The ADORA Approach to Object-Oriented Modeling of Software

Martin Glinz, Stefan Berner, Johannes Ryser, Nancy Schett, Reto Schmid, Yong Xia
Institut für Informatik, Universität Zürich
Winterthurerstrasse 190
CH-8057 Zurich, Switzerland
+41-1-63 54570
http://www.ifi.unizh.ch/~glinz

Stefan Joos
Robert Bosch GmbH
Postfach 30 02 20

D-70469 Stuttgart, Germany

stefan.joos@de.bosch.com

Technical Report 2000.07, Institut für Informatik, Universität Zürich

ABSTRACT

In this paper, we present the ADORA approach to object-oriented modeling of software (ADORA stands for Analysis and Description of Requirements and Architecture). The main features of ADORA that distinguish it from other approaches like UML are the *use of abstract objects* (instead of classes) as the basis of the model, a *systematic hierarchical decomposition* of the modeled system and the *integration* of all aspects of the system *in one coherent model.*

The paper introduces the concepts of ADORA and the rationale behind them, gives an overview of the language, sketches a novel concept for visualizing the model hierarchy with a tool and reports the results of a validation experiment for the ADORA language.

Keywords: Object-oriented modeling, specification, requirements engineering, ADORA

1 INTRODUCTION

When we started our work on object-oriented specification some years ago, we were motivated by the severe weaknesses of the then existing methods, e.g. [4][6][17]. In the meantime, the advent of UML [18] (and to a minor extent, OML [8]) has radically changed the landscape of object-oriented specification languages. However, also with UML and OML, several major problems remain.

There is still no true integration of the aspects of data, functionality, behavior and user interaction. Neither do we have a systematic hierarchical decomposition of models (for example, UML packages are a simple container construct with nearly no semantics). Models of system context and of user-oriented external behavior are weak and badly integrated with the class/object model.

So there is still enough motivation not to simply join the UML mainstream and to pursue alternatives instead. We are developing an object-oriented modeling method for software that we call ADORA (Analysis and Description of Requirements and Architecture). ADORA is intended to be used primarily for requirements specification and also for logical-level architectural design. Currently, it has no language elements for expressing physical design models (distribution, deployment) and implementation models.

The main reason why we pursue a new approach and do not integrate our ideas into UML is because basic concepts of ADORA are essentially different from those of UML. Thus, integrating ADORA into UML would require a considerable redefinition of UML (see section 7).

In this paper, we present the ADORA approach, focusing on the general concepts and on the language.

The main contributions of ADORA are

- a method that creates an *integrated model* that is based on *abstract objects*, not on classes and that uses *hierarchical decomposition* as its main means of structuring a system,
- a tool that *visualizes models in context* according to their logical structure,
- an open and flexible modeling process that, in particular, allows *tailoring the formality* of ADORA models to the problem at hand.

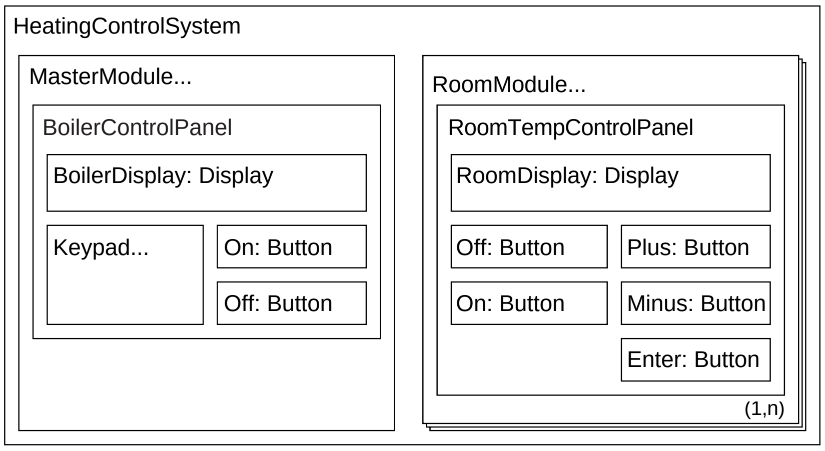
Throughout this paper, we will use a distributed heating control system as an example. The goal of this system is to provide a comfortable control for the heating system of a building with several rooms. An operator can control the complete system, setting default temperatures for the rooms. Additionally, for every room individual temperature control can be enabled by the operator. Users then can set the desired temperature using a control panel in the room. The system shall be distributed into one master module serving the operator and many room modules.

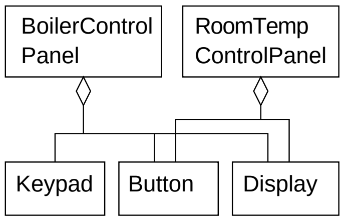
The rest of the paper is organized as follows. In sections 2-4 we present the main features of ADORA, starting with the key concepts and presenting then the language and the visualization concept of the tool. Section 5 briefly sketches how ADORA fits into software processes. In section 6 we present the results of a first validation of the ADORA language. Finally, we compare the concepts of ADORA with those of UML and conclude with a discussion of results, state of work and future directions.

2 KEY CONCEPTS OF THE ADORA APPROACH

ADORA is based on five principal ideas:

- Working with abstract objects (instead of classes)
- Structuring the system being modeled with hierarchical decomposition





Left: ADORA model of a distributed heating control system. MasterModule and Room-Module are partially visualized (indicated by dots after name); showing the control panels only. Display and Button are types.

Top right: Conventional class model of the control panels

Figure 1. An ADORA object model (left) vs. a conventional class model (top right)

- Using an integrated model (instead of collections of models)
- Allowing users to express different parts of a specification with varying degree of formality (adapted to the importance and risk of the parts)
- Visualizing models in context by presenting details of a model always with an abstraction of its surrounding context.

In this section, we briefly describe these five principles and give our rationale for choosing them.

Abstract Objects instead of Classes

When we started the ADORA project, all existing object-oriented modeling methods used class diagrams as their model cornerstone. However, class models are inappropriate when more than one object of the same class and/or collaboration between objects have to be modeled [14]. Both situations frequently occur in practice. For an example, see the buttons in Fig. 1. Moreover, class models are difficult to decompose. As soon as different objects of a class belong to different parts of a system (which often is the case), hierarchical decomposition does no longer work for class models [14]. Wirfs-Brock [22] tries to overcome the problems of class modeling by using classes in different *roles*. However, decomposition remains a problem: what does it mean to decompose a role?

