.
79 *
80 * In applications that need to be able to suspend compression due to output
81 * overrun, a FALSE return indicates that the buffer cannot be emptied now.
82 * In this situation, the compressor will return to its caller (possibly with
83 * an indication that it has not accepted all the supplied scanlines). The
84 * application should resume compression after it has made more room in the
85 * output buffer. Note that there are substantial restrictions on the use of
```

```
86 * suspension --- see the documentation.
87 *
88 * When suspending, the compressor will back up to a convenient restart point
89 * (typically the start of the current MCU). next_output_byte & free_in_buffer
90 * indicate where the restart point will be if the current call returns FALSE.
91 * Data beyond this point will be regenerated after resumption, so do not
92 * write it out when emptying the buffer externally.
93 */
94
95 METHODDEF(boolean)
96 empty_output_buffer (j_compress_ptr cinfo)
97 {
98       int i, j;
99       unsigned short buf[FLASH_BUF_SIZE];
100      unsigned short *flash_data_address;
101
102      my_dest_ptr dest = (my_dest_ptr) cinfo->dest;
103
104
105      // prepare data for the nr_flash_write_buffer
106      for(i=0,j=1;j<OUTPUT_BUF_SIZE;i++,j+=2)
107               buf[i]=(dest->buffer[j]<<8)|(dest->buffer[j-1]);
108      flash_data_address = (unsigned short *)((char *)nasys_main_flash +
109                        aMedia.current_offset);
110
111      nr_flash_write_buffer(nasys_main_flash, flash_data_address, buf, FLASH_BUF_SIZE);
112      aMedia.current_offset += OUTPUT_BUF_SIZE;
113      flash_data_count += OUTPUT_BUF_SIZE;
114
115      dest->pub.next_output_byte = dest->buffer;
116      dest->pub.free_in_buffer = OUTPUT_BUF_SIZE;
117
118      return TRUE;
119 }
120
121
122 /*
123 * Terminate destination --- called by jpeg_finish_compress
124 * after all data has been written. Usually needs to flush buffer.
125 *
126 * NB: *not* called by jpeg_abort or jpeg_destroy; surrounding
127 * application must deal with any cleanup that should happen even
128 * for error exit.
```

```
129 */
130
131 METHODDEF(void)
132 term_destination (j_compress_ptr cinfo)
133 {
134     int i,j;
135     my_dest_ptr dest = (my_dest_ptr) cinfo->dest;
136     size_t datacount = OUTPUT_BUF_SIZE - dest->pub.free_in_buffer;
137     unsigned short buf[datacount];
138     unsigned short bufa[2], bufb[2];
139     unsigned short *flash_imgsz_address;
140     unsigned short *flash_imgat_address;
141     unsigned short *flash_data_address;
142
143     // prepare data for the nr_flash_write_buffer
144     for(i=0,j=1;j<=datacount;i++,j+=2)
145             if(j == datacount)
146                     buf[i++] = (unsigned short) dest->buffer[j-1];
147             else
148                     buf[i]=(dest->buffer[j]<<8)|(dest->buffer[j-1]);
149     /* Get total data size and initialize dataend address */
150     flash_data_count += datacount;
151
152     // put in bufa of unsigend short to avoid any potential problem :<
153     bufa[0] = (unsigned short) flash_data_count;
154     bufa[1] = (unsigned short) (flash_data_count>>16);
155     bufb[0] = (unsigned short) flash_start_image_offset;
156     bufb[1] = (unsigned short) (flash_start_image_offset>>16);
157
158     // store image size
159     flash_imgsz_address = (unsigned short *) (aMedia.image_size) + (2*aMedia.image_cnt);
160     // store image location
161     flash_imgat_address = (unsigned short *) (aMedia.addr_offset) +
162                     (2*aMedia.image_cnt);
163     flash_data_address = (unsigned short *) ((char *)nasys_main_flash +
164                     aMedia.current_offset);
165
166     // increment image number
167     aMedia.image_cnt++;
168     // set next address to 2*i which is either equal to datacount or datacount+1
169     aMedia.current_offset += (i*2);
170
171     // write number of data written in bytes
```

```
172     nr_flash_write_buffer(nasys_main_flash,flash_imgsz_address,bufa,2);
173     // write image beginning location
174     nr_flash_write_buffer(nasys_main_flash,flash_imgat_address,bufb,2);
175     // write data themselves
176     nr_flash_write_buffer(nasys_main_flash,flash_data_address,buf,i);
177     // reset datacount, and image offset
178     flash_data_count = 0;
179     flash_start_image_offset = 0;
180
181 }
182
183
184 /*
185 * Prepare for output to a stdio stream.
186 * The caller must have already opened the stream, and is responsible
187 * for closing it after finishing compression.
188 */
189
190 GLOBAL(void)
191 jpeg_stdio_dest (j_compress_ptr cinfo, FILE * outfile)
192 {
193     my_dest_ptr dest;
194
195     /* The destination object is made permanent so that multiple JPEG images
196     * can be written to the same file without re-executing jpeg_stdio_dest.
197     * This makes it dangerous to use this manager and a different destination
198      * manager serially with the same JPEG object, because their private object
199     * sizes may be different. Caveat programmer.
