Object-Oriented Modeling: A Roadmap

- Draft Version -

Gregor Engels
University of Paderborn
Dept.of Computer Science
D-33095 Paderborn, Germany
Tel.: +49-5251-60 3337
engels@upb.de

Luuk Groenewegen
Leiden University
LIACS, P.O. Box 9512
NL-2300 RA Leiden, The Netherlands
Tel.: +31-71-527 7139
luuk@liacs.nl

ABSTRACT

Object-oriented modeling has become the de-facto standard in the early phases of a software development process during the last decade. The current state-of-the-art is dominated by the existence of the Unified Modeling Language (UML), the development of which has been initiated and pushed by industry.

This paper presents a list of requirements for an ideal object-oriented modeling language and compares it with the achievements of UML and other object-oriented modeling approaches. This forms the base for the discussion of a roadmap for object-oriented modeling, which is structured according to a classification scheme of six different themes, which are language-, model- or process-related, respectively.

Keywords

Object-oriented modeling, UML, profile, views, patterns, frameworks, development process

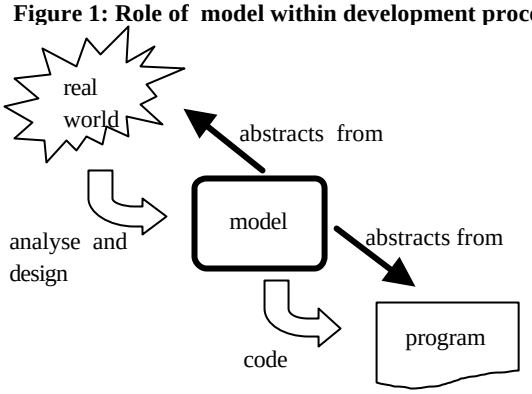
1. INTRODUCTION

It is one of the main objectives of the software engineering discipline to support the complex, and hence error-prone software development task by offering appropriate 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 preceeds the actual implementation in a programming language.

The usefulness of an abstract system model was already recognized in the 70ies, when *structured methods* were proposed as software development methods. These methods offered Entity-Relationship diagrams 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

Figure 1: Role of model within development process



horizontal consistency between the data and behavior part within the overall system model, and the vertical mismatch of concepts between the real world 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 80ies. 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 thoughout 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 phase within a software development process, where object-oriented modeling approaches are becoming more and more the standard ones.

The success of object-oriented modeling approaches was hindered in the beginning of the 90ies due to the fact that surely more than 50 object-oriented modeling approaches claimed to be the right one. This so-called *method war* came to an end through an industrial initiative, which pushed the development of the meanwhile industrially standardized object-oriented modeling language UML (Unified Modeling Language) [4].

But, despite the fact that it has been standardized, UML is still a moving target. In particular, the above mentioned horizontal and vertical inconsistencies are still to be resolved:

- The currently offered language features of UML are not real world-specific, i.e., domain-specific enough, as UML was designed as a general-purpose modeling language.
- Parts of the context-sensitive syntax as well as the semantics has not yet been fixed formally, which allows a lot of different interpretations of a UML model.
- Furthermore, the (possibly automatic) transition from an UML model towards an implementation is still an open issue.

Nevertheless, UML plays a very dominant role in the object-oriented modeling scene. It caused a lot of researchers as well as practitioners to orientate their work along UML issues and to focus on appropriate adjustments of UML. These are, for example, domain-specific extensions of UML as, e..g., in the real-time domain, or semantical definitions for (parts of) UML.

Also this article is influenced by the existence of UML. But, it is not intended to be an article on UML. The interested reader is referred to corresponding web pages (see links at the end of the article) or to numerous articles in journals and conference proceedings (e.g., [3, 22]).

The structure and objective of this article is as follows:

In section 2, requirements for an ideal object-oriented modeling language are presented. Section 3 summarizes the state-of-the-art in the area of object-oriented modeling languages. Due to the prominent role of UML, it is partially a brief survey on UML. Section 4 illustrates the still existing lacks in the field of object-oriented modeling and presents a structured view on the open issues in the field. An important contribution of this section is a layered classification scheme of topics, resp. regions in the landscape of object-oriented modeling. Within each region, examples are given to illustrate drawbacks and approaches to overcome them. Thus, this illustration is combined with a lot of references to existing promising approaches to overcome existing deficiencies. Obviously, such a presentation can not be complete and is influenced by the authors' background. But, the presented classification scheme will help to structure the road for research and development in the field of object-oriented modeling within the next couple of years. The article closes with some conclusions in section 5 and a reference list as well as list of related links to get further informations.

2. REQUIREMENTS

The starting point on the road map is the question: what do we expect, or rather require from an ideal object-oriented modeling approach? So, we first discuss the requirements for an ideal object-oriented modeling approach. As object-oriented modeling aims at being central to the whole software engineering life cycle, the well-known software engineering principles and qualities, so expertly explained in [24], certainly apply.

In the particular context of object-orientation however, we can be substantially more specific about many of these principles and qualities. So, we start studying object-oriented modeling by reopening the discussion on software engineering principles and qualities. The discussion concerns both the modeling language underlying the approach and the steps taken in the approach. Thus, we prepare the ground for the road map.

What do we want? Basically we want object-oriented modeling to be sufficiently *user-friendly* to all kinds of possible stakeholders. That is, for all clients of any model, its relevant parts expressed in the modeling language, must be *understandable*, must be *clear* even. For the modeler as well as for all other persons involved in the modeling activity, any model must be *expressive*, *precise* and *clear* as well. As these are properties for any model, they can be considered as properties of the modeling language in the sense that the language should be such that it comprises only models with the required properties.

The required precision mentioned above, immediately leads to the required qualities of *correctness*, *reliability* and *robustness*. Hence not only the syntax, but also the semantics of the modeling language has to be well defined, in order to make model checking possible. Based on the semantics, at least some mild forms of checking then can be performed, e.g. through prototyping, through animation or through other forms of validation. In addition, stricter forms of checking might be performed through verification, at least partial.

