Abstract dynamic modelling of complex systems

Gregor v. Bochmann
Université de Montréal

(December 1992, revised Febr. 1993)

Abstract: Most object-oriented analysis and design methods support some kind of entity-relationship modelling of the universe of discourse which defines the object types to be considered, as well as the relationships that exist among these types of objects. In addition, each object type is characterized by an interface which defines a number of actions, methods or operations. This paper introduces the concept of "abstract event" which is the joint execution of actions by object instances which are related by a relationship. This allows the modelling of communication between objects at a high level of abstraction, and the definition of the dynamic behavior of the related object types. The paper provides various examples which demonstrate the usefulness of this notion for dynamic system modelling. Combined with the notion of aggregation, which means that an object may consist of a number of components objects, the rendezvous mechanism of abstract events may also be used to define the dynamic behavior of composed objects which may be based on the behavior of its components. It is also shown that the notion of multiple inheritance can be considered a variation of composition.

1. Introduction

The modelling of complex systems is largely an art, although many systematic approaches have been defined for this purpose. One of the most important aspects of a modelling methodology is the ability to decompose a system into parts. In the context of the popular object-oriented approaches, the system parts are called objects, and usually, a system part which is considered at an abstract level of description as a single object, may be considered at a more detailed level as an aggregation of several component objects.

While much attention in the past has been put into the description of the structure of the described system in terms of objects and their interrelation, less attention has been given to the aspect of the dynamic behavior of each object. This aspect has traditionally been handled at the implementation level where the behavior may be coded in terms of procedures written in a (possibly object-oriented) programming language, in the area of communication protocols, the dynamic behavior has always been of prime attention, and several formal specification languages

have been developed for the description of complex systems, including their dynamic behavior (for an overview, see [Bochmann, 90g #275]).

Comparing various methods for describing the dynamic behavior of computer systems, there appears to be some kind of opposition between two approaches: some methods are mainly based on the events that happen (and relate these events to changes of the object states), while other methods are more based on the possible states of the objects (and describe what events are possible depending on the system state). The usual object-oriented approach to system analysis and design seems to be more based on the objects and their states, and much less on the events that may occur. We show in this paper how this view can be complemented with a view which is much more oriented towards the events that may occur in the system to be described.

The events that occur in a system are also related to the communication that occurs between the different system components, or between the described system and its environment. Three basic communication paradigm can be distinguished: (1) asynchronous message passing, (2) (remote) procedure calls, and (3) rendezvous communication which only occurs when all participating parties (which may be two or more objects) are ready for the interaction. The last paradigm seems to be the most abstract one, and is used for the definition of the "abstract event" notion defined in Section 3.

In traditional object-oriented systems, the operations, which can be called or be invoked by sending a message, are associated with a single object. They can be invoked by another object if the latter knows the identity of the former. This is a rather implementation-oriented mechanism, and we think that for abstract system descriptions, it would be better to have some other way of indicating which object communicates with which other object. In fact, sometimes so-called *use* relations are explicitely introduced during the system design. We propose a kind of generalization of this concept.

Most object-oriented analysis and design methods support some kind of entity-relationship modelling of the universe of discourse which defines the object types to be considered, as well as the relations that exist among these types of objects. We use these relations to support the communication between objects. The abstract events, as defined in Section 3, are usually associated with the instances of relations (sometimes of several relations) and involve certain operations that are defined for the objects which are related by the relation in question. An event represents therefore the occurrence of a communication between several objects and may imply state changes in these objects.

After reviewing some basic modelling concepts in Section 2, we present the notion of *abstract event* in Section 3 together with a large number of examples which are intended to demonstrate the flexibility and usability of this new concept. The following discussion relates this concepts to similar approaches described in the literature.

Section 4 deals with descriptions at several levels of details and aggregation. It also deals with multiple inheritance. This concept has been often discussed in the context of object-oriented languages, and it is not clear whether its usefulness for the description of complex systems

warrents the conceptual complexity which it introduces. We show in this paper that, through the use of abstract events, multiple inheritance may be modelled by aggregation.

2. Some basic modelling approaches

2.1. Entity-relationship modelling

Entity-relationship modelling was originally introduced as a method for designing the data model of database systems [Chen, 76 #358]. Similar concepts are included in most recent approaches to object-oriented system analysis and design (see for instance [Rumbaugh, 91 #1110]). The purpose of this approach is to identify the type of objects (also called entities) in the universe of discourse which is of interest, and also to identify the relations that exists between these objects. A simple example is given in Figure 2.1, which shows six types of objects and five types of relations between them. The relation between *Computer* and *Person* may have the meaning "belongs to" while the other relations represent aggregation, that is, the fact that a computer consists of several parts, a powersupply, a screen, a keybord and possibly several disk drives. The multiplicity annotation (in the form a pair, representing minimum and maximum number of occurrences of the relation) provides information about the existential dependencies between the different related objects. For instance, the annotation 0,1 and 0,* for the "belongs to" relation indicates that a *Computer* may belong to zero or (at most) one *Person*, and that there may be zero, one or more *Computers* that belong to a given *Person*.

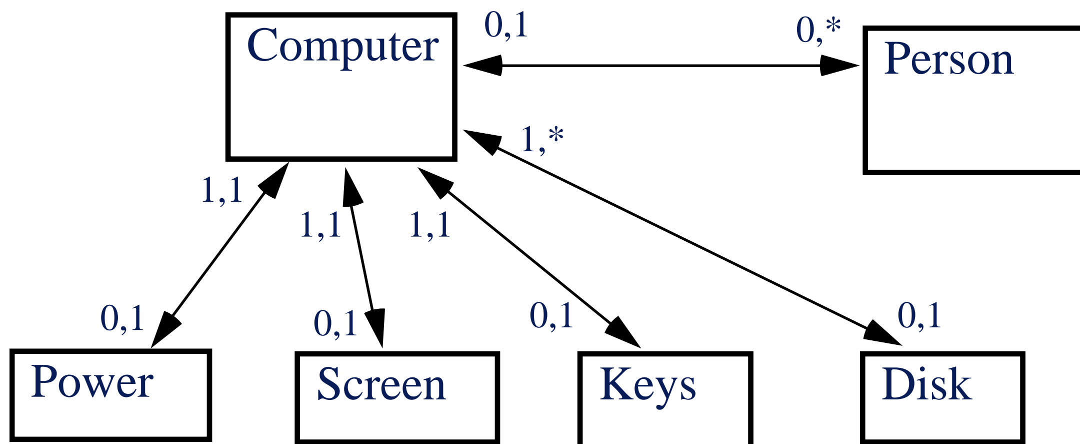


Figure 2.1: Type diagram using entity-relationship modelling

In addition to the "type diagram" shown in Figure 2.1, it is often interesting to consider a particular configuration of object instances, specially if the dynamic interactions between several objects within a system is under consideration. In the simplest case, such "instance diagrams" can be quite similar to the corresponding type diagram. An example involving the persons Fred and Tom, where Fred owns a particular computer with the serial number SN 123 is shown in Figure 2.2.