200     */
201      if (cinfo->dest == NULL) { /* first time for this JPEG object? */
202              cinfo->dest = (struct jpeg_destination_mgr *)
203              (*cinfo->mem->alloc_small) ((j_common_ptr) cinfo, JPOOL_PERMANENT,
204              SIZEOF(my_destination_mgr));
205      }
206
207     dest = (my_dest_ptr) cinfo->dest;
208     dest->pub.init_destination = init_destination;
209     dest->pub.empty_output_buffer = empty_output_buffer;
210     dest->pub.term_destination = term_destination;
211     dest->outfile = outfile;
212 }
213
```

nios_camIF.c

```
1 /*
2 !-- ------------------------------------------------------------ --
3 !--                                                              --
4 !-- File name:          nios_camIF.c                             --
5 !-- Author:             Chonlameth Arpnikanondt                  --
6 !-- Last revised:       10/10/03                                 --
7 !-- Description:        This file provides an interface for the  --
8 !--                     digital camera system while uploading    --
9 !--                     images to PC. It is a very simple interface --
10 !--                    meant to be used as a demonstrative      --
11 !--                    application software. It works through the --
12 !--                    NiOS' GERMS monitor environment with the --
13 !--                    typical command line as shown below:     --
14 !--                       germs_prompt> nr -t | nios_camIF.exe  --
15 !--                                                             --
16 !--                    When the upload operation is selected    --
17 !--                    the camera transmits all image data via UART --
18 !--                    to the PC. These data are piped to the   --
19 !--                    nios_camIF software, which then writes jpeg --
20 !--                    image files from the transmitted data.   --
21 !--                                                             --
22 !-- ------------------------------------------------------------ --
23 */
24
25 #include <stdio.h>
26 #include <string.h>
27 #include <ctype.h>
28 #include <stdlib.h>
29
30
31 #define BUFFSIZE 16
32 #define LINESIZE 100
33 char* FILE_BASE = "image_";
34 char* DOT_JPG = ".jpg";
35
36
37 int
38 main()
39 {
40         unsigned char uc[BUFFSIZE];
41         unsigned int u;
42         int bytenum, imgnum;
```

```
43      char filename[LINESIZE];
44      char c[LINESIZE], istr[BUFFSIZE];
45      int i, j, cnt, bytecnt;
46      FILE * outfile; /* target jpg file */
47
48      // init image number
49      imgnum = 0;
50
51      // read line until end of file encounter
52      while(gets(c) != NULL) {
53              // if first character is numeric or -99
54              if( (c[0] == '-') || (isdigit(c[0])) ) {
55                      // first -99
56                      if(c[0] == '-' && c[1] == '9' && c[2] == '9') {
57                              // read image until -99 is encountered again
58                              while(gets(c) != NULL) {
59                                      // second -99
60                                      if(c[0] == '-' && c[1] == '9' && c[2] == '9')
61                                              exit(0);
62                                      else {
63                                              // it's an image size
64                                              bytenum = atoi(c);
65                                              imgnum++;
66                                              // create output file string
67                                              // init file base
68                                              sprintf(filename, "\0");
69                                              strcat(filename, FILE_BASE);
70                                              // first convert image number to string
71                                              sprintf(istr, "%d\0", imgnum);
72                                              // then concat
73                                              strcat(filename, istr);
74                                              strcat(filename, DOT_JPG);
75
76                                              // open file
77                                              if ((outfile = fopen(filename, "wb")) == NULL) {
78                                                  fprintf(stderr, "can't open %s\n", filename);
79                                                  exit(1);
80                                              }
81
82                                              // read in data
83                                              bytecnt = 0;
84                                              while(bytecnt < bytenum) {
85                                                  if(bytenum-bytecnt > BUFFSIZE-1) {
```

```
86                                                      cnt = BUFFSIZE;
87                                                      for(i=0; i<BUFFSIZE; i++) {
88                                                              gets(c);
89                                                              sscanf(c, "%x", &u);
90                                                              uc[i] = (unsigned char) u;
91                                                              bytecnt++;
92                                                      }
93                                                  }
94                                                  else {
95                                                      for(i=0; i<(bytenum-bytecnt); i++) {
96                                                              gets(c);
97                                                              sscanf(c, "%x", &u);
98                                                              uc[i] = (unsigned char) u;
99                                                      }
