Towards Formal Support for UML-based Development of Embedded Systems

Jozef Hooman
University of Nijmegen, the Netherlands
http://www.cs.kun.nl/~hooman/
E-mail: hooman@cs.kun.nl

Abstract—**We describe ongoing work on the definition of a UML-based development methodology for the software of embedded systems. The aim is to improve current tools and methods by incorporating formal techniques. As a starting point, we define a formal semantics for a selected subset of UML. Next this language is extended to increase expressibility, especially concerning timing.**

Keywords— **UML, embedded systems, real-time, formal methods, development methodology**

I. INTRODUCTION

To increase the quality of software for embedded systems, we investigate the possibility to support UML-based development by formal techniques. The aim is to propose a methodology that incorporates formal techniques during all development phases in a coherent and consistent way. This work is carried out in the context of EU-project Omega (Correct Development of Real-Time Embedded Systems). This project started 1 January 2002 and hence the current paper only describes the main ideas and preliminary results.

As a starting point, we select a subset of UML [3] that is suitable for our application domain and allows the definition of a formal semantics. This subset is extended and adapted to be able to deal with real-time systems in formal way. This leads to the so-called Omega kernel model, which provides an unambiguous, precise meaning to our UML-based modeling language.

Based on the kernel model, we define a development methodology that describes how the selected notations can be used to develop real-time embedded systems, supported by formal techniques.

An important aim of Omega is the development of formal tools for the analysis and verification of design steps. The project addresses various techniques such as model-checking of timed and untimed models, interactive theorem proving to support compositional reasoning, refinement rules relating different levels of abstraction, and synthesis from specifications. The developed formal tools are connected to commercial UML-tools. In this project, we focus on connections with Rhapsody and Tau, supported by the companies that produce these tools.

The basic idea is that users of the Omega technology mainly use commercial UML-tools to design their embedded systems. In addition, they may use some specialized notations for specifications such as Live Sequence Charts (LSCs) [4] and some logical formulas, such as the Object Constraint Language (OCL) [19]. All used notations are interpreted on the common, formal kernel model that ensures the consistency and coherence of the formal verification tools (the main ones are shown in green in Figure 1).

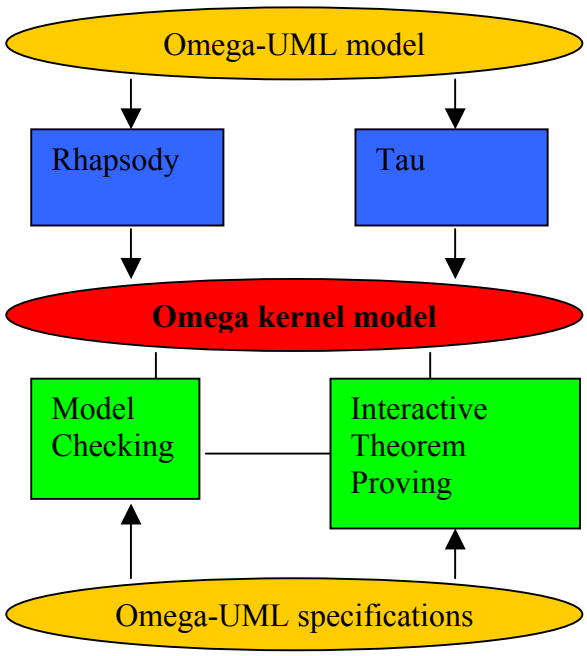

Figure 1: Omega tools connections

The developed methodology and tools will be evaluated on industrial cases studies that are provided and elaborated by the industrial partners of the project. These partners also defined user requirements for the kernel language, the methodology and the tools, based on their experience in the embedded systems domain.

In the rest of this paper, we describe the Omega kernel model in section II. The current, preliminary ideas on the development methodology are described in section III. Concluding remarks can be found in section IV. An appendix contains the list of partners and a link to a web page with more information.

