

Chapter 2

PLATFORM-CENTRIC SOC DESIGN METHODOLOGY

2. INTRODUCTION

In the proposed platform-centric SoC method, a 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 THE PLATFORM CONCEPT

2.1.1 Introduction to Platforms

Karl Sabbagh of Boeing Corporation first defined 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, ranging from aircraft to mobile phones. For instance, the Boeing 777 passenger doors, each of which is composed of a

different set of parts with subtly different shapes and sizes for its position on the fuselage, are yet based on the same platform, resulting in 98% of all door mechanisms being 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. DARPA's RASSP Program introduced the concept of "model year architectures" in early 1993, which may also be considered equivalent to the current definition of platforms for electronics systems (<http://www.eda.org/rassp>). The concepts of model year architectures, in turn, were derived from earlier successful models of production of automobiles over a century.

Even though commonality in platforms often could compromise 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 [138]. Sangiovanni-Vincentelli and Martin [14] argue 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]:

- **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.

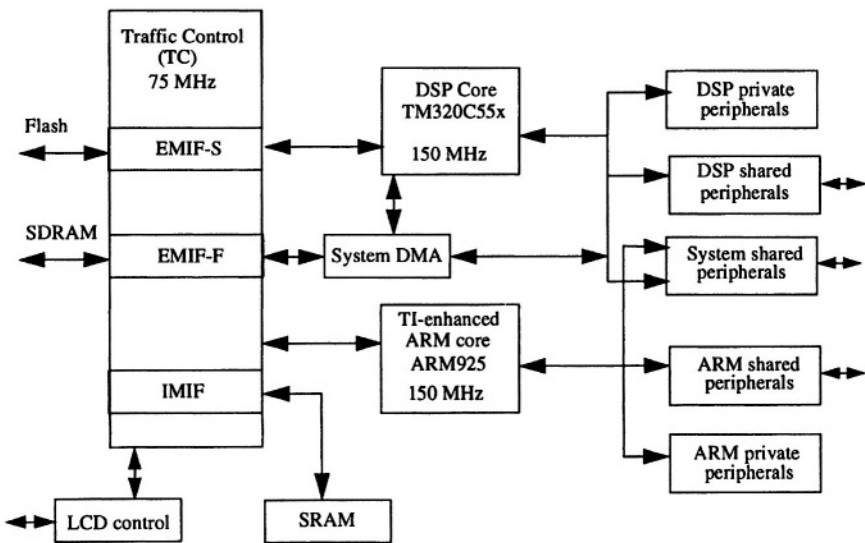


Figure 2-1. A simplified view of TI's OMAP5910 289-pin wireless platform architecture, which has a package size of 12x12 mm² (based on a figure in [70]).

The system platform should 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 [138]. 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 (Ref: 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 (Ref: Jean-Marc Chateau, ST Microelectronics).

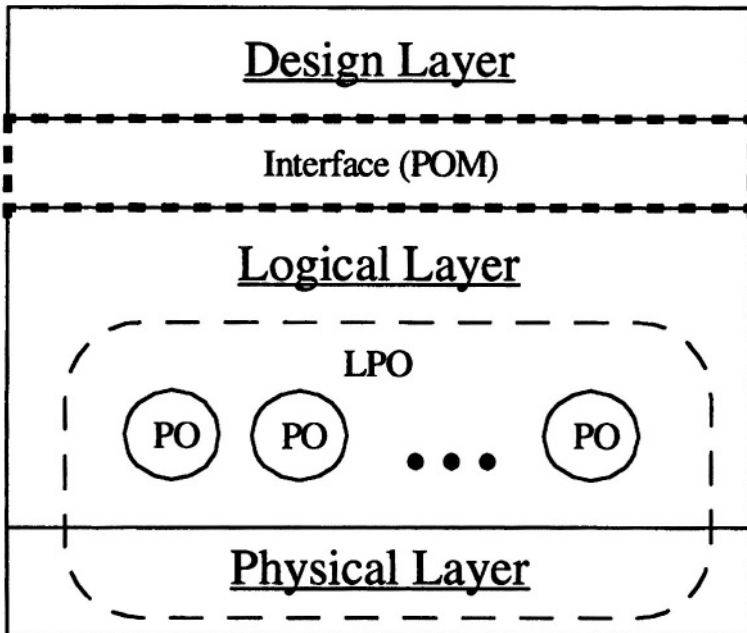


Figure 2-2. A logical model of the platform-centric environment

- 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 (Ref: Virtual Socket Interface Alliance's (VSIA) platform-based design development working group).