A possible way towards user-friendliness as well as to correctness is indicated by another software engineering principle, *separation of concerns*. By means of this principle, components of a model can be formed each with a certain role within the model, views on a model can be defined each conforming to the viewpoint of a certain class of users, aspects of the model can be discriminated such as data, behavior, functionality, communication, security, timeliness.

In particular, the general software engineering principle of separation of concerns combined with object-oriented modeling characteristics has turned out to be very useful. What are these object-oriented characteristics? The basic idea of object-orientation is the consequent application of the *abstract data type* concept, combining data and functionality. The abstract data type concept

is applied in the context of the architecture of any object-oriented model. This architecture first of all advocates the distribution of logically separated model parts, the classes. Secondly, it advocates their (re)integration. The integration actually links the distributed classes into one model consisting of (distributed) parts that are carefully kept consistent, e.g. via the behavioral interfaces of the classes. In addition, the architecture has a number of other properties which are appropriate. Not only is there *consistency* between the model parts, but to a certain extent also aspects as data, behavior, functionality and communication are consistently incorporated and integrated.

Furthermore, apart from consistency between the parts, the object-oriented model architecture advocates *homogeneity* in the model parts: they all are classes. In line with Meyer's pronouncement "Objects are the only modules" [36], compositions of classes in the form of modules or packages should result in something similar: the package or the aggregation itself should be a class. As a class, such a package should unite all aspects of its constituting classes in a consistent manner, such that it can be consistently integrated into the rest of the model.

Through the properties of homogeneity and of consistency with respect to its model parts, object-oriented modeling gives a particular meaning to the software engineering principles of modularization and of separation of concerns. It should lead to full *scalability* of the modeling, as a whole model can be considered as one package, with the properties of a class, which can be integrated in a larger model in a consistent manner.

Furthermore, also *views* on the model should lead to model parts which as packages are homogeneous in form. This should make consistent *view integration* considerably easier.

The particular combination of the software engineering principles separation of concerns and modularization with the object-oriented properties of consistency and homogeneity can be exploited to a far larger extent in the embedding of object-oriented modeling in the overall *development process*. Then, the actual support in the software engineering process becomes apparent from the incrementality in the modeling it allows in combination with a well-guarded consistency between older and newer parts and views. Then, the support also becomes apparent from the smooth transition from informal requirements towards formal model and from the smooth transition from formal model towards programming code. In addition, effective *tool support* can really improve the software engineering process. In particular, the consistency property of object-oriented modeling with respect to the model parts, the views and the aspects can be fully exploited. This helps a modeler to be aware of the consequences of all kinds of modeling choices in an early stage of the modeling.

In view of the great number of different and varying stakeholders of an object-oriented model, there is another point to user-friendliness we want to stress. In the context of the recent globalization as global workbench and global village, *standardization* of the object-oriented modeling language is absolutely necessary. As the homogeneity of small and large model parts is to lead to integration and reuse that should be easier to perform because of the well-guarded consistency, it is crucial that every stakeholder sticks to the standard. This is true within the same project, but also within the same problem domain. But on the other hand there should be enough flexibility in the language or closely related to it, to allow for experiments, large-scale as well as in practice, with not (yet) standardized variants of or extensions to the language. Which balance between standardization and nonconformity is wise, in order to give sufficient opportunity to investigate possible changes to the language? An important guideline for incorporating any form of *anticipation of change* in the language should be, homogeneity of as well as consistency between model parts is to be kept. As we expect, the guideline actually supports the anticipation of change, as it structures possible language extensions.

Summarizing, we come to the following *requirements* for an ideal object-oriented modeling language.

- *User-friendliness*, leading to understandability and precision.
- *Precision* leading to correctness as well as to richness of details, such as model parts, views and aspects.
- Understandability together with *separation of concerns* and modularization, combined with object-orientation, leading to homogeneity of and consistency between model parts, views and aspects.

If we did not mention the other principles and qualities introduced in [24], it is not because we think they are less important. They are important. The particular combination, however, of object-orientation with separation of concerns and modularization gives rise to particular consequences with respect to these other priciples and qualities, as we will see in Section 4 where we discuss the future perspectives. But first comes our discussion of where we are now in the next section.

3. STATE-OF-THE-ART

After having gathered the requirements for an ideal object-oriented modeling approach, we will briefly summarize in this section the current state-of-the-art in object-oriented modeling in industry and research. This forms the basis for identifying drawbacks and, hence, open issues to be investigated within the next couple of years.

Object-oriented modeling in all areas is nowadays dominated by the Unified Modeling Language (UML)[4]. This language has been accepted as industrial standard in November 1997 by the OMG (Object Management Group). UML was developed as solution to the so-called object-oriented method war, which rose up in the beginning of the 1990ies, where more than 50 different object-oriented modeling approaches could be identified in the software industry. 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 [37].

UML was intended as general purpose object-oriented modeling language assembling variants of different already existing modeling languages which are suited to model certain aspects of a system. In particular, the following languages are part of UML:

- *use case diagrams* to model the main functionality of a system as well as the main involved actor roles.
- *class / object diagrams* to model all structural aspects of a system. These diagrams originate from Entity-Relationship diagrams [5] and are useful to model the structure of single objects, their possible structural relationships as well as the signature of operations.
- different forms of behavioral diagrams to model dynamic aspects of a system. These comprise
  - a variant of Harel's *statechart* [26] for modeling system or object states and their possible transitions,
  - *activity diagrams* as dual interpretation of statecharts for modeling procedural control flow, and
  - *sequence diagrams* as a variant of message sequence charts (MSCs) [29] and so-called *collaboration diagrams* to model the interaction between objects over the time.
- two forms of implementation diagrams, i.e., *the component diagram* to model the concrete software units and their interrelations, and the *deployment diagram* to model the arrangement of run-time components on run-time resources such as a computer, device, or memory.

