Chapter #

EMBEDDED SYSTEM DESIGN USING UML AND PLATFORMS

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

Important trends are emerging for the design of embedded systems: a) the use of highly programmable platforms, and b) the use of the Unified Modeling Language (UML) for embedded software development. We believe that the time has come to combine these two concepts into a unified embedded system development methodology. Although each concept is powerful in its own right, their combination magnifies the effective gains in productivity and implementation. This paper defines a UML profile, called UML Platform, and shows how it can be used to represent platforms. As an example, the Intercom platform designed at the Berkeley Wireless Research Center is presented to illustrate the approach.

Key words: UML, Platform-based Design, Embedded System, Quality of Service

1. INTRODUCTION

Embedded System Design (ESD or just ES) is about the implementation of a set of functionalities satisfying a number of constraints ranging from performance to cost, emissions, power consumption and weight. Due to complexity increases, coupled with constantly evolving specifications, the interest in software-based implementations has risen to previously unseen levels because software is intrinsically flexible. Unlike traditional software design, embedded software (ESW) design must consider hard constraints such as reaction speed, memory footprint and power consumption of

software because ESW is really an implementation choice of a functionality that can be also implemented as a hardware component, so that we cannot abstract away the hard characteristics of software. For this reason, we believe ESW needs to be linked 1) upwards in the abstraction levels to system functionality, 2) to the programmable platforms that support it, thus providing the much needed means to verify whether the constraints posed on ES are met.

UML is the emerging standard meta-notation (a family of related notations) used in the software world to define many aspects of object-oriented software systems [4]. Now UML is also capturing much attention in the ESW community as a possible solution for raising the level of abstraction to a point where productivity can be improved, errors can be easier to identify and correct, better documentation can be provided, and ESW designers can collaborate more effectively. An essential deficiency is that UML standardizes the syntax and semantics of diagrams, but not necessarily the detailed semantics of implementations of the functionality and structure of the diagrams in software. In [2], some requirements are identified that have to be satisfied to make UML a suitable development basis for embedded systems design, in terms of notation, semantics, refinement steps, and methodologies. The UML Platform profile described in this paper can be considered as a notation to satisfy some of the requirements of [2].

Platform-based design has emerged as one of the key development approaches for complex systems, including embedded systems in the last several years. In [1], an architecture platform is defined as a specific 'family' of micro-architectures, possibly oriented toward a particular class of problems that can be modified (extended or reduced) by the system developer. 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. Furthermore, [1] defines an API platform, a software layer that abstracts computational resources and peripherals contained within the architecture platform, to hide unnecessary implementation details from embedded software developers. A platform can be described in terms of the type and quality of the services it offers to its users. Quality of Service (QoS) parameters, e.g. processing speed and I/O bandwidth, define platform performance and reliability and therefore are the essential distinguishing factors between platforms. The task of a designer is to find a platform that best meets the QoS requirements of applications [3].

When we combine UML with the platform-based design concept, we see, following the reasoning of [2], that it is necessary to have a way of describing those platforms in UML, i.e., a projection of the platform into the UML notation space. The definition of this projection is the purpose of this chapter that describes a "UML Platform" proposal.

2. RELATED WORK

UML already has the capability to model the most relevant real-time system features, such as performance (using tagged attributes or OCL [10]), resources (using Component or Deployment Diagrams), and time (using classifiers and tagged attributes). However, in absence of a standard and unified modeling approach, the same embedded systems specification may be modeled in several different ways. Therefore, how to use UML for modeling real-time systems has become recently an active area of research and several proposals have been made.

The Real-Time UML profile [3] 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 by designers to build models of real-time systems annotated with relevant QoS parameters. The profile standardizes an extended UML notation to support the interoperability of modeling and analysis tools but touches little on platform representation. UML-RT [11] 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 takes place by means of protocols that define the sequence of signals exchanged through stereotyped objects called 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. Rose RT). HASoC [5] is a design methodology based on UML-RT notation. 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. De Jong in [9] presents an approach that combines the informal notation of the UML Diagrams with the formal semantics of SDL. 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 specified using use case diagrams; the system components and their interactions are described using block diagrams and message sequence charts, respectively. Then the behavior of each module is specified using SDL that provides an executable and simulatable specification.

2.1 Our approach

In this chapter, we propose a new UML profile, called UML Platform, to model embedded system platforms. First, we introduce a subset of UML notation (new building blocks using stereotypes and tags) to represent specific platform concepts. Then, we show the main levels of abstraction for platforms and the most common types of relationships between platform

components, and how to use appropriate UML diagrams along with aforementioned UML notations to model those platforms and relationships. Last, we explain how to represent platform QoS performance and do constraint budgeting.