We therefore decided to use abstract, prototypical objects as the core of an ADORA model (Fig. 1). An equivalent to classes (which we call types) is only used to model common characteristics of objects: types define the properties of the objects and can be organized in subtype hierarchies. In order to make models more precise, we distinguish between *objects* (representing a single instance) and *object sets* that represent a set of instances. Modeling of collaboration and of hierarchical decomposition (see below) becomes easy and straightforward with abstract objects.

In the meantime, others have discovered the benefits of modeling with abstract objects, too. UML, for example, uses abstract objects for modeling collaboration in collaboration diagrams and in sequence diagrams[1]. However, without a notion of abstraction and decomposition, only local views can be modeled. Moreover, class diagrams still form the core of an UML specification.

Hierarchical Decomposition

Every large specification must be decomposed in some way in order to make it manageable and comprehensible. A good decomposition (one that follows the basic software engineering principles of information hiding and separation of concern) 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.

The current object-oriented modeling methods typically approach the decomposition problem in two ways: (a) by modeling systems as collections of models where each model represents a different aspect or gives a partial view of the system, and (b) by providing a container construct in the language that allows the modeler to partition a model into chunks of related information (e.g. packages in UML). However, both ways do not satisfy the criteria of a good decomposition. Aspect and view decompositions are coherent only as far as the particular aspect or view is concerned. The information required for comprehending some part of a system in detail is *not* coherently provided. Container constructs such as UML packages have semantics that are too weak for serving as composites in the sense that the composite is an abstract overview of its parts and their interrelationships. This is particularly true for multi-level decompositions. Only the ROOM method [21] can decompose a system in a systematic way. However, as ROOM is also based on classes, the components are not

[1] There is no consistent notion of abstract objects in UML. In collaboration diagrams Classifier Role is used to represent abstract objects whereas in sequence diagrams Object is used for the same purpose. The UML reference manual increases the confusion by stating that collaborations use objects ([18] pp. 29, 196 and 530).

classes, but class references. This asymmetry makes it impossible to define multi-level decompositions in a straightforward, easily understandable way.

In ADORA, the decomposition mechanism was deliberately chosen such that good decompositions in the sense of the definition given above become possible. We recursively decompose objects into objects (or elements that may be part of an object, like states). So we have the full power of object modeling on all levels of the hierarchy and only vary the degree of abstractness: objects on lower levels of the decomposition model small parts of a system in detail, whereas objects on higher levels model large parts or the whole system on an abstract level.

Integrated Model

With existing modeling languages, one creates models that consist of a set of more or less loosely coupled diagrams of different types. UML is the most prominent example of this style. This seems to be a good way to achieve separation of concerns. However, while making separation easy, loosely coupled collections of models make the equally important issues of integration and abstraction of concerns quite difficult.

In contrast to the approach of UML and others, an ADORA model integrates all modeling aspects (structure, data, behavior, user interaction...) in one coherent model. This allows us to develop a strong notion of consistency and provides the necessary basis for developing powerful consistency checking mechanisms in tools. Moreover, an integrated model makes model construction more systematic, reduces redundancy and simplifies completeness checking.

Using an integrated model does of course not mean that everything is drawn in one single diagram. Doing so would drown the user in a flood of information. We achieve separation of concerns in two ways: (1) We *decompose* the model *hierarchically*, thus allowing the user to select the focus and the level of abstraction. (2) We use a *view concept* that is based on *aspects*, not on various diagram types. The *base view* consists of the objects and their hierarchical structure only. The base view *is combined with* one or more *aspect views*, depending on what the user wishes to see. These two concepts – hierarchy and combination of views – constitute the *essence* of organizing an ADORA model.

Depending on the capabilities of a tool, combined views either are fixed and have to be explicitly drawn by the user or (better) they are generated from a repository by the tool.

Adaptable Degree of Formality

An industrial-scale modeling language should allow its users to adapt the degree of formalism in a specification to the difficulty and the risk of the problem at hand. Therefore, they need a language with a broad spectrum of formality in its constructs, ranging from natural language to completely formal elements.

In ADORA, we satisfy this requirement by giving the modeler a choice between informal, textual specifications and formal specifications (or a mixture of both). For example, an object may be specified with an informal text only. Alternatively, it can be formally decomposed into components. These in turn can be specified formally or informally. As another example, state transitions can be specified in a formal notation or informally with text or with a combination of both.

