UML - A Universal Modeling Language?

Gregor Engels, Reiko Heckel, and Stefan Sauer

University of Paderborn, Dept. of Computer Science, D 33095 Paderborn, Germany
engels|reiko|sauer@upb.de

Abstract. The Unified Modeling Language (UML) is the de facto industrial standard of an object-oriented modeling language. It consists of several sublanguages which are suited to model structural and behavioral aspects of a software system. The UML was developed as a general-purpose language together with intrinsic features to extend the UML towards problem domain-specific profiles. The paper illustrates the language features of the UML and its adaptation mechanisms. As a conclusion, we show that the UML or an appropriate, to be defined core UML, respectively, may serve as a universal base of an object-oriented modeling language. But this core has to be adapted according to problem domain-specific requirements to yield an expressive and intuitive modeling language for a certain problem domain.

Keywords: object-oriented model, UML, OCL, profile, class diagram, interaction diagram, statechart

1 Introduction

Main objectives of the software engineering discipline are to support the complex and hence error-prone software development task by offering sophisticated concepts, languages, techniques, and tools to all stakeholders involved.

An important and nowadays commonly accepted approach within software engineering is the usage of a software development process model where in particular the overall software development task is separated into a series of dedicated subtasks. A substantial constituent of such a stepwise approach is the development of a *system model*. Such a model describes the requirements for the software system to be realized and forms an abstraction in two ways (cf. Fig. 1). First, it abstracts from real world details which are not relevant for the intended software system. Second, it also abstracts from the implementation details and hence precedes the actual implementation in a programming language.

Thus, the system model plays the role of a *contract* between a client, ordering a software system, and a supplier, building and delivering a software system. Therefore, the contract has to be presented in a language which is understandable by both the client, generally not being a computer scientist, and the supplier, hopefully being a computer scientist. This requirement excludes cryptic, mathematical, or machine-oriented languages as modeling languages and favors diagrammatic, intuitively understandable, *visual languages*.

M. Nielsen, D. Simpson (Eds.): ICATPN 2000, LNCS 1825, pp. 24–38, 2000.
© Springer-Verlag Berlin Heidelberg 2000

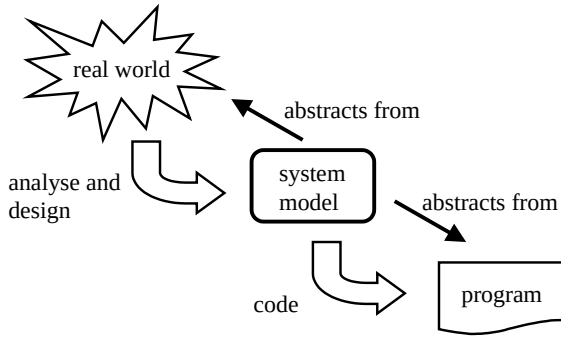

Figure 1. Role of System Model

Besides acting as a contract document between client and supplier of a software system, a system model may serve as a *documentation* document for the realized software system. The existence of such a documentation, which is consistent with the software system, substantially facilitates any required change of the system in the maintenance phase.

In case of missing documentation, respectively system model, a reverse transformation from the software system to the system model has to be performed in order to yield a model of the system on a more abstract and better understandable level where any required change can be discussed with the client and the supplier.

Thus, the system model does not only play an important role in the *forward engineering* process of developing software, but also in *reverse engineering*. Hence techniques are studied and developed to understand and document existing legacy systems, to update their functionality or to integrate them into larger systems. The rebuilding of a generally not existing system model for a legacy system eases the understanding of the consequences of any system update in contrast to dangerous ad-hoc updates of the existing system itself.

The usefulness of an abstract system model was already recognized in the 1970s, when *structured methods* were proposed as software development methods [23]. These methods offered Entity-Relationship diagrams [3] to model the data aspect of a system, and data flow diagrams or functional decomposition techniques to model the functional, behavioral aspect of a system. The main drawbacks of these structured approaches were the often missing horizontal consistency between the data and behavior part within the overall system model, and the vertical mismatch of concepts between the real world domain and the model as well as between the model and the implementation.