2.2 Case study: the Intercom

The Intercom [7] is a single-cell wireless network supporting full-duplex voice communication among multiple mobile users in a small geographical area. Each remote can request one of the following services: subscription to enter active mode, unsubscription to return to idle mode, query of active users, conference with one or multiple remotes, and broadcast communication. The system specification also includes performance requirements on the transmission of voice samples such as latency, throughput, and power consumption. [7] defines a protocol stack that includes Application, Transport, MAC and Physical layers (see Figure 1). Its physical implementation is composed of a reconfigurable embedded processor (Tensilica Xtensa running the RTOS eCos), a memory subsystem, fixed and configurable logic and a silicon backplane (supporting Sonics OCP) that interconnects these components.

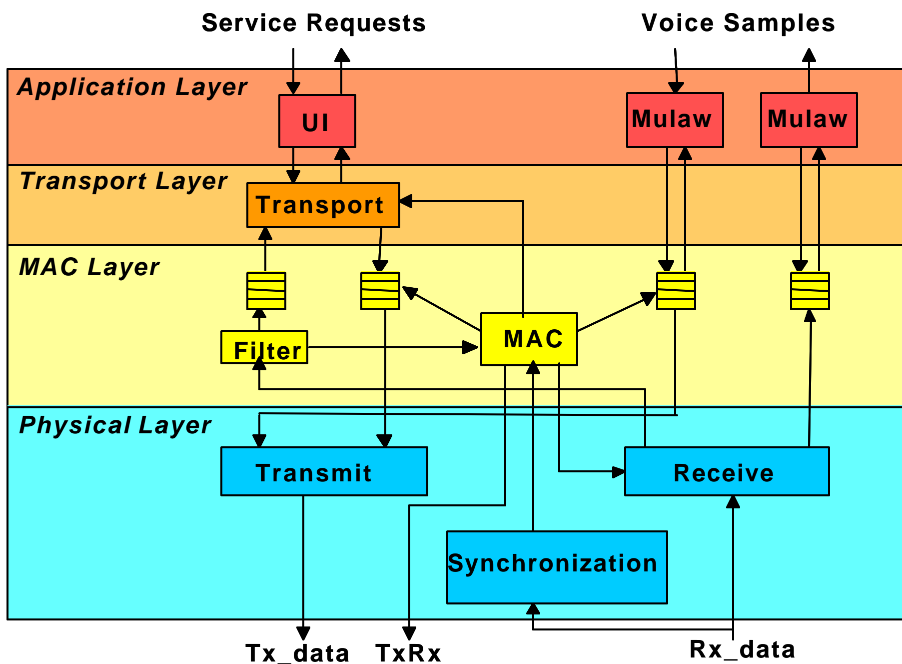

Figure 1. Intercom Protocol Stack

3. THE UML PLATFORM PROFILE

3.1 New stereotypes and tagged values

In this section, we introduce a set of new stereotypes and tagged values for the UML Platform profile. The list of stereotypes and tagged values is derived from the description of several platform examples and, hence, in our opinion it is sufficient to model most embedded platforms.

For each of the building blocks that are frequently used in modeling platform components such as processor, device driver, scheduler, table, buffer, memory, cache, etc., a stereotyped classifier is defined. A stereotyped classifier usually includes a set of attributes and methods that are specified only when the block is instantiated at modeling time. For example, the stereotyped class “cache” is associated with attributes such as "valid", "block index", "tag", and "data", methods such as "write through" or "write back", and QoS parameters such as “hit time” or “miss penalty”.

