# Problems and Deficiencies of UML as a Requirements Specification Language

Martin Glinz
*Institut für Informatik, Universität Zürich*
*Winterthurerstrasse 190*
*CH-8057 Zurich, Switzerland*
*glinz@ifi.unizh.ch*

## Abstract

*In recent years, UML has become a standard language for modeling software requirements and design.*

*In this paper we investigate the suitability of UML as a semiformal requirements specification language. Using the Teleservices and Remote Medical Care (TRMCS) case study as an example, we identify and demonstrate various problems and deficiencies of UML, particularly concerning use case models and system decomposition.*

*We also investigate whether and how the deficiencies can be overcome and how potential alternatives could look.*

**Keywords:** UML, requirements specification, model, use case, decomposition

## 1 Introduction

Semiformal modeling languages are a powerful means of describing requirements. Such languages have a long tradition, starting about 25 years ago with PSL/PSA [23], SADT [20] and Structured Analysis [5]. About ten years ago, object-oriented specification languages appeared ([3], [4], [22] and many others). A few years ago, the object-oriented approaches were consolidated into UML [21].

The structured languages like DeMarco's Structured Analysis [5] were plagued by many problems, in particular the paradigm mismatch between analysis and design, missing locality of data definitions, only partial information hiding and no types [8]. Object-oriented modeling languages were proposed to overcome these problems. However, the early object-oriented specification languages also had serious deficiencies, particularly due to their inability to model dynamic and behavioral aspects of a system. Jacobson [14] tried to overcome these defects by introducing the notion of use cases. UML [21] was created with the goals of unifying the best features of different existing languages and of creating an industry standard.

However, UML is not the ultimate answer to the problem of creating a good language for semiformal modeling of requirements. In this paper, we identify several deficiencies of UML as a language for requirements specification. We select issues from the Teleservices and Remote Medical Care (TRMCS) case study representing typical requirements specification problems. We try to model these issues with UML and discuss the difficulties encountered.

We do not attempt a comprehensive analysis of the weaknesses of UML. In particular, we do not systematically analyze the omissions, inconsistencies, vaguenesses and comprehensibility problems in the partially overcomplex and ill-structured metamodel of UML and in the definition of UML semantics. Considering the size of the UML 1.3 specification (over 800 pages), this would be a major research endeavor. Furthermore, when looking at the rapid evolution of UML versions, the results would probably be outdated before completion.

The rest of this paper is organized as follows. In section 2 we outline the case study. In sections 3 and 4 we identify deficiencies in UML concerning use case models and system decomposition. In section 5 we investigate whether the deficiencies can be overcome by using UML extension mechanisms. Finally, we sketch a potential alternative to UML.

## 2 The case study

As a case study we use the Teleservices and Remote Medical Care System (TRMCS) which was defined by Inverardi and Muccini for this workshop [7]. As this case study is rather open, we add more precise requirements and design decisions where appropriate. The high-level goals and constraints of the TRMCS and some high-level system design decisions are summarized below.

**Business/system requirements for the TRMCS**

**Goal.** The TRMCS shall provide medical assistance to at-home or mobile patients.

**Subgoals.**

1. The TRMCS shall provide two main services for patients:
   - adequately service help calls issued by a patient
   - continuously telemonitor a patient's health condition and automatically generate a help call when necessary.
2. These services shall be available regardless of the actual geographic location of the patient (but within the limits defined by the service contract).

Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD-10), San Diego, November 2000. 11-22.

3. The TRMCS shall support and coordinate multiple and geographically distributed service providers.
4. The services provided by the TRMCS shall have the same level of reliability, safety, security, accessibility and medical ethics as a local service provided by humans would have.

**Constraints.** The TRMCS shall operate on near-future network and computing infrastructures.

**Assumptions.** The TRMCS assumes that a patient using a TRMCS service has access to a highly reliable and available telecommunications system that transmits voice and data.

We assume that the following system design decisions have been taken based on the business/system requirements.

**System design decisions for the TRMCS**

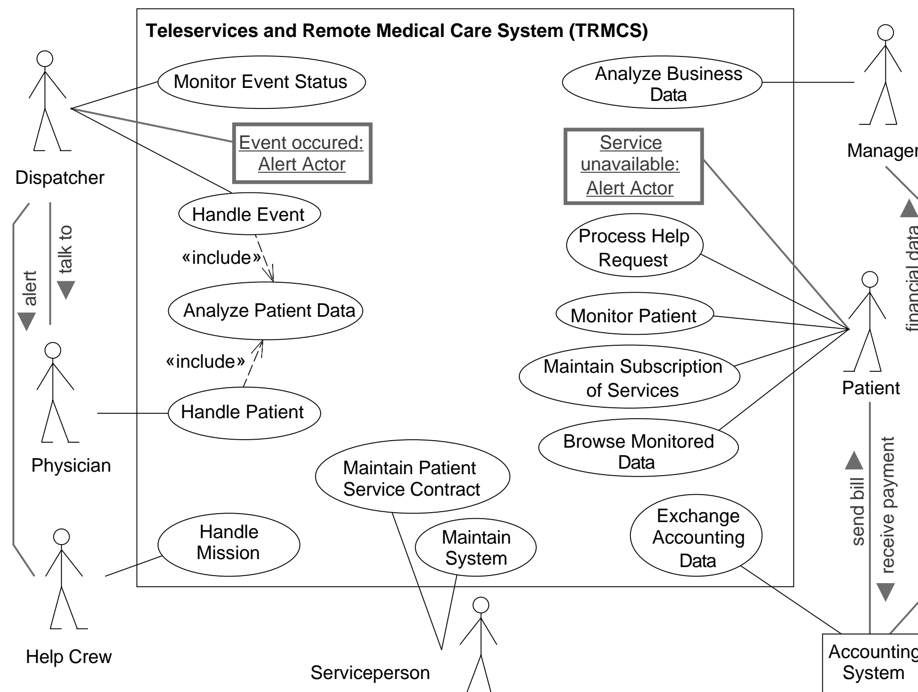
1. The TRMCS will be a distributed system having
   - a subsystem at the site of every patient
   - a subsystem at the site of every service provider
   - a mission management subsystem managing the missions needed to take care of the patients
   - one central subsystem.
2. The central subsystem will communicate with the provider subsystems through an Intranet with guaranteed quality of service.
3. The patient support subsystems will communicate with the other subsystems through the Internet and the telephone network.
4. All events (help calls, alarms) generated by patients are directed to the central site. The central site routes them to an appropriate service provider according to a policy based on patient contracts and provider availability.
5. The TRMCS will not contain an accounting component. It will provide accounting information to an external accounting system instead.
6. The TRMCS will leave all decisions about help or treatments to humans. It only supports the decision-making process by providing information and suggesting solutions.

