PERGAMON

Information Systems 27 (2002) 425–444

www.elsevier.com/locate/infosys

Object-oriented modeling with ADORA

Martin Glinz[a,*], Stefan Berner[b], Stefan Joos[c]

[a] *Institut für Informatik, Universität Zürich, Winterthurerstrasse 190, CH-8057 Zurich, Switzerland*
[b] *FJA, Zollikerstrasse 183, CH-8008 Zurich, Switzerland*
[c] *Robert Bosch GmbH, Postfach 30 02 20, D-70469 Stuttgart, Germany*

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. © 2002 Elsevier Science Ltd. All rights reserved.

Keywords: Object-oriented modeling; Specification; Requirements engineering; ADORA; Object; Scenario; Scenario integration

1. Introduction

Modeling requirements and architectural design has a long tradition, rooted in simple block diagrams back in the early days of computing. Between 1970 and 1980, the first systematic modeling techniques appeared, for example the entity-relationship approach [1] and structured analysis [2]. The entity-relationship approach had a tremendous influence. In particular, it led to the so-called object-oriented analysis and design methods around 1990, for example OOA/OOD [3], OMT [4] or OOAD [5]. However, these approaches had severe deficiencies that were mainly due to two problems: no systematic decomposition of models and the inability to model dynamic and behavioral aspects of a system. Both problems are inherited from the entity-relationship approach, which also has no decomposition and—by its nature—does not model dynamic and behavioral aspects. On the other hand, structured analysis, which had the features lacking in the OO-approaches, had other, equivalently severe deficiencies, in particular the paradigm mismatch between analysis and design, missing locality of data definitions, only partial information hiding and no types [6]. Jacobson [7] tried to overcome some defects of the object-oriented approaches by introducing the notion of use cases. OML [8] was another attempt to create a better object modeling language. The beginning of our own work on object-oriented specification [6,9] was also motivated by the weaknesses of structured analysis

*Corresponding author.
E-mail address: glinz@ifi.unizh.ch (M. Glinz).
URL: http://www.ifi.unizh.ch/~glinz.

and the problems of the early object modeling approaches.

UML [10] was created with the goals of unifying the best features of different existing languages and of creating an industry standard. In achieving the last of these goals, UML was stunningly successful. However, this success was no reason for us to stop our work or redirect it towards UML, because also with UML (and to a minor extent, with OML), several critical 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 [11]. Thus, UML is definitely not the ultimate answer to the problem of creating a good language for modeling requirements and architecture.

We are developing an object-oriented modeling method for software that aims at overcoming some of the major problems mentioned above. We call our approach ADORA which is an acronym standing for Analysis and Description Of Requirements and Architecture [12,13]. 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.

However, as ADORA models are object-oriented, we can implement a smooth transition from an ADORA architecture model to detailed design and code written in an object-oriented programming language.

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 (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 concept that *visualizes models in context* according to their logical structure,
- support for a flexible, incremental 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, consisting of one master module serving the operator and many room modules.

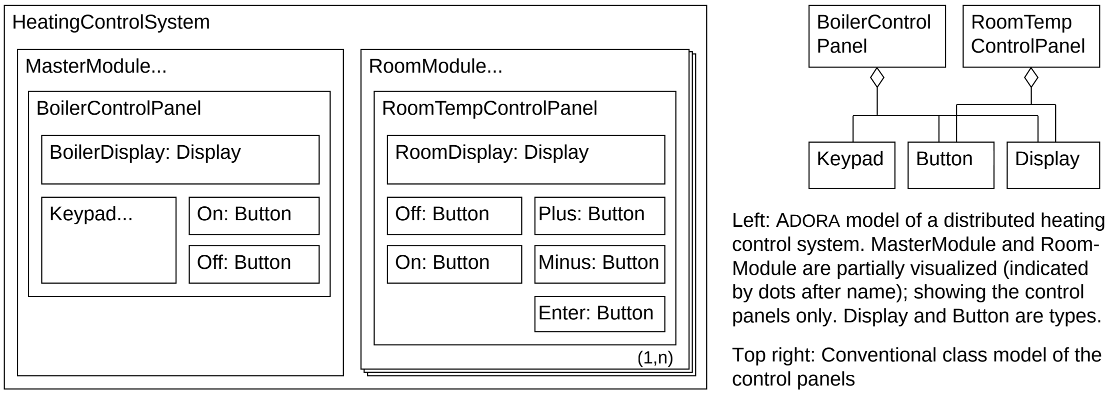
However, ADORA is not only applicable for the specification of industrial control systems. It can as well be used for the specification of information systems, in particular distributed ones. For example, in an experimental validation of the ADORA language (see Section 6), we have modeled a distributed ticket information and vending system.