While all these sublanguages of UML are graphical or diagrammatic languages, an additional textual language is provided by UML, to express static consistency constraints on sets of objects and their interrelations. This is the *object constraint language* (OCL) [44].

In order to manage huge models, a purely syntactical *package concept* is offered by UML, too. It allows to divide a huge model into smaller parts, so-called packages, with clearly defined dependency relations between them.

The main focus of the OMG standardization effort so far was an agreement on a commonly accepted *standard notation* for all these diagram types, while an agreement on a mathematically defined and precise semantics was postponed to the next standardization phase. Currently, the semantics of UML language constructs is only defined in a textual, informal way. By using a layered language definition approach, in particular the abstract syntax of UML diagrams has been defined precisely by the usage of a so-called *metamodel*. This is itself a UML class diagram together with OCL-constraints and it defines the context-free as well as context-sensitive syntax of all UML diagram types.

Despite of the definitely dominant role of UML, competing and extending approaches towards object-oriented modeling exist in industry as well as in academia. Well-known examples of these modeling approaches are the OPEN Modelling Language, OML [20], or the Business Object Notation, BON [38].

In particular, domain-specific approaches have been developed in the past decades. Examples are modeling approaches

- for real-time and embedded systems as e.g. ROOM [40]
- for state-based, interactive systems as e.g. Statecharts [26],
- for reactive and concurrent systems as e.g. Petri Net-based approaches [27] or logic-based approaches as OBLOG [41] or TROLL [31], or
- for concurrent, collaborating systems as e.g. SOCCA [14].

Besides being domain-specific, these approaches differ from the current state of UML in that they often come along with a precisely defined semantics. A reason for this is that these approaches often are centered around one specific diagram type, viewing all other diagram types as extensions of this basic type.

These two drawbacks of being a general-purpose language and lacking a precise semantics have been identified by the UML standardization groups and have led to establishing corresponding task groups and related RFPs (Request For Proposals) by the OMG (cf. [34]). It has to be expected that further versions of UML as well as proposals for domain-specific profiles will be developed and published with the next years.

Besides being to abstract in case of domain-specific applications, the general-purpose nature of UML also missed an appropriate process support for deploying the various UML diagram types. This situation has been improved in the meantime, as several development methods have been proposed in the literature. Prominent examples are the Unified Process ([30]), the Catalysis approach ([11]), or the approach by [10] for real-time applications.

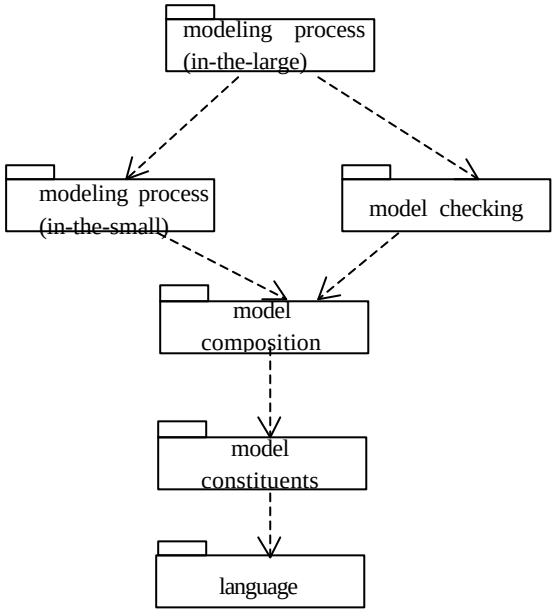
Last but not least, a great variety of commercial software tools are available on the market. Most of them are still under development in the sense that they support only parts of the complete UML, that they do not offer sufficient process support, or that they are hard to integrate within an overall software development environment.

4. PERSPECTIVES

Based on the above discussion of the requirements and of the state-of-the-art, we shall investigate the landscape of drawbacks and open issues in order to find the road for research and development tasks for the next couple of years.

First, we start with structuring the landscape and identifying regions of related topics. Thus, by applying the principle of separation of concern to our own presentation, we yield the following six regions illustrated as packages in UML notation and interrelated by a dependency relation (cf. Fig. 2):

Figure 2: Regions of the object-oriented modeling landscape



1.) *language structure*: the definition and architecture of an object-oriented modeling language itself, ranging from core elements to application-specific extensions;

2.) *model constituents*: the model parts that reflect the way one wants to separate one´s concerns;

3.) *model composition*: grouping the constituents into some integrated unity based on relationships as aggregation, refinement, etc..

4.) *modeling process*: support for the process of constructing a model;

5.) *model checking*: techniques to ensure the quality of the model built and to verify expected properties of the model;

6.) embedding the modeling in the whole *software development process*: consequences for the software life cycle.

The order of the themes from the structure exhibits some interesting structure, too. Let us suppose for a - really very short - moment, we study the themes strictly subsequently. First, we study a language as formalism. Then we study elements - the constituents and their composition - from the descriptions formulated in the language. Next we study how to come up with such a description, followed by studying how to check such a description. Finally, we study integrating the last two processes into their umbrella software engineering process.

This is very similar to the (pre)history of software engineering: the programming language, the program elements, the programming process, the testing process, the integration of the latter two processes into their umbrella business process, being the software engineering process. In the past, when we still had to learn the process of software engineering, the latter series of themes from programming language to process embedding were studied one after the other, spanning a period of over fourty years. In the case of the object-oriented modeling language, we can organize the study of topics from the themes in parallel, while being aware of the structure these themes fit in with. This can considerably reduce the time span of the research, say ten years instead of fourty.

To summarize, it is the structure of the themes that is crucial for understanding the relevance of the topics to be studied. Therefore, in the discussion below some of the topics will be mainly illustrative for the structure in the grouping of the topics. In such a case we will content ourselves with only a very short description of the topic.