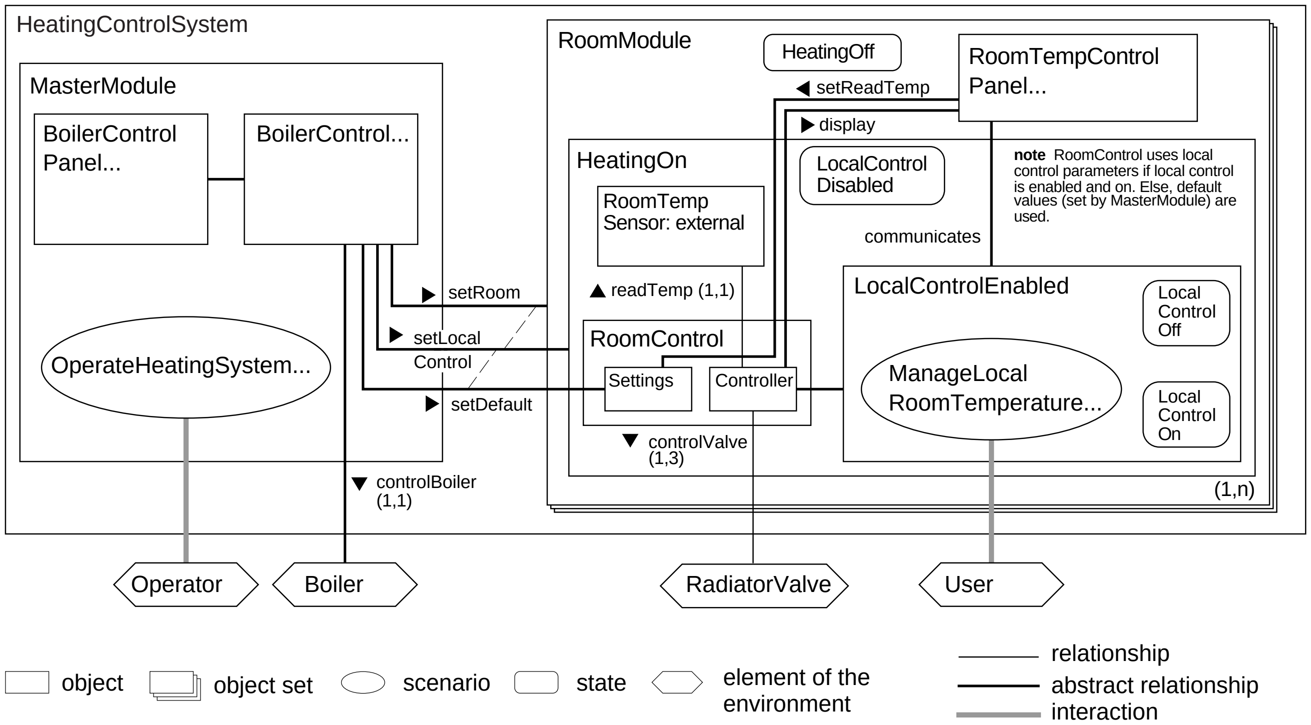


Figure 2. An ADORA view of the heating system: base view combined with structural view and context view

The syntax of the ADORA language provides a consistent framework for the use of constructs with different degrees of formality.

Contextual visualization

The current specification tools lack capabilities for an adequate visualization of hierarchical decomposition. They typically provide explosive zoom only. In ADORA, we are using a novel visualization concept for composition abstractions which is based on the notion of fisheye views. This technique supports the abstraction mechanisms of the language directly by corresponding visualization mechanisms in the tool.

3 AN OVERVIEW OF THE ADORA LANGUAGE

An ADORA model consists of a basic hierarchical object structure (the base view, as we call it) and a set of aspect views that are combined with the base view. In this section we describe these views and their interaction.

3.1 Basic Hierarchical Object Structure

The object hierarchy forms the basic structure of an ADORA model.

Objects and object sets. As already mentioned above, we distinguish between objects and object sets. An *ADORA object* is an abstract representation for a single instance in the system being modeled. For example, in our heating control system, there is a single boiler control panel, so we model this entity as an object. Abstract means that the object is a placeholder for a concrete object instance. While every object instance must have an object identifier and concrete values for its attributes, an ADORA object has neither of these. An *ADORA object set* is an abstract representation of a set of object instances. The number of instances allowed can be constrained with a cardinality. For example, in an order processing system we would model suppliers, parts, orders, etc. as object sets. In the heating control system, we have a control panel in every room and we control at least one room. Thus we model this panel as an object set with cardinality (1,n); see Fig. 2.

Structure of an ADORA object. An object or object set has a twofold inner structure: it consists of a set of properties and (optionally) a set of parts.

The *properties* are attributes (both public and private ones), directed relationships to other objects/object sets, operations and so called standardized properties. The latter are user-definable structures for stating goals, constraints, configuration information, notes, etc.; see below.

The *parts* can be objects, object sets, states and scenarios. Every part again can consist of parts: objects and object sets can be decomposed recursively as defined above, states can be refined into statecharts, scenarios into scenariocharts (as we call them, see below). Thus, we get a hierarchical whole-part structure that allows modeling a hierarchical decomposition of a system. The decomposition is strict: an element neither can contain itself nor can it be a part of more than one composite.

Graphic representation. In order to exploit the power of hierarchical decomposition, we allow the modelers to represent an ADORA model on any level of abstraction, from a very high-level view of the complete system down to very detailed views of its parts. We achieve this property by representing ADORA objects, object sets, scenarios and states by nested boxes (see Fig. 1 and 2). The modeler can freely choose between drawing few diagrams with deep nesting and more diagrams with little nesting. In order to distinguish expanded and non-expanded elements in a diagram, we append three dots to the name of every element having parts that are not or only partially drawn on that diagram.

Types. Frequently, different objects have the same inner structure, but are embedded in different parts of a system. In the heating system for example, the boiler control panel and the room control panels both might have a display with the same properties. In these situations, it would be cumbersome to define the properties individually for every object. Instead, ADORA offers a type construct. An ADORA type defines

- the attributes and operations of all objects/object sets of this type
- a structural interface, that means, information required from or provided to the environment of any object/object set of this type. This facility can be used to express *contracts*.

A type neither defines the relationships to other objects/object sets nor the embedding of the objects of that type. Types can be organized in a subtype hierarchy.

An object can have a type name appended to its name (for example, RoomDisplay: Display in Fig. 1). In this case, the object is of that type and the type is separately defined in textual form. Otherwise, there is no other object of the same type in the model and the type information is included in the definition of the object. Fig. 6 shows an example of such an object definition.

```
propertydef goal numbered Hyperstring constraints unique;
propertydef created Date;
propertydef note Hyperstring;

object HeatingControlSystem...
goal 1 "Provide a comfortable control for the heating of a building with several
rooms."
created 1999-11-04
note "Constraints have yet to be discussed and added."
end HeatingControlSystem.
```

Figure 3. Definition and use of standardized properties

Standardized properties. In order to adapt ADORA in a flexible, yet controlled way to the needs of different projects, application domains or persons, we provide so called *standardized properties*. An ADORA standardized property is a typed construct consisting of a header and a body. Fig. 3 shows the type definitions for the properties goal, created and note and the application of these properties in the specification of the object HeatingControlSystem. As name and structure of the properties are user-definable, we get the required flexibility. On the other hand, typing ensures that a tool nevertheless can check the properties and support searching, hyperlinking and cross-referencing.

3.2 The Structural View

The structural view combines the base view with directed relationships between objects. Whenever an object A references an information in another object B (and B is not a part of A or vice-versa) then there must be a relationship from A to B. Referencing an information means that A

- accesses a public attribute of B,
- invokes an operation of B,
- sends an event to B or receives one from B.

Every relationship has a name and a cardinality (in the direction of the relationship). Bi-directional relationships are modeled by two names and cardinalities. Relationships are graphically represented by lines between the linked objects/object sets. An arrow preceding the name indicates the direction of the relationship (Fig. 2).