As a solution to these drawbacks, the concept of an *abstract data type*, where data and behavior of objects are closely coupled, became popular within the 1980s. This concept then formed the base for the object-oriented paradigm and for the development of a variety of new object-oriented programming languages,

database systems, as well as modeling approaches. Nowadays, the *object-oriented paradigm* has become the standard approach throughout the whole software development process. In particular, object-oriented languages like C++ or Java have become the de facto standard for programming. The same holds for the analysis and design phases within a software development process where object-oriented modeling approaches are more and more becoming the standard ones.

The success of object-oriented modeling approaches was hindered in the beginning of the 90s by the fact that surely more than fifty object-oriented modeling approaches claimed to be the right one, the so-called object-oriented method war. This so-called method war came to a (temporary) end by an industrial initiative which pushed the development of the meanwhile standardized object-oriented modeling language UML (Unified Modeling Language) [18].

UML aims at being a general purpose language. Thus, the question arises whether UML can be termed a *universal language* which is usable to model all aspects of a software system in an appropriate way. In particular, it has to be discussed

- which language features are offered to model a certain aspect like structure or behavior,
- whether horizontal consistency problems are resolved in order to yield a complete and consistent model,
- whether all vertical consistency problems are resolved, such that
  - real world domain-specific aspects can be modeled in an appropriate, intuitive way, and that
  - a transition from a UML model towards an implementation is supported.

It is the objective of this article to discuss these issues. Section 2 will provide an overview on UML and will explain the concepts offered by UML to model system aspects. Section 3 illustrates briefly the UML approach to define the syntax and (informal) semantics of UML. This shows how horizontal consistency between different model elements can be achieved. In addition, extensibility mechanisms of UML are explained which allow to adapt UML to a certain problem domain. Section 4 discusses current approaches to define domain-specific adaptations of UML, so-called *profiles*. The article closes with some conclusions in sect. 5 and a reference list as well as a list of related links to get further information.

2 Language Overview

Object-oriented modeling in all areas is nowadays dominated by the Unified Modeling Language (UML) [18]. This language has been accepted as industrial standard in November 1997 by the OMG (Object Management Group). UML was developed as a solution to the object-oriented method war mentioned above. Under the leadership of the three experienced object-oriented methodologists Grady Booch, Ivar Jacobson, and James Rumbaugh, and with extensive feedback of a large industrial consortium, an agreement on one object-oriented modeling language and, in particular, on one concrete notation for language constructs

was reached in an incremental and iterative decision process. For today, UML version 1.3 represents the currently accepted industrial standard [2,16].

Main objectives for designing the Unified Modeling Language (UML) were the following:

- UML was intended as a general purpose object-oriented modeling language instead of a domain-specific modeling language.
- It was intended to be complete in the sense that all aspects of a system can be described and modeled in an appropriate way.
- It was intended to be a visual, diagrammatic language, as such a language is generally better intuitively understandable than a textual one.
- UML was not intended to be a new language, but an appropriate reuse of best practices of already existing modeling languages which are suited to model certain aspects of a system.
- An important objective was to agree on a formalized syntax and standard notation for modeling constructs, while an informally given semantics definition was found to be acceptable (at least in the beginning of the standardization process).
- As the name says, UML was only intended to be a language. A discussion of an appropriate method or process for deploying UML was intended to be separated from defining the language.
- Despite the fact that UML was intended to be a general purpose language, concepts should already be included in the language which allow to adapt the language towards particular problem domains.

As it is common in structured as well as in object-oriented modeling approaches and in order to meet all objectives stated above, the Unified Modeling Language (UML) was defined as a family or even better Union of Modeling Languages. In particular in order to cover all aspects of a system, several modeling languages are combined where each of them is suited for modeling a specific system aspect. This means that a system model is given by a set of submodels or views where each submodel concentrates on a specific system aspect. On the other hand, the same aspect may be modeled from different perspectives. Thus, the different submodels may overlap and even provide a redundant or conflicting specification of certain system aspects. This approach of providing overlapping, non-orthogonal sublanguages eases the specification process, as the designers may describe the same issue incrementally by interrelating it to other issues. In contrast, the usage of different, even non-orthogonal sublanguages for developing a system model increases the danger of inconsistencies between the different submodels, and thus enforces additional means to handle and to prevent from inconsistencies.