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 then presenting the language and the visualization concept of the tool. Section 5 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 and rationale of the ADORA approach

ADORA is based on five principal ideas:

- working with abstract objects (instead of classes),



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

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

- structuring the system being modeled with hierarchical decomposition,
- using an integrated model (instead of collections of models),
- allowing users to express different parts of a specification with a 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.

2.1. Abstract objects instead of classes

When we started the ADORA project, all existing object-oriented modeling methods used class diagrams as the cornerstone of their model. However, class models are inappropriate when more than one object of the same class and/or collaboration between objects has to be modeled [11,14]. 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 no longer works for class models [14]. Wirfs-Brock [15] 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 also discovered the benefits of modeling with abstract objects. 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 a UML specification.

2.2. Hierarchical decomposition

Every large specification must be decomposed in some way in order to make it manageable and

[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 ([10] pp. 29, 196 and 530).

comprehensible. 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

- 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,
- 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, neither way satisfies 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 [16] can decompose a system in a systematic way. However, as ROOM is also based on classes, the components are not 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 so 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.

2.3. 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, etc.) 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 shown 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.

So the complete model is basically an abstract one—it is almost never drawn in a diagram. The concrete diagrams typically illustrate certain aspects of certain parts of a model in their hierarchical context. However, since every concrete diagram is a view of an integrated model of the complete system, we can build strong consistency and completeness rules into the language and build powerful tools for checking and maintaining them. Readability of diagrams is achieved by selecting the right level of abstraction, by restricting the number of aspects being viewed together, and by splitting complex diagrams into an abstract overview diagram and a number of part diagrams. For example, if Fig. 2 is perceived to be too complex, it can be split into an overview diagram (Fig. 9) and two part diagrams, one for MasterModule and one for RoomModule.

2.4. 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.

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

2.5. Contextual visualization

Current modeling languages either lack capabilities for system decomposition or they visualize decompositions in an explosive zoom style: the composites and their parts are drawn as separate diagrams. Hence, a diagram gives no information about the context that the presented model elements are embedded in. In ADORA, we use a fisheye view concept for visualizing a component in its surrounding hierarchical context, thereby simplifying browsing through a set of diagrams and improving comprehensibility [12,17]. This technique allows the construction of tools that support 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

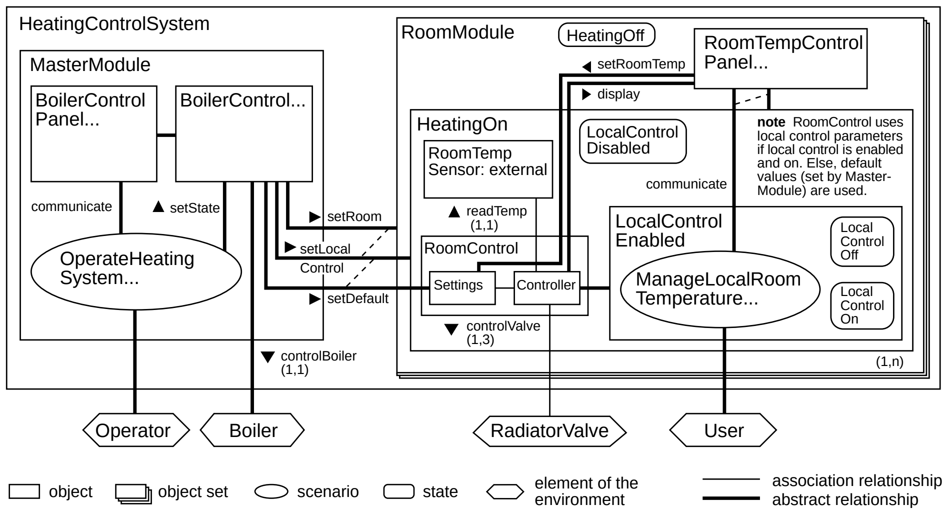


Fig. 2. An ADORA view of the heating system: base view combined with structural view and context view.