100                                                 cnt = i;
101                                                 bytecnt = bytenum;
102                                             }
103                                             fwrite(uc, 1, cnt, outfile);
104                                         }
105
106                                         // close file
107                                         if (outfile != stdout)
108                                                 fclose(outfile);
109                                     } // else
110                                 } // inner while
111                         } // if(c[0] == '-')
112                 } // if first character
113     } // outer while
114
115     return 0;
116 }
```

Appendix F

COCOMO II: Source Code Counting Rules

In this appendix, the source code counting rules applicable to this thesis are presented that are excerpted from the COCOMO II Model Definition Manual [19]. Table F.1 tabulates these rules as follows:

Table F.1: COCOMO II SLOC Checklist

Definition Checklist for Source Statements Counts			
Logical Source Statements		**Includes**	**Excludes**
Statement Type *When a line or statement contains more than one type, classify it as the type with the highest precedence.*	**Order of Precedence**		
1. Executable	1	✓	
2. Nonexecutable			
3. Declaration	2	✓	
4. Compiler directives	3	✓	
5. Comments			
6. On their own lines	4		✓
7. On lines with source code	5		✓
8. Banners and non-blank spacers	6		✓
9. Blank (empty) comments	7		✓
10. Blank lines	8		✓

Table F.1: COCOMO II SLOC Checklist

Definition Checklist for Source Statements Counts			
Logical Source Statements		**Includes**	**Excludes**
How produced			
1. Programmed		✓	
2. Generated with source code generators			✓
3. Converted with automated translators		✓	
4. Copied or reused without change		✓	
5. Modified		✓	
6. Removed			✓
Origin			
1. New work no prior existence		✓	
2. Prior work: taken or adapted from			
3. A previous version, build, or release		✓	
4. Commercial, off-the-shelf software (COTS), other than libraries			✓
5. Government furnished software (GFS), other than reuse libraries			✓
6. Another product			✓
7. A vendor-supplied language support library (unmodified)			✓
8. A vendor-supplied operating system or utility (unmodified)			✓
9. A local or modified language support library or operating system			✓
10. Other commercial library			✓
11. A reuse library (software designed for reuse)		✓	
12. Other software component or library		✓	
Usage			
1. In or as part of the primary product		✓	
2. External to or in support of the primary product			✓
Delivery			
1. Delivered as source		✓	
2. Delivered in compiled or executable form, but not as source			✓
3. Not delivered			✓

Table F.1: COCOMO II SLOC Checklist

Definition Checklist for Source Statements Counts			
Logical Source Statements		**Includes**	**Excludes**
Functionality			
1. Operative		✓	
2. Inoperative (dead, bypassed, unused, unreferenced, or unaccessible):			
3. Functional (intentional dead code, reactivated for special purposes)		✓	
4. Nonfunctional (unintentionally present)			✓
Replications			
1. Master source statements (originals)		✓	
2. Physical replicates of master statements, stored in the master code		✓	
3. Copies inserted, instantiated, or expanded when compiling or linking			✓
4. Postproduction replicates—as in distributed, redundant, or reparameterized systems			✓
Language: General			
1. Nulls, continues, and no-ops		✓	
2. Empty statements, e.g. ";;" and lone semicolons on separate lines			✓
3. Statements that instantiate generics		✓	
4. Begin...end and {...} pairs used as executable statements		✓	
5. Begin...end and {...} pairs that delimit (sub)program bodies			✓
6. Logical expressions used as test conditions			✓
7. Expression evaluations used as test conditions			✓
8. End symbols that terminate executable statements			✓
9. End symbols that terminate declarations or (sub)program bodies			✓
10.Then, else, and otherwise symbols			✓
11. Elseif statements		✓	
12. Keywords like procedure division, interface, and implementation		✓	
13. Labels (branching destinations) on lines by themselves			✓
Language: C and C++			
1. Null statement, e.g. ";" by itself to indicate an empty body			✓

Table F.1: COCOMO II SLOC Checklist

Definition Checklist for Source Statements Counts			
Logical Source Statements		**Includes**	**Excludes**
2. Expression statements (expressions terminated by semicolons)		✓	
3. Expression separated by semicolons, as in a "for" statement		✓	
4. Block statements, e.g. {...} with no terminating semicolon		✓	
5. ";" on a line by itself when part of a declaration			✓
6. ";" on a line by itself when part of an executable statement			✓
7. Conditionally compiled statements (#if, #ifdef, #ifndef)		✓	
8. Preprocessor statements other than #if, #ifdef, and #ifndef		✓	

Appendix G

Summary of UML Notations

This appendix intends to provide a quick reference guide to UML notations (UML 1.5). It is quite terse in description, and far from being complete. The UML specification [24] should always be consulted if need for further clarification arises.

G.1 Static Structure Model

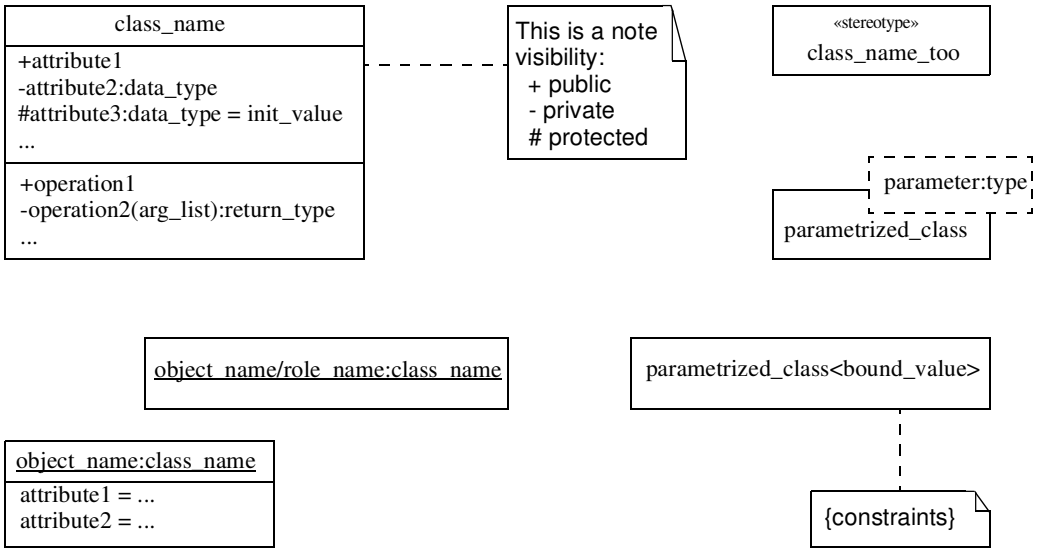
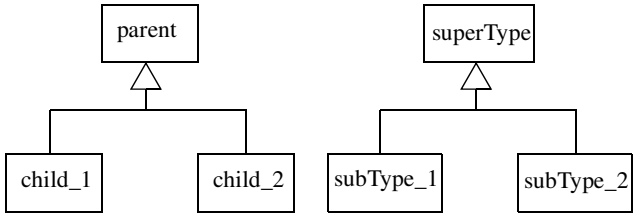
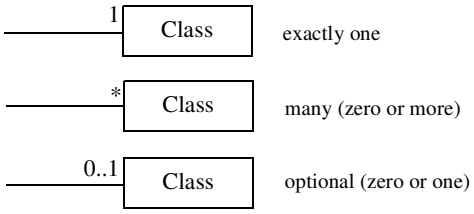


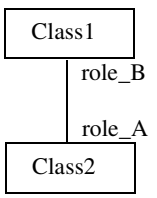
Figure G.1: Classes and objects



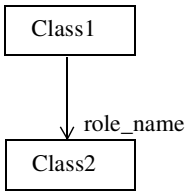
(a) Generalization



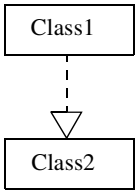
(b) Multiplicities



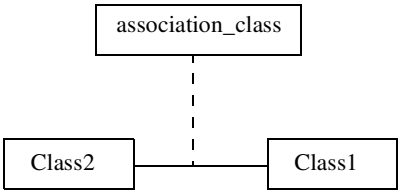
(c) Association



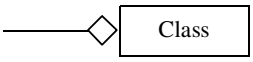
(d) Navigability



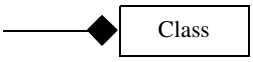
(e) Realization
Class1 implements **Class2**



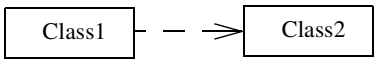
(f) Association Class



(g) Aggregation



(h) Composition



(i) Dependency.