Originating from the concept of an abstract data type, the traditionally distinguished system aspects are the *structural* aspect and the *behavioral* aspect of a system. UML follows this distinction and offers the following sublanguages to specify structural aspects on one side and behavioral aspects on the other side.

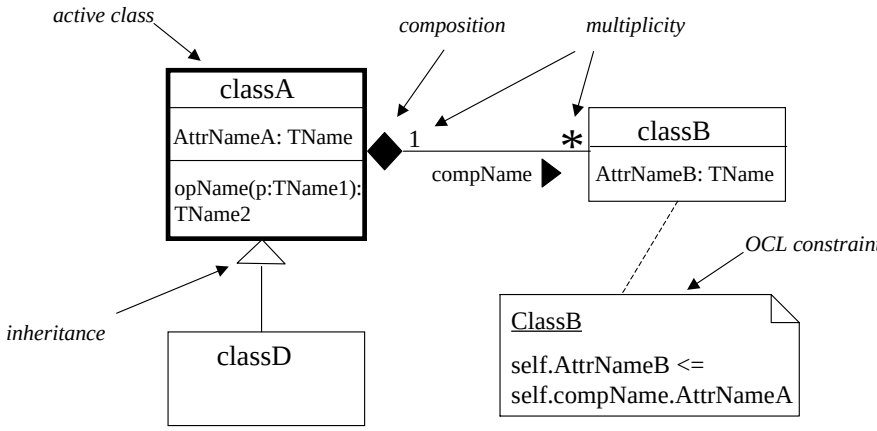

Figure 2. Class Diagram

Modeling the Structural Aspect. UML provides *class* and *object diagrams,* respectively, to model all structural aspects of a system on a type and instance level, respectively. These diagrams originate from Entity-Relationship diagrams [3] and offer means to specify the structure of objects and the possible structural relationships between objects. Objects are described by their attributes as well as by the signatures of operations which may change the state of an object. Structural relationships can be described as general associations or as a weak or strong aggregation relationship between objects, the latter kind of aggregations being so-called compositions. In addition, objects may be specified as passive or active objects, the latter ones having their own, permanently active thread of control. Figure 2 shows the standard notation of UML for these language features in an abstract example.

Allowed object societies, i.e. objects with their interrelations, may be further restricted by additional integrity constraints. These constraints may be formulated in a graphical way and attached to e.g. class diagrams (as for instance, multiplicity constraints of relationships) or may be formulated in the more expressive textual language OCL (*Object Constraint Language*) [22]. OCL is based on predicate logic and may be used in a UML model to specify integrity constraints or invariants for object societies, but also e.g. pre-/post-conditions for operations. For instance, the OCL constraint in fig. 2 states that the value of attribute AttrNameB of an object of ClassB has to be less than or equal to the value of attribute AttrNameA of the object of ClassA reachable via the compName link between these two objects.

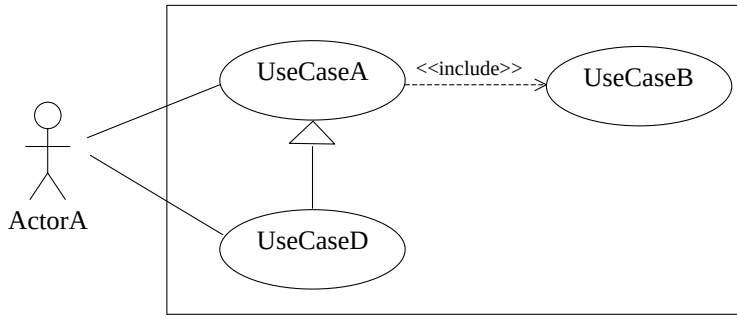

Figure 3. Use Case Diagram

Modeling the Behavioral Aspect. UML offers several diagram types to model the behavioral aspect of a system. Each of them focuses on a certain view on a system and offers appropriate language features.

- A global, coarse-grained, sometimes also called external view on a system can be modeled by *use case diagrams*. This view is restricted to the identification of the main functionality or processes of a system, called use cases, and to (external) actors participating in these use cases. Use cases are only described by their name and an optional textual explanation. Use case diagrams may be structured by include, extend, or inheritance relationships between use cases (cf. fig. 3).
  Use case diagrams have to be refined by the usage of other behavioral diagrams (see below) in order to describe what happens during the execution of a use case (i.e. process).