2.2 PLATFORM-CENTRIC SoC DESIGN APPROACH

The platform-centric method is an enhanced version of a platform-based design approach. It provides a 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 depicted in Figure 2.3.

Definition 2.1. The Platform-Centric SoC Design Method

A reuse-intensive, software-biased, and analysis-driven co-design approach that relies upon the UML-/XML-enhanced tools-integrated environment and the use of platforms to develop a feasible 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.

Definition 2.2. The 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.

identifies such an abstract representation and the relationship with its counterpart in the physical layer, forms 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 ensures that the LPO boundary is as expansive as the Internet itself, and these LPO member modules are un-encumbered by geographical location. 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 is usually a software tool member of the physical layer.

The platform-centric SoC design approach promotes an enhanced tools-integrated environment while also raises a level of design abstraction. Each platform object (PO) represents a collection 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 capture 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 so as 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 is passed back to appropriate LPO modules to be further analyzed, realized and integrated. Detailed discussion regarding each main stage in the proposed approach is attempted next.

2.2.1 Platform-Independent Specification

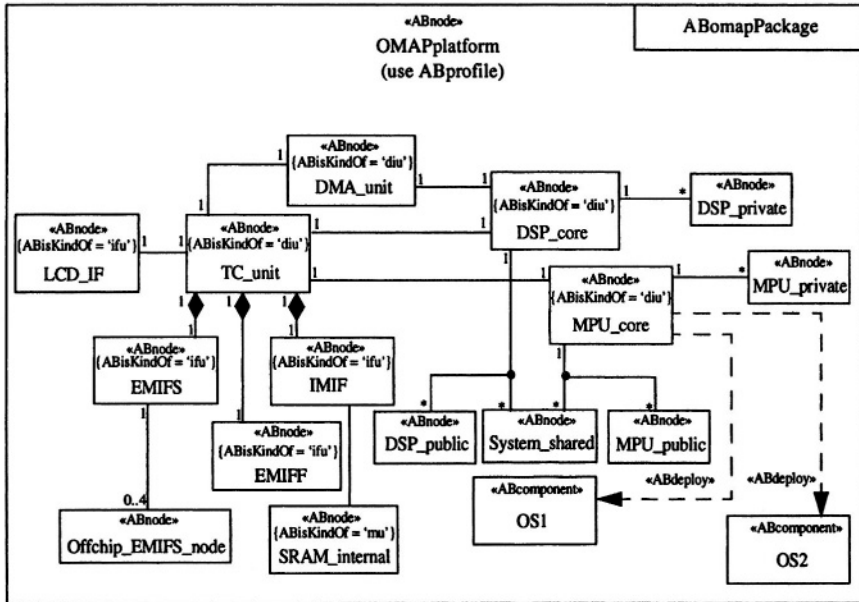
This stage chiefly concerns with the derivation of the functional specification that is still independent of any platform instance binding. Unlike most current co-design approaches that begin with a formal specification of the system, the proposed platform-centric method starts with the requirements 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.

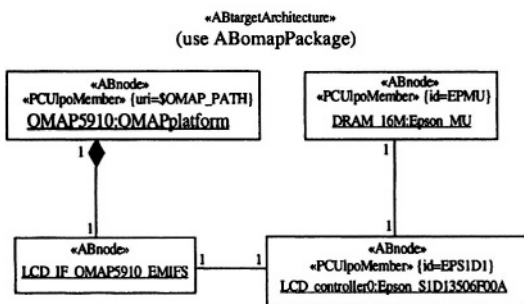
The functional requirements are then modeled using Use Case diagrams; while, those with non-functional characteristics, e.g. speed, size, reliability, robustness, portability, standards compliance, ease of use, etc., may be captured into the supplementary requirements specification (usually a note or textual description) later to be processed and incorporated into the functional specification as constraints [97].