## 3 A use case model of the TRMCS

Based on the system requirements and design decisions given above, we now want to model software requirements for the TRMCS, using UML as a modeling language. Conforming to the process recommended for the use of UML [17] we start with a use case model.

We identify the following actors who interact with the TRMCS: Patient, Dispatcher (person in a provider's office who handles events and help calls), Physician (consulting dispatchers and patients), Help Crew (visiting/rescuing a patient who needs help), Serviceperson, Manager (of the remote health care service), and Accounting System. For these actors we define a first-cut set of use cases (Figure 1; parts drawn in black).

In the following subsections, we examine various parts of this model more closely and identify six modeling issues which cannot adequately be handled with UML.




UML model elements are drawn in black. Model elements drawn in grey are needed, but cannot be modeled in UML.

**Figure 1.** Use case diagram of the TRMCS

### 3.1 System-actor interaction

In a first step, we consider the patient use cases (cf. Figure 1). Process Help Request deals with the situation where a patient actively asks for help. Monitor Patient describes the continuous monitoring of patient data (e.g. heart beat). Maintain Subscription of Services deals with subscription, update and deletion of TRMCS services by a patient. Browse Monitored Data allows a patient to look at the data that the monitoring function has recorded.

A closer look at the patient use cases reveals that they cannot specify the interaction between a patient and the TRMCS completely. For example, the TRMCS should warn a mobile patient when a service becomes unavailable because she or he is moving into a region with no connection to the mobile communication network. So we need an active model element (for example, an active object; see Figure 1) which is able to initiate communication between the system and an actor.

However, such an element cannot be modeled in UML: a use case by definition describes a sequence of actor stimuli and system responses *that is initiated by an actor* ([18], p.2-124). Active objects are not allowed in UML use case diagrams.

**Deficiency 1.** A UML use case model cannot specify interaction requirements where the system shall initiate an interaction between the system and an external actor.

### 3.2 Rich context

When modeling the context of a system, it is important that a rich context can be modeled, including interaction between actors [13].

In the TRMCS for example, a Dispatcher may phone a Physician and alert a Help Crew. Both interactions are related to the TRMCS but take place outside the TRMCS system boundary. We also have TRMCS-external communication between Accounting System, Patient and Manager (cf. Figure 1).

However, UML cannot model context associations because it forbids associations between actors ([18], p. 2-121).

**Deficiency 2.** UML cannot model a rich system context.

### 3.3 Use case structure and decomposition

Next, we consider the dispatcher use cases. The Dispatcher actor has two major use cases (Figure 1). Handle Event specifies how a dispatcher handles a help call (either issued by a patient or raised by a TRMCS monitoring component). A dispatcher shall handle more than one event in parallel. Monitor Event Status specifies how a dispatcher interacts with the TRMCS in order to monitor the status and progress of the events that are currently being handled. Again, an active element is needed alerting a dispatcher when an event occurs (cf. section 3.1 above).

The problem with Handle Event is that this is a large and complicated use case which in fact consists of a structured set of sub-use cases: Handle Event is a sequence of three sub-use cases: Acknowledge Event, Take Actions and Close Event. Take Actions is an iterative sequence of actions, for example to phone the patient, then send a nurse to the patient and finally informing the patient's physician about the findings and treatments of the nurse. Every action again is a sequence of three sub-use cases: Analyze Situation, Decide And Act and Observe And Get Feedback. The sub-use case Decide And Act consists of a set of alternative use cases, for example Inform Physician or Send Emergency Team.

Handle Event itself runs in parallel with Monitor Event Status because the status of all open events must be monitored continuously by the dispatcher.

Finally, there is a sub-use case Analyze Patient Data, which is used both within Handle Event (when analyzing what to do) and within the use cases of the actor Physician.

A good modeling language should allow us to model such kinds of structural relationships among use cases in a straightforward way. We imagine something like the diagram shown in Figure 2. We also expect that the structuring capability comes in combination with a decomposition facility, such that we can draw overview diagrams showing only Handle Event and Monitor Event Status as well as detailed diagrams for the dispatcher use cases showing all the structural details.

We now examine how UML handles the structure and decomposition of use cases. It turns out that this is a rather messy issue because the UML 1.3 specification is inconsistent and contradictory concerning the relationships between use cases.

**A. Structural relationships.** The first problem concerns structural relationships between use cases in UML. On the one hand, the UML specification states that every use case should express a complete sequence of interactions which is independent of any other use case and that use cases specifying the same system or subsystem must not communicate or have associations to each other ([18], pp. 2-122, 2-124 and 2-125). On the other hand, UML provides three kinds of relationships between use cases: Generalization, «Include» and «Extend». The generalization relates general use cases to special case use cases, which is not applicable in our example. Both «Include» and «Extend» imply the existence of use cases describing subsequences which are *not* necessarily complete and *do* require communication between the base use case and the included/extending use case. «Include» corresponds to a procedure call in programming. The relationship between Handle Event and Analyze Patient Data can be modeled this way. «Extend» means that the extending use case is inserted into the extended one at a designated extension point if a guarding condition is true (a mechanism corresponding to macro expansion in assembler programming). The alternative actions (Inform Physician, Send Emergency Team, etc. can be modeled as extensions.

Expressing sequential, parallel or iterative relationships between use cases is impossible in UML. A sequence of extensions could be tried as a workaround for the missing sequence relationships, but the resulting model would look quite cryptic. Another, better workaround is to exploit subsystem decomposition (see below).