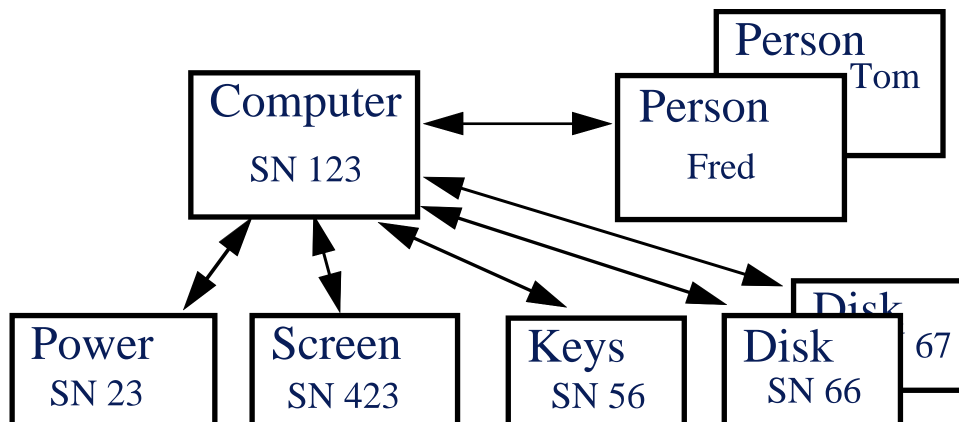


Figure 2.2: An instance diagram corresponding to the type diagram of Figure 2.1

2.2. Rendezvous communication

As mentioned in the Introduction, three basic communication paradigm can be distinguished: (1) asynchronous message passing, (2) (remote) procedure calls, and (3) rendezvous communication. A general form of rendezvous, where more than two objects (usually called processes) may participate in a given interaction is defined by the LOTOS language [Bolognesi, 87 #293]. It is characterized by the fact that each interacting object may impose specific conditions for the execution of a rendezvous interaction, and the interaction may only occur when all these conditions are satisfied. In order to avoid deadlocks due to incompatible conditions within a given system design, a given object, for a given state, often foresees several alternative rendezvous interactions or various ranges of interaction parameters. If several of these alternatives are possible in a given state of the system, the specification does not determine which of the possibilities will be realized; the system specification is nondeterministic.

It is important to note that this notion of rendezvous makes no distinction between caller and called entity; the communication is symmetric. This is in contrast to (remote) procedure calls and message passing, where the initiating object has to "know" the called object instance. We note that a combination of remote procedure call with possibilities for alternatives and conditions imposed by the called object has been defined within the Mondel specification language [Bochmann, 92p #287]. In this paper, we consider symmetric communication, based on the rendezvous concept described above.

It has been observed that rendezvous communication is more suitable for abstract description of system behavior than communication primitives based on message passing [Bochmann, 90a #270]. The possibility of cross-over of messages between two communicating objects introduce additional complications which can be avoided by rendezvous communication and which, if resolved, lead to implementation-oriented solutions that are not appropriate for a high level description.

We note that the definition of LOTOS is based on an interleaving semantics where only a single interaction may occur at a given time throughout the whole system. This interpretation is clearly not realistic, and an interpretation allowing for true parallelism between different rendezvous interactions is required, such as described in [Costa, 92 #1136]. The same issues arise in relation with Petri nets [Peterson, 77 #795] where transitions (involving several tokens -- objects) may occur in parallel if they are not in conflict with one another.

2.3. Object-based systems

Object-oriented programming and system design has become fashionable. The main feature of this approach is "information hiding" which is obtained considering objects with an interface which defines the **actions** (usually called methods or operations) that can be invoked by other objects. The implementation of these actions, in terms of an algorithm and/or internal variables is hidden. In the following, we sometimes distinguish between actions which imply state changes, and actions without state changes, which we also call **attributes**.

An example of the definition of an object type is given in Figure 2.3(a). We note that a complete definition of an object type should include, in addition to the interface which describes the actions and their parameters, also the object behavior which describes the nature of the state changes and the results returned by the invoked actions.

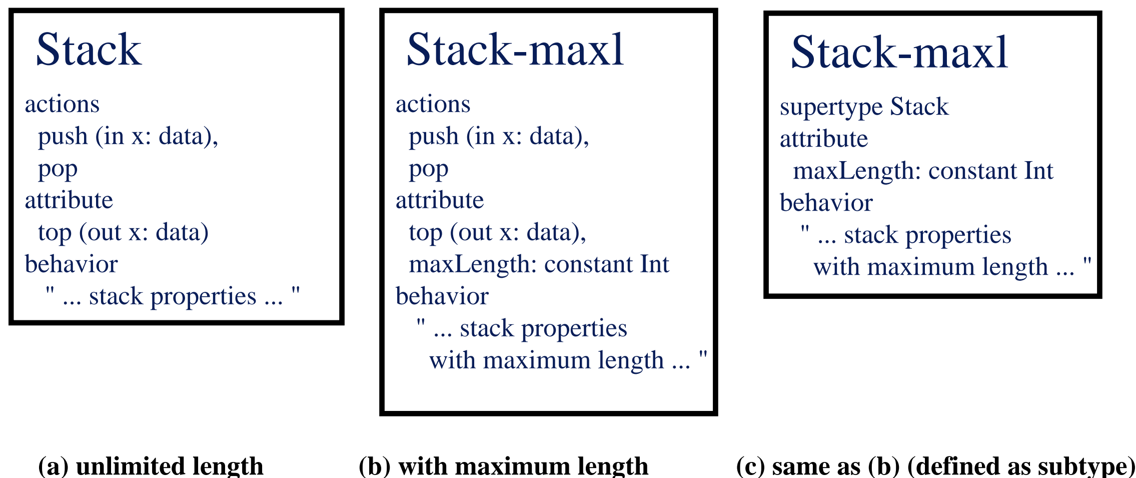


Figure 2.3: Several definitions of Stack types

2.4. Property inheritance

Another feature of object-orientation is the concept of property inheritance, also called subtyping. The idea is that a new type of object (called subtype) may be defined as a specialization of another type (called the supertype, which is already defined) by describing further properties that the new type of objects must satisfy in addition to the properties defined for the supertype. One kind of properties are additional actions or attributes that are defined for the subtype, as the maxLength attribute of the specialized stack type shown in Figures 2.3(b) and

(c). However, there may be other important properties related to the behavior of the objects; for example, a stack with maximum length will behave differently than its supertype with unlimited length for very long sequences of *push* actions.

There are different kinds of properties for which a relation between a subtype and its supertype must be defined, including compatibility of interfaces, specialized behavior, and more detailed description and implementation details. A discussion of some of these aspects is given in [Bochmann, 92a #1027]. We do not allow for "redefinition" of behavior for more specialized objects, that is, all properties specified for a given type of objects remain valid for all specialized types that inherit from the former.

In system analysis and design, property inheritance is often used for defining classification hierarchies of object types. It is usually combined with entity-relationship modelling.