II. Omega Kernel Model

In this section we describe the main notations that are used in the Omega development process. In subsection II.A we briefly present the basic, untimed, kernel model and Subsection II.B describes timing extensions. Subsection II.C briefly mentions the notations to specify properties, namely Live Sequence Charts and OCL.

A. Basic Untimed Kernel Model

Basically, the kernel model [5] is a suitable subset of the current UML-version (*UML 1.4 with Action Semantics*, see [3] for the latest version). When relevant, also concepts from UML 2.0 proposals (e.g. of the U2 Partners [18]) will be adapted.

The basic kernel language contains *class diagrams*, which show a collection of declarative (static) *model elements*, such as *classes* and *relationships*. A class is a description of a set of objects that share the same attributes, operations, signals, relationships, and semantics. So-called *reactive* classes have an associated *state diagram* to process events. We also distinguish *active* and *passive* classes, where active classes have their own thread of control and associated event queue.

Relationships include *generalization* and *association*, with *aggregation* and *composition* as special forms of associations. For simplicity, we exclude association classes here. Multiplicities might be used to specify restrictions on the number of related objects. Other attributes are, e.g., changeability and visibility.

We define a restricted *action language*, which includes the creation and destruction of objects, the call of operations, the computation of primitive operations, and the emission of signals. Operation calls are executed synchronously (the caller is blocked until the call returns), whereas signal-based communication is asynchronous (the sender may continue).

The kernel model defines a formal semantics for the behaviour of a set of related classes and associated state diagrams. It is defined as a symbolic transition system. A snapshot of the system during execution is represented by two system variables, a system configuration *sconf* and a pending request table *prt*.

For each class, there is an infinite array of objects and *sconf* records for each object its status (dormant, executing, suspended, etc.), the values of its attributes, the current state of the associated state diagram (for reactive classes), and the current event queue (for active classes).

The pending request table captures all pending operation requests during system execution in a global table.

The transition predicate defines the possible atomic steps of objects, and system execution is represented as the asynchronous interleaved execution of steps of active objects. We list a few decisions taken in this semantics:

- When an object is executing an operation, no other thread of control is active in the same object. In general, at most one thread of control is active in each object at any point in time.
- We use a run-to-completion semantics, that is, the response to an event from the event queue of an instance of an active class is executed until a stable state is reached in which no local transition (i.e. a transition that does not require an external trigger) can be executed. Pre-emption of such a run-to-completion step is not allowed.

B. Timing Extension

The current UML-specification and proposals for UML 2.0 only mention time in a very limited way. But the adopted UML Profile for Schedulability, Performance and Time [17] defines a large number of timing mechanisms with properties as resolution, skew and drift of clocks and services as set, get, reset, pause, start of clocks. It, however, does not define a precise formal semantics of these concepts. Hence, the first aim is to define a few basic timing extensions that have a proper formal semantics. We mention some of the main ingredients of the proposed timing extensions within Omega.

A special data type *Time* is introduced, with a particular instance called *NOW* that gives the current moment of time. Time values are assumed to be

increasing and divergent (non-Zenoness). A *timer* is an active entity that can be set (activated) and reset (inactivated). It issues a TimeEvent when a time duration elapsed. A *clock* is a passive reactive entity that measures time progress and answers to requests on its current value.

Guards of transitions might include time conditions, triggers can also be TimeEvents, and the action language now includes timing operations. Moreover, with each transition in a state diagram, a notion of *urgency* is attached, which defines when the transition must be taken when it is enabled. Similar to [2], there are three possibilities:

- *eager* transitions are urgent as soon as they are enabled, that is, time cannot progress as long as they are enabled;
- *lazy* transitions are never urgent, their execution can always wait, and they might become disabled by progress of time;
- *delayable* transitions become urgent when they are enabled and progress of time would disable them.

C. UML-based Specifications

In UML-methods usually sequence diagram are used to specify properties, e.g. to show the desired behaviour of use cases. They are closely related to Message Sequence Diagrams (MSCs), which are often used to show the communications between objects. Although MSCs are defined as an ITU standard, there are a number of questions about their semantics. Important is the questions whether certain scenarios or certain messages "may" or "must" occur, i.e. whether they are mandatory.