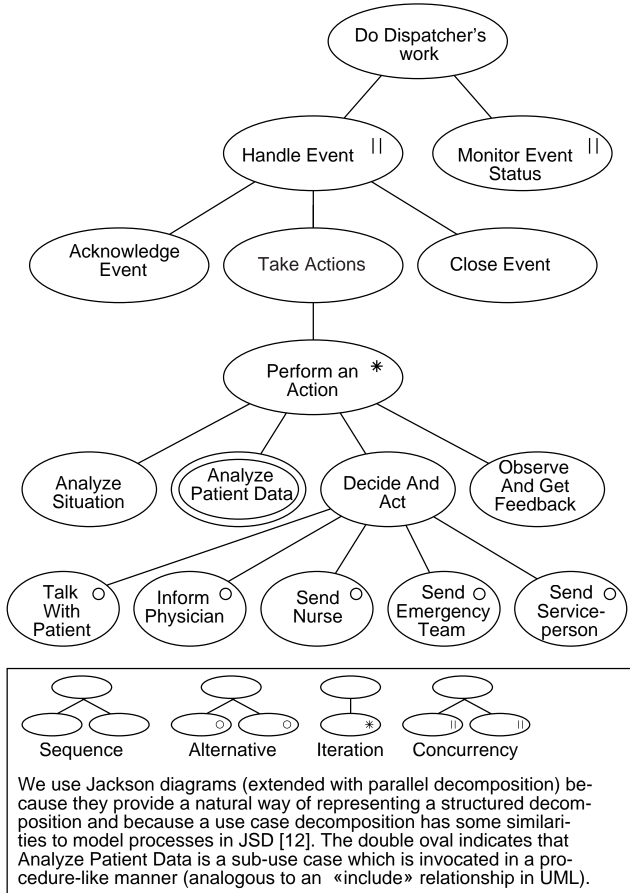



**Figure 2.** An intuitive model of the decomposition of the Handle Event use case (which is not possible in UML, however)

**B. Use case decomposition.** The second problem concerns decomposing a use case into a structured set of subordinate use cases. On the one hand, UML *explicitly forbids decomposing* use cases and *forbids communication among use cases* ([18], pp. 2-122 and 2-125). On the other hand, if a system is decomposed into subsystems, then the use cases of every subsystem *do form a decomposition* of one or more use cases of the system. The subordinate use cases *collaborate* to perform a superordinate one ([18], p. 2-125), which means that they *must communicate*.

It turns out that the property of use case independence (every use case being a complete sequence of interactions which is independent of any other use case) is necessarily violated as soon as we view a system on different levels of decomposition.

Subsystem decomposition gives us a workaround for modeling the structure of the Handle Event use case: we model Handle Event and Monitor Event Status on the system level. The sub-use cases of Handle Event are modeled in the Service Provider subsystem (Figure 3a). The structure of Handle Event is modeled by a collaboration within the Service Provider subsystem (Figure 3b). However, this model is an order of magnitude clumsier and more complex than a facility for directly modeling structure and decomposition of use cases in the style of Figure 2.

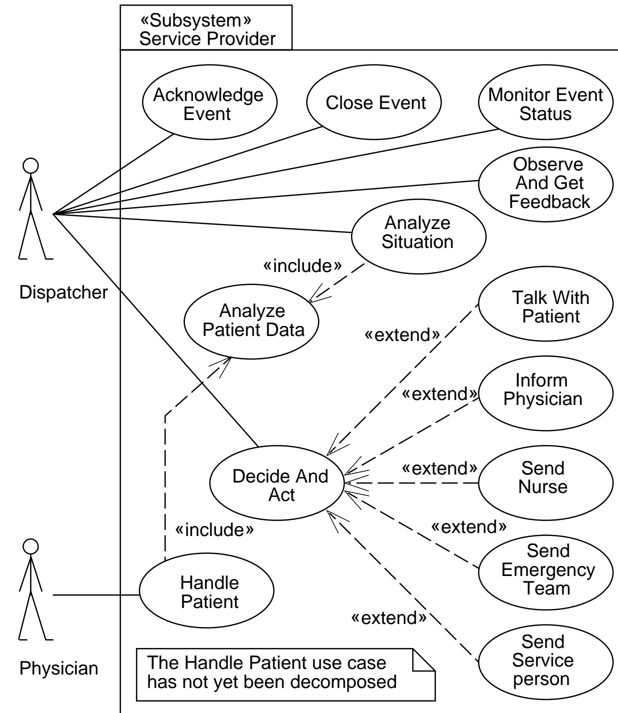



**Figure 3a.** UML model of the Service Provider subsystem with a decomposition of the Handle Event use case

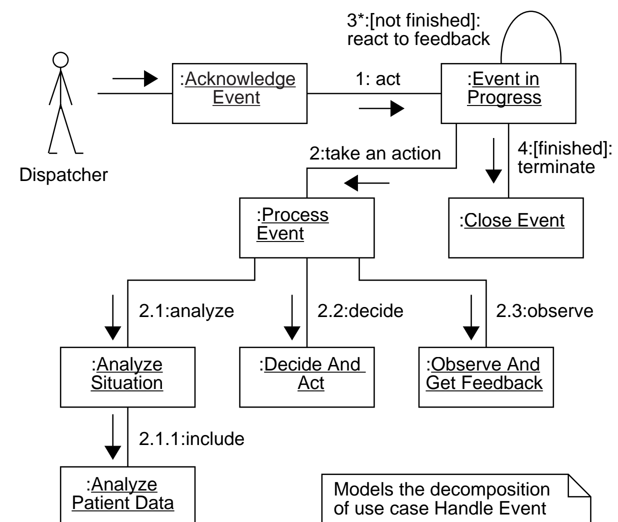



**Figure 3b.** An UML collaboration diagram defining the structure of the Handle Event use case

**Deficiency 3.** UML can neither express structure between use cases nor a structural hierarchy of use cases in an easy and straightforward way.

### 3.4 Use case interaction

UML does not model interaction between use cases. Communication and any associations between use cases are not allowed (see section 3.3). Preconditions such as "use case A requires that use case X has been previously executed" cannot be formally expressed in UML[1].

In reality, however, use cases *do* interact. For example, consider the patient use cases. The way how Process Help Request and Monitor Patient are carried out (and whether they are carried out at all) depends on the services the patient has subscribed to in the Maintain Subscription of Services use case.

Moreover, services itself may interact. Imagine a situation where Monitor Patient automatically issues a help call and transmits patient data, thereby blocking the communication device. If the patient decides to issue a manual help request at that time, the use case Process Help Request is blocked because the communication device is in use by Monitor Patient. In order to recognize and resolve such problems, it is important to identify use case interactions and to model them explicitly.

Use case interaction has similarities to feature interaction [16] as observed by Finkelstein [6].

In order to express use case interaction problems, we must be able to model *states* that can be *accessed and modified* by use cases. UML allows individual use cases to be modeled as state machines. However, UML cannot model states being shared between use cases, because a state machine must be allocated to a single classifier or behavioral element, but not to a subsystem ([18], p. 2-141 and 2-181).

**Deficiency 4.** UML provides no adequate means for dealing with use case interaction.

### 3.5 Use cases alone do not suffice

UML nurses the illusion that the functional requirements of a system can be expressed by a collection of use cases alone, without modeling any persistent state. As evidence, consider that the UML use case semantics say so[2], that the UML-inspired Rational Unified Process basically models requirements with use cases only [17], and that UML does not allow inter-use case state machines (see section 3.4 above).