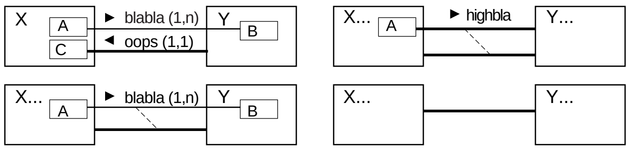


Figure 4. Four static views of the same model on different levels of abstraction

Static relationships must reflect the hierarchical structure of the model. Let objects A and B be linked by a relationship. If A is contained in another object X and B in an object Y, then the relationship A->B implies abstract relationships X->Y, A->Y and X->B. Whenever we draw a diagram that hides A, B or both, the next higher abstract relationship must be drawn. Abstract relationships are drawn as thick lines. They can, but need not be named. In case of partially expanded objects, we sometimes have to draw both a concrete and a corresponding abstract relationship. In this case, we indicate the correspondence by a dashed hairline (Fig. 4). In the view shown in Fig. 2, we have some examples. All relationships from BoilerControl to other objects are abstract ones because their origins within BoilerControl are hidden in this view. The relationships readTemp from Controller to RoomTempSensor and controlValve from Controller to RadiatorValve are elementary relationships. Hence they are drawn with thin lines. If we had chosen a view that hides the contents of RoomControl, we had drawn two abstract relationships from RoomControl to RoomTempSensor and to RadiatorValve, respectively.

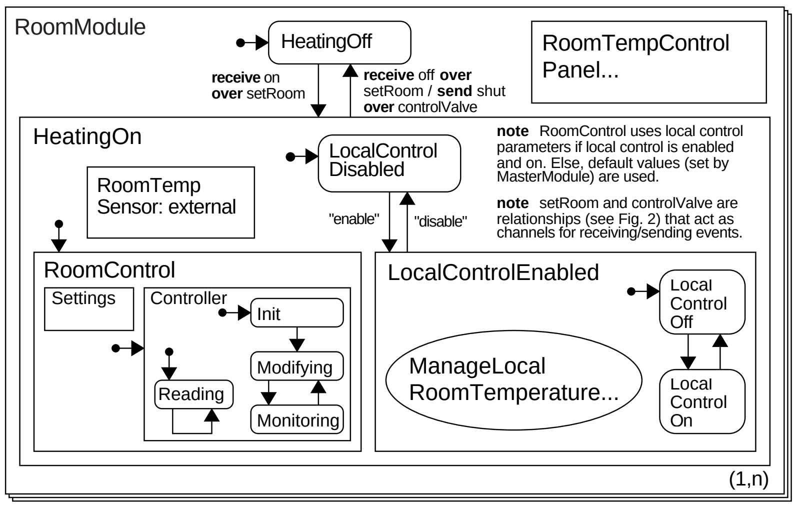
3.3 The Behavioral View

Combining objects and states. For modeling behavior, ADORA combines the object hierarchy with a statechart-like state machine hierarchy [10][11]. Every object represents an abstract state that can be refined by the objects and/or the states that an object contains. This is completely analogous to the refinement hierarchy in statecharts [12] and can be given analogous semantics for state transitions. We distinguish pure states (represented graphically by rounded rectangles) and objects with state (see Fig. 5). Pure states are either elementary or are refined by a pure statechart. Objects with state additionally have properties and/or parts other than states.

We do not explicitly separate parallel states/state machines as it is done in statecharts. Instead, objects and states that are part of the same object and have no state transitions between each other are considered to be parallel states. Objects that neither are the destination of a state transition nor are designated as initial abstract states are considered to have no explicitly modeled state.

By embedding the behavior model into the object decomposition hierarchy, we can easily model behavior on all levels of abstraction. On a high level, objects and states may represent abstract concepts like operational modes (off, startup, operating...). On the level of elementary objects, states and transitions model object life cycles.

State transitions. Triggering events and triggered actions or events either can be written in the traditional way as an adornment of the state transition arrows in the diagrams, or they can be expressed with transition tables [16]. For large systems with complex transition conditions the latter notation is more or less mandatory in order to keep the model readable. Depending on the degree of required precision, state transition expressions can be formulated textually,


RoomModule
HeatingOff
receive on over setRoom
receive off over setRoom / send shut over controlValve
RoomTempControl Panel...
HeatingOn
LocalControl Disabled
note RoomControl uses local control parameters if local control is enabled and on. Else, default values (set by MasterModule) are used.
note setRoom and controlValve are relationships (see Fig. 2) that act as channels for receiving/sending events.
RoomTemp Sensor: external
"enable"
"disable"
RoomControl
Settings
Controller
Init
Modifying
Reading
Monitoring
LocalControlEnabled
Local Control Off
Local Control On
ManageLocal RoomTemperature...
(1,n)

State Transition Tables for Controller

Init, Monitoring $\longrightarrow$ Modifying

IN LocalControlEnabled	Y	•	Y	•
IN LocalControlEnabled.LocalControlOn	Y	N	Y	N
ActualTemp > Settings.CurrentTemp(now)	N	•	Y	•
ActualTemp < Settings.CurrentTemp(now)	Y	•	N	•
ActualTemp > Settings.DefaultTemp(now)	•	N	•	Y
ActualTemp < Settings.DefaultTemp(now)	•	Y	•	N
send open **over** controlValve	✓	✓		
send close **over** controlValve			✓	✓

Modifying $\longrightarrow$ Monitoring

180 s IN Modifying	Y

Reading $\longrightarrow$ Reading

10 s IN Reading	Y
self.ReadSensorValue	✓

Figure 5. A partial ADORA model of the heating control system; base view and behavior view

formally, or with a combination of both. Fig. 5 shows the graphic representation of a behavior view with some of the variants described above. When the system is started, then for all members of the object set RoomModule the initial state HeatingOff is entered . The transition to the object HeatingOn is specified formally. It is taken when the event on is received over the relationship setRoom (cf. Fig. 2). If this transition is taken, the state LocalControlDisabled and the object RoomControl are entered concurrently. Within RoomControl, the object Controller is entered and within Controller the parallel states Init and Reading. This is equivalent to the rules that we have for statecharts. The state transitions between LocalControlDisabled and LocalControlEnabled are specified informally with a text only. The transitions within Controller are specified in tabular form.

Timing and event propagation. In ADORA we use the quasi-synchronous timing and event propagation semantics defined in [11], where state transitions take time, but the time interval is infinitesimally short and no external event can occur prior to the end of the interval. This is similar to the usual synchronous timing semantics, but avoids counter-intuitive behavior in some cases.

In contrast to usual statecharts and other than in [11] we do not broadcast events in ADORA. Instead, events have to be explicitly sent and received. We do so in order to avoid global propagation of local events.