4.1 Language Structure

The classical basics of a language are its syntax and its semantics. From the ongoing research activities concerning UML two important problems emerge. Whereas the *syntax* of this have been precisely described by using a metamodel approach, this does not hold for the *semantics*. When it comes to describing the meaning of the various syntactical language constructs, usually English is used with its inherent informalities and ambiguities. The other problem is the large and complex scope of the language. An extra complication are the built-in features for incorporating anticipation of change into the language, such as stereotypes, tagged values and constraints. On the one hand this makes UML an almost guaranteed part of any future object-oriented modeling language. UML, so to say, can serve as the starting point for such a future language. On the other hand, the nature as well as the possible, meaningful use of these features still needs further study.

Because of the highly complex nature of such an object-oriented modeling language we propose to add a layered structure to the language (cf. Figure 3). This is in line with current work on UML 2.0 as well as of the precise UML group, where an improved layered architecture of the UML is a major point of discussion [34, 18].

The first, innermost layer or shell contains the *core language*, as we call it. It contains the basic language elements of UML, its basic building blocks. It is to be a point of study which language elements will belong to the core. Most probably, classes, relationships, i.e., class diagrams, will belong to the core. Also one or two from the triple use cases, activity diagrams, state machines, representing the more simple behavior modeling elements, will belong to the core, and may be even some form of interaction diagram also will belong to it. It has to be discussed, what are useful, necessary criterions for a language element to belong to the core. A possible criterion could be, whether the semantics of that element is completely defined and can be expressed and understood in a relatively easy manner.

The second layer contains the other *language elements* being the less basic elements of UML. In all likelihood, the constraint language OCL will belong to this second layer. It is an important point of research and part of this second layer to investigate how the different sublanguages of an object-oriented modeling language are interrelated. This hold for instance for the interrelation between statecharts and interaction diagrams, but also the connection between OCL specifications and the remaining UML model [8].

Furthermore, the concrete syntax of an object-oriented modeling has to be studied. In particular, the role of so-called *hybrid languages* - combining textual and graphical notations - have to be investigated (cf. [1]), as it might not always be true that graphical (or textual, resp.,) representations are easier comprehensible for a potential user. For instance, graphical representations for the OCL, so-called constraint diagrams, are currently investigated by [32] as alternative for the originally proposed textual notation.

In addition, the *completeness* of an object-oriented modeling language have to be studied. This means whether appropriate language features are offered for all perspectives of a system to be modelled. For instance, for the description of what happens during execution, i.e, how object structures are rearranged and object values are changed, graph transformations as in ([21]) could be taken into account.

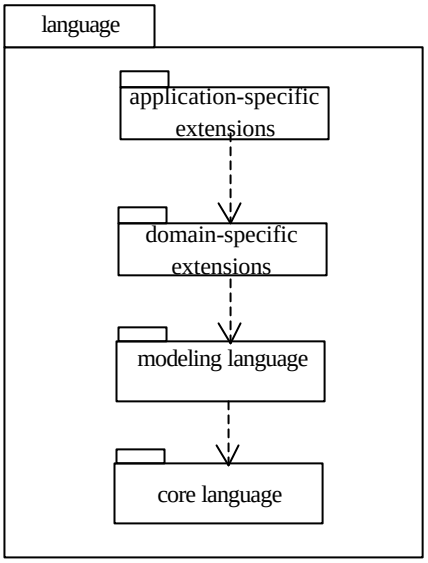
Another point of study that seems interesting, is the binary character of some relations between constituents. For instance, concrete calling of a method of an object normally is always from one other object, the caller - callee relation. Is this necessary? Could it be useful, clarifying even, if the number of two participants in this relation could be generalized into some larger number. A beginning of the study of these so-called *multi-methods* has begun by [39]. This might result in new forms of dependency and influencing and therefore of consistency.

The definition of the *semantic*s of an object-oriented modeling language is a hard and difficult task due to the inherent divergence of the different sublanguages of a fully-fledged object-oriented modeling language like UML. Currently, different approaches are investigated like operational ones based on state transition systems or graph transformation systems, or a denotational one. In any case, it is a fact that a precise and formal semantics is a prerequisite for any type of model checking and of embedding the model into the overall software development process.

The third layer (cf. Fig. 2) consists of the domain-specific extensions to the language, so-called *domain-specific profiles* and *domain-specific frameworks*. Here, we see the extensions to UML in the context of a particular domain. To give an impression of such domains and domain-related ongoing activities, we mention the domains of real-time applications ([10, 40]), multimedia applications ([43]), web applications ([7]) and last but not least the software development process ([30]). The profiles are formulated in terms of the UML´s extensibility mechanisms, i.e., stereotypes, tagged values and constraints.

An alternative approach to yield a domain-specific adaption of a modeling language is the usage of a *domain-specific framework*. Frameworks are architectural patterns, i.e., they form a partial model, which expresses common basic structures and behavior within a certain domain. They can be extended by refinement and specialization to yield a complete model for a certain application. The usage of UML for describing a framework for business processes have been studied, for instance, in [35]. As a framework is defined as an architectural pattern, we thus see patterns return as elements of our third layer. It is intriguing to investigate in what sense and under what circumstances patterns (cf. [23]) can be an alternative to the use of UML´s extensibility mechanisms.

Figure 3: Language layers



The fourth layer contains the *application-specific* extensions to the language on top of the domain-specific extensions from the previous layer. The prefaces as discussed in [9] belong to this layer. Depending on the concrete context of a certain application, so-called semantic variation points occurring in such a preface have to be fixed, thus pin-pointing the pragmatics for that application. There seems to be some similarity with the frameworks mentioned above, so on this level, too, we see the patterns return.

In this subsection, we have proposed a shell-layered language structure. The innermost shell contains the most general elements, the outermost shell the least general ones. As patterns cover the full range of recurring problems, whether this is over all domains, or restricted to one domain but over all applications, or just restricted to one type of application, we see the patterns occur in three out of four shell layers.