However, this approach does not work in practice for any system where system state plays an important role. For the TRMCS, we have already demonstrated that the potential interaction between the patient use cases cannot be modeled without state variables that are shared between the use cases (see section 3.4 above). Moreover, the way a use case has to respond to the stimuli received from an actor frequently depends on the actual state of the system. For example, when a Patient sends a Turn monitoring on stimulus to the Monitor Patient use case, the reaction that has to be specified in this use case depends on

- whether the patient has previously subscribed to a monitoring service,
- whether the patient is allowed to use the service (if he did not pay his bill the TRMCS might block a service),
- whether the current geographical location of the patient allows communication with a provider.

These conditions in turn depend on the outcome of other use cases or on actions of a system entity which actively monitors a system condition (cf. section 3.1). It is impossible to specify the required reaction of the Monitor Patient use case without referring to state variables representing the three conditions listed above.

Theoretically, one could introduce pre- and postconditions for use cases and use global state variables in these conditions. Stereotypes would be the vehicle to do so in UML. However, such a specification would become extremely clumsy for all systems with more than a few state variables. As every system that needs a database belongs to this category, this approach provides no practical solution for the specification of state-dependent requirements. As far as we know, nobody has ever tried to integrate state-dependent behavior into a use case model in this way.

In a practical semiformal requirements language we must be able to combine use cases with a model of objects and states. The use cases capture functional requirements by specifying the behavior of a system as observed from a user's perspective. The object/state model, on the other hand, models both the state space and the events and operations that modify it. In [9] we present an approach that systematically and consistently combines a use case model and a class/object/state-model.

However, UML is quite weak here. Classes and their associated state machines are regarded as realizing use cases, not to augment them with a specification of state-dependent behavior. State machines shared by a set of use cases cannot be modeled (cf. section 3.4 above).

**Deficiency 5.** A UML use case model cannot express state-dependent system behavior adequately.

### 3.6 Tracing information flow

In order to model the requirements stemming from the system design decision about routing events (decision number four, see section 2), we have four options:

- We only model an association between the Dispatcher actor and the Process Help Request use case.

---

[1] In the pre-UML literature on use cases, there existed a notion of preconditions for use cases. UML 1.3 however, defines a precondition to be "a constraint that must be true when an operation is invoked" (Glossary in [18], p. B-13). Preconditions for use cases could at best be expressed informally in a textual description of a use case.

[2] "...the dynamic requirements of the system as a whole can be expressed with use cases." ([18], p. 2-127). "Each use case specifies a service the entity provides to its users (...). The service (...) is a complete sequence. This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again." ([18], p. 2-124).

• We model the complete process of forwarding, routing and delivering an event at its originating point, i.e. in the Process Help Request use case.
• We model the parts of the process where they belong, i.e. forwarding in Process Help Request, routing in the Central Site subsystem and delivery in the Service Provider subsystem, and we model the flow of information between these entities.
• We model the parts as in the third option, but instead of modeling the information flow, we describe the sequence of subprocesses (forwarding-routing-delivery) in the originating use case (as done in the second option).

The first option ignores the information flow between subsystems, which is incompatible with the notion of decomposition. The second option concentrates the flow requirements in one use case, which contradicts the principles of information hiding and separation of concerns. Furthermore, the first two options both leave the specification of Central Site and Service Provider substantially incomplete.

The fourth option introduces too much redundancy into the specification and also breaks information hiding.

So we decide in favor of the third option. (By the way, modeling components and flow of information were strengths of the structured analysis methods of the Eighties.) However, modeling such a problem in UML turns out to be quite clumsy.

What we would expect is that we have to model the items shown in Figure 4a. An Event Router entity specifies the routing requirements in subsystem Central Site. An Event Delivery entity specifies the delivery requirements in subsystem Service Provider. Associations specify the information flow.

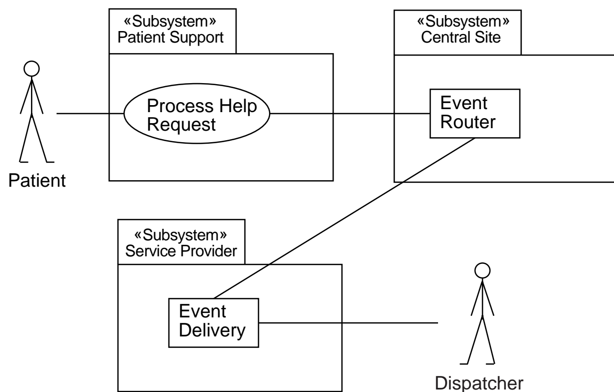



**Figure 4a.** Information flow from end to end

However, the information flow must also be modeled in the context of every individual subsystem and on the system level.

• On the subsystem level, we therefore must add the following model elements (Figure 4b):
  • Event Router as an actor in the context of subsystem Patient Support.
  • Process Help Request as an actor in the context of Central Site
  • Event Delivery as an actor in the context of Central Site
  • Event Router as an actor in the context of Service Provider.

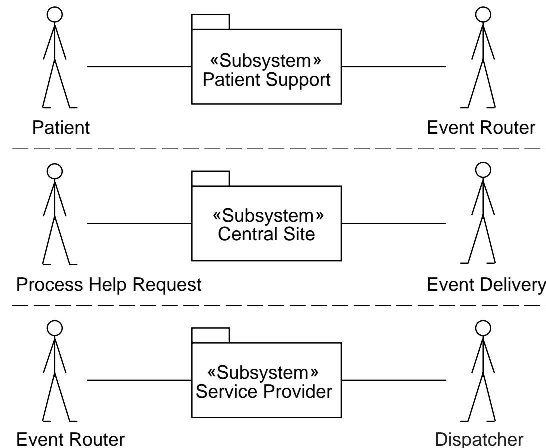



**Figure 4b.** Information flow on the level of individual subsystems

• On the system level, we have to model the following associations (part of Figure 4c):
  • between Patient Support and Central Site, and between Central Site and Service Provider, expressing the information flow between the subsystems
  • between Process Help Request and Dispatcher to indicate the information flow in the use case model (to be added to Figure 1).