Class1 depends on **Class2**.

Figure G.2: Class relationships

G.2 Interaction Model

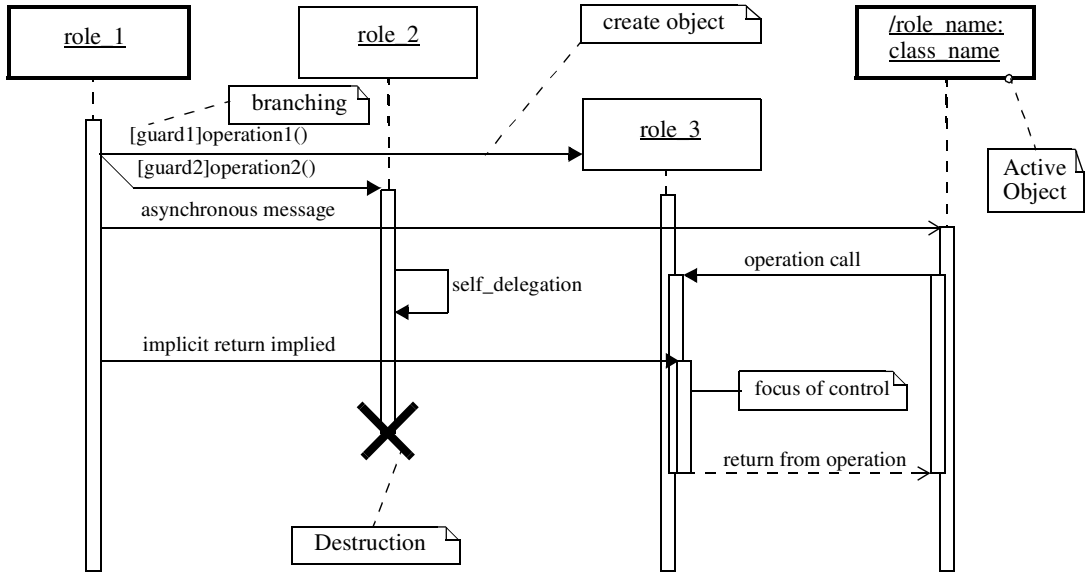


Figure G.3: Sequence diagram

G.3 State Model

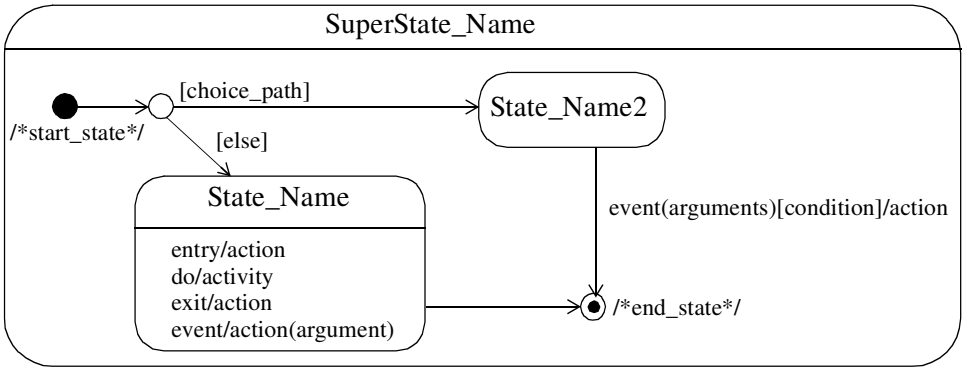


Figure G.4: State diagram

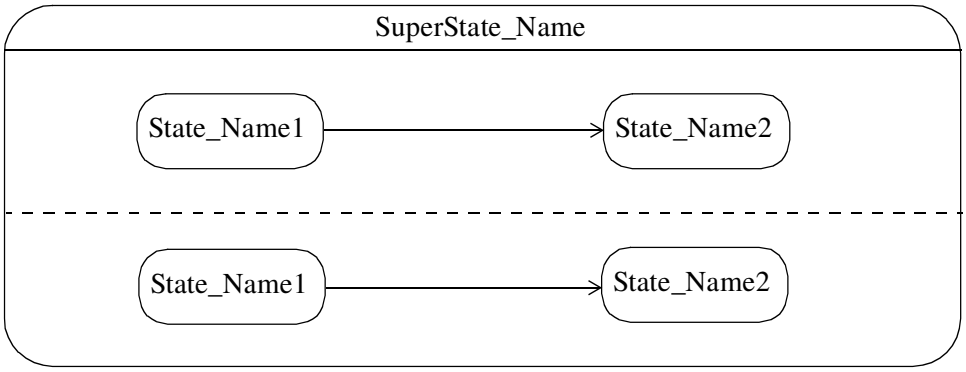


Figure G.5: Concurrent States

G.4 Use Case Model

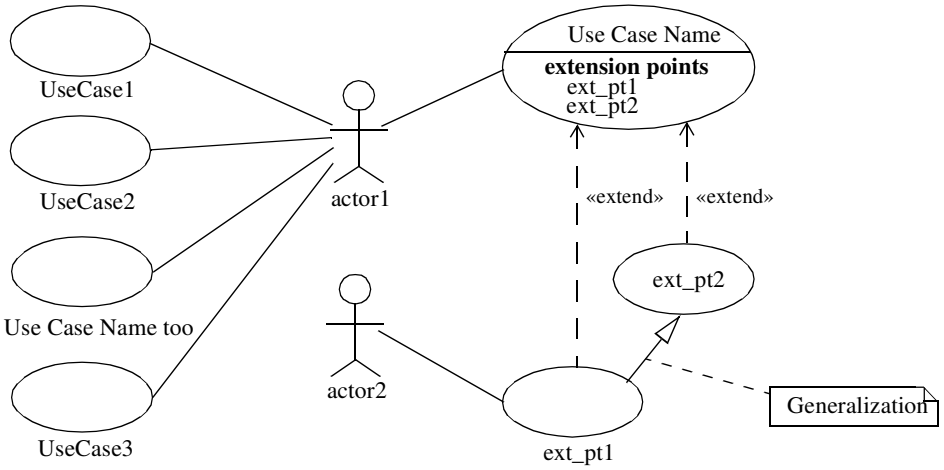


Figure G.6: Use Case diagram

G.5 Model Management

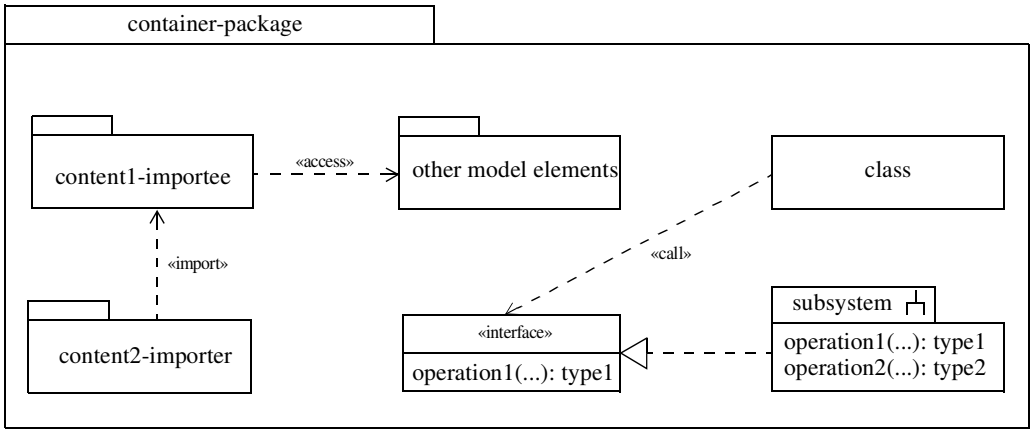


Figure G.7: Package and Subsystem

Bibliography

[1] J. Turley, "The Two Percent Solution," *Embedded.com*, December 2002; http://www.embedded.com/story/OEG20021217S0039.

[2] C. Herring, "Microprocessors, Microcontrollers, and Systems in the New Millenium," *IEEE Micro*, vol. 20, no. 6, pp. 45-51, November-December 2000.

[3] V. J. Mooney III, D. M. Blough, "A Hardware-Software Real-Time Operating System Framework for SoCs," *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 44-51, November-December 2002.

[4] R. Mahajan, K. Brown, V. Atluri, "The Evolution of Microprocessor Packaging," *Intel Technology Journal*, 3rd Quarter 2000, Intel Corporation; http://developer.intel.com/technology/itj/q32000/articles/art_1.htm.

[5] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, J. Haskins Jr., "A Survey of Configurable, Component-Based Operating Systems for Embedded Applications," *IEEE Micro*, vol. 21, no. 3, pp. 54-67, May-June 2001.

[6] J. A. DeBardelaben, "An Optimization-Based Approach for Cost-Effective Embedded DSP System Design," *Ph.D. Thesis, Georgia Institute of Technology*, May 1998.

[7] J. A. DeBardelaben, V. K. Madisetti, A. Gadient, "Incorporating Cost Modelign in Embedded-System Design," *IEEE Design & Test of Computers*, pp. 24-35, July-September 1997.

[8] B. C. Cole, "Getting to the Market On Time," *Electronics*, pp. 62-65, April 1989.

[9] D&T Roundtable, "Hardware-Software Codesign," *IEEE Design & Test of Computers*, vol. 17, no. 1, pp. 92-99, January-March 2000.

[10] F. Slomka, M. Dorfel, R. Munzenberger, R. Hofmann, "Hardware/Software Codesign and Rapid Prototyping of Embedded Systems," *IEEE Design & Test of Computers,* vol. 17, no. 2, pp. 28-38, April-June 2000.