It is very well imaginable that the impact of the elements in the domain-specific or the application-specific shell on the language support may vary in the course of time. For instance, there may

be a shift in impact of a specific pattern from the domain-specific shell towards a more general shell. As a concrete example, we mention the role of the communicator in the Model-View-Communication-Controller pattern ([42]), which may lead to new language features in the language shell or even the core shell, to express different alternatives of communication between objects in a more explicit way than it is supported by currently available language features.

Summarizing, we see the following four main open issues in the language region.

Open issues in the ***Language*** region:

- language architecture (core vs. profiles)
- hybrid notations (textual vs. graphical)
- completeness
- semantics

4.2 Model Constituents

The constituents of a concrete model (cf. Figure 2) reflect the modeler´s way of separating the various concerns seen as relevant for the concrete situation to be modeled. Consider the situation of an architect designing a prefab house. Parts of the house are to be made in some factory, and the parts have to be put together on the spot where the final house has to be built. A part is a complete wall or floor element, containing the (local) pipes and tubes for gas, water, central heating, the electricity wiring and such things as radiators, windows, doors. So in this case, the archtectural design of the house physically consists of the drawings of the prefab parts. When a potential buyer of such a house wants to see a drawing of the house, he actually gets a series of interrelated drawings, one for each floor. These drawing do not in the least reflect drawings of the prefab parts the house is composed of, although the floor representations are thoroughly consistent with these parts. Moreover, if the potential buyer wants to be completely informed about the central heating system, he will also get a drawing on scale containing all heating-tubes, radiators, the stove, and the thermostat. Also this drawing does not in the least reflect drawings of the prefab parts, nor does it reflect all information from the floor representations. But again, there is full consistency with the other representations.

Analogous to the prefab house, an object-oriented model has many *constituents*, which are different in nature. Firts of all, a model consists of the constituents it has been - really, physically - made of: the classes, with their data, behavior, functionality, communication, and of relationships as class connectors. Also packages belong to the physically real constituents of a model. These constituents exactly reflect how the model has been constructed, similar to drawings of the above prefab parts of the house. Second, a model also has other representations. Such a representation is a different form the model takes, when represented depending on a particular use of the model one has in mind. Not the model as it is, but as one can think of it, the model with its representative constituents. Examples of such different representations are *views, subjects* and *patterns*. They themselves consist of the same type of constituents, classes and relationships, as the model physically does, but differently represented, similar to the drawings of the floors versus drawings of the various prefab parts. Third, the classes actually combine things that, unless on metalevel, are not to be represented as classes and relationships themselves. Examples are data, behavior, functionality, communcation. They are *aspects* that are bound together in the various classes, similar to the above central heating system. The aspects are so to say woven into the classes.

Where there are so many different components in a model, the real ones, the representative ones as views and patterns, and the aspects, the interrelations between the components have to be carefully watched over. There is really much consistency to take care of. Important parts of this *consistency management* issues have already been studied, as, for instance, in the viewpoint approach ([19]), where consistency between different views could be kept by an explicit consistency management. A different approach of an (automatic) integration of possibly overlapping views or subjects into an overall model has been discussed in the view-based development approach of [16], or the subject-oriented design technique of [6].

The real model, the one with the real components, actually describes its own architecture, although probably from a logical point of view, and not so much from a hardware and system software point of view. Therefore, further study of the relation with *architecture description languages* (ADL) seems promising in view of mutual usefulness of both approaches for each other ([13]). Architectural patterns may turn out to be important, so the above consistency insights could be relevant for this situation, too.

Patterns as a different kind of constituent of a model have been studied e.g. in [23]. Similar to views, also patterns, once specified, are to be composed with other patterns. Pattern integration should be such that necessary or perhaps only desired or optional consistency can be guaranteed.

A third but different type of model constituent is the *aspect* or *feature.* Aspects or features are not classes, but types of general characteristics or concerns for the model as a whole, see e.g. [33, 46]. Examples for aspects are communication, resource sharing, replication. Such an aspect normally is entangled, intertwined with everything else in the model. Small bits of the aspects can be found in many different parts of the model. The parts are the classes, so the classes bear small parts of the aspects. Further research is required to shift the aspect-oriented from the programming level to the modeling level. Here, it has to be investigated which aspects can be specified separately and what are the interrelations between different aspects or features (see e.g. [2] for an investigation on feature interactions).

Summarizing, we see the following open issues in the model constituents region:

Open issues in the ***Model Constituents*** region:

- modeling units and their interdependencies
  - views, subjects
  - aspects, features
  - patterns, frameworks

4.3 Model Composition

With so many model constituents around, one needs flexibility in changing from a small number of constituents to a large number and back, without straining the model elements and expressivity to much. This is known as *scalability*. Scalability has direct consequences for reuse and for change. For instance, extending a model or a small part of it with some other model(´s part) that has new or more sophisticated functionality should be relatively easy and also well understood.

We see scalability into two directions, horizontal and vertical. *Horizontal scalability* consists of adding or removing model parts as classes, packages, patterns, views or aspects. *Vertical scalibility* consists of refinement or its reverse aggregation.

When in the horizontal direction, model parts are being put together, one has to perform the actual integration of these parts with the already present ones. Not only in a syntactical sense, but also in a functional, semantical sense. This means that we need composition techniques for the different types of constituents. For instance, we need a better understanding how to refine behavioral descriptions like statechart-based descriptions. A discussion on two different types of statechart inheritance together with concrete construction rules can be found in [12].

Horizontal compositionality might result in the requirement to treat all constituents of a model as a *semantic unit* with a clear defined *interface* through which such a model part looks like a class. In particular, this might mean to lift packages from the syntactical level to the semantical level.