- The behavior of single objects over time is described by *state machines*. Objects have a control state (in contrast to a data state) which may change in reaction to received events (triggering state transitions). Such an event may be a signal or call event from another object or a time signal which causes the object to change its state. Thus, state machines are used to model the lifecycle of an object and provide a so-called intra-object view. State machines in UML are based on Harel's statecharts [11] and offer means like concurrent and sequential composite states, history states, or junction states to model complex behavior of an object. Figure 4 gives an abstract example of such a state machine.
  Summarizing, state machines are mainly used to model a state- and event-based view on a system. What is missing in such a description is a model of the cooperation and interaction between different objects in a system. This is provided by the following three behavior diagrams where each of them focuses on a certain aspect.

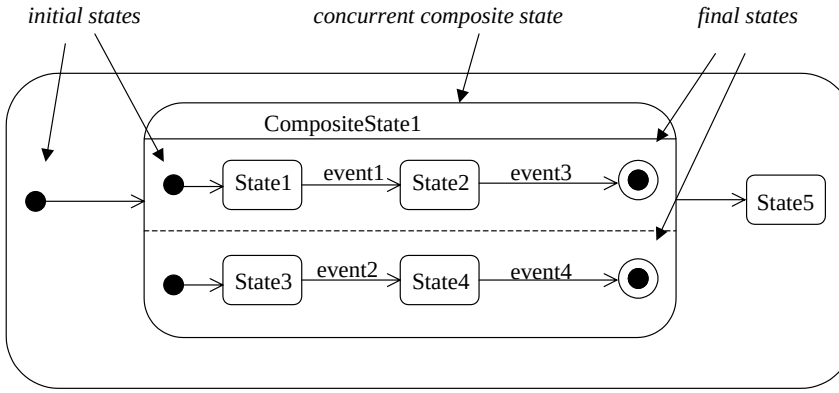

Figure 4. State Machine

The inter-object view on a system, i.e. the communication between and collaboration of different objects, can be described by a UML activity, sequence or collaboration diagram:

- A control-flow oriented description can be given by a UML *activity diagram*. Syntactically, an activity diagram is a special form of a state machine where states are interpreted and labelled by activities. In contrast to usual state machines, a state change is automatically triggered when the execution of an activity has been finished. Activity diagrams do not relate activities to certain objects and represent mainly a procedural, possibly concurrent flow within a system. Objects may be exchanged as in-/out-parameters between different activities, which may be indicated by additional object flow links between states. Figure 5 illustrates the used notations in activity diagrams.

- A scenario-oriented description of the interaction between objects can be given by *sequence diagrams*. They originate from message sequence charts (MSC) [12] and focus mainly on the sequence of message exchanges over time between objects involved in a certain activity. Each object is represented by a vertical life line on which the active and passive periods of an object are shown. Different forms of message exchange like synchronous or asynchronous ones can be indicated by different shapes of arrows between object life lines. Figure 6 illustrates the used notations in sequence diagrams.

- An object structure-oriented description of the interaction between objects can be given by *collaboration diagrams*. They represent mainly the same information as sequence diagrams, but focus on the objects and their structural

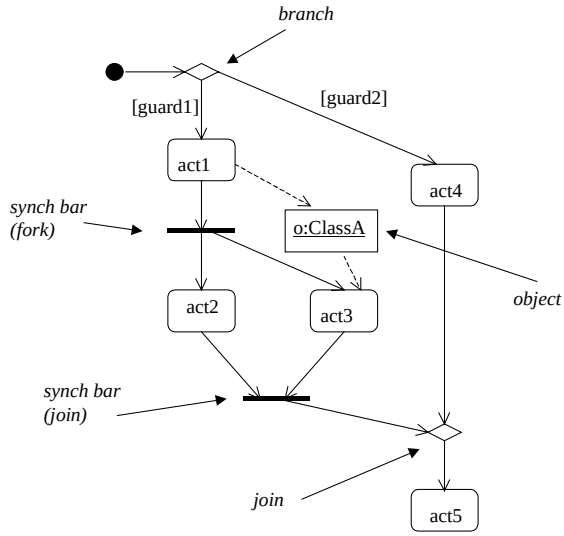

Figure 5. Activity Diagram

interrelations. Thus, the base of a collaboration diagram is an object diagram where the links between objects are additionally labelled by messages which are sent between a sending object and a receiving object. Links can be distinguished into those based on structural relationships, parameters, local variables, etc. The sequencing of messages is described by sequence numbers which are attached to messages and describe a sequential, nested, or concurrent sending of messages. Figure 7 illustrates the used notations in collaboration diagrams.