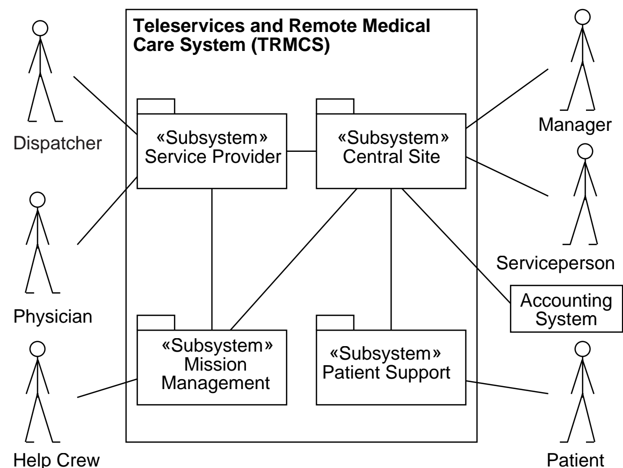



**Figure 4c.** Subsystem decomposition of the TRMCS

As the semantic relationships between all these elements cannot be expressed formally, it is impossible to establish or maintain them automatically with a tool. Consistency must be secured manually, which is difficult and expensive in large models.

**Deficiency 6.** Modeling information flow in a system consisting of subsystems is awkward in UML.

# 4 Decomposition of the TRMCS

In this section we investigate the decomposition problems which arise when UML is used as a requirements specification language.

Every large system needs to be decomposed in order to make it comprehensible and manageable. A good decomposition (one that follows the basic software engineering principles of information hiding and separation of concerns) decomposes a system recursively into parts such that

(i) every part is logically coherent, shares information with other parts only through narrow interfaces and can be understood in detail without detailed knowledge of other parts,

(ii) every composite gives an abstract overview of its parts and their interrelationships.

If high-level design decisions imply a subsystem structure, then the decomposition of the requirements model on the next lower level may also follow this structure. Design decisions concerning system decomposition should follow information hiding criteria anyway. Figure 4c shows such a decomposition for the TRMCS.

UML basically has three model elements that can be decomposed hierarchically:

- A *package* is a container and a namespace for an arbitrary set of model elements. The decomposition has no other semantics.
- A *subsystem* is a specialization of a package. Subsystems partition a system into a set of behavioral subunits. Every subsystem encapsulates its behavior.
- A *class* can be a composite aggregation of a set of part classes.[3] [4]

## 4.1 Subsystems need high-level behavior

If a subsystem is constructed according to information hiding criteria, it is typically not a mere container for the model elements which make up the subsystem. In particular, a subsystem frequently exhibits behavior of its own, specifying the high-level behavior of the subsystem as a whole.

The TRMCS for example, comprises a set of Service Provider subsystems. Each of these subsystems has a high-level behavior expressed by the following states:

- non-operational: the provider is known to the TRMCS, but currently it is not in business,
- operational: the provider is in business and is providing services (or is principally ready to do so),
- starting up: the provider is in the process of becoming operational,
- closing down: the provider is in the process of becoming non-operational.

The operational state in turn has three substates: idle (waiting for events to serve), active (handling at least one event), and overloaded (unwilling to accept further events).

An adequate model of these states and their associated behavior would be a statechart (or state machine in UML terminology) on the level of the subsystem (Figure 5). However, UML regards subsystems as containers of behavioral entities only and hence disallows associating a state machine with a subsystem[5].

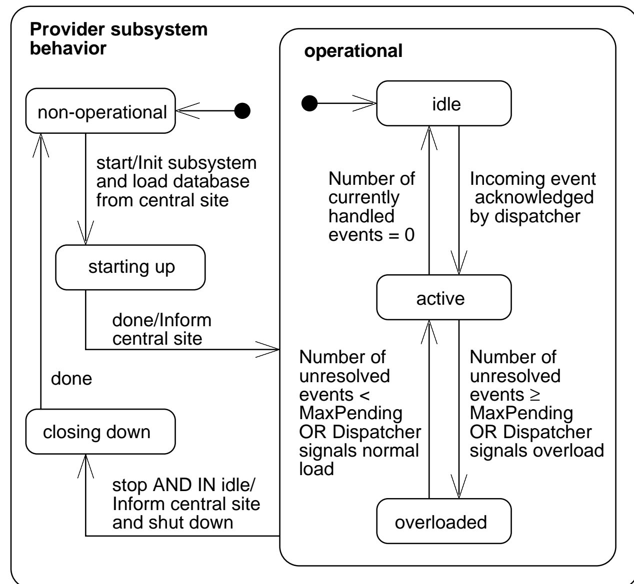



**Figure 5.** High-level behavior of the Service Provider subsystem (not possible in UML)

As a workaround, one could consider using classes and composition aggregation for decomposition instead of subsystems. This would make it easy to model behavior on any level of the decomposition. However, classes are unsuitable for modeling a subsystem decomposition (see section 4.2 below).

Thus it is impossible in UML to model subsystem behavior as described above for the TRMCS.

**Deficiency 7.** UML cannot model the behavior of high-level system components such as subsystems.

## 4.2 Subsystems are objects

A closer look at the Service Provider subsystem reveals that this subsystem not only has a behavior of its own, but also has subsystem-level *operations* and *attributes*. For example, start, stop (cf. Figure 5) and isAvailable (deter-

[3] Theoretically, composition is a specialization of aggregation and hence defined on any set of classifiers (Class, Actor, Use Case, Component,...). However, semantics are defined for class composition only.

[4] A classifier is also a namespace which may contain a (restricted) set of model elements in its scope. The semantics is the same as if the classifier additionally would be a package (except that a classifier cannot import model elements from other namespaces).

[5] In the UML metamodel, Subsystem is defined as a specialization both of Package and of Classifier. Since state machines can be associated with classifiers, this should also be possible with subsystems. However, the definition of subsystem semantics does not allow us to do so: “A subsystem has no behavior of its own” ([18], p. 2-181).