3. Abstract events

3.1. The concept

We note that communication between different objects in object-oriented systems is usually done in a client / server relationship, where a calling object invokes an operation on the called object. The latter must be "known" by the former, that is, the caller has to know the object identifier of the called object. Two modes of communication may be considered: asynchronous and synchronous. In the asynchronous mode of communication, the caller sends a "message" to the called object and may proceed immediatly afterwards with its own processing. In the synchronous mode, the caller waits for the called object to return the result of the operation before proceeding with other activities; in this case, an operation has two kinds of parameters, inputs and results. In both cases, it is the calling object that makes the decision to invoke the operation.

In the following, we introduce the notion of **abstract event** which is more abstract than the common synchronous operation call described above. The following are the main differences:

(a) There is no asymmetric caller / callee relationship: It is not said which object makes the decision for the execution of an event.
(b) There may be more than two objects participating in a given event. These objects are related to one another through one or several relations (see below).
(c) Each participating object may impose certain conditions which must be satisfied when the event occurs. Each participating object may also define some local state changes that occur during the execution of the event.

The semantics of an abstract event is defined as explained through the example shown in Figure 3.1. As usual for object-based systems, each object has a certain number of defined actions (including the attributes, i.e. actions without state changes). For example, object O1 in the figure

has actions a1, a2, and a3. In addition, we assume that with each type of relation, there may be an associated rendezvous interactions between certain actions of the related objects. Such a rendezvous interaction is called an **abstract event**. Each occurrence of an abstract action is associated with an instance of the corresponding relation. For example, the relation R1 in the figure specifies rendezvous between the a1 action of an instance of an O1 object with the a4 action of an instance of a related O2 object (we use the symbol # to denote such rendezvous). The relation R2 specifies rendezvous between the actions a4 of instances of the object types O2 and O3. R3 is a ternary relation and defines a three-way rendezvous for the action a3 common to three instances of the object types O1, O3, and O4, respectively. The definition of an abstract event may introduce additional conditions that must be satisfied when an occurrence of the event is executed, as for instance for relation R2. If there are several instances of a given relation for a given object instance, such as in the case of relation R1 for an object of type O2, an event involving this relation will only rendezvous with **one** instance of the related object type (in this example, one instance of object O1).

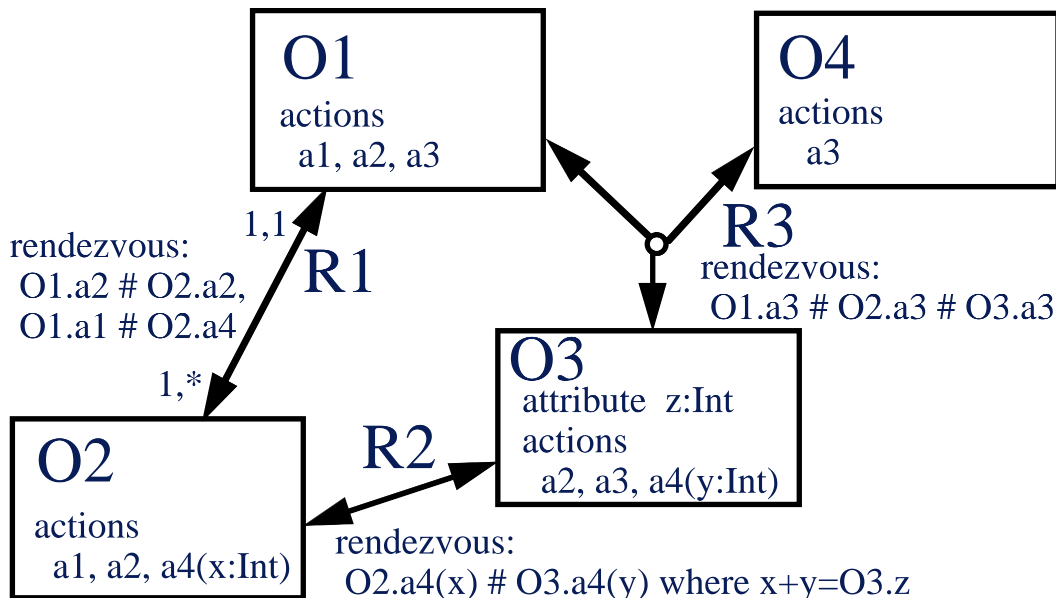


Figure 3.1: Example of objects, relations and abstract events

The fact that the action a4 of an instance of type O2 is constraint by R1 to a rendezvous with the action a1 of the related object instance of type O1, and also constraint by R2 to a rendezvous with the action a4 of an instance of type O3 implies that these three actions form a single abstract event. This is a case of "multiple rendezvous". The possible events of the system shown in Figure 3.1 are therefore the following: O1.a2 with O2.a2, O1.a1 with O2.a4 with O3.a4, O1.a3 with O3.a3 with O4.a3, O2.a1 (alone), and O3.a2 (alone). The relation R2 is an example where an additional condition for rendezvous is required; this condition involves in this case the two parameters of the two respective actions and a state attribute of one of the object instances involved.

The multiplicity annotation for relation R1 indicates that a given instance of object O1 may be related to several object instances of type O2. Each occurrence of the defined rendezvous interactions will be related to a particular instance of the relation. This means that the execution of the action a2 of an object O2, for instance, will be synchronized with the execution of the action a2 of (exactly) **one** of the related O2 object instances.

3.2. Examples

In the following we describe several more meaningful examples in order to show how the concept of abstract events may be used to describe various situations.

Client-server relationship: The following diagram shows a client object which uses two different server objects, a database for searching and updating, and a powerful processor for performing certain calculations. Not all the available operations are used by the client, as its list of actions indicates. The relations between the client and the two servers define rendezvous for all actions that are common to the two respective objects, that is, there are the abstract events *find* and *update* common to the client and the database, and the event *calc* common to the client and the processor; the action *other* of the client is not part of a rendezvous.

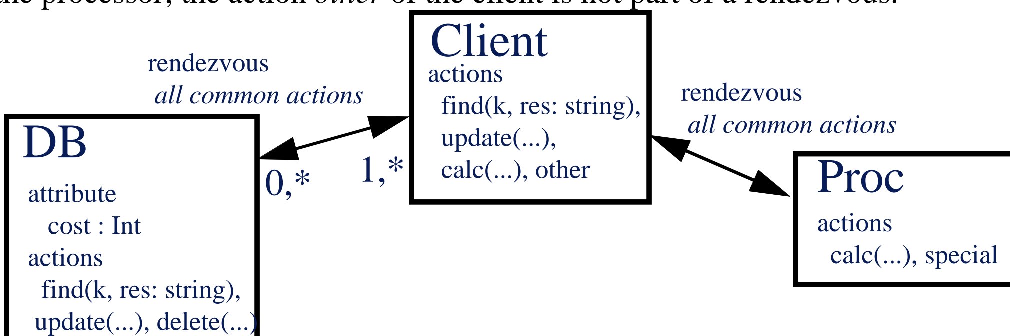


Figure 3.2: Type diagram showing client with database and powerful processor

If we assume that the parameters *k* and *res* of the *find* actions represent the search key and the result of a search, respectively, and if we assume that a *DB* object imposes a condition for the execution of the *find* action, namely that an entry for the search key exist in its database, then the specification of Figure 3.2 implies the possibility of a parallel search in all the *DB* objects to which a given *Client* is related, and the rendezvous *find* will be executed with one of the *DB* objects which contains an entry for the given search key in its respective database.