In addition to these diagrams for modeling the structural as well as behavioral aspects of a system, UML provides two diagram types to describe the transition from a model to the corresonding implementation. These so-called implementation diagrams are the component diagram and the deployment diagram.

The *component diagram* describes the software architecture of a system which consists of components, their interfaces and their interrelations. A component itself encapsulates the implementation of elements as e.g. classes from the system model.

The *deployment diagram* goes even one step further and describes the hardware architecture of a system, consisting of nodes as physical objects and their interrelations. The deployment diagram describes the distribution of objects and components to nodes, and thus links the software architecture to the hardware architecture.

Compared to the UML diagrams explained above to model the structure and behavior of a system, these two implementation diagrams are still in a very rudimentary form in the current UML version. Ongoing discussions to combine UML with language features from the ROOM (Real-time Object-Oriented Modeling)

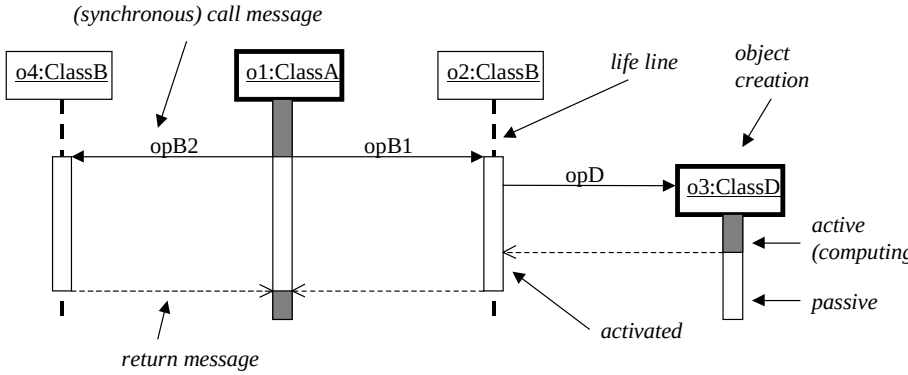

Figure 6. Sequence Diagram

approach [19] to model real-time, embedded systems will result in an improved, more expressive form of these implementation diagrams.

This concludes the overview on the UML language. UML offers a lot of additional features which could not be explained in this brief overview. An example is the package concept which allows to divide a model into smaller parts with clearly defined dependencies and thus supports to manage huge models also. The interested reader is referred to [2,18] and to the links to related web-sites (at the end of this article).

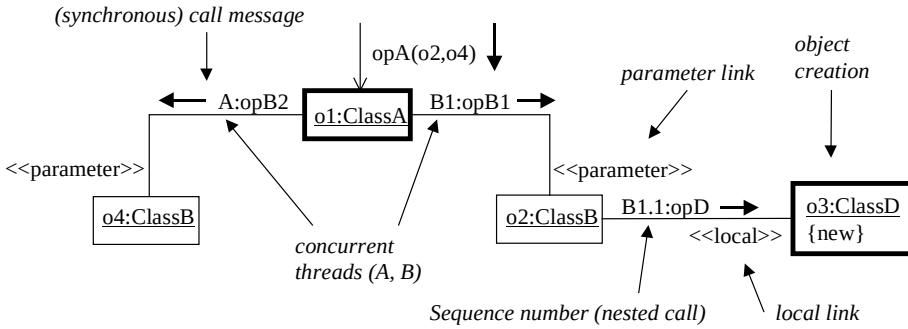

Figure 7. Collaboration Diagram

3 Language Definition

The main focus of the OMG standardization effort so far was an agreement on a commonly accepted concrete notation as well as abstract syntax for all these diagram types of the UML. The semantics of the UML is currently only

informally, textually defined, and its further development towards a precise semantics is postponed to the next standardization phase. In this section, we will briefly sketch the syntax definition approach followed by the OMG and explain UML-intrinsic features to adapt UML to a problem-domain specific modeling language.

