Chapter 5

UML AND PLATFORM-BASED DESIGN

Rong Chen[1], Marco Sgroi[1], Luciano Lavagno[2], Grant Martin[2], Alberto Sangiovanni-Vincentelli[1], Jan Rabaey[1]
[1]University of California at Berkeley [2]Cadence Design Systems

Abstract: This chapter presents a specification technique based on UML for the design of embedded systems and platforms. It covers stereotypes and extended notations to represent platform services and their attributes in embedded software development. It also presents a design methodology for embedded systems that is based on platform-based design principles.

Key words: platform-based design, embedded system design, UML

1. INTRODUCTION

The embedded system design approach currently used in the industry is informal especially in the initial phase, where the requirements and the functionality of the system are usually expressed in natural language. The inherent ambiguities of an informal specification prevent a meaningful analysis of the system and may result in misunderstandings with customers and in incorrect or inefficient decisions at the time when the design is partitioned and the tasks are assigned to different teams.

Hence, a key ingredient in a well defined methodology is a specification language with formally defined semantics that allows designers to describe the structure and the behavior of an embedded system at multiple levels of abstraction starting from the purely conceptual level. Embedded systems must satisfy tight performance and cost constraints. Therefore, embedded software design, in comparison with traditional software development, requires one not only to verify the functional correctness but also to check the satisfaction of these constraints. Performance and cost analysis depends

on the selected architecture and therefore requires tools and models for a formal definition of the implementation platform resources and the quality of the services they offer. Furthermore, early stages of the design process would benefit from the use of graphical interface tools that visualize the system specification and allow multiple team members to share the relevant information.

This chapter presents a specification technique for the design of embedded systems and platforms that addresses these issues and is based on the Unified Modeling Language (UML).

1.1 Platform-based Design

Platform-based design has emerged recently as one promising approach to solve the problems caused by the ever-increasing complexity and time-to-market pressure in embedded system design. According to [1], a platform is an abstraction layer in the design flow that facilitates a number of possible refinements into a subsequent, lower-level abstraction layer (platform). In other words, a platform is an abstract representation of a set of possible implementations, which is used by the application designer as a design target, and is implemented by the platform provider.

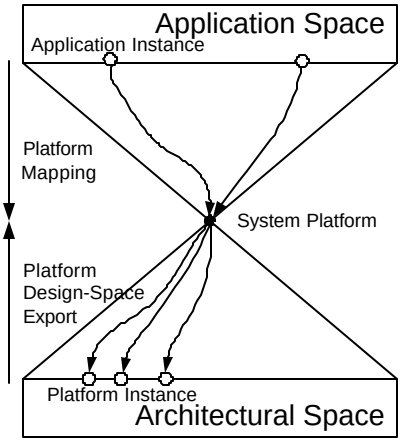

Figure 5-1. Platform-based Design

As shown in Figure 5-1, the basic tenets of the platform-based design methodology are:

1. the treatment of design as a “meet-in-the-middle process”, where successive refinements of specifications meet with abstractions of potential implementations;

2. the identification of precisely defined layers, where the refinement and abstraction process take place. The layers then allow designs built upon them to be isolated from lower-level details, but let enough information be transmitted about lower levels of abstraction to allow design space exploration with a fairly accurate prediction of the properties of the final implementation.

Among many platforms existing in a design flow, two important abstractions are identified at the articulation point between system definition and implementation: the (micro-) architecture platform and the application programming interface (API) platform. The (micro-) architecture platform concept originates from the fact that integrated circuits used for embedded systems are usually derived from some related micro-architectures rather than assembled from a collection of independently developed blocks of silicon functionality. Consequently, a (micro-) architecture platform is defined as a specific family of micro-architectures, possibly oriented toward a particular class of problems, which can be modified (extended or reduced) by the system developer. Examples are the AMBA/ARM platform, the CoreConnect platform, a family of FPGA chips. A platform instance can be derived from a platform by choosing a set of components from the platform library and setting parameters of configurable components. The choice of a platform is driven by cost and time-to-market considerations and is done after exploration of both the application and architecture design spaces. The API platform concept results from the fact that embedded software developers need a platform abstraction that hides architecture details and defines the services that the platform offers. More precisely, an API platform is the Programmer's Model for the abstraction of a multiplicity of computational resources and available peripherals contained within the architectural platform; it is a unique abstract representation of the architecture platform via the software layer. This abstraction usually consists of a software layer that wraps the essential parts of the architecture platform and includes, among others, RTOS and device drivers. Examples are the VxWorks platform, the OSEK platform, the software DSP task control of the OMAP platform. On top of the API platform there is an application specific programmable platform, which consists of commonly used functionalities in a particular application domain. This platform usually consists of embedded software components and directly interacts with embedded system designers. Examples are the TCP/IP platform and the top level of the Nexperia platform. There are other platforms such as the silicon implementation platform (SIP), which is beyond the scope of this chapter, and the network platform, which is discussed in section 4.4.