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 of 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 two-fold 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 the modeling of a hierarchical decomposition of a system. The decomposition is strict: an element can neither contain itself nor can it be a part of more than one composite. We stick to a strict decomposition due to its inherent simplicity and elegance. Commonalities between objects in different positions of a decomposition hierarchy can be modeled by assigning them the same type (see the paragraph on types below and Fig. 1).

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 Figs 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 defines neither 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.

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 the 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 the 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 information about relationships between objects. Relationships in ADORA express more than classic associations do: they model all kinds of information flow between objects.

Associations. Whenever an object A accesses a public attribute of another object B or invokes an operation of B (and B is not a part of A or vice-versa) we model a directed *association* relationship from A to B. The association has a name and may have a cardinality. In contrast to UML and other object or entity-relationship modeling languages, ADORA associations are always directed binary relationships. Modeling with directed associations has several advantages: it helps decouple objects, supports information hiding and enables modeling of contracts between objects. For example, in a phone directory system one may want to associate

```
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 2001-10-15
note "Constraints have yet to be discussed and added."
end HeatingControlSystem.
```

Fig. 3. Definition and use of standardized properties.

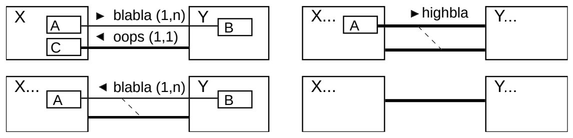


Fig. 4. Four static views of the same model on different levels of abstraction.

persons with phone numbers, but—for privacy reasons—not vice-versa. Bi-directional relationships can be modeled by two directed ones with corresponding names. Associations are graphically represented by thin lines between the associated objects. An arrowhead preceding the name of the association indicates the direction (cf. Figs. 2 and 4).

Abstract relationships. Relationships in ADORA must reflect the hierarchical structure of the model. That means, a relationship between two objects implies relationships on all hierarchical levels lying above these objects. Otherwise, the model would become inconsistent concerning information flow between objects.

We call these implied higher-level relationships *abstract relationships*. Fig. 4 illustrates the concept of abstract relationships by examples: let objects A and B linked by an association relationship. If A is contained in another object X and B in an object Y, then the relationship $A \rightarrow B$ implies abstract relationships $X \rightarrow Y$, $A \rightarrow Y$ and $X \rightarrow B$. 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 have names and cardinalities.[2] In the case of partially expanded objects, we sometimes have to draw both a relationship and a corresponding higher-level abstract relationship. In this case, we indicate the correspondence by a dashed hairline (Fig. 4).

Modeling information flow. Besides associations, information flow between two objects A and B may also stem from A sending an event to B or

[2] Cardinalities for abstract relationships can easily lead to inconsistencies. The cardinality of an abstract relationship r must not be more restrictive than the least restrictive cardinality of the relationships that r is derived from.

from A referencing names defined within B. The latter typically happens when a scenario interacts with data objects in the model. These two kinds of information flow are also modeled with abstract relationships.

Thus, the structural view models the hierarchical structure and all relationships caused by referencing, accessing or transmitting information.

Direction of abstract relationships. An abstract relationship has a direction, if and only if all its underlying lower level relationships have the same direction. So if, for example, we have two high-level subsystems S_1 and S_2 that are solely connected by a directed abstract relationship from S_1 to S_2, then we know that S_2 is independent of S_1 and that S_1 is built upon S_2.

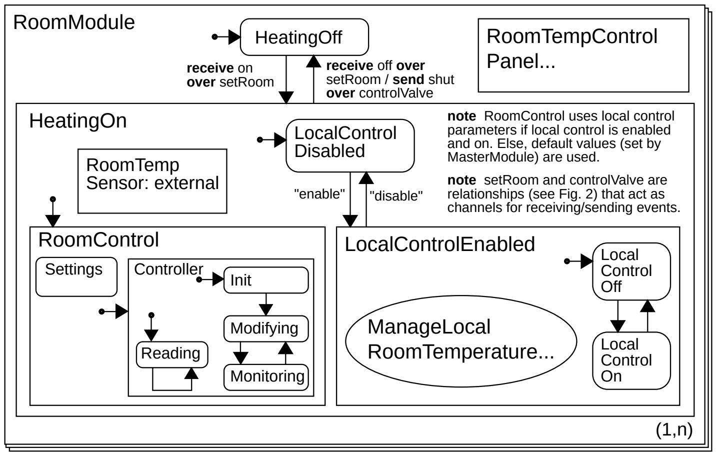
In the view shown in Fig. 2, we have some examples of Adora relationships. All relationships from `BoilerControl` to other objects are abstract ones because their origins within `BoilerControl` are hidden in this view. The relationships `readTemp` from `Controller` to `RoomTempSensor` and `controlValve` from `Controller` to `RadiatorValve` are associations. 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

The behavioral view combines the base view with a model of dynamic system behavior.