For the definition of the abstract syntax of the UML, the OMG follows a four-layered meta modeling approach. These four layers are the following:

- The *MetaMetaModel (M3)* layer provides a so-called Meta Object Facility (MOF) to define meta models on the next lower layer. The MOF consists of language features for defining an Entity-Relationship diagram or class diagram as well as a constraint language to define additional integrity constraints.
- The MOF is used on the *MetaModel (M2)* layer to define a concrete meta model for a modeling language. This meta model consists of a concrete class diagram with additional integrity constraints which defines the allowed abstract syntax features of a modeling language and their interrelations. Thus, the meta model defines the abstract syntax of a modeling language.
- A concrete UML model is an element of the *Model (M1)* layer and is an instance of the meta model layer M2.
- Finally, a conrete runtime extension of a UML model is an element of the *Objects (M0)* layer and is an instance of the model layer M1.

Thus, the UML meta model on layer M2 is specified as an instance of the meta-meta model of layer M3 by a UML class diagram together with OCL constraints, i.e., partly deploying the UML itself. While the class diagram part defines the abstract context-free syntax, the OCL constraints define the context-sensitive syntax of UML. By providing one overall meta model for all sublanguages of the UML, the horizontal consistency problem between different submodels is resolved. Particularly, the OCL constraints, also called well-formedness rules, take care that the different submodels written in different sublanguages of UML are syntactically well integrated.

This agreement on a well-defined abstract as well as concrete syntax together with a (yet informally) defined semantics has the advantage that all users of UML have the same understanding of a system model described by UML. The disadvantage of such an approach is that one has to agree on a general-purpose language with high-level language features which might not be expressive enough to model problem-domain specific details in an appropriate, intuitive way. Therefore, two types of language extensions have been discussed within the OMG to adapt the UML to problem-domain specific needs [17]. These are the *heavyweight* and the *lightweight extension* mechanisms.

The heavyweight extension mechanism is provided by the MOF which means that it is possible in principle to change and adapt the UML by modifying the UML metamodel on layer M2. As the name says, this kind of extension has great impact on the UML language and, therefore, is not possible for an individual user of UML.

In contrast to this, the lightweight extension mechanisms are built-in mechanisms of UML and allow any individual modeler to tailor the UML to her needs. This provides the opportunity to adapt the UML to the requirements of a certain problem domain by tailoring the general-purpose, universal UML to a problem domain-specific modeling language. As this tailoring means that the syntax as well as the semantics of UML constructs might be changed, it is obvious that it has to be done with great care. Thus, it is unlikely and not intended that an individual user starts to adapt the UML. It will be mainly the task of, e.g., a user group or a tool vendor, to propose and to do such an adaptation of the UML for a specific problem domain. The result of such an adaptation is a UML dialect which is termed a *UML profile*.

Three lightweight extension mechanisms are distinguished for the UML. These are constraints, tagged values, and stereotypes.

- *Constraints* are expressed in the OCL (Object Constraint Language) and specify additional restrictions on a UML system model. They are comparable to integrity constraints in the database field and can be added to any model part in an UML system model.
- *Tagged values* are pairs of strings - a tag string and a value string - which can be added to any model element in a UML system model. This feature allows to attach additional information to a UML system model which can not directly be expressed by UML language features.
- The most powerful and thus most heavily discussed extension mechanism are *stereotypes*. Stereotypes allow to give existing UML model elements an additional classification, and thus to tailor them for a specific purpose. Stereotyped model elements of UML are indicated by an additional annotation or they may even have a different concrete notation. Stereotypes can range from modification of concrete syntax to redefinition of original semantics of model elements (cf. [1]).

Summarizing, it can be stated that the abstract and concrete syntax of UML has been formally defined by following a four-layered meta model approach. In addition, lightweight extension mechanisms are provided to adapt the general-purpose UML to a specific problem domain.