1.2 UML and Embedded System Design

The Unified Modeling Language (UML) is an object-oriented modeling language introduced by Booch, Rumbaugh and Jacobson [2] to support software development. Recent work [3][4][5][6] has shown the potential of UML also for embedded system design. Due to its rich graphical notation and its modeling capabilities that allow the capture and visualization of the system structure and behavior at multiple levels of abstraction, UML has been used in the embedded system domain mainly as a documentation aid and a modeling language. UML includes a rich set of modeling elements that can be used for a wide range of applications and has already built in the capabilities to model the most relevant features of real-time embedded systems, such as performance (using tagged attributes or OCL [7]), physical resources (using Deployment Diagrams), and time (using classifiers and tagged attributes), as discussed in Chapters 11, 12 and 16. However, the following factors should also be considered.

- First, modeling specific applications would be easier using a more specialized (domain-specific) notation representing the basic elements and patterns of the target domain.
- Second, formally defined domain-dependent use semantics are required to avoid multiple interpretations of the same models and to support analysis tools.
- Third, multiple diagrams can be used to capture related aspects of a design. The possibility of viewing and describing the same object from different perspectives makes the system specification phase easier, but may result in inconsistencies between diagrams, when the use of UML is not coupled with a rigorous design discipline.

For these reasons, the use of UML for a specific application domain such as the design of embedded system platforms requires:

- a domain specific language, called a profile, built on the basic UML infrastructure and including domain-specific building blocks (defined using stereotypes) to model common design elements and patterns,
- a methodology that defines how and when the profile notation should be used.

A UML profile for embedded system platforms should include specialized notations to represent the structure and the behavior of the platform resources, and the services they provide, with particular emphasis on the performance and cost aspects. It should also have the capability to

visualize multiple implementation alternatives of a specification to facilitate a quick comparison.

The rest of this chapter is organized as follows. First, we give an overview of related work, and describe the research project Metropolis, within which we are developing our concepts (section 2). Then, we describe a UML profile for embedded system platforms, called UML Platform, (section 3) and a methodology for using it (section 4). Finally we present its practical application to the design of wireless networks platforms.

2. BACKGROUND

2.1 Related work

The UML Profile for Schedulability, Performance and Time (also informally called the Real-Time UML Profile) [8], recently standardized by the Object Management Group (OMG) and summarized in Chapter 11, defines a unified framework to express the time, scheduling and performance aspects of a system. It is based on a set of notations that can be used to build models of real-time systems annotated with relevant Quality of Service (QoS) parameters. External tools can perform formal analysis based on these models and return information on performance and schedulability before the system is built. The profile consists of the Generalized Resource Modeling (GRM) Framework that defines a notation for modeling resources, time, concurrency and schedulability parameters. In addition to GRM, sub-profiles are defined with extensions to the basic notation that are specific to certain types of analysis, e.g. schedulability. The Real-Time UML Profile standardizes an extended UML notation to support the interoperability of modeling and analysis tools but does not define a full methodology for the use of this notation.

Several methodologies based on UML have been proposed. They all couple the UML notation with a formal model with precise semantics that allows the capture of system behavior and support simulation and synthesis. UML-RT [9] is a profile that extends UML with stereotyped active objects, called capsules, to represent system components. The internal behavior of a capsule is defined using statecharts; its interaction with other capsules is by means of protocols that define the sequence of signals exchanged through stereotyped objects called ports. Capsules have run-to-completion semantics and their execution is defined by the sequence of actions that are taken upon reception of messages from the input ports. The UML-RT profile defines a model with precise execution semantics, hence it is suitable to capture

system behavior and support simulation or synthesis tools (e.g. Rational Rose-RT). UML-RT has limited architecture and performance modeling capabilities and therefore should be considered complementary to the Real-Time UML Profile standardized by OMG.

HASoC [10] is a design methodology based on UML-RT notation described in Chapter 6. The design flow begins with a description of the system functionality initially given in use case diagrams and then in a UML-RT version properly extended to include annotations with mapping information. The authors argue that UML-RT is too restrictive a model because capsules' behavior is defined by statecharts and propose instead to associate capsules with additional models of computation (MoC) such as synchronous dataflow, codesign finite state machines etc. Another full system design methodology that uses UML is presented by de Jong et al. [4]. It consists of a flow from the initial specification phase to the deployment level that specifies the target architecture. The high-level system specification is first built using use-case diagrams; then the system components and their interactions are described using block diagrams and message sequence charts, respectively. As a next step the behavior of each module is specified using SDL that provides an executable and simulatable specification. This approach combines the informal notation of the UML Diagrams with the formal semantics of SDL and moves one step further the integration between these two models.

2.2 The Metropolis design environment

Metropolis [11] is an on-going research project at UC Berkeley that addresses embedded system design using the following novel approaches. First, Metropolis does not commit a priori to any particular communication semantics or firing rule. Hence, it leaves the designer free to use the specification mechanism of choice (graphical or textual language), as long as it has a sound semantic foundation (model of computation). Secondly, it uses a single formalism to represent both the embedded system and some abstract relevant characteristics of its environment and implementation platform. Finally, it separates orthogonal aspects, such as:

1. Computation and communication. This separation is important because:
a) refinement of computation is generally done by hand, or by compilation, or by scheduling or using other complex techniques;
b) refinement of communication is generally done by use of patterns (such as circular buffers for FIFOs, polling or interrupts for hardware to software data transfers, and so on).

2. Function and Architecture. They are often defined independently by different design teams (e.g. video encoding and decoding experts versus hardware and software designers in multimedia applications). Function (both computation and communication) is then "mapped" to architecture in order to derive an implementation.
3. Behavior and performance parameters, such as latency, throughput, power, energy.