As established techniques in object-oriented design, the UML’s 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 diagrams may be used to describe the system characteristics.

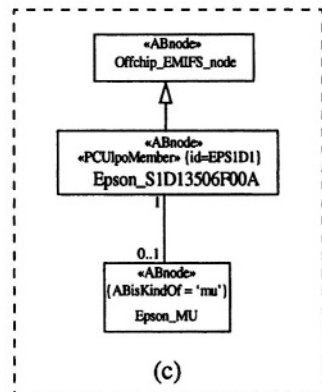
For example, Sequence and/or Activity diagrams can capture concurrent and/or sequential interactions; or, Component diagrams can depict components connection within the system.



(a)



(b)



(c)

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 represents a pre-defined communication. The DRAM object comes as a derivative requirement when instantiating the LCD controller Pomm module (c)

The resulting functional specification can have the class methods implemented using a programming language of choice. This specification may later be verified for behavioral correctness and handed off to the next stage.

2.2.2 Platform Analysis

The primary 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. The configuration of the selected platform object then follows, resulting 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 Texas Instruments' 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.

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 tabular search. The reader may wish to consult references [23], [91] for detailed treatments of these algorithms on the architectural selection process. In [92], [93], [94], other related algorithms are presented.

It may be noted that, in developing a system where flexibility and technology-to-market time are the predominant factors, software may often carry 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 techniques in the software engineering discipline. In Chapter 5, the UML profile for Co-design 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 P0mm 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 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.

The platform-dependent specification stage is iterative, and during this phase, 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 is carried out 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 extraction. 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 CURRENT APPROACHES

Table 2.1 shows how the proposed approach compares with previous related approaches in current practice, albeit in the research domain. The general framework for the hardware/software co-synthesis approach often involves a pre-determined 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 co-synthesis approach includes the work by Ernst, et.al. [39]. They describe the COSYMA (Co-synthesis of eMbedded Architectures) system which is used to extract, from a given program, a slice of functionality 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 co-synthesis 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 co-synthesizing 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 excellent co-synthesis approaches represent state-of-the-art efforts in co-design, they do have certain limitations—some of the more notable ones are:

- In the COSYMA and Vulcan systems, the fixed architecture is presumed without elaborating on such a decision. This could be the scope of applicability of these approaches.
- Yen and Wolf's approach supports only bus-oriented architectural topologies, which may 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. They are unable 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 designing 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 has some 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 requirements.

The POLIS [11] co-design 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 Co-design Finite State Machine (CFSM).

Table 2-1. Feature support of current co-design approach: “supported” (++) , “partially supported” (+) and “not supported” (BLANK). The survey approaches include the Model-based [43, 44], POLIS [11], Corsair [10], SpecC [20], SystemC [131], Chip-in-a-day [46]

Features	Model-Based	POLIS	Corsair	SpecC & SystemC	Chip-in-a-day	Platform-Centric
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						++

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.

The Corsair [10] integrated framework methodology is similar to POLIS. The Corsair framework contains several tools for the automated 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.

Similar in motivation to SpecC, a new language called SystemC has been proposed [131]. 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 co-simulation and/or co-verification of heterogeneous systems under development. The SystemC's support for executable

specification modeling within a single language facilitates HW/SW co-design while, simultaneously, easing the co-simulation/co-verification tasks. In addition, its generalized communication model permits an iterative refinement of communication through the concepts of ports, channels, and interfaces. 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 requirements.
- SystemC is relatively new, especially as a hardware description language (HDL). The current version (SystemC 2.0) has yet to provide support for hardware synthesis. The language lacks also the wide industry and engineering backing that the UML enjoys.
- The interface-based approach adopted by SystemC singularly might not suffice to tackle current issues in the development of SoC systems.