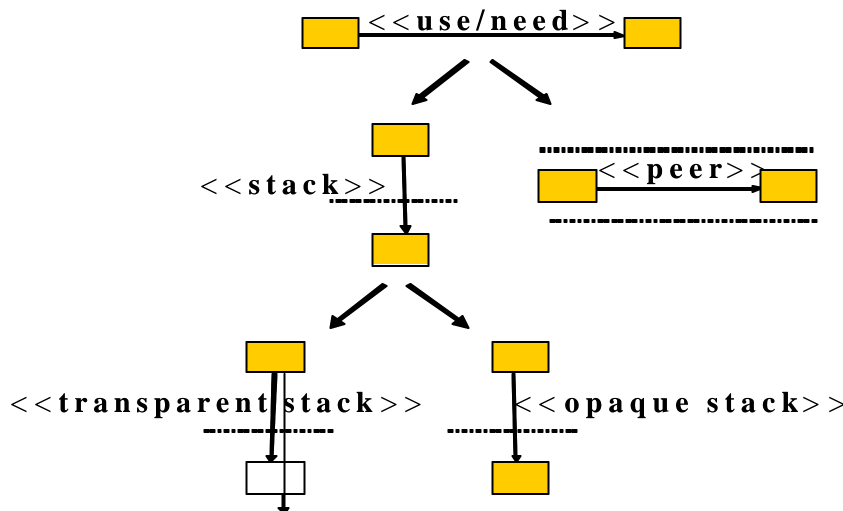

Figure 2. *Use*, *Stack*, *Peer* Stereotyped Relationship

The following stereotypes model common relationships between platform components. At the top of the hierarchy shown in Figure 2, the stereotype <<use>> represents a relationship in which an entity uses a service provided by a resource, while the stereotype <<need>> indicates when an entity needs a service from another entity, but the service is not currently available, i.e. not implemented. Thus it represents a request for future service enhancements.

<<stack>> and <<peer>> are refinements of the stereotype <<use>>. <<stack>> is used when the platform component providing the service and the one using it are at different levels of abstraction. We further specialize this stereotyped relationship into <<transparent stack>> and <<opaque stack>>. <<transparent stack>> models the case where the upper level component knows how the service is implemented within the lower level. So, it is possible for the upper entity to bypass the lower one in search of a service that is simpler, but more suited to the requirements. For example, normally a data transfer function interacts with a medium access control (MAC) function to transfer data, but if a faster transfer rate is desired the data transfer function may bypass the MAC function to directly call a device driver to access the network inter-connector. In such case, the data transfer function and the MAC function are related by a transparent stack relationship. <<opaque stack>> describes the case when the upper level component has no knowledge of how the service is implemented by the lower level component. Thus, the upper entity has to always rely on the lower one to provide the necessary service. For example, a platform service function written in a high-level language declared as an interrupt service routine always relies on an RTOS to save the microprocessor context, identify the interrupt source, and invoke it whenever the interrupt arrives. Due to the insufficient power of the high-level language, it cannot bypass the RTOS to run on top of a bare microprocessor. In this case, the platform service function and the RTOS form an opaque stack relationship. <<peer>> is used when both the platform component using and the one providing the service are at the same abstraction level. In general, peers can only exist within the same level platform, but stack can exist both within and across one level platform.

<<communicate>> is used to relate two components sharing some information. It can be further specialized by stereotypes representing specific models of computation, e.g. <<asynchronous>>, <<RPC synchronous>>, <<rendezvous>>, <<Kahn process>>, etc. <<coupling>> reveals the limited freedom in choosing platform components. There are two types of couplings: <<weak coupling>> and <<strong coupling>>. If whenever one entity is chosen, one from a certain group of entities must also be chosen in order to achieve some functionality, then we say a weak coupling exists between this entity and the group; if whenever one entity is chosen, exactly one other entity has also to be chosen in order to achieve some functionality, then a strong coupling exists between these two entities. Note that, although the <<coupling>> relationship can be also described in OCL [10], the stereotype form is preferred because it is more visible to users. Figure 3 shows an example of the use of <<weak coupling>> between an RTOS and a CPU. These two entities are coupled because when a CPU is used also an RTOS must be used, and vice versa. This coupling is weak in both directions

because there are several types of RTOS that can run on the Xtensa CPU and several CPUs that can support the eCOS RTOS. Finally, we call <<share>> the relationship among multiple entities that use services provided by the same resource (e.g. in Figure 3, tasks sender and receiver share the same CPU). In the presence of <<share>>, it is frequently necessary to deploy an allocation or arbitration scheme, such as a scheduler.

