A UML-Based Design Methodology for Real-Time and Embedded Systems

Gjalt de Jong – Telelogic
Naamsesteenweg 539 / 15
B 3001 Leuven, Belgium
gjalt.dejong@telelogic.com

Abstract

The fast growing complexity of today's real time embedded systems necessitates new design methods and tools to face the problems of design, analysis, integration and validation of complex systems. We present a system level design method for embedded real-time systems combining the informal strengths of UML with the formal strengths of SDL. We demonstrate our flow by the design example of a telecommunications application from the wireless or access domain, showing the applicability of the flow to control and data-dominated types of systems. Finally we will show how the application results and other end-user needs and requirements influenced the current UML 2.0 proposal with support for real-time and embedded systems.

1. Introduction

The design of real time embedded systems copes with the problems of ever growing complexity. Integration and validation of complex systems becomes a bottleneck in the traditional design flow. The full separation of hardware (HW) and software (SW) design can not longer be sustained for system-on-chip designs, comprising of general purpose microprocessors, digital signal processors (DSP) and dedicated hardware.

HW design and real-time SW design are very similar, as both mainly deal with concurrent objects and use very similar concepts to describe and reason about a system. Their *implementation* description may differ in the time granularity used and whether the system is specified synchronously or asynchronously. At the RT level, the (clock-cycle) instruction is the granularity of specification, whereas real-time software design may be written on packet-based time granularity. Code can be written as periodic, or as data- and event driven. However with today's complex systems, all these concepts are now present in embedded systems, and the use of a single modelling concept is insufficient.

A typical telecommunication design contains data-paths and control planes. Data-path functions represent transformations on the data as it flows through the system. A control plane configures and schedules the data-paths. Control flow and data operations can be described independently, *both* in hardware and software. In most cases, the control "plane" constitutes the real system IP, and, typically due to underlying implicit assumptions, is the most difficult to design and validate.

In this paper we mainly address the control, communication and synchronization refinement during the conceptual design stages. As complimentary to the full functional view, which typically concerns the algorithmic design, they together constitute a full system design methodology, reflecting the concurrent engineering system and architecture design within industrial system houses.

The key aspects of our flow are the use of OO concepts to specify hardware and software components. As in UML 2.0, the informal strengths of UML are married with the formal strengths of SDL in this specification capturing flow.

2. System design methodology

Traditional design starts from written specifications. Written specifications have a number of well-known shortcomings. They are written in natural language and are ambiguous. Typically these informal, incomplete specifications are immediately refined into an RT level description. The SW components are then integrated into a complete system at RT level or later.

To alleviate the problems due to late integration and validation, most research in the context of system-on-chip and HW/SW co-design is on (formal) specification and to provide an automated link to implementation. Existing system-level design tools, such as Cossap, SPW, Bones, and Opnet, also address such full functional specification. Assuming zero-delay models and the availability of infinite resources, performance and the impact of HW system architecture are only addressed at the lower-levels of implementation.

But this is only a limited view on system level design. As important is the specification capturing phase [9] and the overall methodology, i.e. the upper part of the flow as illustrated in Figure 1.

Environments and methodologies are also required to refine and decompose the control, communication and synchronization within heterogeneous real-time systems. Nowadays, it is all specified at a single level of abstraction, and not able to disentangle its higher-level structure. State-transition oriented languages like Esterel,

SDL, Statecharts are presented to model these types of systems, and which also target implementation.

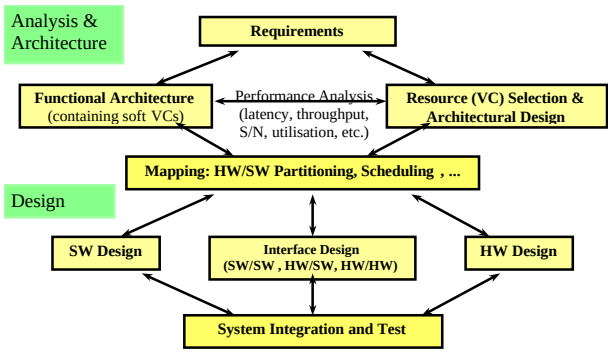

Figure 1: System level design flow

Within the domain of software engineering, more support is available for a complete system level design methodology and for the capture of specifications. (OMT, UML, ...). We believe that the object-oriented paradigm is the first, widely-accepted, approach to cover the "external" and declarative view of a system. At the same time, it bridges seamlessly with the internal implementation, or specification, view of a system, which mainly has been addressed up to now.