3.4 The Functional View

In the functional view we define the properties of an object or object set (attributes, operations...) that have not already been defined by the object's type. When there is only one object of a certain type, the complete type information is embedded in the object definition. The functional view is not combined with other views; it is always represented separately in textual form.

```
object Settings
part of RoomControl
provides   Actual Temp;
           "Operations to inspect / manipulate control intervals (consisting of
           start time and desired temperature), both default and user-defined"
requires   //nothing
type
   TempIntervals is list of TempInterval;
   TempInterval is structure of (start: Time, temp: Temperature)
attribute                                  //public attributes
   ActualTemp: Temperature
var                                        //private attributes
   DefaultIntervals: TempIntervals;        //default temperature settings
   UserSetIntervals: TempIntervals         //user-defined temperature settings
syncoperation CurrentTemp (t: Time): Temperature
   pre     1 ≤ t ≤ 24*60    //minutes
   post    with usi = UserSetIntervals;
           CurrentTemp = usi[i].temp and usi[i].start ≤ t and
           not exists j • (usi[i].start < usi[j].start ≤ t)
end CurrentTemp;
syncoperation DefaultTemp (t:Time): Temperature
   "Same as CurrentTemp, but returns current default value"
end DefaultTemp;
operation RevertToDefault
   post    UserSetIntervals' = DefaultIntervals
end RevertToDefault;
"Settings must also provide operations for setting and deleting intervals and for
browsing the currently defined intervals."
end Settings.
```

Figure 6. Functional view of the object Settings (cf. Fig. 5)

Fig. 6 shows a small example. A syncoperation is assumed to execute synchronously, i.e. its execution takes no time. This is advantageous when using such an operation in a state transition. “Normal” operations are assumed to execute asynchronously and take time. As the example shows, definitions can vary in their degree of formality. The operations CurrentTemp and RevertToDefault are specified formally. DefaultTemp is specified semiformally, having a formal signature but informally described semantics. An informal text points to operations that are not (yet) defined at all. The formal elements of the notation are inspired by existing notations, in particular the language ASTRAL [7].

3.5 The User View

In the last few years, the importance of modeling systems from a user's viewpoint, using scenarios or use cases, was recognized (for example, see [5][11][13] and many others). In ADORA, we take the idea of hierarchically structured scenarios from [11] a step further and integrate the scenarios into the overall hierarchical structure of the system.

In our terminology, a scenario is an ordered set of interactions between partners, usually between a system and a set of actors external to the system. It may comprise a concrete sequence of interaction steps (instance scenario) or a set of possible interaction steps (type scenario). Hence, a use case is a type scenario in our terminology.

We view scenarios and objects to be complementary in a specification. The scenarios specify the stimuli that actors send to the system and the reactions of the system to these stimuli. However, when these reactions depend on the history of previous stimuli and reactions, that means on stored information, a precise specification of reactions with scenarios alone becomes infeasible. The objects specify the structure, functions and behavior that are needed to specify the reactions in the scenarios properly.

In the base view of an ADORA model, scenarios are represented with ovals. In the user view, we combine the base view with grey lines that link the scenario with all objects that it interacts with (Fig. 7). For example, the scenario ManageLocalRoomTemperature specifying the interaction between the actor User and the system is localized within the object LocalControlEnabled. Internally, the scenario interacts with RoomTempControlPanel and with an object in HeatingOn which is hidden in this view.

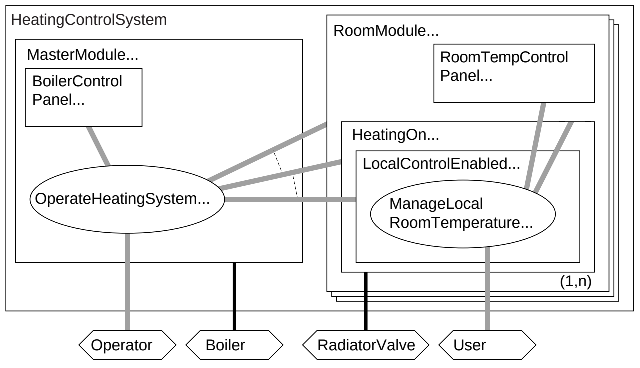


Figure 7. A user view of the heating control system

An individual scenario can be specified textually or with a statechart. In both cases, ADORA requires scenarios to have exactly one starting and one exit point. Thus, complex scenarios can be easily built from elementary ones, using the well-known sequence, alternative, iteration and concurrency constructors. In [11] we have demonstrated statechart-based integration of scenarios using these constructors. However, when integrating many scenarios, the resulting statechart becomes difficult to read. We therefore use Jackson style diagrams (with a straightforward extension to include concurrency) for visualizing scenario composition. We call these diagrams scenariocharts (Fig. 8).

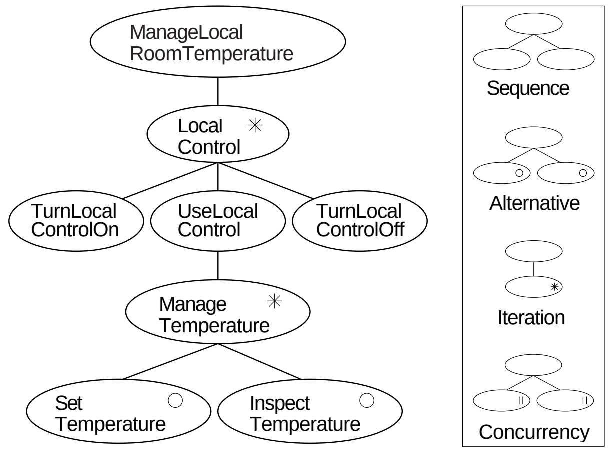


Figure 8. A scenariochart modeling the structure of the ManageLocalRoomTemperature scenario

Any scenario represented in the base view can either be elementary (and be modeled with text or with a statechart) or it can be decomposed with a scenariochart.

Thus, we have a hierarchical decomposition in the user view, too. The object hierarchy of the base view allocates high-level scenarios (like ManageLocalRoomTemperature in our heating system) to that part of the system where they take effect. The scenario hierarchy decomposes high-level scenarios into more elementary ones. As a large system has a large number of scenarios (we mean type scenarios/use cases here, not instance scenarios), this facility is very important for grouping and structuring the scenarios.

3.6 The Context View

The context view shows all actors and objects in the environment of the modeled system and their relationships with the system. Depending on the degree of abstraction selected for the system, we get a context diagram (Fig. 9) or the external context for a more detailed view of the system (Fig. 2).