All these separations result in better design re-use, because they decouple independent aspects that would otherwise tie, for example, a given functional specification to low-level implementation details, or to a specific communication paradigm, or to a scheduling algorithm. It is very important to define only as many aspects as needed at every level of abstraction, in the interest of flexibility and rapid design space exploration. They also allow extensive use of synthesis, system-level simulation and formal verification techniques in order to speed up the design cycle.

In Metropolis, a system is represented as a netlist, and a netlist can be further decomposed into subnetlists or components. Components of a netlist or a subnetlist include processes, media, ports, interfaces, constraints, and quantities. Processes are active objects (running on their own threads) used for modeling computation; media are passive objects used for modeling communication. Ports, specified as interface types, reside in processes and are the only places through which communication can take place. Interfaces declare all and only the methods that can be called through ports; constraints deal with coordination and performance specifications; quantities annotate behaviors so that constraints can be specified precisely.

3. UML PLATFORM PROFILE

3.1 Modeling Platforms Using UML

The UML Platform profile is a graphical language for specification of embedded system platforms. It includes domain-specific classifiers and relationships specialized with stereotypes, in addition to the notation defined in the standard UML [2] and in the UML Profile for Schedulability, Performance and Time Specification [12], to model the structure and the behavior of embedded systems and to represent the relationship between platforms at different levels of abstraction. The profile has been derived from the model and design of several wireless protocols and therefore is especially suited for this application domain.

UML Platform structural models capture the components of a system and their relationships using stereotyped modeling elements. The model of a platform, especially when it relates to another platform at a different abstraction level, often requires modeling elements of different types (e.g. classes to represent the logical functions, components for the software implementation and deployment nodes for the physical resources running them) and therefore is not identifiable with a specific UML Diagram.

The behavior of an embedded system can be captured at different levels of abstraction using Use Case, Interaction, State Machine and Activity Diagrams. Use Case Diagrams provide an abstract representation of the services that the system as a whole provides to the environment, Interaction Diagrams define just the interaction among system components, State Machine and Activity Diagrams allow the specification of the detailed action-level behavior of individual components. Specifying the behavior of a system requires to choose a model of computation that formally defines the semantics of the execution and the interaction among the system components and to describe the behavior of the components. As discussed in Section 2.2, the specification mechanism should not commit a priori to a specific MoC but should be flexible enough to let the designer choose the most appropriate MoC for the application. The UML Platform profile defines stereotypes representing standard MoCs (such as Kahn Process Networks, Synchronous Dataflow etc.) and elementary building blocks, such as buffers, protocols etc., that can be used to specify a MoC. The behavior of individual components is specified using graphical (State Machine or Activity Diagrams) or textual notation.

The syntax of the UML Platform profile is defined by the set of standard and stereotyped UML modeling elements and by the rules for using and composing them. UML Platform follows the rules of UML for standard classifiers and relationships and explicitly defines composition rules for the newly introduced notation. The semantics of UML Platform is defined in terms of the Metropolis Metamodel [11] by establishing a direct correspondence between modeling elements of UML Platform and elements of the Metamodel.

3.2 Stereotypes

The UML Platform profile includes the following stereotypes:

- *<<Netlist>>* is a top-level class that identifies the overall system structure as a set of connected components
- *<<Resource>>* specializes classes, components or deployment nodes and is used to represent platform components. Resource attributes (for

example, bus attributes such as width, number of masters/slaves, arbitration policy...) are specified by annotating tags or as attributes of the corresponding class. *<<Bus>>*, *<<Memory>>*, *<<Processor>>*... further specialize the *<<Resource>>* stereotype for deployment nodes and are used to model physical resources

– *<<Process>>* is an active class [2] modeling logical objects performing computation
– *<<Medium>>* is a class modeling logical or physical objects that implement a communication function
– *<<Scheduler>>* is a class modeling objects and algorithms that perform coordination and arbitration of access to shared resources
– *<<Port>>* represents a part of a resource that allows its interaction with its environment. A port has an aggregate relationship with the resource to which it belongs and is associated with the services it provides access to.

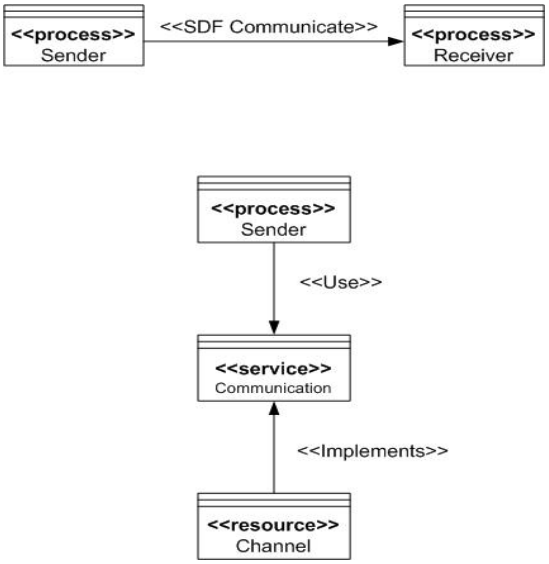

Figure 5-2. Stereotyped Relationships

The following stereotypes specialize the UML relationships:

<<Communicate>> specializes the association relationship and is used to relate classes of objects that interact and exchange messages. The *<<Communicate>>* relationship can be further specialized with stereotypes indicating a particular communication mechanism (e.g. Synchronous,

Asynchronous, Buffered, Unbuffered) or a MoC (e.g. SDF, Kahn Process Networks, Synchronous FSMs, CFSMs...).