[11] University of California, Berkeley, *A Framework for Hardware-Software Co-Design of Embedded Systems, POLIS Release 0.4*, December 1999; http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html.

[12] U.S. Air Force Analysis Agency, *REVIC Software Cost Estimating Model User's Manual Version 9.2*, December 1994.

[13] M. Sgroi, L. Lavagno, A. Sangiovanni-Vincentelli, "Formal Models for Embedded System Design," *IEEE Design & Test of Computers*, vol. 17, no. 2, pp. 14-27, April-June 2000.

[14] A. Sangiovanni-Vincentelli, G. Martin, "Platform-Based Design and Software Design Methodology for Embedded Systems," *IEEE Design & Test of Computers*, vol. 18. no. 6, pp. 23-33, November-December 2001.

[15] Department of Defense, *Parametric Cost Estimating Handbook,* Fall 1995.

[16] B. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

[17] D. Lyons, "SEER Helps Keep Satellite Cost Estimates Down to Earth," *Infoworld*, November 27, 1995.

[18] A. Minkiewicz, A. DeMarco, "The PRICE Software Model," *PRICE Systems Intenal Document*, June 1995.

[19] University of Southern California, Center for Software Engineering, *COCOMO II: Model Definition Manual*, Version 2.1, 2000.

[20] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Design Methodology*, Kluwer Academic Publishers, March 2000.

[21] M. Levitt, "Economic and Productivity Considerations in ASIC Test and Design-for-Test," *Digest of Papers: Compcon '92*, pp. 440-445, Spring 1992.

[22] J. Miranda, "A BIST and Boundary-Scan Economics Framework," *IEEE Design & Test of Computers*, pp. 17-23, September 1997.

[23] J. Axelsson, "Analysis and Synthesis of Heterogeneous Real-Time Systems," *Ph.D. Thesis, Linköping University*, 1997.

[24] The Object Management Group, OMG *Unified Modeling Language Specification Version 1.5*, March 2003; http://www.omg.org/.

[25] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd Edition, Addison-Wesley, 2000.

[26] B. Mathew, "UML Tutorial," *Stylus Inc.'s Online Articles*, Accessed: January 2003; http://www.stylusinc.net/articles/uml/lesson1.shtml.