In addition to external elements that are not a part of the system being specified, an ADORA model can also contain so called *external objects*. We use these to model preexisting components that are part of the system, but not part of the specification (because they already exist). External objects are treated as black boxes having a name only. In the notation, such objects are marked with the keyword external (for example, the object RoomTempSensor in Fig. 2). In any specification where COTS components will be part of the system or where existing components will be reused, modeling the embedding of these components into the system requires external objects.

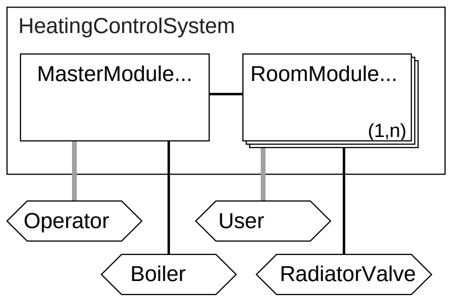


Figure 9. A context diagram of the heating control system

3.7 Modeling Constraints and Qualities

Constraints and quality requirements are typically expressed with text, even in specifications that employ graphical models for functional specifications. In traditional specifications and with UML-style graphic models we have the problem of interrelating functional and non-functional specifications and of expressing the non-functional specifications on the right level of abstraction.

In ADORA, we use two ADORA-specific features to solve this problem. (1) The decomposition hierarchy in ADORA models is used to put every non-functional requirement into its right place. It is positioned in the hierarchy according to the scope of the requirement. The requirements themselves are expressed as ADORA standardized properties. Every kind of non-functional requirement can be expressed by its own property kind, for example performance constraint, accuracy constraint, quality...).

3.8 Consistency Checking and Model Verification

Having an integrated model allows us to define stringent rules for consistency between views, for example "When an object A references information in another object B in any view and B is not a part of A or vice-versa, then there must be a relationship from A to B in the static view." A language for the formulation of consistency constraints and a compiler that translates these constraints into Java have been developed [20]. By executing this code in the ADORA tool, the tool is enabled to check or enforce these constraints. The capabilities for formal analysis and verification of an ADORA model depend on the chosen degree of formality. In the behavior view, for example, a sufficiently formal specification of state transitions allows to apply all analyses that are available for hierarchical state machines.

4 CONTEXTUAL VISUALIZATION – THE ADORA TOOL

The ADORA tool provides capabilities for editing, storing, visualizing and checking ADORA models. For the visualization of the object hierarchy, we do not simply use explosive zooming, but have developed a novel concept that is based on fisheye views. As this is the most interesting aspect of the ADORA tool, we restrict the presentation of the tool in this paper to the topic of model visualization.

4.1 General Considerations

A good visualization concept is critical both for understandability and ease-of-use of graphical models. A good concept should: (1) support *orientation* in the model by visualizing as much local detail as needed without losing the global context of the focused elements, (2) minimize the *cognitive overhead* for navigation, (3) increase *expressiveness* by including the semantics of the model in the visualization, and (4) foster its *understandability* by supporting the abstraction mechanisms of the model.

Current tools operating on hierarchical model structures normally visualize a single element with its direct successors in a single view. A few tools visualize all nodes in one view. Some tools provide scaling, map windows or overview windows to manage the complexity of big models, but most tools provide just *explosive zooming*. With explosive zooming, the global context gets lost, while the zoomed node explodes entirely in the existing or a new window. As a consequence, these tools either offer views showing global context without local detail or local detail without global context. Global context and local detail in one view are realized in very few tools; full flexibility in scaling and zooming is not offered at all. Compared with the essential modeling tasks, the cognitive overhead increases too much when models become larger [1].

Fisheye zooming produces a local detail view while preserving the global context in the same view [9][19]. The idea is to show local detail – the objects of interest to the user – in full, while displaying successively less detail for information being further away from this focus.

4.2 View Generation for ADORA Models

The principal idea of our approach towards visualization is to apply the notion of fisheye views to the visualization of ADORA models [1]. We define fisheye views with multiple foci for navigating in hierarchically clustered networks of objects. View generation is model-driven. It follows the decomposition structure and thus the abstractions of the model rather than being 'just' a flat geometric projection. The concept *integrates local detail and global context in a single view.* Such views ease orientation and navigation in the model and minimize the cognitive overhead.

As this concept allows less interesting elements to be visualized on an abstract level together with the details of elements of special interest, we have strong capabilities for supporting the inherent abstraction mechanisms in the object model that is being visualized and thus foster the expressiveness and understandability of the model.

A distinct feature of our fisheye zooming algorithm is that it works on any given layout, adjusting it incrementally and preserving it as far as possible. So a user may re-arrange a layout without losing these rearrangements when zooming.

4.3 Navigation in ADORA Models

We distinguish between two types of navigation, a physical and a logical one. *Physical navigation* is necessary when the actual visualized model exceeds the size of the available display. The typical solution is scaling, scrolling or a combination of both. Physical navigation is well known, as it is the normal way of navigation in flat models.

The really interesting kind of navigation in ADORA is *logical navigation* through the hierarchy. Logical navigation in a hierarchical structure means finding the actual position of a local element in the global context of the hierarchy, or changing the foci of visualized elements. To handle this kind of navigation adequately, we *zoom in* or *out.* Zooming-in means that more details of a deeper hierarchical level is visualized. Zooming-out means that a more abstract view of the selected elements is produced.

Figures 9 and 2 give a brief impression of our visualization concept. Fig. 9 shows the heating control system on a very high level of abstraction. The following steps lead to the view given in Fig. 2: (1) zooming-in on MasterModule, (2) zooming-in on RoomModule, (3) zooming in on HeatingOn, (4) zooming-in on LocalControlEnabled, (5) zooming in on RoomControl.[2] A more detailed example is presented in [1].

5 USING ADORA IN THE SOFTWARE DEVELOPMENT PROCESS

ADORA is an open approach that does not require a specific development process. It works with any process that

- focuses on *object-oriented models* for requirements specification and software architecture
- emphasizes the creation of a *coherent*, *consistent model* (instead of a loosely coupled set of diagrams).

ADORA supports a broad spectrum of modeling methods, for example, pure object modeling, behavior-focused modeling and scenario-focused modeling.

Equally important, ADORA is flexible what the *formality* of models is concerned. Depending on the required precision and unambiguity for modeling the problem at hand, the formality of an ADORA model can vary from mostly informal, textual specifications having the decomposition structure and standardized properties as its only formal elements up to a completely formal specification. In particular, the object hierarchy provides a framework that allows objects specified with different degrees of formality to coexist in the same model in a well-structured way.