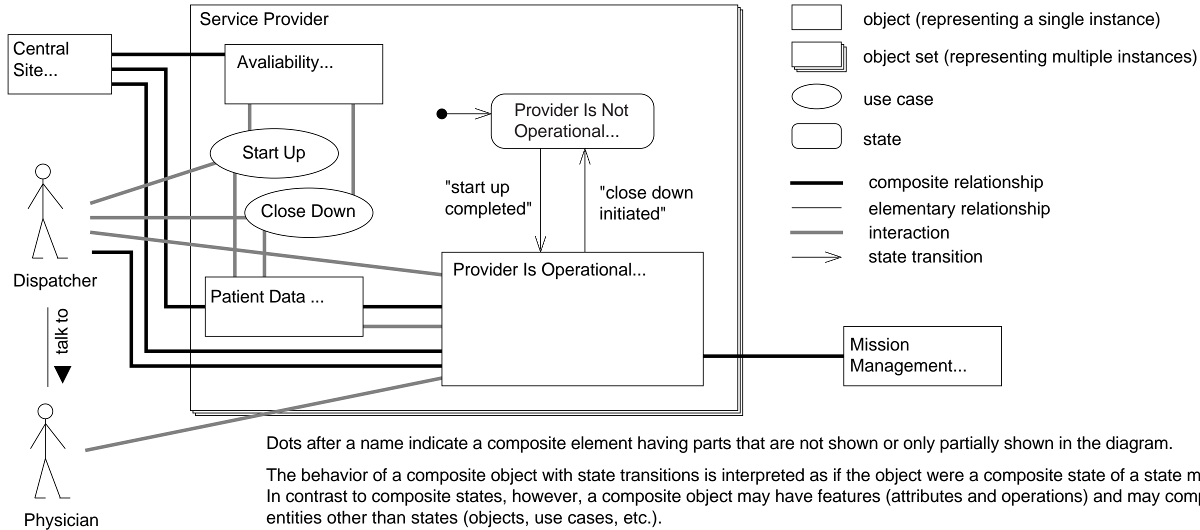



**Figure 6.** An aspect-integrated model of the Service Provider subsystem and its context (not possible in UML)

mines whether the subsystem currently accepts events) are operations on the subsystem as a whole. The subsystem name, its service capacity and MaxPending (the maximum number of events that can be pending before the subsystem becomes overloaded; cf. Figure 5) are examples of attributes characterizing the subsystem as a whole.

Taking these facts into account and recalling that a subsystem may have behavior of its own (see section 4.1), we can conclude that a subsystem should not be simply a container for the model elements that make up a behavioral unit of a system, as UML sees it. Instead, a subsystem would be better characterized as a composite object that plays a given role in the system. Entities of this kind are modeled by classifier roles in UML. However, a UML classifier role can only occur within collaborations and there is no decomposition defined for classifier roles – hence classifier roles cannot do the job.

A potential alternative might be to model subsystems as UML classes and use composition aggregation as a vehicle for hierarchical decomposition. However, this approach does not work either. The main reason is that we frequently have situations where objects of the same class occur in different subsystems, playing different roles there. In the TRMCS for example, we have an active object alerting a dispatcher when an event arrives. We also have an active object alerting a patient when a service becomes unavailable. Both objects belong to the same class Alert Actor, but they are embedded in different subsystems where they play different roles, collaborating with different actors and objects. Event List is another example. In the Service Provider subsystem, we need Event List objects in three roles: Pending Events, Events In Progress (list of currently handled events) and Local Event History. In the Central Site subsystem, another Event List object plays the role of Global Event History.

As we cannot allocate a class in two different places (by definition, a decomposition is strictly partitioning), decomposing classes would require defining a subclass for every role, resulting in highly complex and artificial models.

**Deficiency 8.** UML cannot adequately model the decomposition of a distributed system like the TRMCS, neither with the language element Subsystem nor with another UML language element.

### 4.3 Aspect-integrated component views

Structured Analysis, which was the standard language before object-oriented approaches took over [5], [11], had an outstanding strength: the hierarchical decomposition of dataflow diagrams. This feature made it possible to decompose a system recursively into smaller, less complex parts. Every part was a comprehensive local specification of the aspects of functionality (activities), data (stores) and behavior (control specs). Every composite was an abstraction of its parts and of the information flow between the parts.[6] In more abstract terms, Structured Analysis provided a separation of concerns by separating subproblems.

In an object-oriented requirements specification, it would also be quite valuable to have a hierarchical decomposition that separates subproblems, but keeps all aspects of a subproblem together. With such a decomposition, understanding a selected subproblem would be much easier because we no longer need to assemble the required information from a collection of different (and possibly separately decomposed) aspect models.

For example, consider the Service Provider Subsystem. To get an overview of this subsystem, we want to model the subsystem as a set of abstract, high-level components: classes or objects and their interrelationships, use cases and the entities they communicate with, and the high-level behavior of the subsystem (Figure 6). For every complex

[6] It must be mentioned, however, that this paradigm was broken by the global data dictionary which was not decomposable.

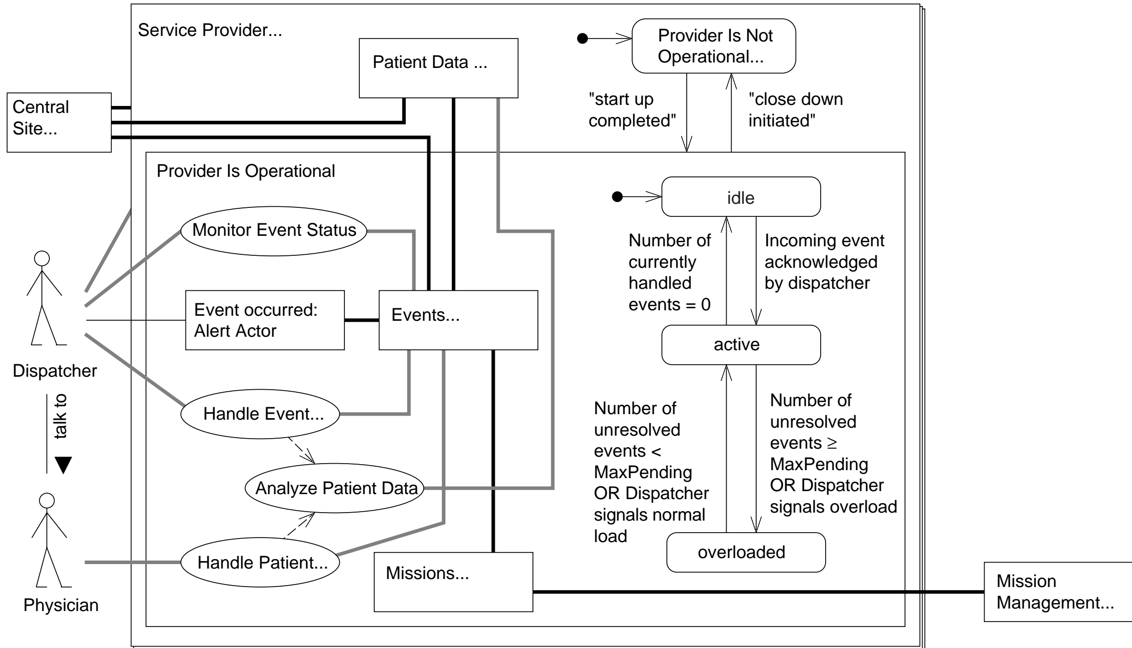



**Figure 7.** Decomposition of the composite object Provider Is Operational and its context (not possible in UML). Elements of Service Provider that do not communicate with Provider Is Operational are suppressed.