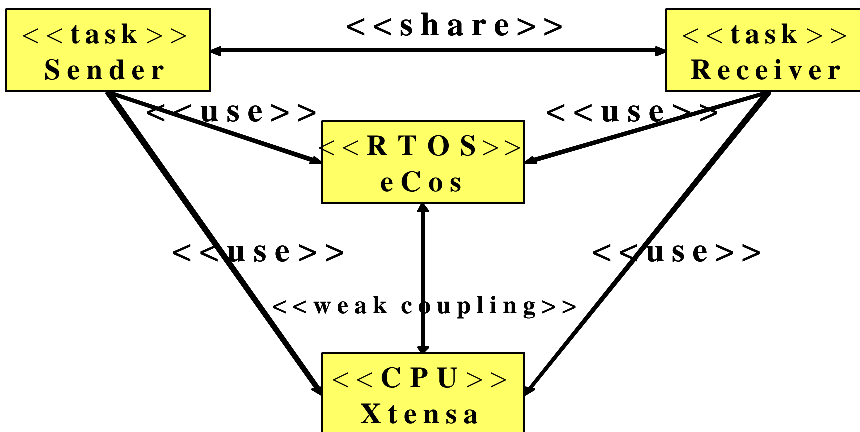

Figure 3. *Share*, *Coupling* Relationship

Tagged values (or called tags) are used to extend the properties of a UML building block with additional information. In the UML Platform profile, they are mainly used to specify QoS parameters, application requirements, as well as communication types, usage types, etc. Examples of relevant tags are: {throughput} for communication throughput, {delay} for time delay between request and response, {precision} for calculation precision, {power} for power consumption, {size} for memory size.

3.2 Platforms

We classify embedded system platforms into three abstraction levels: architecture (ARC), application programming interface (API), and application specific programmable (ASP) platforms (see Figure 4). The ARC Layer includes a specific family of micro-architectures (physical hardware), so that the UML deployment diagram is naturally chosen to represent the ARC platform. The API layer is a software abstraction layer wrapping ARC implementation details. API should be presented by showing what kinds of logical services (represented as interfaces) are provided and how they are grouped together. For example, it is important for users to know that preemption is supported by an RTOS, but not how this service is

implemented inside the RTOS because users (either platform users or developers) rarely need to modify the RTOS itself. For such a purpose, RTOS, device-driver and network communication subsystem are treated as components, i.e., the physical packaging elements for logical services. In UML, their natural representation is the component diagram.

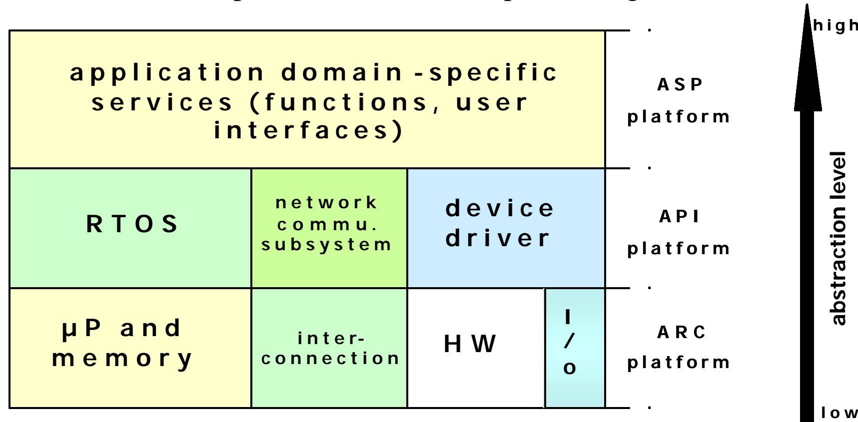

Figure 4. Platforms at Different Levels

ASP is a platform, which makes a group of application domain-specific services directly available to users. For example, the function to set up a connection in the Intercom is such a domain-specific service. In addition to calling these existing services, users sometimes also need to modify or combine them, or even develop new services to meet certain requirements. Consequently, unlike API, here it becomes essential to show not only what functionality these services offer, but also how such services are supported by their internal structures, and how they relate to each other. In UML, the class diagram best represents such information.

Figure 5 shows how Intercom platforms are represented by UML Platform, where:

- ARC, API and ASP platforms are represented by deployment, component and class diagrams respectively;
- A <<transparent stack>> relationship exists within the ASP platform (such as the one indicated by the dotted line between Transport and MAC); an <<opaque stack>> relationship exists between ASP and API (such as the one indicated by the solid line between Transport and eCos), and between API and ARC (such as the one indicated by the solid line between eCos and Xtensa). This implies that Transport may bypass MAC in search of better-suited performance, but it can never bypass eCos;
- <<share>> is used twice: Application and Transport share eCos, while MAC and Physical share device driver.