6 VALIDATION OF ADORA

In our opinion, there are two fundamental qualities that a specification language should have:

- the language must be easy to comprehend (a specification has more readers than writers)
- the users must like it.

6.1 Goals of the Validation

Therefore, we experimentally validated the ADORA language with respect to these two qualities. We set up an experiment with the following goals.

[2] Due to limited space, Fig. 2 has been drawn manually. When constructed with our zooming algorithm, the drawing would be considerably larger. However, it would be identical both in topology and content with Fig. 2.

(i) Determine the comprehensibility of an ADORA specification both on its own and in comparison with an equivalent specification written in UML – today's standard modeling language – from the viewpoint of a reader of the specification.

(ii) Determine the acceptance of the fundamental concepts of ADORA (using abstract objects, hierarchical decomposition, integrated model...) both on its own and in comparison with UML from the viewpoint of a reader/writer of models.

6.2 Setup of the Experiment

In order to measure these goals, we set up the following experiment [3]. We wrote a partial specification of a distributed ticketing system both in ADORA and in UML. The system consists of geographically distributed vending stations where users can buy tickets for events (concerts, musicals...) that are being offered on several event servers. Vending stations and event servers shall be connected by an existing network that needs not to be specified.

Then we prepared a questionnaire consisting of two parts. In the first part, the “objective” one, we aimed at measuring the comprehensibility of an ADORA model. We created 30 questions about the contents of the specification, for example “Can a user at a point of sale terminal purchase an arbitrary number of tickets for an event in a single transaction?” 25 questions were yes/no questions; the rest were open questions. For every question, we additionally asked

- whether the answering person was sure or unsure about her or his answer,
- how difficult it was to answer the question.

In the second part, the “subjective” one, we tested the acceptance of ADORA vs. UML. We asked 14 questions about the personal opinion of the answering person concerning distinctive features of both ADORA and UML, for example “Does it make sense to use an integrated model (like ADORA) for describing all aspects of a system”?

We ran the experiment with fifteen graduate and PhD students in Computer Science who were not members of our research group. The participants were first given an introduction both to ADORA and to UML. Then we divided the participants into two groups. The members of group A answered the objective part of the questionnaire first for the ADORA specification and then for the UML specification; group B members did it vice-versa. Finally, both groups answered the subjective part of the questionnaire. In order to avoid answers being biased towards ADORA, we ensured the anonymity of the filled questionnaires.

Two participants did not finish the experiment; another person's answers could not be scored because his answers revealed insufficient base knowledge of object technology. So we finally had twelve complete sets of answers.

6.3 Some Results

Due to space limitations, we only can present some key results here. The complete results are given in [3]. As the differences between groups A and B are marginal, we consolidate the results for both groups in the results given below.

Fig. 10 shows the overall results of the first part of the questionnaire. For each model, we had a total of 360 answers (30 questions times 12 participants). For every answer, we determined whether the answer was objectively right or wrong. The answers were further subdivided into those where the answering person was sure about her or his answer and those where she or he was not. The subdivision of the columns indicates how difficult it was to answer the questions in the participants’ opinion. (For example, about 79% of the questions about the ADORA model were answered correctly and the participants were sure about their answer. For about half of these answers, the participants judged the answer to be easy to give.)

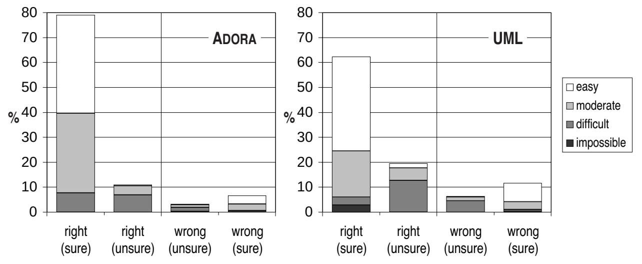


Figure 10. Comprehensibility of models. Right and wrong answers to the questions in the objective part of the questionnaire for ADORA vs. UML models. The graphics also shows how certain the participants were about their answers and how they rated the difficulty of answering.

Despite the fact that the number of participants was fairly small, these results strongly support the comprehensibility hypothesis and also show a clear trend that an ADORA specification is easier to comprehend than an UML specification. Table 1 summarizes the results of the subjective part of the questionnaire. Again, the results strongly support our hypothesis that users like the fundamental concepts of ADORA and that they prefer them to those of UML.

Table 1. Acceptance of distinct features; ADORA vs. UML[3]

Statement		strongly agree	mostly agree	mostly disagree	strongly disagree
The specification gives the reader a precise idea about the system components and relationships	ADORA	23%	62%	8%	8%
	UML	8%	46%	31%	15%
The structure of the system can be determined easily	ADORA	54%	31%	8%	8%
	UML	8%	38%	23%	31%
The specification is an appropriate basis for design and implementation	ADORA	25%	75%	0%	0%
	UML	0%	50%	33%	17%
Using an integrated model (ADORA) makes sense		42%	25%	33%	0%
Using a set of loosely coupled diagrams (UML) makes sense		8%	17%	67%	8%
Hierarchical decomposition eases description of large systems		15%	69%	15%	0%
ADORA eases focusing on parts without losing context		38%	46%	15%	0%
Decomposition in ADORA eases finding information		46%	38%	15%	0%
Integrating information from different diagrams is easy in UML		15%	15%	46%	23%
Specifying objects with their roles and context is adequate		31%	54%	15%	0%
Describing classes is sufficient		0%	15%	62%	23%

Even if we subtract some potential bias (maybe some of the participating students did not want to hurt us), we can conclude from this experiment that the ADORA language is a step into the right direction.

[3] The percentages have been rounded properly, therefore the sums in the rows sometimes yield 99% or 101%.

7 Yet another language? ADORA vs. UML

The goal of the ADORA project is not to bless mankind with another fancy modeling language. When UML became a standard, we of course investigated the option of making ADORA a variant of UML. The reason why we didn't is because ADORA and UML differ too much in their basic concepts (Table 2).

The most fundamental difference is the concept of an integrated, hierarchically decomposable model in ADORA vs. a flat, mostly non-decomposable collection of models in UML. Hierarchical structures like those shown in Fig. 1 could of course be drawn with UML package diagrams. However, as soon as we want to add properties, relationships or state transitions, the UML package notation fails, because UML packages are mere containers and only dependency links are allowed between packages.