Scalability in the *vertical direction* is very often modularization, resp. more concrete either refinement or aggregation. In the case of refinement, an existing model part changes its content into a new, more detailed content, but from the outside it still looks as before. So the old part is replaced by some more detailed part, for instance one class is replaced by a whole group of cooperating classes. Such a group then is known as a module. The module replaces the old class, by representing itself to the rest of the model as if it is that class. The other way round is the grouping of a set of already existing model parts, e.g. classes, into something new, the module. The module represents the set towards the rest of the model. Quite often we want a module to look as much as a class as possible. This is in line with the aggregation concept, where a group of classes constitutes the aggregation class. Adapting to the type level, the above cited remark of Meyer now becomes, classes - instead of objects - are the only modules. It has to be investigated how *class-like descriptions* of certain modules or packages can be defined and eventually constructed, also with respect to non-structural aspects as behaviour, functionality and communication.

Open issues in the ***Model Composition*** region:

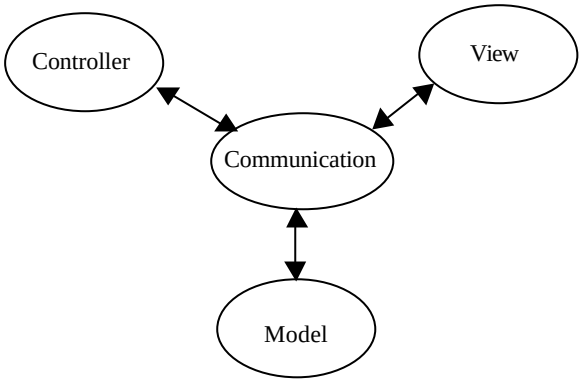
- scalability
- horizontal / vertical composition techniques

4.4 Modeling Process (in-the-Small)

After having discussed constituents of object-oriented models and requirements for having full scalability in a model, we now discuss the process of developing such a model. We see the following question as crucial for the development process: how to take care, from the beginning, of the coordination between all constituents, such that the scalability requirements are met? As a matter of fact, in UML many diagrams exist that have to do with this coordination: use cases, sequence diagrams, collaboration diagrams, state diagrams, activity diagrams, deployment diagrams. A use case specifies one behavioral scenario which among other things addresses coordination. So does a sequence or collaboration diagram, it also specifies one behavioral scenario, but this scenario concentrates much more strongly on the interaction and hence on the coordination between objects involved. State diagrams are based on statecharts, so they specify behavior in terms of separate sequential parts, the separate state transition diagrams. The coordination between these sequential parts is explicitly indicated. Activity diagrams have features from control flow diagrams as well as from Petri nets. So they too specify coordination. In combination with so-called swimlanes, the specification moreover expresses between which classes the coordination takes place. A deployment diagram, among other things, specifies whether some communication between its components exists, but it does not express how the corresponding coordination is to happen.

What is clearly lacking, is first the *consistency* between all different coordination specifications, and second, a technique or approach for finding some reasonable coordination specification. In our opinion, an explicit *coordination* model is needed as submodel of an object-oriented model, together with an approach how to build such a submodel and how to make and keep it consistent with the whole model. It might be a good idea to extend the well-known Model-View-Controller (MVC) pattern to a so-called Model-View-Communication-Controller (MVCC) pattern ([42]) (cf. Figure 4). Here, the often implicit communication and coordination becomes explicit and has to be handled by a separate component. This, is very much in line with current developments in middleware software, where e.g. CORBA or DCOM components provide explicit support for the communication part within a software system.

Figure 4: Model-View-Communication-Controller Pattern



A different approach could be, to see communication and its coordination as so fundamental for behavior and functionality modeling, that the object-oriented modeling language has to be extended or adapted with a separate aspect for it (in addition to data, behavior and functionality). The new aspect should be such that all communication and coordination is being covered, that the aspect is sufficiently integrated with the already present aspects as data, behavior and functionality, and that the specification of the aspect is sufficiently scalable. This is the approach taken in SOCCA ([15]), where based on purely sequential state transition diagrams, used for both the visible behaviors and the hidden functionalities, new notions are defined for specifying phases of behavior (temporary behaviour restrictions) and how to control these.

How to integrate new aspects, that means adding a new aspect to an existing model in a well-structured manner, is an open problem yet. Similar to the integration of the communication aspect by some form of distribution over the classes, as it is done in SOCCA [14], one can try to add a new aspect via local arrangements in the classes. The multi-dimensional approach of [45] might be of help here. He presents a (generic) framework by regarding the whole software development process as a path through a multi-dimensional grid of aspects. Each node in the grid represents a (still to be discussed form of) composition of aspects. A concrete form of this framework might help to understand better the process of developing a concrete model.

Summarizing, we see the following three main open issues in the modeling process (in-the-small) region:

Open issues in the ***Modeling Process (in-the-Small)*** region:

- consistency
- coordination and communication
- model architectures

4.5 Model Checking

After, during or perhaps even before the actual construction of the model, one certainly wants to know whether the model is good, one wants to check the quality of the model. In order to be able to check it, one at least needs full semantics of the modeling language.

Two types of model checking can be distinguished. These are model animation or simulation techniques on one hand, and analytical methods on the other hand.

As in the field of discrete event simulation, *animation* of the model is a very quick and intuitively very convincing way of telling both a designer and a user of the model whether the model indeed reflects what one thinks it should do. In combination with extensive what-if facilities this comes very close to testing the model.

Animation, being a discrete event simulation, is a form of validation, as only some example behaviors, some example functionalities and some example communications are being imitated. It seems worthwhile to investigate under which extra conditions the animation can be replaced by a so-called *analytical* computation model. That means, one can compute the behaviour of the original model in a direct way, instead of getting an estimation for it through the simulation / animation. Such computations are analogous to the various computational analyses that can be performed in Petri nets. The extra conditions may concern assumptions concerning the duration times of behavioral and of functional steps, or assumptions concerning the queueing mechanisms, or assumptions concerning the composition of the model from certain logical parts. In case of an analytical computation model, we have model verification with respect to the results computed. This can be viewed as a sound basis for reasoning about the model.