<<Use>> is a type of association that relates a service with a user of the service. *<<Need>>* is similar to *<<Use>>* and indicates that a user needs a service that is not currently available. Thus, it represents a request for future service extensions. An object may use multiple services and the same service may be used by multiple objects. A stereotyped relationship called *<<Share>>* can be used to relate the users of a service provided by the same resource. *<<Stack>>* and *<<Peer>>* specialize the relationship between a service user and a service provider (a resource): *<<Stack>>* is used if the service user and service provider belong to different layers of abstraction, *<<Peer>>* if they belong to the same layer. *<<Transparent Stack>>* is used when the service user uses a service provided by a resource that is not in the adjacent lower layer.

<<Implement>> is a type of realization relationship between a service and a resource (or set of resources) implementing it. In figure 5-2, the channel resource is an implementation of a communication service. *<<Refine>>* is a type of realization relationship between an object (or set of objects) and an object or set of objects that describe it at a greater levels of details.

4. UML PLATFORM DESIGN METHODOLOGY

The design methodology, based on the UML Platform profile and Metropolis [11], is shown in Figure 5-3. In the first step the design problem is formulated, i.e. the functionality of the system as a whole is specified using Use Case Diagrams and the constraints are annotated to the model. Then, the functionality is decomposed into components and captured using the UML Platform stereotypes within the Class (structure), State Machine, Activity and Sequence Diagrams (behavior). Constraints are propagated and budgeted to the components.

As a next step, the UML Platform specification is compiled into a Metropolis Metamodel specification to conjugate the convenience of using the graphical UML Platform interface for specification with the possibility to use the analysis and synthesis tools available in the Metropolis framework. The Metamodel functional specification can be validated using the Metropolis simulator [11].

Then, Communication Refinement and Mapping take place. Platforms that implement the functional components are specified in UML Platform as a netlist of resources providing services. The UML Platform model is

compiled into a mapped Metamodel specification, and performance analysis and validation take place in the Metropolis simulation environment.

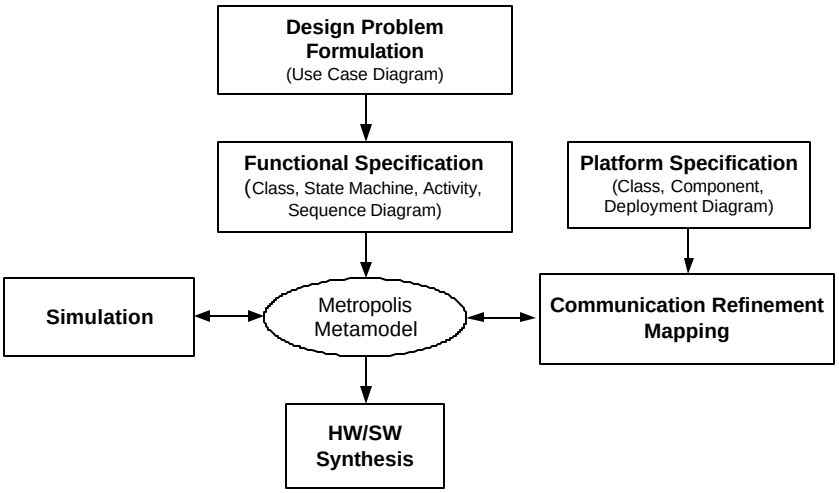

Figure 5-3. UML Platform Design Flow

4.1 Design Problem Formulation

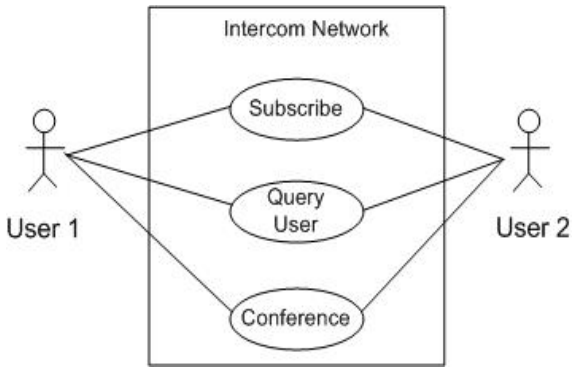

Figure 5-4. Intercom Use Case Diagram

The system requirements and the overall functionality are expressed using Use Case Diagrams. The system as a whole is described in terms of the services (modeled as use cases) it provides to the environment users (modeled as actors). Requirements are annotated to the model. Figure 5-4

shows the Use Case Diagram model of the Intercom Network [13]: use cases model the services provided by the network (subscription, conferences, query user status), actors represent the mobile users of the Intercom network services.

4.2 Functional Specification

Specifying the functionality of a system usually requires to identify a number of functional components (functional decomposition), select a MoC that defines the semantics of their interaction, and specify the behavior of each component.

In UML Platform the functionality of a system is specified in two steps.