To our knowledge, Octopus [1] is the first approach to transform a generic OO methodology into a concrete flow for real-time systems. Earlier approaches from structured analysis design such as Shlaer-Mellor, are much more state-transaction based, and therefore implementation oriented.

SDL is most known as a formal system specification and design language, typically used for control-oriented telecommunication systems. However, it is being designed for and it supports a combined structured analysis and object-oriented design methodology, as for example described in SOMT [8]. This is also exemplified in the current definitions of UML 2.0 that combines the formal strengths of SDL for rigorous specification, design and implementation with the strengths of UML during the more informal requirements analysis phases. Such a methodology is described next.

2.1 System specification flow

The complete specification flow is illustrated in Figure 2. Each phase of the system level design flow is characterized by a set of workproducts.

We have used the UML language and methodology/process [3],[6] for the first ***formulation*** phase of system design and the specification capture. SDL, or rather the SDL concepts as defined in UML 2.0, is then used, with additional patterns, during the subsequent ***formalization*** phase. In both these external view oriented phases, the static and dynamic views of UML and SDL are to be used for the specification of the system communication and coordination.

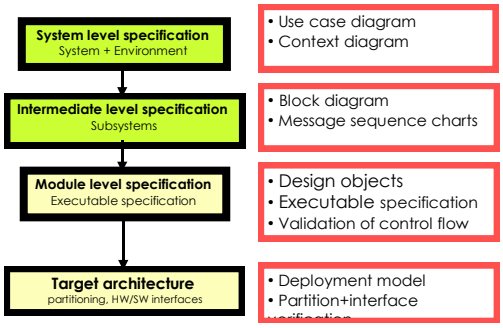

Figure 2: Design flow and workproducts

Thereafter, the detailed ***design*** phase addresses the internal specification of the modules and other constituting elements. Full-functional specifications are to be described with detailed SDL, i.e. formal and rigorously defined state charts, and C behaviours.

The fourth level in the design flow is the ***deployment level*** specifying the target architecture to which the functionality of a system is mapped, and the mapping itself. We refer to [4] how it is to be addressed in UML.

The formulation phase starts from the idea of what the system is envisioned to do. The system level specification shows the system in its real environment by means of *use cases* and *context diagrams*.

For the formalization phase, we propose a gradual progression from a conceptual model to the final and fully formal specification model, through a set of increments and refinements. Documents at this level show how subsystems work together to realize the use cases using two kinds of graphs. The blocks and inter-block communication channels are depicted in a *block diagram* (or: UML superstructure); the dynamic behaviour is depicted in a *message sequence chart*. At this level, we also add block diagrams specifying the subsystems, the testbench and the “clocks” in the system.

The next step is the definition and formalization of the interfaces and the communication, or rather the coordination, between the system and its environment, and between the subsystems and other components. For this definition of attributes that go with the graphical documentation, and their semantics are key. The stereotyping capabilities of UML are appropriate for this and the standardized attributes and types in UML 2.0 allow for such specification and analysis of performance, schedulability and QoS aspects. See [7] for details.

Furthermore stereotypes for interface patterns as in [2] are defined and which are associated with patterns and scenario’s describing their (internal) behaviour. All concepts translate easily to a number of simple

constructs in (more) typical interaction oriented and/or state-transition based specification formalisms, such as SDL and state charts. They can be implemented as remote procedure calls, services, or as direct signals and transitions, in a synchronous and asynchronous way. As patterns, they do not restrict the design and implementation, but show typical uses and templates.