4 Language Adaptations

The main advantage and at the same time the main drawback of UML is that it is a general-purpose language. This objective of the UML designers automatically yielded a language which is not capable of providing features which are appropriate to express problem domain-specific situations. This drawback of being a general-purpose language and, in addition, the lacking of a precise semantics has been identified by the UML standardization groups and has led to establishing corresponding task groups and related RFPs (Request For Proposals) by the OMG (cf. [14]) to overcome these shortcomings. It has to be expected that

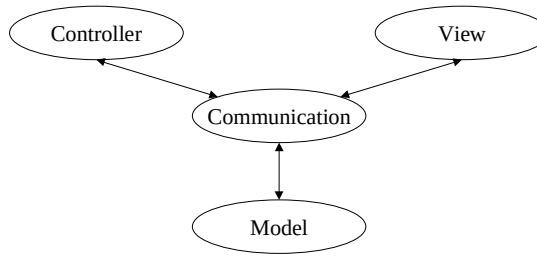

Figure 8. MVCC Architecture

further versions of UML as well as proposals for domain-specific profiles will be developed and published within the next years.

As an example of such a problem domain, we discuss the requirements for applications in the field of embedded, interactive software systems. Such software systems are typical for today's software systems and can be found, for instance, in banking terminals, as infotainment software in advanced automotive systems, or in production control systems. These systems support on one hand a window-based user interface and at the same time the connection to hardware components. Often, these software systems have to obey real-time constraints, too, in order to react to events caused by the user or an embedded system component appropriately.

A closer look at the architecture of those systems shows that the classical Model-View-Controller (MVC) architecture [15] has to be extended to a Model-View-Communication-Controller (MVCC) architecture [20] where the communication between the different components is treated as a first-class object, too (cf. fig. 8).

Summarizing, a modeling language which offers appropriate support for all aspects of such an interactive, embedded system has to provide language features to model

- the model part, i.e., the problem domain specific objects,
- the view part, i.e., the user interface of the system,
- the communication part, i.e., the interaction between the different constituents of the system, and
- the controller part which might include a real-time behavior.

In addition, appropriate means should be provided to model the component-based style of the software architecture which reflects the hardware architecture of those embedded systems.

A comparison with the language features of UML shows that the language elements are mainly suited to describe the model part. Specific language features are missing in the UML to specify the user interface and particularly the layout of a user interface, to specify the interaction between different components in a fine-grained way, or to specify real-time aspects.

This observation caused several groups to work on the development of specific profiles. A prominent example is the work on a profile for component-based, real-time systems based on ROOM [19]. Ongoing own research work comprises the investigation of fine-grained modeling of interaction [7] based on SOCCA [5,6], and the development of a profile for multimedia applications which, in particular, supports the modeling of the layout of the user interface [21].

5 Conclusion and Perspectives

In this paper, we have discussed the role of a system model within the software development process and the appropriateness of the Unified Modeling Language (UML) as a language to specify these system models. In particular, we have illustrated the main features of the sublanguages of the UML to model structural as well as behavioral aspects of a system.

UML has been designed as a general-purpose language, but also as a language which can easily be extended to a problem domain-specific language. The development of so-called problem domain-specific profiles is supported by UML-intrinsic extension mechanisms as constraints, tagged values, and particularly stereotypes.

Thus, returning to the question posed in the title of this paper, whether the *U* in UML can also be interpreted as *universal*, the following can be concluded:

At first glance, UML is a *union* of modeling languages as several already existing modeling languages have been gathered under one umbrella. But it is more than a disjoint union of these languages, as due to the definition of one common meta model for all sublanguages, an integration, at least on the syntactical level, has taken place. Thus, the UML can be termed a *unified* language on the abstract syntax level, but also on the concrete syntax level, since an agreement on concrete notations has taken place, too.

Discussing the question, whether the UML is a *universal* language which can be deployed for modeling all of today's software systems, it has to be concluded that the UML forms an ideal base for any problem domain-specific language. But the UML can not fulfill the role of a universal language which should or could be deployed in any problem domain. In order to provide intuitive and expressive modeling features for a certain domain, UML has to be extended and adapted by appropriate profiles.