Star network with switch: The system below represents a central switch (we assume, there is only a single instance) with a number of connected stations. The stations exchange data with other stations via the switch. For this purpose, they send and receive messages that contain three fields, the source and destination addresses and the data. The relation specifies rendezvous for the send and receive actions of the stations and the switch; since this is a 1:N relation, each execution of an action of the switch will perform a rendezvous with the corresponding action of **exactly one** of the related stations. Which of the n stations will participate in the rendezvous is (in this case) determined by the conditions of the stations for the execution of their actions.

Similarly, the switch associates certain conditions and state changes with the execution of its actions. These conditions ensure that the data is transmitted correctly to the intended destination. (Notation: If x is the name of an attribute representing a state variable, its value before the execution of the action is denoted by x, and its value after the execution by x').

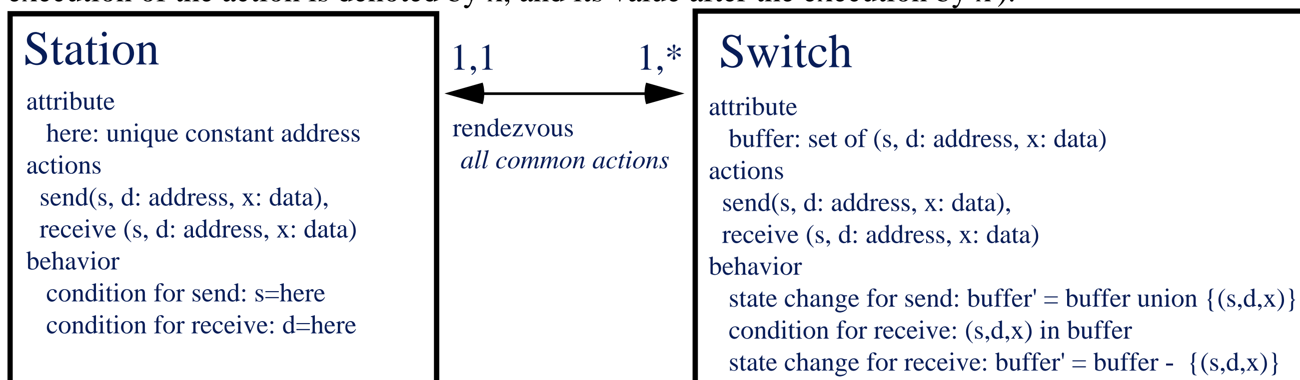


Figure 3.3: Type diagram of a star network

Service access point: The concept of a service access point has been introduced in the OSI Reference Model for the design of layered protocol systems [Knightson, 88 #639]. Often certain local rules concerning the order of execution of interactions, called service primitives, are specified for a given access point [Bochmann, 90g #275], [Bochmann, 90a #270]. These rules should be followed by the service provider and service user components which use the access point as their interface. The following diagram shows a system structure where the local rules are defined by the behavior of the *Access-Point* object and the service interactions are abstract events that are executed jointly by the *Access-Point* and the service *Provider* and *User* objects (in the form of a three-way rendezvous). Three different notations for the description of the behavior (in this case a simple state transition model) are shown in figures (b), (c) and (d).

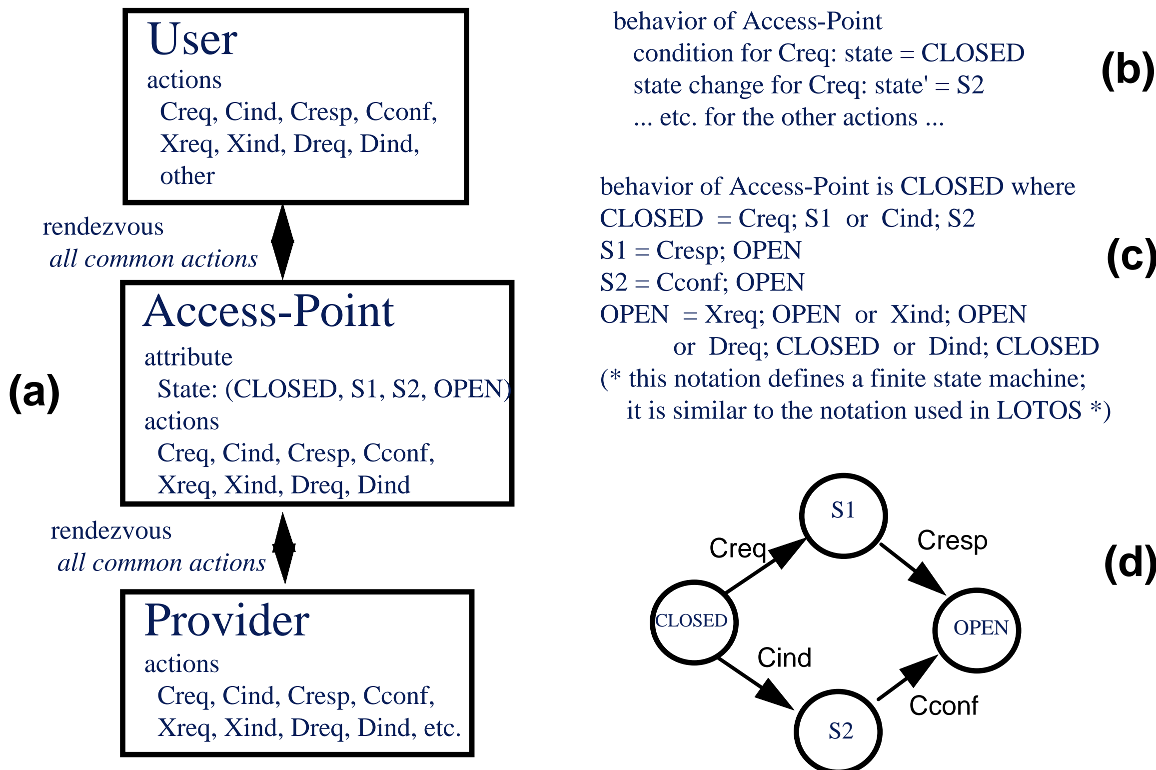


Figure 3.4: (a) Type diagram of an access point with service provider and user (b), (c) and (d): Different (equivalent) behavior descriptions

Physical relationships: Many physical relationships between different objects can be modelled by relations defining abstract events concerning changes of certain physical properties, such as position, temperature, etc. (see for instance [Mili, 90 #1107]). The following example represents two objects, which are cylinders filled with water and subjected to varying pressure. They are connected through a thin tube which allows the water to flow between them.