– First, a netlist of stereotyped classes and relationships is defined to capture the structural decomposition into interacting components. Computation objects are modeled as stereotyped *<<Process>>* classes, while their connectivity is expressed using *<<Communicate>>* relationships or explicitly using stereotyped *<<Medium>>* classes. When a stereotyped class is instantiated, its attributes, such as name of the process, number and type of its ports, internal variables, are set.
– Second, the system behavior is specified refining the *<<Process>>* classes and the *<<Communicate>>* relationships to specific MoC types and describing the behavior of each process using State Machine Diagrams, Activity Diagrams or textual languages depending on what mechanism is more convenient for the target specification.

In UML Platform an MoC can be specified either using the stereotyped classes and relationships (e.g. *<<Kahn Process Networks Process>>*, *<<Kahn Process Networks Communicate>>*...) if it is a standard MoC or by composing finer granularity modeling elements such as types of channels (e.g. FIFO, shared memory...), interface functions (e.g. blocking/non-blocking read/write...) and coordination expressions (e.g. to specify synchronization or coordination of reads and writes...). A UML Platform model can be translated into a Metamodel description using the Metropolis Metamodel libraries, and instantiating the library elements that correspond to the stereotypes in the UML Platform model. The stereotyped classes of type process and medium have a corresponding Metamodel element. Ports and interfaces, which may be explicitly visualized in a UML Platform model, are part of the specification of processes and media in the Metamodel. The MoC's stereotyped relationship has a corresponding element in the library that includes a Metamodel description of the interface functions of the medium and the execution policy of the processes.

Let us consider the specification of the filter *o(n) = k2 * i(n) + k1 * o(n-1)*.The UML Platform model of the filter is shown in Figure 5-5. First, the computation components are described as stereotyped <<*Process*>> classes and are connected by <<*Communicate*>> relationships. The filter performs pure data processing functions, therefore the most suited MoC is Synchronous Dataflow (SDF) [14]. SDF is a MoC that represent the system as a network of actors that communicate over single-reader single-writer blocking read non-blocking write channels with infinite FIFOs holding tokens. The execution of an SDF model is based on a sequence of actor firings, each producing and consuming a fixed number of tokens.

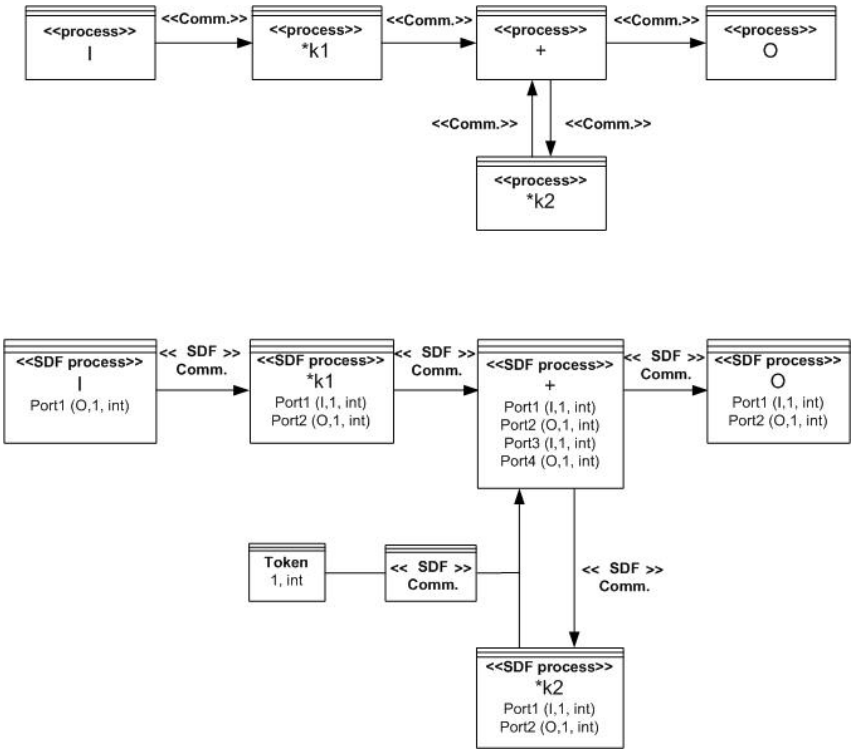

Figure 5-5. IIR Filter Model

The UML Platform model is refined using the corresponding SDF stereotypes, as shown in the lower part of Figure 5-5, so that the firing rules and the communication semantics of the specification are the ones of the SDF MoC. Parameters are set in the model for each class: the number of input and output ports, the data type and the number of the tokens produced or consumed at each port. The core function of each process in this case is conveniently specified with a fragment of code that describes the

corresponding function (e.g. *o_adder = in1_adder + in2_adder*). The token class represents the one tap delay on one of the channels.

The information visualized in the UML Platform model is compiled together with the Metamodel description of SDF in the Metropolis library (built by H.Hsieh and L.Jin [15]). The SDF library defines:

1. the SDF medium, i.e. the interface functions read, write and n (that returns the token count) and the memory buffer (array list) storing the data,
2. the SDF process, i.e. the constructor, the declaration of the number and type of ports, and the functions readport, writeport defining the firing rules (e.g. SDF process executed when all the expected input tokens are present).