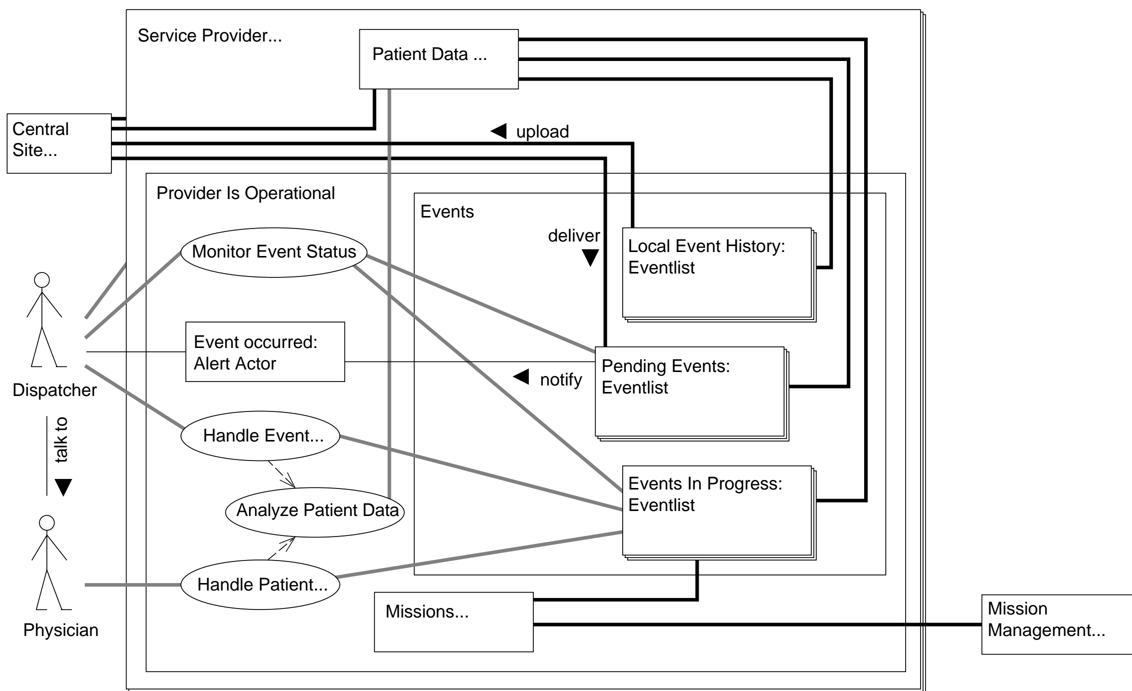



**Figure 8.** Decomposition of the composite object Events within Provider Is Operational and its context (not possible in UML). Again, some elements of Service Provider are suppressed. The elements of Events are not further decomposable.

component such as Provider Is Operational in Service Provider, we would like to do the same (Figure 7) and proceed recursively until we arrive at simple classes or objects, use cases and states (Figure 8).

However, modeling such an aspect-integrated decomposition of a system is impossible in UML[7]. A basic principle underlying the design of UML is to provide separate

[7] The diagrams of Figures 6-8 have been drawn using an alternative object modeling language called ADORA [10], [15] (see section 6).

models for different aspects; the separation of class diagrams and use case diagrams being the most prominent example. Subsystems were added later to UML when the need for a system decomposition mechanism was recognized by the creators of UML. However, subsystems cannot do the job: they can neither act as a composite object or composite state nor can they have behavior of their own (cf. Deficiency 7 in section 4.1).

**Deficiency 9.** UML cannot model all aspects of a composite entity like a subsystem together in a single view.

## 5 Can we overcome the deficiencies?

In this section we investigate whether the problems identified in sections 3 and 4 can be overcome using the UML extension and tailoring facilities.

UML has a powerful built-in extension facility: the so-called *stereotypes*. A stereotype in a modeling language is a well-formed mechanism for expressing user-definable extensions, refinements or redefinitions of elements of the language without (directly) modifying the metamodel of the language. Although stereotypes do not change the metamodel, they are conceptually powerful enough to completely redefine a language [2]. However, in the more recent versions of UML, the power of UML stereotypes has been seriously constrained by the decision to restrict the number of stereotypes per model element to one and by disallowing a stereotype to have features and associations.

*Profiles* are a mechanism for adapting UML to the needs of specific kinds of systems or specific problem domains. The basic idea is to tailor the language by selecting a subset of the metamodel and introducing additional well-formedness rules, standard elements, and constraints. However, for those profiles that the OMG is intending to adopt as a standard, the OMG also allows a profile to modify the metamodel if absolutely necessary ([19], p. 23).

As we will see below, some of the deficiencies can be fixed with stereotypes. Profiles, on the other hand, do not provide much help in our case, because they are primarily oriented towards tailoring by subsetting and constraining. At best, a "Distributed Systems" profile could tie together the minor metamodel modifications required to fix or alleviate the deficiencies concerning subsystems and decomposition. Doing major metamodel modifications in a profile would be a misuse of this mechanism.

A fundamental point in the analysis of a deficiency is whether the nature of the problem is accidental, essential or fundamental. *Accidental* deficiencies in a language can be fixed by minor modifications that fully conform to the paradigm of the language, i.e. to its basic ideas, structures and properties. Overcoming *essential* deficiencies requires modifications affecting major concepts of the language. *Fundamental* deficiencies cannot be removed without modifying basic concepts of the language.

Analyzing our nine deficiencies, we can say that

- deficiencies 1 and 2 are accidental,
- deficiencies 3, 4, 5, and 7 are essential,
- deficiencies 6, 8, and 9 are fundamental.

*Deficiency 1* (missing active model elements in use case diagrams) can be removed with a stereotype «active» for use cases or with a slight modification in the metamodel allowing the inclusion of active objects in use case diagrams. *Deficiency 2* (no rich context) can be removed with a small, local modification in the metamodel, allowing associations between actors.

*Deficiency 3* (no adequate modeling of use case structure and hierarchy) can partially be treated by defining a use case structure diagram in the style of Figure 2. This can be accomplished with a set of stereotypes for use cases and dependency relationships. However, consistency between such use case structures and the structure introduced by the subsystem decomposition cannot be ensured. Removing this problem would require a uniform model decomposition concept – a fundamental modification.

*Deficiency 4* (inadequate treatment of use case interaction), *Deficiency 5* (inadequate modeling of state-dependent system behavior) and *Deficiency 7* (no models of high-level component behavior) are related. As first aid for alleviating these problems, the following three measures could be taken. They imply moderate modifications of the UML metamodel and the UML semantics.