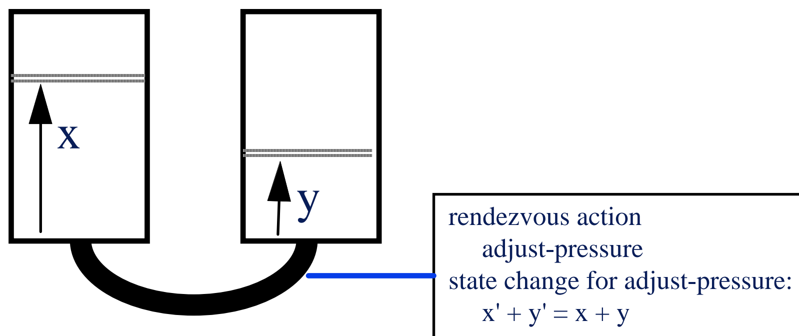


Figure 3.4: Two interconnected cylinders containing a liquid

Petri nets: Petri-nets (see for instance [Peterson, 77 #795]) are often used to model systems with concurrent activities. As the following example shows, they can be modelled using our formalism. The Petri-net places are represented by objects and each transition of the net is represented by a relation and a corresponding abstract event. Note that Figure 3.5(b) does not

include the conditions for the transitions concerning the presence of tokens in the places P1 and P2, neither the specification of the state change corresponding to the change of token configuration in the different places.

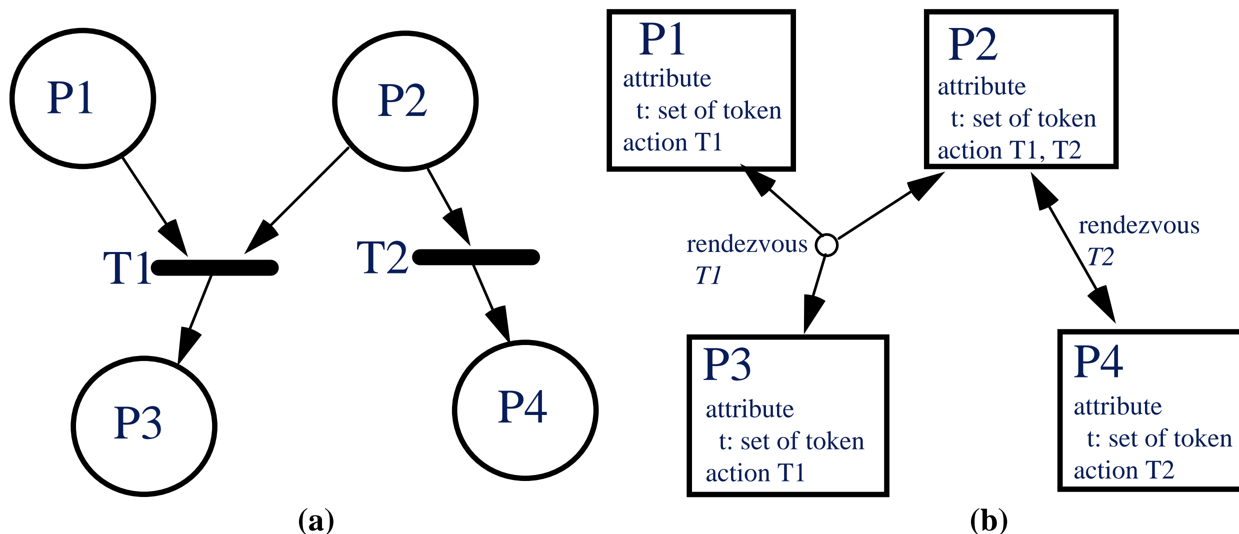


Figure 3.5: Example of a Petri net and its representation as a type diagram

Transactions: A transaction in a database system usually involves several objects which are the parameters of the transaction (see for instance [Gray, 81 #492; Bochmann, 90l #280]). Like the abstract events considered here, a transaction is characterized by conditions that must be satisfied before the transaction can be executed. Its execution is characterized by state changes in the objects to which it is related. Therefore, we may consider a transaction as a dynamic instance of a relation (between a certain number of objects, its parameters) which disappears after its execution.

3.3. Discussion

It is important to note that the above concept of abstract event does not include the notion of caller - callee. This means that it is not specified how an abstract event is "triggered". Only the conditions for its occurrence are specified. A related notion is the distinction between input and output signals for finite state machines and the distinction between input and result (output) parameters of operations. Such distinctions are not explicitly made for the parameters of abstract events. These parameters may be considered input or output depending on the conditions that are imposed by the participating objects. For instance, if one object imposes a specific value for a particular parameter of an event, one may consider this parameter an output provided by the object during this event.

In the case of the examples above concerning the physical relations and Petri nets, there is conceptually no object that is the "caller". The same situation often occurs during the modelling

of business processes where it is more important to define the possible events that may occur (and the order of occurrence) than the actors that are involved in these events [Stanley, 92 #1108]. It seems that the concept of abstract events is a good paradigm for describing the possible events in the system, usually in the form of a relatively abstract, nondeterministic description, leaving such question as triggering of actions, their scheduling and the involved actors for a later, more detailed system description. For instance, in the case of the star network described above, the scheduling of the transmissions of the datablocks contained in the *buffer* of the switch is completely left open. Various priorities and scheduling disciplines could be defined in the form of additional conditions to be satisfied by the *send* event, or by introduing a scheduler object which has access to the *buffer* and can explicitely "call" the *send* events for the different datablocks.

The concept of abstract events described above is closely related to two well-known approaches to system modelling: (1) rendezvous communication and (2) the specification of operations through pre- and postconditions. The rendezvous concept of communication can be used for describing the interactions between finite state machines (see for instance [Merlin, 83 #722]) and is used in Petri nets, CSP [Hoare, 85 #526] and LOTOS . The paradigm of defining the conditions and state changes of an abstract event (as shown in the example of the star network) in terms of enabling predicates and change predicates involving the "before" and "after" values of variables is common to many specification languages (e.g. the SPECIAL language developed at SRI International around 1980) and also corresponds the specification of functions or procedures by assertions in the form of pre- and postconditions (e.g. as in the language Eiffel) .

The main new idea in this paper seems to be the combination of these known concepts with entity-relationship modelling by associating abstract events with relationships. The second new point is the idea that relations, in addition, may be used as a vehicle to "inherit" actions from one object to another, as discussed in the next section. This also leads to an integrated view of aggregation and multiple inheritance.

4. Aggregation and multiple inheritance

Object-oriented design methodologies often consider that certain relations have an additional semantic meaning called "aggregation" or "is-part-of". A typical example is shown in Figure 2.1 which shows that a computer consists of different parts, namely always one power supply, one screen, one keyboard, and possibly several disk units. The multiplicity annotation in this example indicates that the parts may exist without being part of a computer.

4.1. Step-wise refinement

The notion of aggregation is usually related to the notion of composition / abstraction where the same (composed) object may be considered at two different levels of abstraction:

(1) At the abstract level, where the object is considered as a whole, and the components are not visible, and
(2) at the detailed level, where the components of the object are visible.