In most existing modeling languages (both general-purpose such as UML and behavior-specific such as Statemate [18]), the behavior model and the class or activity model are modeled and kept separately. In languages supporting hierarchical structure, the behavioral hierarchy and the hierarchy of activities or classes are also modeled separately [19]. In Adora we take a different approach, integrating the object and the behavior model.

Combining objects and states. Adora combines the object hierarchy with a statechart-based state machine hierarchy in a single hierarchical structure [9,20]. 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 [21] 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.



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.

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	✓

Fig. 5. A partial Adora model of the heating control system; base view and behavior view.

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 are neither 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, etc.). On the level of elementary objects, states and transitions model object life cycles.

Timing and event propagation. Harel's semantics of statecharts [21] is based on event broadcasting and synchronous timing. While this is quite straightforward for small, simple models, it has serious drawbacks for complex models. Event broadcasting breaks the principles of localizing effects and of information hiding. Hence, modifying and maintaining models becomes hard. Synchronous timing can have counter-intuitive effects [20] and is inappropriate for modeling problems that are asynchronous by nature.

Therefore, we use a different approach to timing and event propagation in ADORA, building on our own previous work [20] and on the work by Leveson and Heimdahl [22,23]. An event is broadcast only within the object where the event originates. In order to make an event available in another object, it must be explicitly sent by the generating object and received by the target object. By default, both broadcasting and sending events use the quasi-synchronous timing semantics defined in [20], 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 its problematic and often counter-intuitive effects.

When explicitly sending an event from one object to another, we may also specify the transmission to be *asynchronous*. In this case, the event leaves the time envelope of quasi-synchronicity and arrives sometimes later at its destination. Thus, we can model systems that operate synchronously within subsystems but asynchronously between subsystems, a situation that frequently occurs in distributed systems.

Notation of state transitions. The 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 [23]. While the traditional way is easy and straightforward for small models with simple transition conditions, it becomes unmanageable when the triggering event is a complex combination of conditions and when there are many actions to be taken. In this situation, the tabular notation is more or less mandatory in order to keep the model readable.

Depending on the degree of precision required, state transition expressions can be formulated textually, formally, or by 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 text only. The transitions within `Controller` are specified in tabular form.

3.4. The functional view

In the functional view we define the properties of an object or object set (attributes, operations, etc.) 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.

The language for expressing the functional view is inspired by existing notations, in particular the language ASTRAL [24]. We have chosen a pre-/postcondition style. The syntax supports the expression of information on any level of formality, ranging from informal text to logic formulae.

Fig. 6 shows a small example. A syncoperation is assumed to execute quasi-synchronously, i.e. its execution takes an infinitesimally short 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.

```
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;
         //returns desired temperature at current time t
         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.
```

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

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 [25,7,20] and many others). In ADORA, we take the idea of hierarchically structured scenarios from [20] 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 consider 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, i.e. on stored information, a precise specification of reactions with scenarios alone becomes unfeasible. 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 by ovals. In the user view, we add to the base view the actors in the system environment that the scenarios interact with and all those abstract relationships that model interactions between scenarios and objects. Frequently the user view is combined with the context view (see below) which adds those environmental elements that interact with objects, but not with scenarios (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`. `RadiatorValve` is an environmental object that is controlled from an object inside `HeatingOn`. It is part of the context view and would not be visible in a pure user view.

An individual scenario can be specified textually or by a statechart [26]. In both cases, ADORA requires scenarios to have exactly one starting and one regular exit point. Other exit points may be used for exception handling purposes only. Given this structure, complex scenarios can easily be built from elementary ones, using the well-known sequence, alternative, iteration and concurrency constructors. In [20,26] 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).

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 `Manage-LocalRoomTemperature` 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.

Note that the ADORA user view is a logical view of user–system interaction only; it does not include the design of the user interface.