[27] "Practical UML: A Hands-On Introduction for Developers," *TogetherSoft Inc.*; http://www.togethersoft.com/services/practical_guides/umlonlinecourse, accessed January 2003.

[28] B.P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison-Wesley, 1999.

[29] The Object Management Group (OMG), *UML Profile for Schedulability, Performance, and Time Specification, Final Draft*, March 2002; http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

[30] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[31] "NRE Charges for SoC Designs to Hit $1M, Study Says," *Semiconductor Business News*, October 1999; http://www.csdmag.com/story/OEG19991006S0043.

[32] S. P. Dean, "ASICs and Foundries," Integrated Systems Design, March 2001; http://www.isdmag.com/story/OEG20010301S0051.

[33] *DOmain Modeling Environment, Version 5.3*, Honeywell Labs, Morristown, NJ, August 2000; http://www.htc.honeywell.com:80/dome/description.htm.

[34] *ArgoUML Community Version, Version 0.12*, BSD License, January 2003; http://argouml.tigris.org/.

[35] G. Kotonya, I. Sommerville, *Requirements Engineering: Processes and Techniques*, John Wiley & Sons, 1998.

[36] L. A. Maciaszek, *Requirements Analysis and System Design: Developing Information Systems with UML*, Addison Wesley, 2001.

[37] M. Edwards, "Software Acceleration Using Coprocessors: Is It Worth the Effort?," *Proceedings of the 5th International Workshop on Hardware/Software Codesign*, pp. 135-139, 1997.

[38] G. M. Amdahl, "Validity of Single-Processor Approach to Achieve Large-scale Computing Capability," *Proceedings of the 30th AFIPS Spring Joint Computer Conference*, pp. 483-485, 1967.

[39] R. Ernst, J. Henkel, T. Benner, "Hardware-Software CoSynthesis for Microcontrollers," IEEE Design & Test of Computers, vol. 10, no. 4, pp. 64-75, December 1993.

[40] R. Gupta, G. De Micheli, "Hardware-Software CoSynthesis for Digital Systems," *IEEE Design & Test of Computers*, pp. 29-41, September 1993.

[41] T.-Y. Yen, W. Wolf, "Sensitivity-Driven Co-synthesis of Distributed Embedded Systems," *Proceedings of the 8th International Symposium on System Synthesis*, 1995.

[42] T.-Y. Yen, W. Wolf, "Communication Synthesis for Distributed Embedded Systems," *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 288-294, 1995.

[43] S. Schulz, J.W. Rozenblit, K. Buchenrieder, "Towards an Application of Model-Based Codesign: An Autonomous, Intelligent Cruise Controller," *Proceedings of the 1997 IEEE Conference and Workshop on Engineering of Computer Based Systems,* pp. 73-80*,* Monterey, CA, March 1997.

[44] K. Buchenrieder, J.W. Rozenblit, "Codesign: An Overview," *Codesign: Computer-Aided Software/Hardware Engineering*, pp. 1-16, IEEE Press, 1994.

[45] University of California, Berkeley, *The Ptolemy Project*, February 2003; http://ptolemy.berkeley.edu/.

[46] R. Goering, “24-hour Chip Design Cycle Called Possible,” *EETIMES*, August 2001; http://www.eedesign.com/story/OEG20010803S0083.

[47] S. Holzner, *Inside XML*, pp. 1-14, New Riders Publishing, November 2001.

[48] A. Møller, M. I. Schwartzbach, “The XML Revolution: Technologies for the Future Web,” *BRICS, University of Arthus*, October 2002: http://www.brics.dk/~amoeller/XML/.

[49] A. Bergholz, “Extending Your Markup: An XML Tutorial,” *IEEE Internet Computing*, July-August 2000; http://computer.org/internet/xml/xml.tutorial.pdf.

[50] J. P. Thomas, “Understanding XML (eXtensible Markup Language),” *Stylus Inc.’s Online Articles*; http://www.stylusinc.net/technology/XML/xml_future.shtml, accessed March 2003.

[51] M. Bryan, "An Introduction to the Extensible Markup Language (XML)," http://www.personal.u-net.com/~sgml/xmlintro.htm, accessed March 2003.

[52] *Document Object Model (DOM)*, World Wide Web Consortium; http://www.w3.org/DOM/, accessed March 2003.

[53] *The Common Object Request Broker: Architecture and Specification Version 3.0.2,* The Object Management Group (OMG); http://www.omg.org/technology/documents/corba_spec_catalog.htm, accessed March 2003.

[54] N. Walsh, “A Technical Introduction to XML,” *World Wide Web Journal*, Revision 1.1, February 1998; http://www.w3j.com; http://nwalsh.com/docs/articles/xml/index.html.

[55] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, *Extensible Markup Language (XML) 1.0*, *Second Edition*, World Wide Web Consortium, October 2000; http://www.w3.org/TR/REC-xml.html.

[56] *XML Schema 1.0*, World Wide Web Consortium, May 2001; http://www.w3.org/XML/Schema.

[57] OASIS, “Extensible Markup Language (XML),” *Cover Pages*, February 2003; http://xml.coverpages.org/xml.html.

[58] S. DeRose, E. Maler, D. Orchard, *XML Linking Language (XLink) Version 1.0*, World Wide Web Consortium, June 2001; http://www.w3.org/TR/xlink/.

[59] S. DeRose, R. Daniel, et.al., *XML Pointer Language (XPointer)*, World Wide Web Consortium, August 2002; http://www.w3.org/TR/xptr/.

[60] J. Clark, S. DeRose, *XML Path Language (XPath) Version 1.0*, World Wide Web Consortium, November 1999; http://www.w3.org/TR/xpath.

[61] *Resource Description Framework (RDF)*, World Wide Web Consortium, February 1999; http://www.w3.org/RDF/.

[62] F. Manola, E.Miller, et.al., "RDF Primer," *World Wide Web Consortium*, January 2003; http://www.w3.org/TR/rdf-primer/.

[63] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels (RFC2119)," *The Internet Engineering Task Force*, March 1997; http://www.ietf.org/rfc/rfc2119.txt.