There is some similarity with two other levels of description that are often distinguished:

(a) At a more abstract level, only the so-called "interface" of the object is defined in terms of the names and parameters of the available operations (see for example the Stack definition given in Figure 2.3(a)). It is important to note that the "behavior" of these operations is not defined at this level.

(b) At the so-called implementation level (e.g. the body of an Ada package), the semantics of an operation is defined by a procedure (e.g. algorithm) that may invoke, in turn, certain operations performed on "internal" objects or attributes, and sometimes also on "external" objects (see for instance the stack implementation shown in Figure 4.2(a)).

At the levels (2) and (b), the internal objects may be considered to be "part of" the composed object being defined. This view is shown in Figure 4.1 for the Stack example.

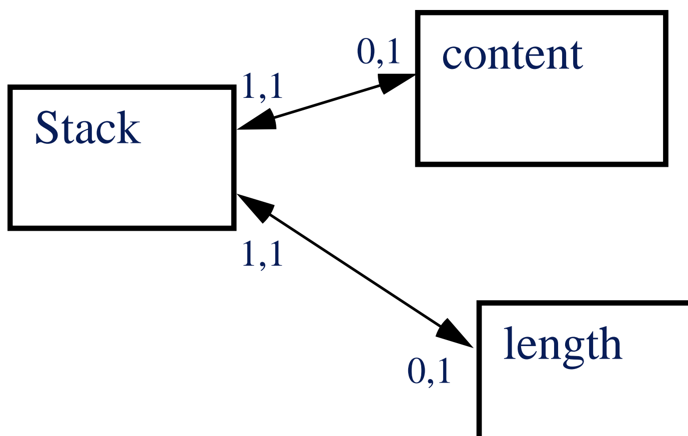


Figure 4.1: Stack object with two components serving implementation

In the following we are interested in integrating, into the aggregation relation, the description of the behavior of the composed object. This behavior can be understood from the behavior of its components and the behavioral inter-relation between the components and the whole. We consider two examples.

The first example defines the *Stack* behavior in terms of abstract events that involve the stack object itself, as well as its components *content* and *length,* which are not visible at the abstract level. Two definitions are shown in Figures 4.2(a) and (b), respectively. The first definition is a relatively straightforward design which uses the sequential programming paradigm for the definition of the semantics of the *Stack* operations. This means that an abstract operation, such as *push*, is defined in terms of a sequence of operations on the components; for instance, *push* is realized by two operations on the components: increasing the *length* variable, and then assigning the *push* parameter to the indexed *content* variable.

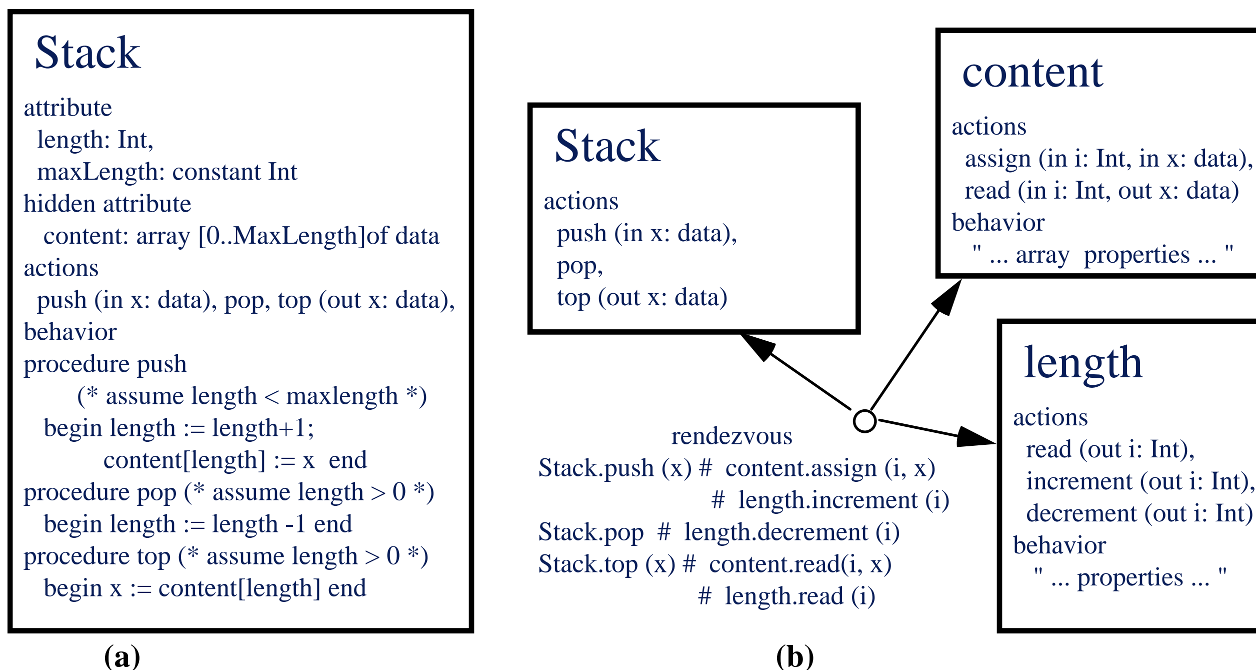


Figure 4.2: Complete specifications of Stack object (including behavior)
(a) Behavior description in terms of internal variables and procedures
(b) Behavior description in terms of component objects and common abstract events

In the case of the design of Figure 4.2(b), we have defined the abstract operations through synchronous rendezvous with certain operations of the components. In order to be able to describe the same semantics with a single rendezvous, we have introduced the ternary relation between the *Stack* and its components, as shown in the figure, and the operations *increment* (returning the incremented value) and *decrement* on the *length* object. Although there is a single rendezvous to realize each operation of the stack, there is some data flow among the operation parameters which also implies some sequentiality; for instance during the execution of the *push* operation, the *increment* operation on the *length* must be executed before the *assign* operation on the *content*, since the former provides the output parameter *i* which is used by the latter.

These two definitions are examples of sequential and parallel decomposition of abstract operations in terms of more detailed operations performed on the components of the abstract object. We note that in general, more powerful formalisms will be required for the operations declared at a more abstract level, in order to describe their semantics in terms of more detailed actions performed on the internal components and other related objects. Traditional approaches include execution control in terms of sequential execution (see for instance [Guttag, 77 #506]), algebraic specifications (see for instance [Ehrig, 85 #423]), or the refinement of actions in Petri nets.

4.2. Inheritance through relations

The next example is introduced in order to discuss in detail situations where certain operations of components may remain visible at the higher level of abstraction. Since at that level the components are not visible, it is necessary to consider that the operation in question is directly "offered" by the abstract object. We consider as an example the *Computer* shown in Figure 2.1, and the action of switching on the power supply, which is realized by the *Power* component. At the higher level of abstraction, the power supply is not visible, but the operation remains. More details are shown in Figure 4.3.