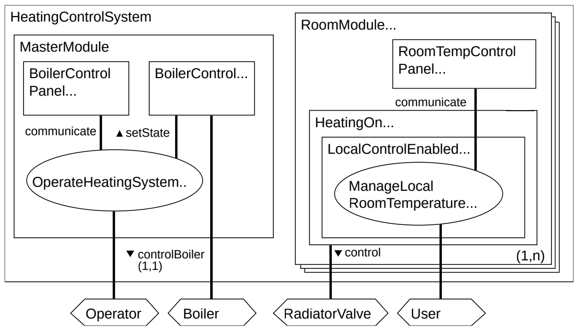


Fig. 7. A combined user and context view of the heating control system.

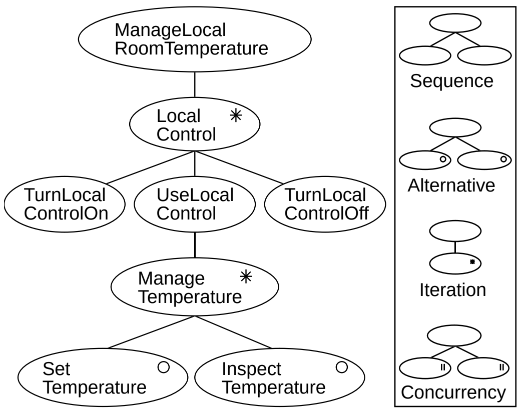


Fig. 8. A scenariochart modeling the structure of the Manage-LocalRoomTemperature scenario.

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). ADORA allows modeling of a rich context, i.e. we can model relationships between actors or objects in the system environment. This is an important feature, because for properly situating a system in its environment we frequently must know not only about the system-environment interactions, but also about interactions between actors in the environment.

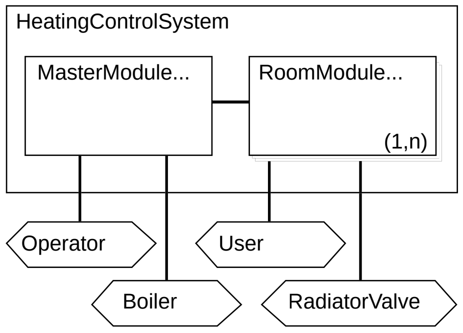


Fig. 9. A context diagram of the heating control system (base view combined with context view and structural view).

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.

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. (2) 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`, etc.).

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 has been developed [27]. 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 the application of all analyses that are available for hierarchical state machines.

4. Contextual visualization—the ADORA tool

We have developed a concept for visualizing hierarchical object models with a tool and are developing a prototype of this tool.

The ADORA tool shall provide capabilities for editing, storing, visualizing and checking ADORA models. The distinguishing feature of the ADORA tool is the way we visualize the object hierarchy: we do not simply use explosive zooming, but have developed a novel concept that is based on fisheye views. In this paper, we restrict the presentation of the ADORA tool to this important aspect.

4.1. General considerations

A good visualization concept is critical both for understandability and ease-of-use of graphical models. A good concept should:

- support *orientation* in the model by visualizing as much local detail as needed without losing the global context of the focused elements,
- minimize the *cognitive overhead* for navigation,
- increase *expressiveness* by including the semantics of the model in the visualization,
- 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 [12].

For the visualization of Adora models we adapt an alternative concept for viewing graphic structures that was developed nearly 20 years ago by Furnas [28]: the so-called *fisheye views*. Furnas uses the analogy of a fisheye (which has a very wide angle lens) for a view concept that shows full detail in a focal point, while displaying information being further away from this focus in successively less detail. Zooming in fisheye views basically works by moving the angle and the focus of the display.

The principal idea behind fisheye view visualization is to display local detail and global context together in a well-balanced way. Local detail is required for getting all the relevant information that a user is currently focused on, while global context is needed for situating the current focus in the general context of the information being viewed (which in turn is a prerequisite for quick and easy orientation and navigation).

4.2. View generation for Adora models

For Adora, we modified the original fisheye view concepts in order to make them suitable for our purposes [12,17]. Firstly, we do not generate views by geometric projection of a large flat model (the way that a real fisheye lens would work). Instead we generate so-called logical fisheye views that are based on the hierarchical structure of an Adora model and its inherent abstraction capabilities. Secondly, we allow views with more than one focus.

Thus, we get a model-driven visualization that fully exploits the power of a hierarchically structured model and is able to integrate *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.