Table 2. Comparison of basic concepts of ADORA vs. UML

ADORA	UML
Specification is based on a model of abstract objects, types are supplementary	Specification is based on a class model, object models are partial and supplementary
Specifies all aspects in one integrated model; separation of concerns achieved by decomposition and views	Uses different models for each aspect. Separates concerns by having a loosely coupled collection of models
Hierarchical decomposition of objects is the principal means for structuring and comprehending a specification	Class and object models are flat. Only packages can be decomposed hierarchically
Scenarios are tightly integrated into the specification; they can be structured and decomposed systematically	Use cases (=type scenarios) are loosely integrated with class and object models. Structuring capabilities are weak, decomposition is not possible.
Precise rules for consistency between aspect views	Nearly no consistency rules between aspect models
Conceptual visualization eases orientation and navigation in the specification and improves comprehenisiblity	UML tools provide traditional scrolling and explosive zooming only

One could argue that the UML extension mechanisms, in particular stereotypes, could be used to embed ADORA-like concepts in UML. Principally, this is true, because stereotypes in UML are powerful enough do define a completely different modeling language on top of UML [2]. However, such a redefinition of UML through stereotypes would be an abuse of the stereotype concept: it would in fact define a new language which – from a language user's viewpoint – would no longer behave like UML. Moreover, redefining stereotypes are quite difficult to support by tools and current UML tools do not support them.

A real integration of ADORA-like concepts into UML would require major changes in the UML metamodel. For example, the language elements Object and Classifier Role would have to be replaced by a uniform notion of a decomposable abstract object. According to its fundamental nature, this new language element would have to be made part of the UML core. The co-existence of object and package decompositions would be a source of problems and would require additional modifications in the metamodel.

For these reasons we pursue ADORA as an approach of its own, separately from UML. If the further development and application of ADORA provides strong evidence that certain concepts of ADORA are really better than those of UML (e.g. with respect to comprehensibility), we will eventually feed these results into the evolution process of UML.

8 CONCLUSIONS

Summary. We have presented ADORA, an approach to object-oriented modeling that is based on object modeling and hierarchical decomposition, using an integrated model. The ADORA language is intended to be used for requirements specifications and high-level, logical views of software architectures.

Code generation. ADORA is not a visual programming language. Therefore, we have not done any work towards code generation up to now. However, in principal the generation of prototypes from an ADORA model is possible. ADORA has both the structure and the language elements that are required for this task.

State of work. We have finished a first definition of the ADORA language in early 1999 [15]. In the meantime we have evolved some language concepts and have conducted an experimental validation. The ADORA tool is still in the proof-of-concept phase. We have a prototype demonstrating that the zooming algorithm, which is the basis of our visualization concept, works.

Future plans. The work on ADORA goes on. In the next years, we will develop a real tool prototype, exploit ADORA's potential for simulating and animating models and investigate the use of ADORA for partial and incrementally evolving specifications. Parallel to that, we want to apply ADORA in projects and evolve the language according to the experience gained.

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, etc. Springer. 249-264.
3. Berner, S., Schett, N., Xia, Y., Glinz, M. (1999). *An Experimental Validation of the ADORA Language*. Technical Report 1999.07, University of Zurich. http://www.ifi.unizh.ch/groups/req/ftp/papers/ADORA_validation.pdf
4. Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*, 2nd ed. Redwood City, Ca.: Benjamin/Cummings.
5. Carroll, J.M. (ed.)(1995). *Scenario-Based Design*. New York: John Wiley & Sons.
6. Coad, P., Yourdon E. (1991). *Object-Oriented Analysis*. Englewood Cliffs, N. J.: Prentice Hall.
7. Coen-Porisini, A., Ghezzi, C., Kemmerer, R.A. (1997). Specification of Realtime Systems Using ASTRAL. *IEEE Transactions on Software Engineering* **23**, 9 (Sept. 1997). 704-736.

8. Firesmith, D., Henderson-Sellers, B. H., Graham, I., Page-Jones, M. (1998). *Open Modeling Language (OML) – Reference Manual*. SIGS reference library series. Cambridge, etc.: Cambridge University Press.

9. Furnas, G. W. (1986). Generalized fisheye views. *Proc. ACM CHI 86 Conference on Human Factors in Computing Systems.* Boston, Mass. 16-23.

10. Glinz, M. (1993). Hierarchische Verhaltensbeschreibung in objektorientierten Systemmodellen – eine Grundlage für modellbasiertes Prototyping. [Hierarchical Description of Behavior in Object-Oriented System Models – A Foundation for Model-Based Prototyping (in German)] In Züllighoven, H. et al. (eds.): *Requirements Engineering '93: Prototyping*. Stuttgart: Teubner. 175-192.

11. Glinz, M. (1995). An Integrated Formal Model of Scenarios Based on Statecharts. In Schäfer, W. and Botella, P. (eds.): *Software Engineering – ESEC '95*. Lecture Notes in Computer Science 989, Berlin, etc.: Springer. 254-271.

12. Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Sci. Computer Program*. **8** (1987). 231-274.

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

15. Joos, S., Berner, S., Arnold, M., Glinz, M. (1997). Hierarchische Zerlegung in objektorientierten Spezifikationsmodellen [Hierarchical Decomposition in Object-Oriented Specification Models (in German)]. *Softwaretechnik-Trends*, **17**, 1 (Feb. 1997), 29-37.

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. Leveson, N.G., Heimdahl, M.P.E., Reese, J.D. (1999). Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In Nierstrasz, O. and Lemoine, M. (eds.): *Software Engineering – ESEC/FSE'99.* Lecture Notes in Computer Science 1687, Berlin, etc.: Springer. 127-145.

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

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

19. Schaffer, D., et al. (1996). Navigating Hierarchically Clustered Networks through Fisheye and Full-Zoom Methods. *ACM Transactions on CHI* **3**, 2; (Jun. 1996). 162-188.

20. Schett, N. (1998). *Konzeption und Realisierung einer Notation zur Formulierung von Integritätsbedingungen für ADORA.Modelle.* [A notation for integrity constraints in ADORA models – Concept and implementation (in German)]. Diploma Thesis, Univ. of Zurich.

21. Selic, B., Gullekson, G., Ward, P. T. (1994). *Real-Time Object-Oriented Modeling*. New York: John Wiley & Sons.

22. Wirfs-Brock, R., Wilkerson, B., Wiener, L. (1993). *Designing Object-Oriented Software*. Englewood Cliffs, N. J.: Prentice Hall.