To express this more clearly, *Live Sequence Charts* (LSCs) have been proposed [4], where charts, or parts of it, have an attribute "hot" or "cold" to specify whether it is mandatory (or live) or not. Within Omega also notions to express timing constraints in LSCs are developed.

As an alternative, textual way to express specifications, we consider the *Object Constraint Language* (OCL) [19], and especially the proposal for OCL in UML 2.0 [13]. Note that this proposal also contains a formal semantics of OCL based on some kind of kernel model.

III. OMEGA DEVELOPMENT METHODOLOGY

The aim is to propose a development methodology that improves current UML-based development by incorporating formal techniques in a coherent and consistent way. In this document we present some very preliminary ideas on this methodology.

Since the goal is to incorporate formal methods in the industrial development process, we first propose – in subsection III.A – a basic development process, inspired by current industrial (real-time) development. Note that this is not a formal method; it mainly describes the use of UML-based notations. Next – in subsection III.B – we list the possibilities for formal support of this development process within the Omega project.

A. Development Process

We start with a simple development process, inspired by well-know approaches which are currently used in industry. Examples are the Rational Unified Process (RUP) [11], Octopus [1], ROOM [16], the Rose RealTime approach [15], the Real-Time Perspective method of ARTISAN [12], and ROPES (Rapid Object-Oriented Process for Embedded Systems) [6][7][8].

As a starting point, consider the following development process for a particular System Under Development (SUD), typically a real-time embedded system:

1. Requirements specification and analysis
2. System analysis, definition of an architecture for the SUD
3. Iterate the following steps, for an increasing part of the SUD
   3.1. Analysis and design of a part of the SUD
   3.2. Refine this diagram until it is close to a concrete implementation
   3.3. Produce a (next) version of the SUD by realizing it on a concrete platform

Step 3 iterates on increasing parts of the system, first aiming at quickly producing a prototype and later producing releases of the system with increasing functionality. Often step 3 is use-case driven, that is, in each iteration one of the use cases is realized, starting with the one that contains the largest risks concerning the development of a successful product.

Above we ignored testing, since it is outside the scope of Omega. Also note that this is a highly simplified view

on the development process. Current industrial methods have much more detailed steps and guidelines. Omitted here is, e.g., explicit domain analysis.

Our development process is based on four core workflows:

1. Requirements,
2. Architecture,
3. Analysis & Design, and
4. Implementation.

In the next subsections we explain the UML-notations used in these workflows.

1) Requirements

Requirements are expressed by means of a *use case diagram*, containing actors, use cases and relations between them.

Actors define the environment of the SUD. Especially for embedded systems it is important to model the environment in detail, expressing all assumptions about the behaviour of the environment explicitly. An actor is a role of a person or a device that interacts with the SUD. It is a class with stereotype <<actor>>. Hence it can have attributes and operations. Assumptions about the (timing) behaviour of the actor can be expressed by means of (timed) state diagrams.

Use cases define a subset of the behaviour of the SUD. Usually, a use case is represented by a number of representative scenarios, depicted using sequence diagrams. In Omega, we propose to replace this by Live Sequence Charts (LSCs) with timing annotations. As an alternative, also state diagrams can be used to represent the required behaviour of use cases. It is often also useful to express some global behaviour of the SUD (such as mode changes) in terms of a (timed) state diagram. Finally, there might be a possibly to use (timed) OCL expressions, maybe in addition to the visual notations above, to express additional requirements.

Relations between actors and use cases express which actors are involved in a certain use case. For simplicity, we do not consider relations between use cases, such as the <<extends>> relation.

2) Architecture

The architecture of a software system is described using *components* interacting through *interface*s. Components can be composed of successively smaller components and interfaces. Usually a software architecture also describes an architectural style that guides this organization, e.g. concerning the type of collaborations and compositions.

Our approach closely follows the UML 2.0 proposal of the U2 Partners [18], proving it with a formal semantics based on the kernel model. It contains the following concepts.