The chip-in-a-day approach [46] proposed 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 Mathworks' Simulink to capture a high-level data flow and control flow diagrams. Based on pre-characterized 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 co-design together. Embedded UML coalesces various existing ideas currently used in real-time UML and HW/SW co-design 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 test benches 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 Co-design 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 Co-design 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], many propose that semiconductor systems manufacturers & integrators will design systems, as opposed to designing processors for the short term, though it may be to their advantage

in the longer term to design their own processors. The proposed UML profile for Co-design Modeling Framework is flexible to meet with new demands from these organizations as they adapt to new technologies and business models. This is to contrast to the Embedded UML profile whose approach seems to come from the opposite direction where system houses are seen as the primary forces in the development process. Under such a circumstances, 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 book may eventually be more robust in a long run as it is designed to better adapt to technological changes.

Proposed as part of the Yamacraw Embedded Software (YES) effort at the Georgia Institute of Technology's Georgia Electronic Design Center (GEDC), 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]:

- Behavior of analog interfaces within the embedded specification,
- Real-time related behavior 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 may be clear from this discussion that, like the Embedded UML, the YES-UML also bears a number of similarities to the Co-design Modeling Framework (CMF) profile even though its intended development paradigm is larger in scope than that of the proposed platform-centric methodology.

2.5 A PERSPECTIVE ON COLLABORATION WITH NON-PLATFORM APPROACHES

Being a graphical language, the 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 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 or 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 illustrates possible synergy between the proposed UML-based 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 SoC design method where it can still benefit from a unified design environment and extensive reuse at the interface level, even if/when the platform-level design is not possible.

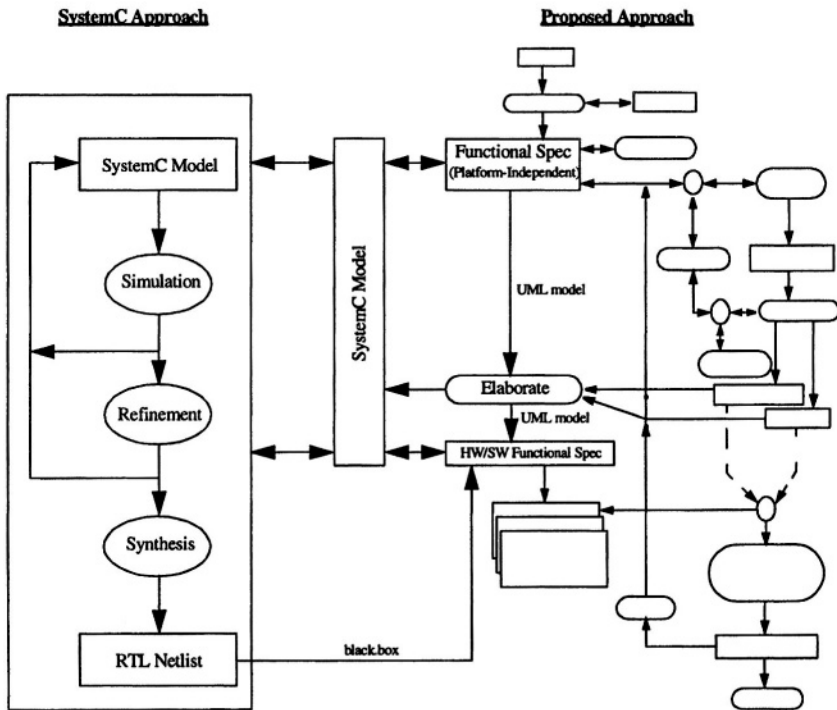


Figure 2-5. Collaborative usage model for the proposed platform-centric approach (CMF) and the SystemC approach (adapted from [132]).

One of the major advantages of the proposed platform-centric SoC design methodology appears to be the creation of an 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 book 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 applicable to SoC design is given.

<http://www.springer.com/978-0-387-23895-1>

A Platform-Centric Approach to System-on-Chip (SOC)
Design

madisetti, v.; Arpnikanondt, C.

2005, XII, 206 p. 67 illus., Hardcover

ISBN: 978-0-387-23895-1