This conclusion is also reflected by ongoing research and development work in the UML field. In particular, it is currently discussed which constituents of the UML belong to a *core UML* which then might serve as the base for developing problem domain-specific profiles [9].

Besides that, the embedding of the UML into the software development process is an important topic. This includes the definition of a software process model [13,4] as well as techniques to transform a UML model into a corresponding implementation in a programming language [8].

References

1. St. Berner, M. Glinz, St. Joos: A Classification of Stereotypes for Object-Oriented Modeling Languages. In [10], 249-264.
2. G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, 1999.
3. P. Chen: The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions on Database Systems, 1(1), 1976, 9-36.
4. D. D'Souza, A. Wills: Objects, Components, and Frameworks with UML - the Catalysis Approach. Addison-Wesley, 1998.
5. G. Engels, L.P.J. Groenewegen: SOCCA: Specifications of Coordinated and Cooperative Activities. In A. Finkelstein, J. Kramer, B.A. Nuseibeh (eds.): Software Process Modelling and Technology. Research Studies Press, Taunton, 1994, 71-102.
6. G. Engels, L.P.J. Groenewegen, G. Kappel: Object-Oriented Specification of Coordinated Collaboration. In N. Terashima, Ed. Altman: Proc. IFIP World Conference on IT Tools, 2-6 September 1996, Canberra, Australia. Chapman & Hall, London, 1996, 437-449.
7. G. Engels, L.P.J. Groenewegen, G. Kappel: Coordinated Collaboration of Objects. In M. Papazoglou, St. Spaccapietra, Z. Tari (eds.): Object-Oriented Data Modeling Themes. MIT Press, Cambridge, MA, 2000.
8. G. Engels, R. Hücking, St. Sauer, A. Wagner: UML Collaboration Diagrams and Their Transformation to Java. In [10], 473-488.
9. A. Evans, St. Kent: Core Meta-Modelling Semantics of UML: The pUML Approach. In [10], 140-155.
10. R. France, B. Rumpe (eds.): ≪UML≫ '99 - The Unified Modeling Language - Beyond the Standard. Second Intern. Conference. Fort Collins, CO, October 28-30, 1999. LNCS 1723. Springer, Berlin, 1999.
11. D. Harel: Statecharts: A Visual Formalism for Complex Systems. Science of Comp. Prog., 8 (July 1987), 231-274.
12. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.
13. I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process, Addison-Wesley, Reading, MA, 1999.
14. C. Kobryn: UML 2001: A Standardization Odyssey. CACM, 42(10), October 1999, 29-37.
15. G.E. Krasner, S.T. Pope: A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1(3), August/September 1988, 26-49.
16. Object Management Group. OMG Unified Modeling Language Specification, Version 1.3. June 1999.
17. Object Management Group, Analysis and Design Platform Task Force. White Paper on the Profile Mechanism, Version 1.0. OMG Document ad/99-04-07, April 1999.
18. J. Rumbaugh, I. Jacobson, G. Booch: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading, MA, 1999.
19. B. Selic, G. Gullekson, P. Ward: Real-Time Object-Oriented Modeling. Wiley, New York, 1994.
20. St. Sauer, G. Engels: MVC-Based Modeling Support for Embedded Real-Time Systems. In P. Hofmann, A. Schürr (eds.): OMER Workshop Proceedings, 28-29 May, 1999, Herrsching (Germany). University of the German Federal Armed Forces, Munich, Technical Report 1999-01, May 1999, 11-14.

21. St. Sauer, G. Engels: Extending UML for Modeling of Multimedia Applications. In M. Hirakawa, P. Mussio (eds.): Proc. 1999 IEEE Symposium on Visual Languages, September 13-16, 1999, Tokyo, Japan. IEEE Computer Society 1999, 80-87.
22. J. Warmer, A. Kleppe: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Reading, MA, 1998.
23. E. Yourdon, L.L. Constantine: Structured Design: Fundamentals of a a Discipline of Computer Program and Systems Design. Prentice-Hall, Englewood Cliffs, NJ, 1979.

LINKS

www.omg.org - OMG home page
www.cs.york.ac.uk/puml - precise UML group
www.rational.com/uml/index.jtmpl - UML literature
uml.shl.com - UML RTF home page