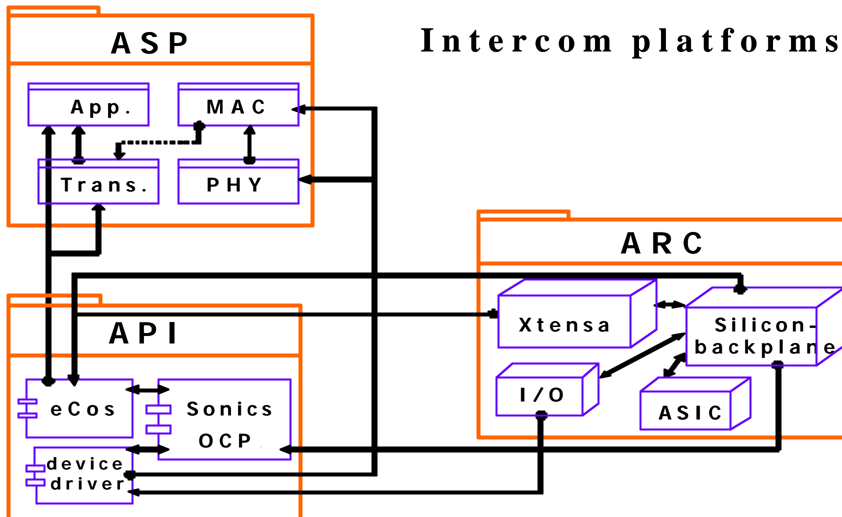

Figure 5. Intercom Represented by UML Platform

3.3 Quality of Service (QoS) and Constraints

QoS parameters identify key performance properties and therefore allow classifying and comparing of different platforms. A set of QoS parameters that completely characterize the needs of any application is obviously impossible to define. However, it is possible to decompose QoS properties into just a few orthogonal dimensions (called QoS parameters). For example, for the ARC platform, such dimensions can be power consumption, memory size, processing capability, communication throughput, and for the API platform, task-handling number, task-waiting time, etc. The design constraints can be decomposed along the same dimensions, and this will conceivably enable some form of automatic matching and analysis between QoS properties and design constraints. In [6], the former are called *offered QoS* (the values are given by the platform providers), the latter *required QoS* (the constraints on these variables are specified by the platform users), and we adopt such terminologies here. In general, we model QoS parameters by annotating classifiers and relationships with tagged values.

Constraints complement the functional specification of an ES with a set of formulae that declare the valid range for variables in the implementation. At the beginning of the design process, constraints are defined for the global system performance and then propagated (through *budgeting*) to lower levels of abstraction to bind the performances of local components as the design gets refined. The objective of this chapter is neither to define a methodology for constraint budgeting and verification nor to propose a new constraint specification formalism in addition to the ones in [8] and [10], but to provide

guidelines on using the UML notation in the budgeting process: annotate diagrams with tags describing constraints and use the graph structure to show how the lower-level components are connected (either sequentially or concurrently) to drive the budgeting process. Assume a constraint ϕ is given on the minimum throughput of a node, and this node is refined into multiple components. If two components are composed in sequence, ϕ is propagated to both, because throughput is non-additive for series composition. Instead, if two components are composed in parallel, two constraints, $\phi 1$ and $\phi 2$, can be derived, provided their sum does not exceed the original constraint ϕ.

4. CONCLUSIONS

In this paper we have presented a new UML Profile, called UML Platform. We have introduced new building blocks to represent specific platform concepts; selected proper UML diagrams and notations to model platforms at different abstraction levels including QoS performance and constraints. As future work we will develop a full design methodology and tool set based on UML Platform.

REFERENCES

1. A. Sangiovanni-Vincentelli, Defining Platform-based Design, EEDesign, Feb 2002.
2. G. Martin, L. Lavagno, J. Louis-Guerin, Embedded UML: a merger of real-time UML and co-design, Proceedings of CODES 2001, Copenhagen, Apr. '01, p.23-28.
3. Bran Selic, A Generic Framework for Modeling Resources with UML, IEEE Computer, June 2000, pp.64-69
4. J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language User Guide, Addison-Wesley, 1998
5. P. N. Green, M. D. Edwards, The modeling of Embedded Systems Using HASoC, Proceedings of DATE 02.
6. 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.
7. 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 the 8th IEEE International Workshop on Hardware/Software Codesign, *CODES '00*, S.Diego, CA, USA, May 2000.
8. Rosetta, www.sldl.org
9. G. de Jong, A UML-Based Design Methodology for Real-Time and Embedded Systems, Proceedings of DATE 02.
10. J. Warmer, A. Kleppe, The Object Constraint Language: Precise Modeling with UML, Object Technology Series, Addison-Wesley, 1999.
11. Selic, J. Rumbaugh, Using UML for Modeling Complex Real-Time Systems, White paper, Rational (Object Time), March 1998.