- Allow a subsystem to have behavior of its own, i.e. allow the attaching of state machines to subsystems.
- Augment the use case model in the specification part of a subsystem with a class/object model which models state variables and operations/events modifying them. Consider this class/object model not as a realization of the use case model; instead view the two models as being complementary.
- Establish consistency between the two models with systematic cross-referencing [9].

In order to remove these deficiencies completely, a semantic integration of the aspects of structure, functionality, behavior and user interaction would be necessary – again a fundamental modification.

The treatment of *Deficiency 6* (awkward information flow models), *Deficiency 8* (inadequate decomposition concepts), and *Deficiency 9* (no aspect-integrated views of composite entities) as well as the complete removal of deficiencies 3, 4, 5, and 7 would require modifications in the very foundations of UML – abandoning the concept of a loosely coupled collection of aspect models and moving towards an integrated model with a uniform decomposition mechanism (cf. section 6).

## 6 Is there an alternative to UML?

For the practical use of a universal semiformal requirements specification language in industry, there is currently no alternative to UML.

However, from a research point of view there is life beyond UML. UML is built upon two fundamental concepts:

1. A class model is the core of a UML specification. This can easily be seen when analyzing the contents of the Core Package in the UML metamodel. The UML class concept still preserves the basic paradigm of its ancestor, the entity-relationship model: it is basically a flat structural description of the objects that the system has to deal with.
2. Specifications in UML consist of a collection of loosely coupled models (class model, use case models, collaborations, activity models, etc.). These are tied together by very few and semantically quite weak rules.

Both these concepts are not at the heart of an object-oriented modeling approach. If we put them aside, we open a design space for object-oriented modeling languages which are conceptually different from UML.

In our research group we have developed such a language which we call ADORA (Analysis and Description of Requirements and Architecture) [1], [10], [15]. The foundation of ADORA is a hierarchy of abstract objects where each object truly integrates the aspects of structure, functionality, behavior and user interaction.

In ADORA we get rid of the problems related to decomposition and aspect interaction plaguing UML (Deficiencies 3 to 9). A detailed description of ADORA is beyond the scope of this paper. Figures 6, 7 and 8 give an impression of how an ADORA model looks.

# 7 Conclusions

In this paper, we have described a set of deficiencies of UML as a language for semiformal requirements specification. We have taken a pragmatic approach, identifying the problems that become apparent when using UML for the specification of a distributed system. We have also analyzed the nature of these deficiencies and discussed how to overcome them. It turns out that some problems can be fixed, while major deficiencies are rooted in fundamental concepts of UML and thus are here to stay with UML.

Our findings provide insight and guidance both for the further evolution of UML and for research on alternative modeling languages for requirements specification that might replace UML in the future.

# References


[1] Berner, S., Joos, S., Glinz, M. Arnold, M. (1998). A Visualization Concept for Hierarchical Object Models. *Proceedings 13th IEEE International Conference on Automated Software Engineering (ASE-98)*. 225-228.

[2] Berner, S., Glinz, M., Joos, S. (1999). A Classification of Stereotypes for Object-Oriented Modeling Languages. *Proceedings 2nd International Conference on the Unified Modeling Language*, Fort Collins. Berlin: Springer. 249-264.

[3] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, Ca.: Benjamin/ Cummings.

[4] Coad, P., Yourdon E. (1991). *Object-Oriented Analysis*. Englewood Cliffs, N. J.: Prentice Hall.

[5] DeMarco, T. (1978). *Structured Analysis and System Specification*. New York: Yourdon Press.

[6] Finkelstein, A. (2000). Personal communication at the IFIP WG 2.9 Workshop on Requirements Engineering in Flims, Switzerland.

[7] Inverardi, P., Muccini, H. (2000). Case study for the Tenth International Workshop on Software Specification and Design (IWSSD-10). http://www.ics.uci.edu/IRUS/iwssd/case-study.pdf

[8] Glinz, M. (1991). Probleme und Schwachstellen der Strukturierten Analyse. [Problems and weaknesses of structured analysis (in German)] In Timm, M. (ed.): *Requirements Engineering '91*. Informatik-Fachberichte Vol. 273, Berlin: Springer. 14-39.

[9] Glinz, M. (2000). A Lightweight Approach to Consistency of Scenarios and Class Models. *Proceedings 4th IEEE International Conference on Requirements Engineering*. Schaumburg, Ill. 49-58.

[10] Glinz, M., Berner, S., Joos, S., Ryser, J., Schett, N., Schmid, R., Xia, Y. (2000). The ADORA Approach to Object-Oriented Modeling of Software. Research report 2000.07, Institut für Informatik, University of Zurich. http://www.ifi.unizh.ch/groups/req/ftp/papers/ADORA_approach.pdf

[11] Hatley, D.J., Pirbhai, I.A. (1988). *Strategies for Real-Time System Specification*. New York: Dorset House.

[12] Jackson, M.A. (1983). *System Development*. Englewood Cliffs, N.J.: Prentice Hall.

[13] Jackson, M.A. (1995). Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices. Wokingham: Addison-Wesley.

[14] Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G. (1992). *Object-Oriented Software Engineering – A Use Case Driven Approach*. Reading, Mass.: Addison-Wesley.

[15] Joos, S. (1999). *ADORA-L – Eine Modellierungssprache zur Spezifikation von Software-Anforderungen* [ADORA-L – A modeling language for specifying software requirements. In German]. PhD Thesis, University of Zurich.

[16] Keck, D.O., Kuehn, P.J. (1998). The Feature and Service Interaction Problem in Telecommunications Systems: A Survey. *IEEE Transactions on Software Engineering* **24**, 10 (Oct. 1998). 779-796.

[17] Kruchten, P. (1998). *The Rational Unified. Process: An Introduction*. Reading, Mass.: Addison-Wesley.

[18] OMG (1999). *OMG Unified Modeling Language Specification Version 1.3*. OMG document ad/99-06-08. ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf

[19] OMG (1999). *UML Profile for Scheduling, Performance, and Time – Request for Proposal.* OMG document ad/99-03-13.

[20] Ross, D. (1977). Structured Analysis (SA): A Language for Communicating Ideas. *IEEE Transactions on Software Engineering*, **SE-3**, 1 (Jan. 1977). 16-34.

[21] Rumbaugh, J., Jacobson, I., Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Reading, Mass.: Addison-Wesley.

[22] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs, N. J.: Prentice Hall.

[23] Teichroew, D., Hershey III, E.A. (1977). PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems. *IEEE Transactions on Software Engineering*, **SE-3**, 1 (Jan. 1977). 41-48.