4.3. Navigation in Adora models

We distinguish between two types of navigation, a physical and a logical one [29]. *Physical navigation* is necessary when the view 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 are visualized. Zooming-out means that a more abstract view of the selected elements is produced.

4.4. The zooming algorithm

Our fisheye zooming algorithm 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. The principal idea of the algorithm for zooming-in is illustrated in Fig. 10. The details are beyond the scope of this paper; they can be found in [17]. Assume that $X\ldots$ is the object we want to zoom-in. The algorithm works in four steps.

(1) Determine the size of the expanded object (dotted shape of X in Fig. 10). This size depends on the size and layout of the elements

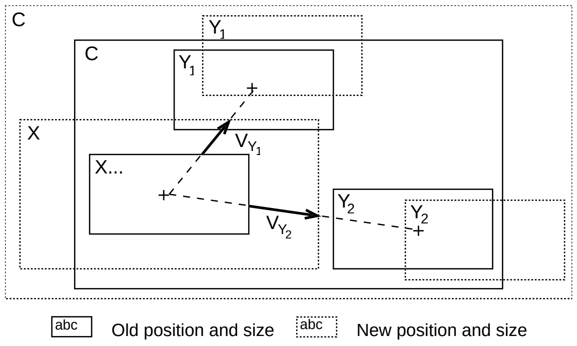


Fig. 10. The ADORA zooming algorithm.

that have to be displayed within X in the zoomed-in view.

(2) Let C be the composite object that directly[3] contains X.[4] For every object Y_j on the diagram which is contained in C and lies outside X; compute a *shift vector* V_{Yj} as follows: draw a virtual line from the center of X to the center of Y_j. The two intersection points of this line with the original and the expanded shape of X define the shift vector for Y_j. Then shift Y_j geometrically by its shift vector.
(3) Determine the new size of C (such that it geometrically surrounds the resized or shifted objects it contains). Recursively apply steps (2) and (3) of the algorithm to C.
(4) Draw the contents of X into its expanded shape.

As long as the shapes of all objects on the diagram are geometrically convex *and* the expanded shape of X is geometrically proportional to the original one, this algorithm always produces a diagram that looks similar to the original one and does not have any overlapping shapes.

The convexity constraint is no problem, because all graphic elements in ADORA are geometrically convex. However, the expanded shape is frequently not proportional to the original one. Relaxing the proportionality constraint while keeping the resulting diagram free of overlappings requires some modifications to the algorithm sketched above. The details are discussed in [17].

4.5. Example

Figs. 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`, and (5) zooming in on `RoomControl`.[5] A more detailed example is presented in [12].

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 concerning the *formality* of models. 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 (using the decomposition structure and standardized properties as the only formal elements) up to a completely formal specification. In particular, the object hierarchy provides a

[3] "Directly" means that there is no object between C and X in the decomposition hierarchy.

[4] If there is no such composite object, step (2) applies to all objects of the diagram lying outside X.

[5] In order to save 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.

framework that allows objects specified with different degrees of formality to coexist in the same model in a well-structured way.

Adora provides strong support for iterative and incremental development processes. Such processes typically start with a high-level requirements and architecture specification and identify work packages for incremental development based on this high level structure. Work packages are then developed by specifying their detailed requirements, designing their architecture in the framework of the general architecture and implementing them on this basis.

The hierarchical structure of Adora fits such processes quite well: the high-level specifications can be expressed by high-level objects and their interrelationships. Goals, objectives and general constraints can be attached to these objects using Adora's standardized properties. Then, the structure of the model can be exploited for determining the work packages. When developing a work package, those objects of the high-level model belonging to the work package are specified in detail by decomposing and refining them.

Adora also provides strong support for multi-level systems engineering and business engineering processes.

A systems engineering process is typically characterized by a hierarchical series of requirements and design steps: after determining the principal goals (requirements), basic design decisions are taken which impose a structure on the system. In a recursive procedure, we then determine the requirements for every component of this structure, design the components, yielding subcomponents, determine the requirements of the subcomponents, etc. until we arrive at concrete hardware and software building blocks.

When using such a process, the documentation of the requirements and architecture naturally has a hierarchical structure, which can be expressed quite easily with Adora.

The Heating Control system is a typical example, where the basic structure of a single master module and one room module per room is a very high-level design decision that shapes the requirements on the lower levels.