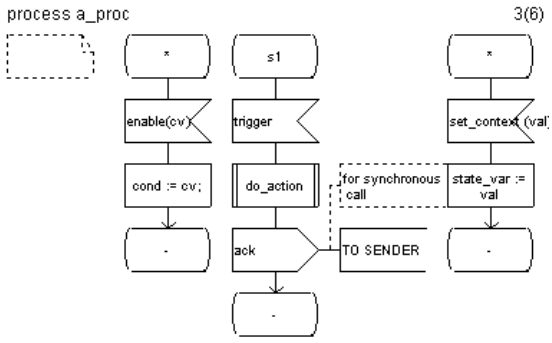

Figure 3: Interface patterns as SDL template specifications

The design intent is captured immediately and intuitively by the use of such stereotyped concepts, thereby greatly increasing the understandability, and transferability of the knowledge, and facilitating the step towards executable specification. Automated refinement and implementation generation can easily support this attribute and pattern-based capture of design intent.

2.2 Diagrams

The diagrams used in the flow are briefly described next. See e.g. [7] for more details.

A use case diagram captures the functional requirements of the design from external actors' point of view. A use case answers the question: what does the system do for each actor. Use cases enumerate the system activities, divided into autonomous (periodic/clocked) ones and activities performed under the effect of an external actor.

A context diagram shows the external actors and the interfaces of the system with its environment.

A block diagram shows the decomposition of the system or a subsystem into a number of smaller subsystems and the related interfaces. It shows the context of the subsystems in the total system.

The above captures the *architectural or static view* of a system; the *behaviour or dynamic view* is captured with the following two types of diagrams.

A message sequence chart provides a dynamic view of a (sub-) system. With each block diagram corresponds one message sequence chart. These charts are use case driven: 1) what are the sequences of messages that realize a particular use case? and 2) how do the entities collaborate to realize a given scenario?

A state-transition graph describes the internal dynamic behaviour of a component. A state-transition diagram or process graph is complementary to a message-sequence chart that describes the external view of typically a single scenario, and which must be compatible. It is used to specify in more detail the internal synchronization and ordering constraints.

3. Conclusions

We have presented a system design flow for real-time and embedded systems. The flow combines the informal strengths of UML with the formal strengths of SDL, as of UML 2.0. The specification capturing methodology is a more formal approach to support joint hardware and software system and architecture design. We have mainly focused on the control aspects that need to be refined during the conceptual design stages. Due to space restrictions, the target architecture aspects and use of UML deployment views have not been addressed in this paper, but are also covered in [4].

The design flow has been demonstrated a.o. by the system level design of a VDSL modem [5]. This application also showed that the flow seamlessly links with the specification, detailed design and implementation phases for a full HW and SW development process.

4. References

[1] M. Awad, J. Kuusela, and J. Ziegler, "Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion", Prentice-Hall, 1996.

[2] B. Bailey, G. de Jong, P. Schaumont, and C. Lennard, *"Interface Based Design Using the VSI System-level Interface Behavioral Documentation Standard"*, System-on-chip Methodologies & Design Languages, pp. 317-331, Kluwer, 2001

[3] I. Jacobson, G. Booch and J. Rumbaugh, "The Unified Software Development Process", Addison- Wesley, 1999.

[4] G. Martin, *"UML for Embedded Systems Specification and Design: Motivation and Overview"*, these proceedings.

[5] A. Niemegeers and G. de Jong, *"System Level Design at Work"*, Medea Conference on System Level Design, Antwerp, September 1999.

[6] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1999.

[7] B. Selic, *"The Real-Time UML Standard: Definition and Application"*, these proceedings.

[8] *"SDL-oriented Object-Modeling Technique"*, Telelogic

[9] P.H.A. van der Putten, and J.P.M. Voeten, "Specification of Reactive Hardware/Software Systems", Ph.D. Thesis, Eindhoven University of Technology, Eindhoven, the Netherlands, 1998.