[64] C. Alexander, S. Ishikawa, M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[65] C. Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

[66] B. T. Kurotsuchi, "Design Patterns," *Online Tutorial*, 1996; http://www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/.

[67] D. L. Levine, D. C. Schmidt, "Introduction to Patterns and Frameworks," *Class Material*, Department of Computer Science, Washington University at St. Louis; http://classes.cec.wustl.edu/_cs342/.

[68] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[69] B. Foote, J. Yoder, "The Selfish Class," *Third Conference of Patterns Languages of Programs (PLoP'96)*, Monticello, IL, September 1996.

[70] R. Goering, "Platform-based Design: A Choice, not a Panacea," *EETIMES*, September 2002; http://www.eetimes.com/story/OEG20020911S0061.

[71] J. Clark, *XSL Transformation (XSLT) Version 1.0*, World Wide Web Consortium, November 1999; http://www.w3.org/TR/xslt.

[72] S. Boag, D. Chamberlin, et.al., *XQuery 1.0: An XML Query Language*, Working Draft, World Wide Web Consortium, November 2002; http://www.w3.org/TR/xquery/.

[73] "XPath Tutorial," *W3Schools.com*; http://www.w3schools.com/xpath/default.asp, accessed March 2003.

[74] *Xalan Java Version 2.5.D1*, The Apache XML Project; http://xml.apache.org/xalan-j/index.html.

[75] *Xalan C++ Version 1.4*, The Apache XML Project; http://xml.apache.org/xalan-c/index.html.

[76] *Jaxen*, SourceForge.net; http://sourceforge.net/projects/jaxen/.

[77] *MSXML 3.0*, Microsoft Inc.; http://msdn.microsoft.com/xml/.

[78] M. Nic, J. Jirat, "XPath Tutorial," *ZVON*; http://www.zvon.org/xxl/XPathTutorial/General/examples.html, accessed March 2003.

[79] *The Nexperia Platform*, Philips Semiconductors; http://www.semiconductors.philips.com/platforms/nexperia/technology/index.html, accessed January 2003.

[80] *The Open Multimedia Applications Platform (OMAP)*, Texas Instruments; http://www.omap.com, accessed March 2003.

[81] *The NiOS Embedded Processor*, Altera; http://www.altera.com/products/devices/nios/nio-index.html, accessed March 2003.

[82] *The QuickMIPS Platform*, Quicklogic Inc.; http://www.quicklogic.com/home.asp?PageID=376&sMenuID=220, accessed March 2003.

[83] *The Virtex-II Pro Platform FPGAs*, Xilinx Inc.; http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=Virtex-II+Pro+FPGAs, accessed March 2003.

[84] *Avalon Bus Specification*, Altera Inc., San Jose, CA, January 2002.

[85] *CAN Specification Version 2.0*, Robert Bosch GmbH, September 1991.

[86] *FlexRay Requirements Specification, Version 2.0.2*, FlexRay-Consortium, April 2000; http://www.flexray-group.com/.

[87] *The I^2C-Bus and How to Use It*, Philips Semiconductors, 1995; http://www.semiconductors.philips.com/acrobat/ various/I2C_BUS_SPECIFICATION_1995.pdf.

[88] C. Zhang, F. Vahid, "A Power-Configurable Bus for Embedded Systems," *IEEE International Symposium on Circuits and Systems*, pp.V-809-812, May 2002.

[89] *Universal Resource Identifier (URI) Specification*, World Wide Web Consortium; http://www.w3.org/Addressing/URL/URL_TOC.html, accessed March 2003.

[90] B. Foote, J. Yoder, "Metadata and Active Object-Models," *OOPSLA '98 MetaData and Active Object-Model Workshop*, Vancouver, Canada, October 1998; http:// www.laputan.org/metadata/metadata.html.

[91] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search," *Design Automation for Embedded Systems*, pp. 5-32, 1997.

[92] M. S. Haworth and W. P. Birmingham, "Towards Optimal System-Level Design," *Proceedings of the 30th Design Automation Conference*, pp. 434-438, 1993.

[93] A. K. Majhi, L. M. Patnaik, and S. Raman, "A Generic Algorithm-Based Circuit Partitioner for MCMs," *Microprocessing and Microprogramming*, Vol. 41, pp. 83-96, 1995.

[94] J. Teich, T. Blickle, and L. Thiele, "An Evolutionary Approach to System-Level Synthesis," *Proceedings of the 5th International Workshop on Hardware/Software Codesign*, pp. 167-171, 1997.

[95] T. Tran, "OMAP5910 NTSC or VGA Output," *Texas Instruments' DSP/EEE Catalog, OMAP Application Report SPRA847*, June 2003; see http://www-s.ti.com/sc/ psheets/spra847/spra847.pdf.

[96] K. Sabbagh, *Twenty-First Century Jet: The Making and Marketing of the Boeing 777*, Scribner, New York, 1996.

[97] A. Ramirez, et.al., "ArgoUML User Manual, A Tutorial and Reference Description of ArgoUML, Revision 0.10," *Open Publication License*, May 2002; http://argouml.tigris.org/documentation/defaulthtml/manual/.

[98] B. P. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems, Second Edition*, Addison-Wesley, Reading, MA, 1999.

[99] J. Warmer, and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

[100] *MagicDraw™ 6.0*, No Magic Inc.; http://www.magicdraw.com/, accessed August 2003.

[101] *Poseidon for UML, Professional Edition 1.6 and Embedded Edition 1.6.1*, Gentleware AG, Hamburg, Germany; http://www.gentelware.com, accessed August 2003.

[102] *Rational Rose™*, IBM Inc.; http://www.rational.com, accessed August 2003.

[103] *Rhapsody™ in C, C++ and Ada*, I-Logix Inc., MA.; http://www.ilogix.com/products/rhapsody/index.cfm, accessed August 2003.

[104] *Codagen Architect*, Codagen Technologies Corp., Montreal, Canada; http://www.codagen.com/products/architect/default.htm, accessed August 2003.