Figure 5-6 shows the Metamodel specification of the Adder process and a fragment of the netlist that corresponds to the UML Platform Class Diagram.

```
process Adder extends sdfprocess{
  Adder(String name){
    super(name,2,2);
    inports_token = new int[2];
    outports_token = new int[2];
    inports_token[0] = 1;
    inports_token[1] = 1;
    outports_token[0] = 1;
    outports_token[1] = 1;
  }
  void execute(){
    double vo0;
    double vi0;
    double vi1;

    vi0 = i_buffer[0].get(0);
    vi1 = i_buffer[1].get(0);
    vo0 = vi0 + vi1;
    o_buffer[0].set(0,(Object)vo0);
    o_buffer[1].set(0,(Object)vo0);
  }
}

public netlist IIRFilter{
```

```
    public IIRFilter(String name){
        ClassAdder Adder_instance = new ClassAdder("sdfadder");
        sdfchannel M0_instance = new sdfchannel("sdfchannel0");
        sdfchannel M1_instance = new sdfchannel("sdfchannel1");
        addcomponent(Adder_instance, this,"adder");
        addcomponent(M0_instance, this, "channel0");
        addcomponent(M1_instance, this, "channel1");
              ...
        connect(Adder_instance, inports[0], M0_instance);
        connect(Adder_instance, outports[0], M1_instance);
    }
}
```

Figure 5-6. Metamodel Filter Specification

4.3 Platform Specification

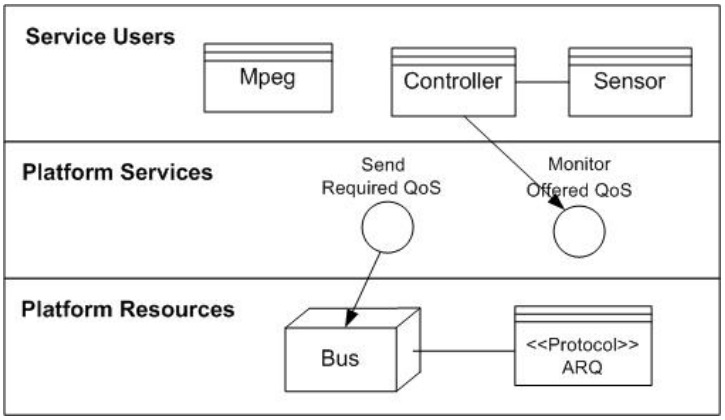

Figure 5-7. Key Elements of a Platform Model

The following components are usually part of a platform specification (Figure 5-7):

- the (physical or logical) resources modeled using the stereotyped classifiers defined in the UML Real-Time [5] and in the UML Platform profiles
- the services offered by individual or groups of resources modeled using standard UML interfaces.(an interface does not have attributes, but may have operations to specify the service primitives [2])
- the QoS performance expressed using tags that annotate the interface modeling the corresponding service

– the QoS constraints specified annotating OCL [7] or expressions to the modeling elements to which they are applied
– the relationships between resources, services and service users captured using the stereotypes defined in the UML Real-Time and UML Platform profiles

Packages are useful modeling elements. A package is ``a general purpose mechanism for organizing elements into groups in order to manipulate them as a set" [2]. Hence, packages are used to provide abstractions and improve the readability of the models, as they group together modeling elements that are closely related.

4.4 Communication Refinement

Communication refinement is the procedure that defines, through a sequence of steps, how to implement the interaction among objects. A *Network Platform (NP)* is a library of resources that can be selected and composed together to form *Network Platform Instances (NPIs)* and to support the communication among a group of interacting objects, called NPI users. An NP library includes logical resources (e.g. protocols, virtual channels...) that are defined only in terms of their functionality and physical resources such as memory, processor, physical medium.

During communication refinement, UML Platform is used to model NPIs and their relationships. In the model of an NPI the platform users are the components in the upper platform layers and are represented as stereotyped classes, the services are of communication type (send and receive primitives) and are modeled as interfaces, the resources are protocols and channels and are represented as stereotyped classes of type *<<Process>>* and *<<Medium>>* respectively. These elements of the model are annotated with QoS requirements and may be related by the stereotypes *<<Use>>*, *<<Communicate>>*, *<<Implement>>* and *<<Refine>>*.

Figure 5-8 shows the UML Platform model of an Application Layer (AL) protocol and its refinement into a Network Layer (NL). The UML Platform model of the AL includes two Process classes representing a Controller and a Sensor, a (one-directional) *<<Communicate>>* relationship, a description of the communication services and a Medium modeling the AL network platform that provides the services. To communicate the Sensor and Controller *<<Use>>* a communication service, whose primitives, send() and receive(), are accessible through the PortSensor and PortController ports. The service is implemented by a platform that consists of a medium with buffer size equal to 1. The platform is refined into a NL platform that includes a protocol entity at each node and a FIFO interconnecting them.