We also find a hierarchical intertwining of requirements and design decisions in business engineering and information systems development. There we typically have three hierarchical levels in the process: the business level, the system level (information systems comprising hardware, software and people), and the software level.

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:

- 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.
- Determine the acceptance of the fundamental concepts of Adora (using abstract objects, hierarchical decomposition, integrated model, etc.) 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 conducted an experiment as follows [30]. 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, etc.) 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?" Twenty-five questions were yes/no questions; the rest were open questions. For every question, we additionally asked

- whether the person answering 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 person answering 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 Ph.D. 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. Results

We present only the key results here; the complete results are given in [30]. As the differences between groups A and B are marginal, we consolidate the results for both groups in the results given below.

Fig. 11 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 person answering 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

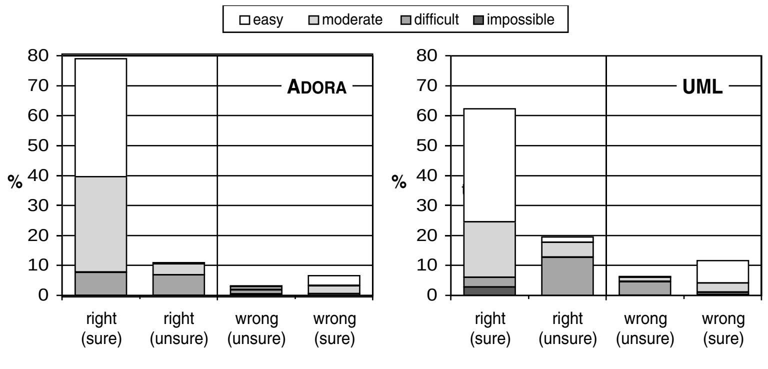


Fig. 11. Comprehensibility of models. Right and wrong answers to the questions in the objective part of the questionnaire for ADORA vs. UML models. The graphics also shows how certain the participants were about their answers and how they rated the difficulty of answering.

their answer. For about half of these answers, the participants judged the answer to be easy to give.)

An analysis of these data reveals two important results that are statistically significant at a level of 0.5%:

- The percentage of errors when reading a model was 9.9% for ADORA vs. 18.1% for UML. That means, reading ADORA models is less prone to errors than reading UML models.
- 87.8% of the participants who correctly answered questions about the ADORA model were confident about their answer. For the UML model, the corresponding figure was 76.1% only. That means, readers of an ADORA model are more confident about the correctness of their interpretation of the model than readers of a UML model.

These results strongly support the comprehensibility hypothesis and also show a clear trend that an ADORA specification is easier to comprehend than a UML specification.

Table 1 summarizes the results of the subjective part of the questionnaire. Due to the small number of answers, we did not attempt a quantitative statistical analysis here. However, a qualitative analysis of the results strongly supports our hypothesis that users like the fundamental concepts of ADORA and that they prefer them to those of UML.

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.

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 did not is because ADORA and UML differ too much in their basic concepts (Table 2).

Moreover, UML has several major problems and weaknesses when used as a requirements specification language [11]. These can be avoided in ADORA because ADORA is founded on a different conceptual basis.

Table 1
Acceptance of distinct features; ADORA vs. UML

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

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

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 comprehensibility.	UML tools provide traditional scrolling and explosive zooming only.

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.

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 to define a completely different modeling language on top of UML [31]. 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 with 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 [11]. 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 coexistence 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

8.1. 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.

8.2. Code generation

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

8.3. State of work

We completed a first definition of the ADORA language in 1999 [13]. 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.

8.4. Future plans

The work on ADORA goes on. In the years to come, we plan to 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] P.P. Chen, The entity-relationship model—toward a unified view of data, ACM Trans. Database Systems 1 (1976) 9–36.

[2] T. DeMarco, Structured Analysis and System Specification, Yourdon Press, New York, 1979.

[3] P. Coad, E. Yourdon, Object-Oriented Analysis, Prentice Hall, Englewood Cliffs, 1991.

[4] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, 1991.

[5] G. Booch, Object-Oriented Analysis and Design with Applications, 2nd ed, Benjamin/Cummings, Redwood City, 1994.

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

[7] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, Object-Oriented Software Engineering—A Use Case Driven Approach, Addison-Wesley, Reading, 1992.