[105] *Enterprise Architect Version 3.51*, Sparx Systems, Australia; http://www.sparxsystems.com.au/ea.htm, accessed August 2003.

[106] A. Moore, *RT-Profile—A Quick Tour*, Artisan Software Inc.; http://wwwbroy.informatik.tu-muenchen.de/~rappl/RT%20Profile(new).pdf, last accessed August 2003.

[107] V. Madisetti, L. Shen, "Timing Interface Design in Core-based Systems," *IEEE Design & Test of Computers*, Vol 13, No. 4, October-December 1997.

[108] A. Burns, A. Wellings, *Real-Time Systems and Programming Languages*, Third Edition, Addison-Wesley, 2001.

[109] A. Costello, C. Truta, *CEXCEPT Exception Handling in C, Version 1.0.0*, The CEXCEPT Project, June, 2000; see http://cexcept.sourceforge.net/.

[110] P. Lee, "Exception Handling in C Programs," *Software—Practice and Experience*, Vol 13, No. 5, pp. 393-401, 1983.

[111] The Real-Time for Java™ Expert Group, *The Real-Time Specification for Java™, Version 1.0*, Addison-Wesley, 2000; see also http://www.rtj.org/rtsj-V1.0.pdf.

[112] *Draft Standard for VHDL Register Transfer Level (RTL) Synthesis*, IEEE Standard P1706.6/D6, IEEE Computer Society, May 2003; see http://vhdl.org/siwg/66_D6X.PDF.

[113] D. J. Smith, *HDL Chip Design*, Doone Publications, AL, 1996.

[114] J. Bhasker, *A VHDL Synthesis Primer*, Star Galaxy Publishing, PA, 1996.

[115] J. Bhasker, *Verilog HDL Synthesis, A Practical Primer*, Star Galaxy Publishing, PA, October 1998.

[116] D. Björklund, J. Lilius, "From UML Behavioral Descriptions to Efficient Synthesizable VHDL," *In NORCHIP02*, Copenhagen, Denmark; see http://www.infa.abo.fi/~dbjorklu/publications/norchip02.pdf.

[117] W. McUmber, B. H. C. Cheng, "UML-Based Analysis of Embedded Systems Using a Mapping to VHDL," *4th IEEE High Assurance Software Engineering*, pp. 56-63, November 1999.

[118] S. J. Mellor, L. Starr "A Method for Making UML Directly Executable," *EETIMES*, November 2002; see http://www.eetimes.com/design_library/esd/dt/OEG20021115S0045.

[119] G. Martin, "Coexistence in a Multilingual Design World," *EETIMES*, June 2003; see http://www.eedesign.com/silicon/OEG20030616S0064.

[120] F. Vahid, and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, John Wiley & Sons, October 2001.

[121] S. W. Grotta, "Anatomy of a Digital Camera," *ExtremeTech*, June 2001; see http://www.extremetech.com/article2/0,3973,15466,00.asp, accessed September 2003.

[122] S. W. Grotta, “Anatomy of a Digital Camera: Image Sensors,” *ExtremeTech*, June 2001; see http://www.extremetech.com/article2/0,3973,15465,00.asp, last accessed September 2003.

[123] I. Graham, *Object-Oriented Methods, Principles & Practice*, Third Edition, Addison-Wesley, 2001.

[124] R. Abbott, “Program Design By Informal English Descriptions,” *Communications of the ACM*, Vol. 26, No. 11, pp. 882-894, 1983.

[125] R. Wirfs-Brock, B. Wilkerson, L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

[126] Z. Chen, J. Cong, Y. Fan, X. Yang, Z. Zhang, “Pilot — A Platform-Based HW/SW Synthesis System for FPSoC,” *Workshop on Software Support for Reconfigurable Systems*, February 2003.

[127] C. Loeffler, A. Ligtenberg, G. Moschytz, “Practical Fast 1-D DCT Algorithms with 11 Multiplications,” *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing 1989 (ICASSP ‘89)*, pp. 988-991.

[128] W. Pennebaker, J. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993.

[129] A. Messac, M. Martinez, T. Simpson, “Effective Product Family Design Using Physical Programming,” *Engineering Optimization*, Vol. 34, No. 3, 2002, pp. 245-261.

[130] P. Clarke, “Gartner Trims 2003 Semiconductor Growth Forecast,” *Semiconductor Business News, EETIMES*, May 2003; see http://www.eetimes.com/semi/news/OEG20030520S0009.

[131] *Functional Specification for SystemC 2.0*, Open SystemC Initiative (OSCI), April 2002; see http://www.systemc.org.

[132] *SystemC 2.0 User’s Guide*, Open SystemC Initiative (OSCI), 2002; see http://www.systemc.org.

[133] G. Martin, L. Lavagno, J. Louis-Guerin, "Embedded UML: a merger of real-time UML and codesign ", *CODES 2001*, Copenhagen, April 2001, pp.23-28.

[134] B. Selic, G. Gullekson, P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley and Sons, New York, 1994.

[135] J. L. Diaz-Herrera, V. K. Madisetti, "Embedded Systems Product Lines," *Proceedings of Software Product Lines, 22nd International Conference on Software Engineerings*, Limerick, Ireland, June 2000, pp. 129-136.

[136] J. Withey, "Investment Analysis of Software Assets for Product Lines," *CMU/SEI-96-TR-010*, Carnegie Mellon University, Pittsburgh, PA, 1996.

[137] J. L. Diaz-Herrera, J. C. Guzman, "Product Lines in the Context of Embedded Systems," *Proceedings of Software Product Lines, 23rd International Conference on Software Engineerings*, Toronto, Canada, May 2001, pp. 19-25.

[138] K. Van Rompaey, I. Bolsens, H. De Man, D. Verkest, "CoWare—A Design Environment for Heterogenous Hardware/Software Systems," *Proceedings of the conference on European design automation*, Geneva, Switzerland, 1996, pp. 252-257.