Due to the still missing complete semantics definition of object-oriented modeling languages, overall model checking support is still missing, too. Nevertheless, first results can be found in the literature, in particular, related to state machine or state transition systems (e.g., [25, 28]). In our opinion, this region in the landscape of object-oriented modeling languages deserves and will receive much more attention in the near future.

Open issues in the ***Model Checking*** region:

- animation / simulation techniques
- analytical techniques

4.6 Embedding into Development Process

In the above five regions, we can see some contours from the umbrella software engineering process. Examples of these contours are: designing a model, checking a model, integrating whatever model part into the rest of the model. Designing a model has to be understood in a broad sense, covering the whole lifecycle of model engineering. This means concretely that the modeling language and

its use have to be embedded in the software engineering lifecycle process, covering all phase where some form of model development is situated. Adopting for a moment the waterfall view with respect to software engineering, we speak about front-end embedding and of back-end embedding of model design into the software lifecycle.

The *front-end embedding* addresses the transformation or translation from any informal description of the problem situation towards the object-oriented model. Often an informal description is written in natural language, but also video or other images can serve as such. This form of transformation occurs in the feasibility phase, in the requirements engineering phase and in the design phase. It is not very common to speak about reverse engineering in the context of this transformation, but animation of an object-oriented model, or of whatever part of it, falls in this category: through animation an informal visualization is presented of what the model or a part of it specifies. It is even imaginable to reverse this form of reverse engineering. First an animation of the model-to-be is developed. As soon as every stakeholder is satisfied with it, the model can be developed on the basis of the animations that actually play the well-known role of simulation in information system development.

Back-end embedding addresses the transformation from object-oriented model towards program code, code generation ([17]). Here, it is has to be investigated, where modeling stops and where programming starts. It has to be understood, how much information, in particular concerning functionality, has to still to be added to the program code. It seems that the border between visual modeling and visual programming will be diminished in the near future or will even disapear.

Object-oriented modeling will become a solid basis for *round-trip* engineering. When during the maintenance phase some change is being proposed, this leads to the situation where some newly developed model part has to be integrated with an often large part of the original model. Hence, the above discussed scalability as well as horizontal and vertical composition techniques can be of great help here. Again, animation as a kind of prototyping could be of substantial help, too, in particular if the new scenario's can be combined with the old scenario's.

All issues discussed above have to be incorporated into an overall software development *process model* and have to be supported by appropriate *software tools*. For instance, the role of animation, prototyping and model checking have to be appropriately integrated with all other model development tasks.

.

Open issues in the ***Modeling Process (in-the-Large)*** region:

- front-end / back-end transformations
- round-trip engineering
- process models
- support tools

5. CONCLUSIONS

In this article, we tried to identify the road map, i.e., the main open issues in the field of object-oriented modeling to be investigated within the next couple of years. As a base for the presentation and future discussion, we introduced a structured landscape consisting of six different regions. They address all perspectives of object-oriented modeling, i.e., the underlying language, the developed models, as well as the development process. Within each region, we identified concrete topics to be investigated and gave references to some research results in order to illustrate possible ways to find solutions.

The whole discussion was biased towards two software engineering principles, which were identified in section 2 to be the most crucial ones in the field of object-oriented modeling. These were user-friendliness of the modeling approach and separation of concerns. As object-oriented modeling is central to the whole software development process, a great variety of stakeholders are dealing with the model and the underlying language. Thus, an intuitive and clear understanding of any model is an important prerequisite. Second, due to the complexity of the models to be developed, in particular horizontal and vertical structuring techniques are desperately needed.

The Unified Modeling Language (UML), currently playing a dominant role in the field, also influenced the presentation and discussion within this article. The discussion on forthcoming versions of the UML will be a major task in the near future in the field of object-oriented modeling. It is obvious that a convincing solution to all open issues discussed above can only be reached if fundamental, scientific research results will be combined in a synergetic way with industrial requirements and restrictions.

6. REFERENCES

[1] M. Andries, G. Engels: A Hybrid Query Language for the Extended Entity Relationship Model. Journal of Visual Languages and Computing, 7(3), September 1996, 321-352.

[2] E. Astesiano, G. Reggio: A Discipline for Handling Feature Interaction. In M. Broy, B. Rumpe (eds.): RTSE'97 - Workshop on "Requirements Targeting Software and Systems Engineering", Technical Report TUM-I9807, April 1998, Technical University of Munich, Gemany, 1 - 22.

[3] G. Booch (guest editor): UML in Action. CACM, Oct. 1999, 42(10).

[4] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, 1999.

[5] P. Chen: The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions on Database Systems, 1(1), 1976, 9-36.

[6] S. Clarke, W. Harrison, H. Ossher, P. Tarr: Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In Proceedings of the OOPSLA '99, Denver, CO, USA, Nov. 1 - 5, 1999, ACM, New York, 1999, 325 - 339.

[7] J. Conallen: Modeling Web Application Architectures with UML, CACM, October 1999, 42(10), 63 - 70.

[8] St. Cook, A. Kleppe, R. Mitchell, J. Warmer, A. Wills: Defining the Context of OCL Expressions. In In R. France, B. Rumpe (eds.): <<UML>>'99, Proc. Second Intern. Conference, Fort Collins, CO, October 1999. LNCS 1723, Springer, 1999, 372 - 383.

[9] St. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, A. Wills: Prefaces: Defining UML Family Members (in preparation).

[10] B. P. Douglas: Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns, Addison-Wesley, 1999.

[11] D. D'Souza, A. Wills: Objects, Components, and Frameworks with UML - the Catalysis Approach. Addison-Wesley, 1998.