An *interface* is a kind of classifier that declares a set of public features and obligations. An interface specifies a kind of contract, which must be fulfilled by any instance of a classifier that realizes the interface. We distinguish *provided* interfaces, representing the obligations to clients, and *required* interfaces, needed to fulfill these obligations.

A *component* represents a modular, deployable, and replaceable part of a system that encapsulates its contents and exposes a set of interfaces. A *port* is a named interface of a component. Components are reused by connecting ("wiring") them together using *connectors*.

Following [18], a component has no behavior of its own. A component has an *external view* (or "black-box" view) by means of its publicly visible properties that are defined as ports. A component also has an *internal view* (or "white-box" view) by means of its private properties, which are its internal classifiers and how they are connected. This view shows how the external behavior is realized internally.

Interfaces might be specified by LSCs, state diagrams, (extended) OCL assertions or a combination of these notations. Similar to [16], a protocol state machine might be attached to a port. State diagrams can also be used to describe the overall coordination mechanism.

3) Analysis & Design

Basic notation in analysis and design is the *class diagram*. A class may use a set of interfaces to specify collections of operations it provides to its environment. OCL constraints can be used to restrict the set of allowed object structures. OCL can also be used to specify pre- and post-conditions of operations. Main part of the behaviour of a reactive class is specified using a state diagram, similar to [9] and [16].

Instead of sequence diagrams, we use LSCs to illustrate typical interactions between objects. Usually, first the class diagram is used to analyze the application domain, representing the main relevant concepts in the domain. Next this is turned into a design of a particular

application. Gradually, details are added to obtain a class diagram that is very close to an implementation of the application. Note that [17] contains concepts that might be useful when describing (detailed) design, such as concurrency unit, atomicity, priority, etc.

4) Implementation

During implementation the detailed design of the SUD is mapped onto a particular physical architecture, including a certain real-time operating system. The UML-models are enriched with bounds on execution times of actions, mappings of processes to processors, scheduling policies, priorities, etc.

We mention a few relevant concepts from [18]. A *node* is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well. *Deployment* is the allocation of an artifact to a node. A deployment specification specifies a set of properties that determine execution parameters of a component artifact that is deployed on a node.

B. Formal Support

We investigate the possibilities to support the development process described in the previous subsection by formal techniques (i.e., methods and tools), mainly focusing on the research activities within the Omega project.

1) Requirements

Concerning LSCs, we provide tool support for this notation to check consistency of specifications. Work on the play-in/play-out approach using LSCs provides a methodology for requirements capture allowing some form of rapid prototyping [10]. Other relevant topics:

- Deducing a formal requirements specification from a combination of notations.
- Formal approaches for requirements analysis, for instance, by applying formal techniques to the closed system consisting of the specified SUD and the assumed behaviour of the external actors. Aim is to detect inconsistencies, ambiguities, incomplete specifications, etc.
- Techniques for refinement of specifications, allowing requirements descriptions at several levels of abstractions (e.g. relating untimed and timed specifications).
- Formal approaches for dealing with requirements evolution during system development and proposals for impact analysis, i.e., possibilities to investigate consequences of changes.

2) Architecture

A formal definition of components and system architecture diagram will be given; here we propose to use the concepts and notations of [18]. We develop compositional techniques to deduce properties of the system from properties of the components. This can be used to related architecture and requirements formally.

3) Analysis and Design

Model-checking possibilities of UML-based design are developed. Also the compositional techniques mentioned above can be used to verify design steps. In some cases, it might be possible to synthesize state diagrams directly from LSC-based specifications. Various types of formal refinement, e.g. between different levels of abstraction and between models of consecutive iterations are studied, for instance, aiming at the definition of refinement steps that preserve certain types of (timing) properties.

4) Implementation

We will develop methods for modeling schedulers of real-time systems. More specifically, we address techniques for consistency checking of scheduler specifications, schedulability analysis, and the generation of schedulers.