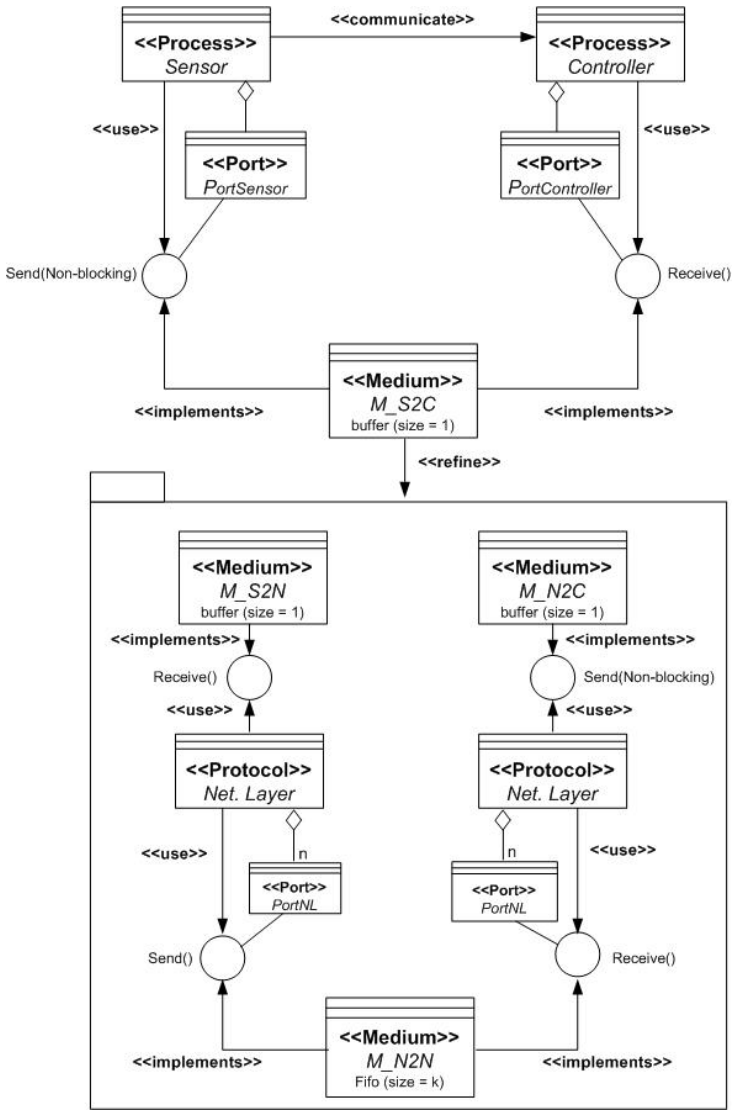

Figure 5-8. Network Platform example

4.5 Mapping

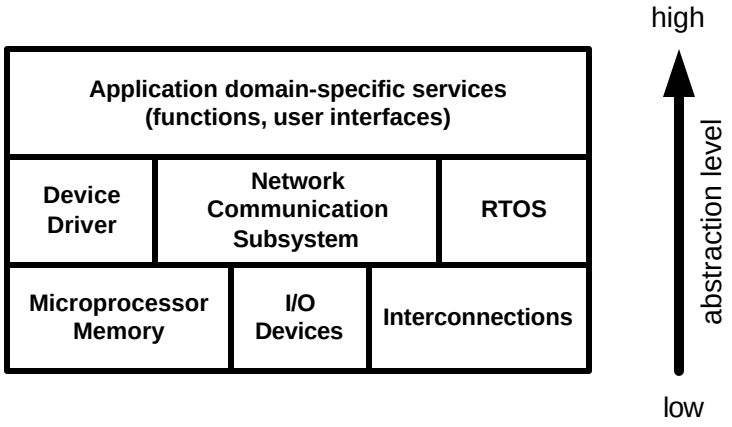

Figure 5-9. Platform Layers

Mapping a functional specification onto an architecture requires the definition of a model of the implementation platform and the establishment of the relationships of the platform resources and services with the application-level functional components. The components assigned to the HW partition are mapped directly onto a HW resource. The components in the SW partition are implemented as SW tasks that use the services provided by an intermediate SW layer. Hence, the elements of embedded system platform models can be conveniently grouped within the three layers shown in Figure 5-9: Architecture (ARC), Application Programming Interface (API) and Application Software (AS) Platforms, where the ARC layer includes a family of micro-architecture HW resources such as microprocessors, memories, ASICs, FPGAs, I/O devices and inter-connectors that in UML are typically modeled as deployment nodes; the API layer is a software abstraction layer and includes RTOS, device-driver, and network communication subsystem that in UML are represented as components; the AS layer includes the software tasks that implement application-level functional components.

Figure 5-10 shows the model of the embedded system platform implementing the Intercom Protocol Stack [13]. The Intercom protocol layers are visualized on the top part of the figure: they are modeled as classes and are annotated with QoS constraints on parameters like error rate (error-free transport layer connection) and throughput (the Mu-law Quantization block must process voice samples at 64 kbps, while the required throughput at the Physical Layer, due to the TDMA policy and the

CRC redundancy, is higher than 1.5 Mbps). Two layers, UI and Transport, are implemented as SW tasks and define the AS Layer; all other protocols are mapped onto HW resources (ASICs or FPGAs). The API Layer includes the RTOS and the Device Drivers. The RTOS implements the communication, arbitration and execution services used by the software tasks. Some of its parameters are the scheduling policy and the context switch overhead. It uses the services offered by the device drivers. The ARC layer includes the Tensilica Xtensa processor, the I/O devices, the Sonics Silicon backplane, ASICs and FPGA blocks that are implemented as deployment nodes. The processor provides to the upper layers ISA execution services, the I/O devices provide HW interface services