[12] J. Ebert, G. Engels: Structural and Behavioural Views on OMT-Classes. In E. Bertino, S. Urban (eds.): Proceedings International Symposium on Object-Oriented Methodologies and Systems (ISOOMS), Palermo, Italy, September 21-22, 1994, LNCS 858, Springer, Berlin 1994, 142-157

[13] A. Egyed, N. Medvidovic: Extending Architectural Representation in UML with View Integration. In R. France, B. Rumpe (eds.): <<UML>>'99, Proc. Second Intern. Conference, Fort Collins, CO, October 1999. LNCS 1723, Springer, 1999, 2- 16.

[14] 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.

[15] 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.

[16] G. Engels, R. Heckel, G. Taentzer, H. Ehrig: A Combined Reference Model- and View-Based Approach to System Specification. International Journal on Software Engineering and Knowledge Engineering, Vol. 7, No. 4, December 1997, 457-477.

[17] G. Engels, R. Hücking, St. Sauer, A. Wagner: UML Collaboration Diagrams and Their Transformation to Java. In R. France, B. Rumpe (eds.): Proc. Second International Conference on the Unified Modeling Language - UML'99, October 28-30, 1999, Fort Collins, Colorado, USA. LNCS 1723, Springer 1999, 473-484.

[18] A. Evans, St. Kent: Core Meta-Modelling Semantics of UML: The pUML Approach. In R. France, B. Rumpe (eds.): <<UML>>'99, Proc. Second Intern. Conference, Fort Collins, CO, October 1999. LNCS 1723, Springer, 1999, 140 - 155.

[19] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, M. Goedicke: Viewpoints: a Framework for Integrating Multiple Perspectives in System Development. International Journal of Software Engineering and Knowledge Engineering, 2(1), March 1992, 31 - 57.

[20] D. Firesmith, B. Henderson-Sellers, I. Graham: OPEN Modeling Language (OML) Reference Manual, Cambridge University Press, New York, 1998.

[21] T. Fischer, J. Niere, L. Torunski, A. Zündorf: Story Diagrams: A New Graph Grammar Language based on the Unified Modeling Language and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.): Proc. of the 6th Intern. Workshop on Theory and Application of Graph Transformation. Paderborn, November 1998, LNCS 1764, Springer , Berlin, 2000 (to appear).

[22] 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, 1999.

[23] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns. Addison-Wesley, Reading, MA, 1995.

[24] C. Ghezzi, M. Jazayeri, D. Mandrioli: Fundamentals of Software Engineering. Prentice-Hall Intern, 1991.

[25] D. Giannakopoulou, J. Magee, J. Kramer: Checking Progress with Action Priority: Is it Fair? In O. Nierstrasz, M. Lemoine (eds.): Proc. ESEC/FSE '99, Toulouse, France, Sept. 1999, LNCS 1687, Springer, Berlin, 1999, 511 - 527.

[26] D. Harel: Statecharts: A Visual Formalism for Complex Systems. Science of Comp. Prog., 8 (July 1987), 231 - 274.

[27] K. M. van Hee: Information Systems Engineering: A Formal Approach. Cambridge Univ. Press, Cambridge, UK, 1994.

[28] C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw: Automated Consistency Checking of Requirements Specifications. In ACM TOSEM, 5(3), July 1996, 231 - 261.

[29] ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, 1996.

[30] I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process, Addison-Wesley, Reading, 1999.

[31] R. Jungclaus, G. Saake, T. Hartmann, C. Sernadas: TROLL - A Language for Object-Oriented Specification of Information Systems. ACM Transactions on Information Systems, 14(2), April 1996, 175 - 211.

[32] St. Kent, J. Howse: Mixing Visual and Textual Constraint Languages. In R. France, B. Rumpe (eds.): <<UML>>'99, Proc. Second Intern. Conference, Fort Collins, CO, October 1999. LNCS 1723, Springer, 1999, 384 - 398.

[33] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin: Aspect-Oriented Programming. In Proceedings of ECOOP'97, LNCS 1241, Springer, 1997.

[34] C. Kobryn: UML 2001: A Standardization Odyssey. CACM, 42(10), October 1999, 29 - 37.

[35] G. Larsen: Designing Component-Based Frameworks using Patterns in the UML. CACM, 42(10), October 1999, 38 - 45.

[36] B. Meyer: Object-Oriented Software Construction, Prentice Hall, 1997.

[37] Object Management Group. OMG Unified Modeling Language Specification, Version 1.3. June 1999.

[38] R. F. Paige, J. S. Ostroff: A Comparison of the Business Object Notation and the Unified Modeling Language. In R. France, B. Rumpe (eds.): <<UML>>'99, Proc. Second Intern. Conference, Fort Collins, CO, October 1999. LNCS 1723, Springer, 1999, 67 - 82.

[39] M. Schrefl, G. Kappel: Cooperation Contracts. In T. J. Theorey (ed.): Proc. of the 10th Intern. Conf. on the ER Approach, October 1991, 285 - 307.

[40] B. Selic, G. Gullekson, P. Ward: Real-Time Object-Oriented Modeling. Wiley, 1994.

[41] A. Sernadas, C. Sernadas, H.-D. Ehrich: Object-Oriented Specification of Databases: An Algebraic Approach. In P. M. Stoecker, W. Kent (eds.): Proc. 13th Intern. Conf. on Very Large Databases VLDB'87, VLDB End. Press, Saratoga (CA), 1987, 107 - 116.

[42] 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.

[43] 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.

[44] J. Warmer, A. Kleppe: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, Reading, MA, 1998.

[45] A. Zamperoni: GRIDS - GRaph-Based Integrated Development of Software: Integrating Different Perspectives of Software Engineering. In Proc. of the 18th International Conference on Software Engineering, March 25-29, 1996, Berlin, Germany, IEEE Computer Society Press, 1996, 48-59

[46] P. Zave: Feature Interactions and Formal Specifications in Telecommunications. Computer, 26(8), 1993, 20 - 29.

7. LINKS

www.omg.org - OMG home page

www.cs.york.ac.uk/puml - precise UML group

www.rational.com/uml/index.jtmpl - UML literature

www.shl.com - UML RTF home page