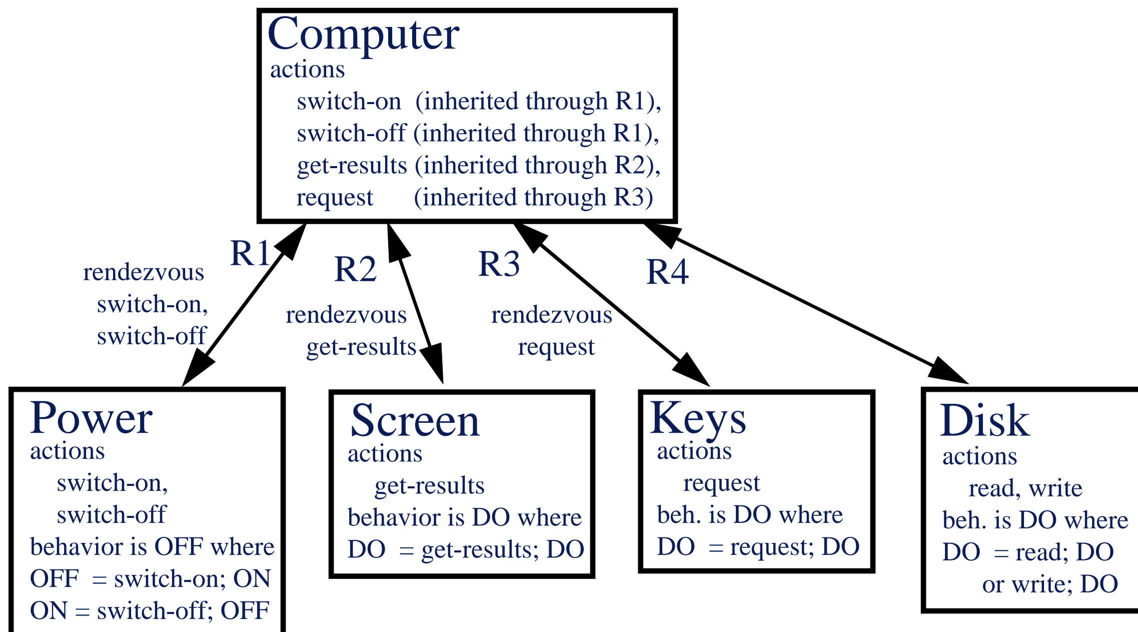


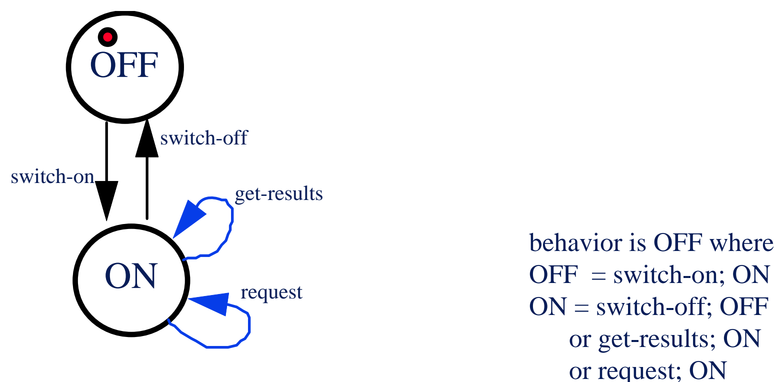
Figure 4.3: Abstract events associated with the computer of Figure 2.1

In this example, we say that the actions of the *Computer* object are **inherited through the relations** *R1, R2* and *R3*, respectively. This means that no behavior for these actions is explicitely specified for the *Computer* object, but that the actions of the related objects are directly visible as actions of the former. For instance, the *switch-on* action of the *Power* object is inherited by the *Computer* object through the rendezvous over the relation *R1*. In this manner, the *Computer* object inherits the operations *switch-on, switch-off, get-results* and *request*, which can be invoked by other objects directly on the *Computer* object, as well as on the components (if they are visible).

This concept is similar to the approach of Wand [Wand, 89 #1137] where certain properties of components may be "inherited" by the composite object, and other new properties may "emerge" at the composite level. A similar approach is also taken for "aggregation inheritance" as introduced by Liu [Liu, 92 #1109], where all operations of a component are inherited by the composite. In contrast, our approach permits "selective" inheritance. Moreover, we do not limit ourselves to aggregation relations; we believe that it is important to be able to associate abstract operations, jointly executed by objects instances which are related through arbitrary

relationships, in order to describe the communication between arbitrary objects within the general context of distributed systems.

The above example of "inheritance through relations" is an extreame case of abstract events where one of the objects involved (here the composite *Computer* object) leaves the behavior of the inherited actions completely undefined. As discussed in [Bochmann, 92a #1027], an explicitely undefined behavior may be interpreted as "arbitray behavior" or "most general". More specific behaviors may be defined by introducing additional constraints and conditions. For example, the *Computer* defined by the specification of Figure 4.3 allows execution of the events *request* and *get-results* before the power is switched on. To make the specification more realistic, one may introduce an additional (global) constraint at the level of the *Computer* object, for instance by imposing that the order of execution of the events should satisfy the rules of the finite state machine shown in Figure 4.4 (i.e. the operations *request* and *get-results* are only possible after the computer is switched on).



(a) State diagram notation (b) Program-like notation

Figure 4.4: Behavior constraint imposed by the *Computer* object

4.3. Multiple inheritance

There has been much discussion in the literature about the question whether the concept of multiple inheritance should be included in object-oriented specification and/or implementation languages (e.g. [Cardelli, 88a #330]). The purpose of this section is to point out that most of the features of multiple inheritance may be obtained through the use of the concept of aggregation plus the notion of inheritance through relations, as explained above. In fact, the *Computer* object has multiple inheritances from its components *Power*, *Screen* and *Keys*.

In general, instead of saying that an object type A has multiple supertypes B1 and B2, we may declare A to be a composite object type with components B1 and B2 where all visible properties of B1 and B2 are inherited through the aggregation relation by the object type A (see also Figure 4.5(a)). The main difference between traditional multiple inheritance and multiple inheritance through aggregation relations is the fact that the latter introduces the component objects with separate object identifiers, while traditional multiple inheritance leads to "simple" objects (with a

single identifier). However, for external objects interacting with objects of type A there seems to be no difference, since in both cases they can invoke the same actions and attributes directly on the objects of type A.