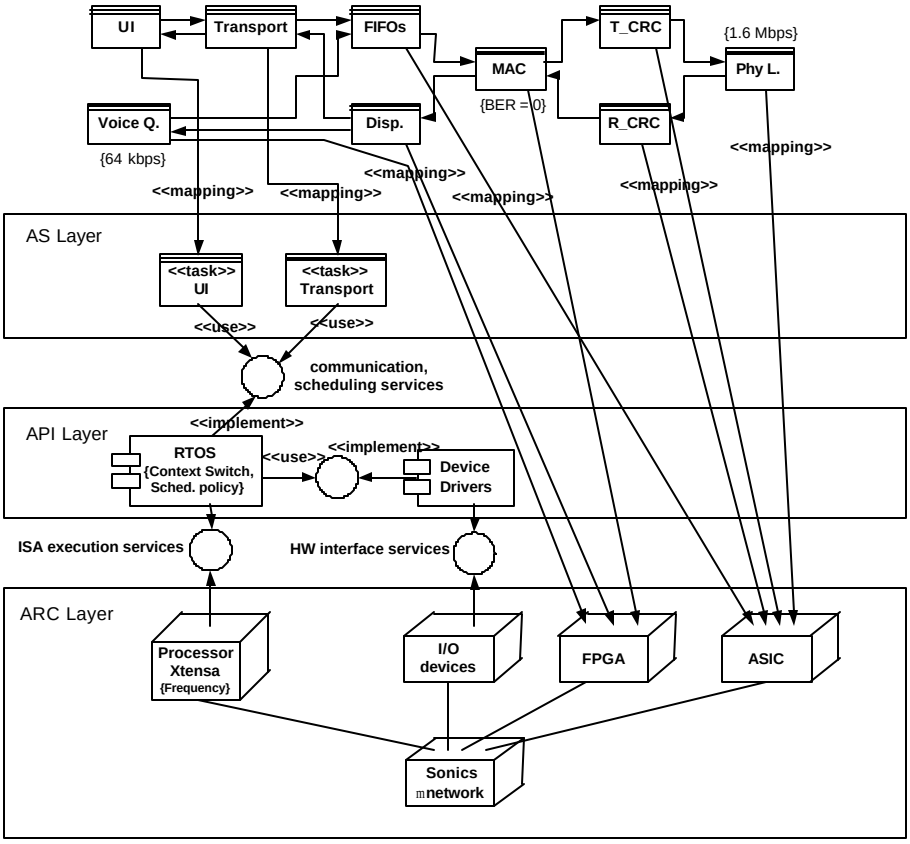

Figure 5-10. Mapping Intercom Protocol onto an embedded system platform.

5. CONCLUSIONS

This chapter has discussed the issues of using UML in the context of platform-based design. A new UML profile, UML Platform, has been proposed by introducing new building blocks (e.g. new stereotypes) to represent platform resources and services. Further, an embedded system design methodology that uses UML Platform and follows the platform-based design principles was presented. We believe that industry convergence on a standard platform profile in UML will be vital for the development of a variety of tools and methods that will better support embedded system design and help automate the flow from specification to platform-based implementation.

REFERENCES

[1] A. Sangiovanni-Vincentelli, "Defining Platform-based Design", *EEDesign*, February 2002.

[2] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998

[3] G. Martin, L. Lavagno, J. Louis-Guerin, "Embedded UML: a merger of real-time UML and co-design", *Proceedings of CODES 2001*, Copenhagen, April 2001, pp.23-28.

[4] G. de Jong, "A UML-Based Design Methodology for Real-Time and Embedded Systems", Proceedings of DATE 2002, Paris, March 2002.

[5] B. Selic, "Complete High-Performance Code Generation from UML Models", *Proceedings of Embedded System Conference*, San Francisco, CA, USA, March 2002.

[6] C. Raistrick, "Executable UML for Embedded System Development", *Proceedings of Embedded System Conference*, San Francisco, CA, USA, March 2002.

[7] J. Warmer, A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Object Technology Series, Addison-Wesley, 1999.

[8] B. Selic, "A Generic Framework for Modeling Resources with UML", *IEEE Computer*, June 2000, pp.64-9

[9] B. Selic, J. Rumbaugh, "Using UML for Modeling Complex Real-Time Systems", White paper, Rational (Object Time), March 1998.

[10] P. N. Green, M. D. Edwards, "The Modeling of Embedded Systems Using HASoC", Proceedings of DATE 2002, Paris, March 2002.

[11] F. Balarin, L. Lavagno, C. Passerone, Y. Watanabe, "Processes, interfaces and platforms. Embedded software modeling in Metropolis", *Proceedings of EMSOFT 2002*, Grenoble, France, October, 2002

[12] ARTISAN Software Tools, Inc. et al., "Response to the OMG RFP for Schedulability, Performance, and Time", OMG document number: ad/2001-06-14, June, 2001

[13] J. da Silva Jr., M. Sgroi, F. De Bernardinis, S.F Li, A. Sangiovanni-Vincentelli and J. Rabaey, "Wireless Protocols Design: Challenges and Opportunities", *Proceedings of CODES 2000*, SanDiego, CA, USA, May 2000.
[14] E. Lee, D. Messerschmitt, "Synchronous Data Flow", *Proceedings of the IEEE*, September, 1987.
[15] H. Hsieh, J. Lin. Modeling SDF in Metropolis, Private Communication