[8] D. Firesmith, B.H. Henderson-Sellers, I. Graham, M. Page-Jones, Open Modeling Language (OML)—Reference Manual, SIGS Reference Library Series, Cambridge University Press, Cambridge, 1998.

[9] M. Glinz, 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: H. Züllighoven et al., (Eds.), Requirements Engineering '93: Prototyping, Teubner, Stuttgart, 1993, pp. 175–192 (in German).

[10] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, Reading, 1999.

[11] M. Glinz, Problems and deficiencies of UML as a requirements specification language, Proceedings of the Tenth International Workshop on Software Specification and Design, San Diego, 2000, pp. 11–22.

[12] S. Berner, S. Joos, M. Glinz, M. Arnold, A visualization concept for hierarchical object models, Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE-98), 1998, pp. 225–228.

[13] S. Joos, ADORA-L—Eine Modellierungssprache zur Spezifikation von Software-Anforderungen [ADORA-L—A modeling language for specifying software requirements (in German)], Ph. D. Thesis, University of Zurich, 1999.

[14] S. Joos, S. Berner, M. Arnold, M. Glinz, Hierarchische Zerlegung in objektorientierten Spezifikationsmodellen [Hierarchical Decomposition in object-oriented specification models], Softwaretechnik-Trends 17 (1) (1997) 29–37 (in German).

[15] R. Wirfs-Brock, B. Wilkerson, L. Wiener, Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, 1993.

[16] B. Selic, G. Gullekson, P.T. Ward, Real-Time Object-Oriented Modeling, Wiley, New York, 1994.

[17] S. Berner, Modellvisualisierung für die Spezifikationssprache ADORA [Model visualization for the specification language ADORA (in German)], Ph.D. Thesis, University of Zurich, 2002.

[18] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, M. Trakhtenbrot, STATEMATE: a working environment for the development of complex reactive systems, IEEE Trans. Software Eng. 16 (1990) 403–414.

[19] D. Harel, E. Gery, Executable object modeling with statecharts, IEEE Computer 30 (7) (1997) 31–42.

[20] M. Glinz, An integrated formal model of scenarios based on statecharts, in: W. Schäfer, P. Botella (Eds.), Software Engineering—ESEC '95, Lecture Notes in Computer Science, Vol. 989, Springer, Berlin, 1995, pp. 254–271.

[21] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Computer Programming 8 (1987) 231–274.
[22] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, J.D. Reese, Requirements specification for process-control systems, IEEE Trans. Software Eng. 20 (1994) 684–707.
[23] N.G. Leveson, M.P.E. Heimdahl, J.D. Reese, Designing specification languages for process control systems: lessons learned and steps to the future, in: O. Nierstrasz, M. Lemoine (Eds.), Software Engineering—ESEC/FSE'99, Lecture Notes in Computer Science, Vol. 1687, Springer, Berlin, 1999, pp. 127–145.
[24] A. Coen-Porisini, C. Ghezzi, R.A. Kemmerer, Specification of realtime systems using ASTRAL, IEEE Trans. Software Eng. 23 (1997) 704–736.
[25] J.M. Carroll (Ed.), Scenario-based Design, Wiley, New York, 1995.
[26] M. Glinz, Improving the quality of requirements with scenarios, Proceedings of the Second World Congress on Software Quality, Yokohama, 2000, pp. 55–60.
[27] N. Schett, Konzeption und Realisierung einer Notation zur Formulierung von Integritätsbedingungen für ADORA-Modelle [A notation for integrity constraints in ADORA models—Concept and implementation], Diploma Thesis, University of Zurich, 1998 (in German).
[28] G.W. Furnas, Generalized fisheye views, Proceedings of the ACM CHI 86 Conference on Human Factors in Computing Systems, Boston, 1986, pp. 16–23.
[29] D. Schaffer, et al., Navigating hierarchically clustered networks through fisheye and full-zoom methods, ACM Trans. CHI 3 (1996) 162–188.
[30] S. Berner, N. Schett, Y. Xia, M. Glinz, An experimental validation of the ADORA language, Technical Report 1999.05, University of Zurich, 1999. http://www.ifi.unizh.ch/groups/req/ftp/papers/ADORA_validation.pdf.
[31] S. Berner, M. Glinz, S. Joos, A classification of stereotypes for object-oriented modeling languages, Proceedings of the Second International Conference on the Unified Modeling Language, Fort Collins, Springer, Berlin, 1999, pp. 249–264.