IV. Concluding Remarks

This paper presents work in progress on the Omega development methodology. The coming years, these ideas will be evaluated on industrial case studies and more experience of the industrial partners will be incorporated. The aim is to incorporate more detailed guidelines on the use of our kernel language, especially concerning our proposed timing extensions. Also the kernel model will be extended, to provide a formal semantics for our component model at the architectural level and to extend it with concepts such as exceptions and interrupts. The amount of formal support for the development process will be extended gradually, mainly focusing on model-checking, synthesis and theorem proving techniques.

APPENDIX: THE OMEGA PROJECT

The IST-2001-33522 project Omega, "Correct Development of Real-Time Embedded Systems", started 1 January 2002 and has a duration of 3 years. Partners:

Technology Providers:

- VERIMAG, Grenoble, France (Coordinator)
- OFFIS, Oldenburg, Germany
- Christian-Albrechts-Universität Kiel, Germany
- Weizmann Institute, Rehovot, Israel
- Centrum voor Wiskunde en Informatica (CWI), Amsterdam, the Netherlands
- University of Nijmegen (KUN), the Netherlands

Users:

- EADS Launch Vehicles, France
- France Telecom R&D, France
- Israeli Aircraft Industries, Israel
- National Aerospace Laboratory (NLR), the Netherlands

The project is supported by two companies that built the commercial UML-based CASE-tools, Rhapsody and Tau, namely iLogix and Telelogic, respectively.

For more information, see http://www-omega.imag.fr/

REFERENCES

[1] M. Awad, J. Kuusela, J. Ziegler. *Object-Oriented Technology for Real-Time Systems*. Prentice-Hall Inc., 1996. See also http://www-nrc.nokia.com/octopus/

[2] S. Bernot, J. Sifakis, S. Tripakis. *Modeling Urgency in Timed Systems*. Symp. on Compositionality – The Significant Difference, LNCS 1536, pp. 103-129, 1998.

[3] Catalog of OMG Modeling Specifications, http://www.omg.org/technology/documents/modeling_spec_catalog.htm

[4] W. Damm, D. Harel. *LSCs: Breathing life into Message Sequence Charts*. In FMOODS'99 IFIP TC6/WG6.1, Conference on Formal Methods for Open Object-Based Distributed Systems, 1999.

[5] W. Damm, B. Josko, A. Votintseva, A. Pnueli, I. Ober, S. Graf. *A Formal Semantics for a UML Kernel Language*. Omega Deliverable IST/33522/WP1.1/D1.1.1, June 2002.

[6] B.P. Douglass. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Addison Wesley, 1998.

[7] B.P. Douglass. *Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison Wesley, 1998.

[8] B.P. Douglass. *ROPES: Rapid Object-Oriented Process for Embedded Systems*. 1999. Available via: http://www.ilogix.com/quick_links/white_papers/index.cfm

[9] D. Harel, E. Gery. *Executable Object Modeling with Statecharts,* IEEE Computer, 31 – 41, July 1997.

[10] D.Harel, H.Kugler, R. Marelly. *The Play-in / Play-out Approach and Tool: Specifying and Executing Behavioral Requirements.* The Israeli Workshop on Programming Languages & Development Environments (PLE'02), 2002.

[11] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[12] A. Moore, N. Cooling. *Developing real-time Systems using Object Technology, Real-Time Perspective: Foundation and Overview version 1.3*. http://www.artisansw.com , 2000.

[13] Response to the UML 2.0 OCL RfP, available via Klasse Objecten, http://www.klasse.nl/

[14] UML resources: http://www.omg.org/technology/uml/

[15] Rose RealTime, http://www.rational.com/products/rosert/

[16] B. Selic, G., Gullekson, P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[17] UML *Profile for Schedulability, Performance and Time*, http://www.omg.org/cgi-bin/doc?ptc/2002-03-02, see [3].

[18] UML 2.0 Proposal v. 0.671 (draft) – (OMG doc# ad/02-01-12) Revised *UML 2.0 proposal* for public review and comment by U2Partners (http://www.u2-partners.org/).

[19] J. Warmer, A. Kleppe. *The Object Constraint Language – Precise modelling with UML*. Addison-Wesley, 1999.