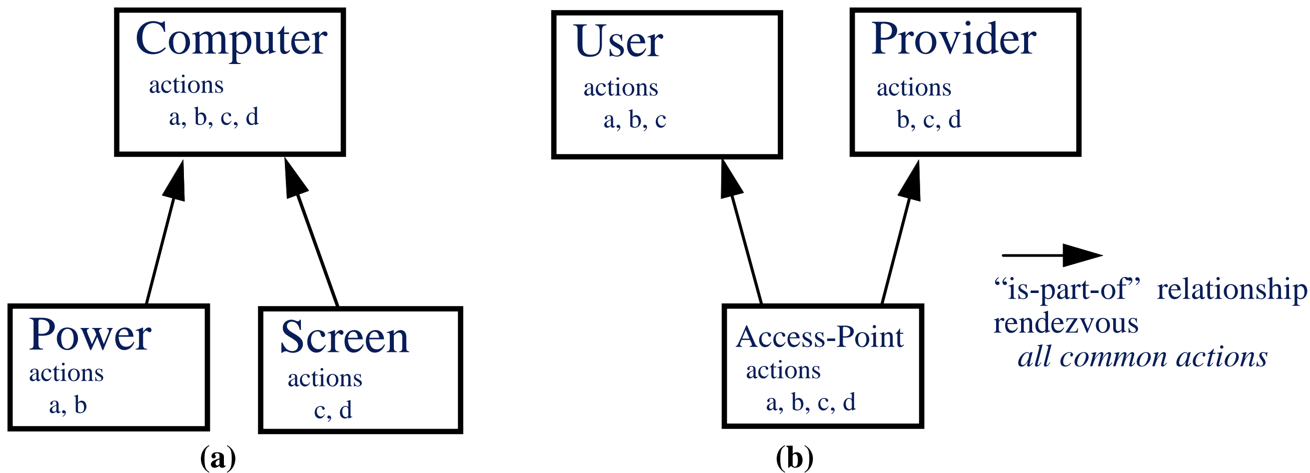


Figure 4.5: Aggregation relationships corresponding to (a) multiple inheritance and (b) a communication interface

We note that for the description of a connection or (communication) interface between two physical system components, it is convenient to allow for the connection (or interface) to be "part of" both system components which it connects. An example is the service access point described in Section 3.2. Figure 4.5(b) shows the corresponding aggregation diagram, which has a structure complementary to the one for multiple inheritance shown in Figure 4.5(a).

5. Conclusions

We have defined the notion of abstract events which seems to be a good concept for the abstract modelling of dynamic interactions in object-oriented systems, and which can be used in relation with system analysis and design methodologies for the description of the dynamic behavior of complex objects and systems. This concept is based on a combination of well-known approaches, including entity-relationship modelling, rendezvous communication and object-orientation including property inheritance. Combined with the notion inheritance through relations, which means that objects may inherit properties from other objects through the relationships between them, we can model multiple inheritance by an aggregation relationship and inheritance through relations.

The various examples given in the paper demonstrate the usefulness of these concepts. However, further experience with larger examples is required to provide a more detailed evaluation of this approach.

Most object-oriented analysis and design methods (e.g.[Rumbaugh, 91 #1110]) proceed in several steps where the first step is usually the identification of the object types of interest, and a

later step identifies the operations that are supported by the objects. Some high-level modelling approaches, however, start with the identification of the major events in the system to be modelled (e.g. [Stanley, 92 #1108]). We believe that the concept of abstract events described in this paper can provide a solid foundation for such kinds of modelling approaches.

Acknowledgements

The ideas presented in this paper where influenced by several research projects in collaboration with industry, such as a project on object-oriented modelling supported by the Centre de Recherche Informatique de Montreal (CRIM) and Bell-Northern-Research during 1989 through 1991, an ongoing project on the modelling of IT architectures funded by DMR-Macroscope, and the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols and the Canadian Institute of Telecommunications Research (CITR), which support the author's research group. The author would like to thank all collaborators in these different projects for many interesting discussions, and in particular M. Barbeau, H. Mili, D. Ramazani and F. Stanley. F. Lustman and R. Dssouli provided useful comments for improving the manuscript.

References

[Boch 90a] G. v. Bochmann, *Specifications of a simplified Transport protocol using different formal description techniques*, Computer Networks and ISDN Systems, Vol. 18, no.5, June 1990, pp. 335-377.

[Boch 90g] G. v. Bochmann, *Protocol specification for OSI*, Computer Networks and ISDN Systems 18 (April 1990), pp.167-184.

[Boch 90l] G. v. Bochmann, M. Barbeau, M. Erradi, L. Lecomte, P. Mondain-Monval and N. Williams, *Mondel: An object-oriented specification language*, publication départementale no. 748, Dépt. IRO, Université de Montréal, 1990.

[Boch 92a] G. v. Bochmann and R. Gotzhein, *Specialization of object behaviors and requirement specifications*, publication départementale no.853, Dépt. IRO, Université de Montréal, janvier 1993.

[Boch 92p] G. v. Bochmann, S. Poirier and P. Mondain-Monval, *Object-oriented design for distributed systems and OSI standards*, Proc. of IFIP Int. Conf. on Upper Layer Protocols, Architectures and Applications, Vancouver, May 1992, G. Neufeld and B. Plattner (Eds.), North-Holland Pub., pp. 265-280. A shorter version is also included in the proceedings of the Int. Workshop on ODP, Berlin, Oct. 1991.

[Bolo 87] T. Bolognesi and E. Brinksma, *Introduction to the ISO Specification Language Lotos*, Computer Networks and ISDN Systems, vol. 14, no. 1, pp.25-59, 1987.

[Card 88a] L. Cardelli, *A semantics of multiple inheritance*, Information and Computation 76 (1988), pp. 138-164.

[Chen 76] P. P. Chen, *The Entity-Relationship model - Toward a unified view of data*, ACM Trans. on Database Systems, Vol. 1, No. 1, March 1976, pp.9-36.

[Cost 92] R. J. C. d. Costa and J. P. Courtiat, *A true concurrency semantics for LOTOS*, Proc. FORTE'92 (IFIP), M. Diaz and R. Groz (Eds.), North Holland Publ. 1993.

[Ehri 85] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1*, Springer Verlag, 1985.

[Gray 81] J. Gray, *The transaction concept: virtues and limitations*, Proc. Conf. on VLDB, Cannes, Sept. 1981 (IEEE), p. 144-154.

[Gutt 77] J. Guttag, *Abstract data types and the development of data structures*, Comm. ACM 20, 6 (June 1977) pp. 396-404.

[Hoar 85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[Larm 88] K. G. Knightson, T. Knowles and J. Larmouth, *Standards for Open Systems Interconnection*, McGraw-Hill, 1988.

[Liu 92] L. Liu, *Exploring semantics in aggregation hierarchies for object-oriented databases*, ???

[Merl 83] P. Merlin and G. v. Bochmann, *On the Construction of Submodule Specifications and Communication Protocols*, ACM Trans. on Programming Languages and Systems, Vol. 5, No. 1 (Jan. 1983), pp. 1-25.

[Mili 90] H. Mili, J. Sibert and Y. Intrator, *An object-oriented model based on relations*, Journal of Systems and Software, Vol. 12 (1990), pp. 139-155.

[Pete 77] J. L. Peterson, *Petri Nets*, Computing Surveys, Vol.9 No.3, September 1977, pp.223-252.

[Rumb 91] J. Rumbaugh and e. al., *Object-oriented modeling and design*, Prentice Hall, 1991.

[Stan 92] F. H. Stanley, *Conceptual framework for analysing business infrastructure as an integrated dynamic model*, DMR Group Inc., Montreal, Canada.

[Wand 89] Y. Wand and R. Weber, *A model of systems decomposition*, in Proc. 10th Int. Conf. on Inform. Systems (Boston), Dec. 1989, pp. 41-51.