Modelling Global Behaviour in Object-Oriented Analysis: Scenarios, Use Cases and Interaction Diagrams[1]

Dorothea Beringer
Laboratoire du Génie Logiciel
LGL-DI-EPFL
CH - 1015 Lausanne
d.i.beringer@ieee.org

December 1996

Abstract

Most software development methods define notations for modelling global behaviour. Many of these notations use interaction diagrams to model possible sequences of interactions between a system and its environment, or between the objects of the system. Other methods use structured textual descriptions or regular expressions for specifying use cases or use scenarios. Although many methods use similar notations, there are often substantial differences in their underlying concepts and in the content expressed by the diagrams. On the other hand, completely different notations use sometimes the same concepts and show the same aspects of the global behaviour of a system. In this technical report we investigate the characteristics of different notations. We show what their underlying concepts are and how they relate to each other. Finally, we give an overview of the techniques that are used by some of the current object-oriented analysis methods for modelling global behaviour.

1. This is an enhanced and updated version of the report of the project of the postgraduated course "Génie Logiciel" of November 1995, with the title "Modelling Global Behaviour in Object-Oriented Analysis; Overview of the Usage of Scenarios, Use cases and Interaction Diagrams".

Table of content

Table of content i
List of figures iii
1 Introduction 1
2 Modelling global behaviour 1
2.1 The global behaviour of an object-oriented system 1
2.1.1 Global versus local behaviour 1
2.1.2 Internal versus external view of the global behaviour 2
2.1.3 Conceptual versus technical events 2
2.1.4 Requests versus notifications 3
2.1.5 Design versus analysis 4
2.2 Modelling global behaviour by modelling scenarios 6
2.2.1 Scenarios of one object versus scenarios between several objects 6
2.2.2 Scenario instances versus scenario types 6
2.2.3 Scope of a scenario, classification schemes 7
2.3 Notational limits in modelling interacting objects 9
2.3.1 Modelling the dynamics versus modelling the statics 10
2.3.2 Dynamic models: transitions between states versus interactions between objects 10
2.3.3 Interactions between objects: time-line diagrams versus object diagrams 13
2.3.4 Some further aspects of interaction diagrams 14
2.4 Summary 15
3 Overview on various methods and modelling techniques 17
3.1 Introduction 17
3.2 OOSE 17
3.2.1 Terms 17
3.2.2 Mappings and classifications 18
3.2.3 Notations 18
3.2.4 The use of the models in the development process 19
3.3 Fusion 20
3.3.1 Terms 20
3.3.2 Mappings and classifications 20
3.3.3 Notations 21
3.3.4 The use of the models in the development process, weaknesses 22
3.4 OBA 24
3.4.1 Terms, mappings and classifications 24
3.4.2 Notations 24
3.4.3 The use of the models in the development process 25
3.5 FORAM 25
3.5.1 Terms and notations 25
3.5.2 The use of the models in the development process 27
3.6 BON 27
3.6.1 Terms 27
3.6.2 Mappings and classifications 27
3.6.3 Notations 28
3.6.4 The use of the models in the development process 29
3.7 Booch and OMT 30
3.7.1 Notations 30
3.7.2 The use of the models in the development process 31

3.8 Modern Structured Analysis 31
3.8.1 Terms 31
3.8.2 Mappings and classifications 31
3.8.3 Notations 32
3.8.4 The use of the models in the development process 32
3.9 Further notations for modelling global behaviour 33
3.9.1 Path Expression Groups 33
3.9.2 Scenario trees 33
3.9.3 Composition of scenarios based on statecharts 35
3.9.4 CRC cards 35
3.9.5 Requirements scripts in OSMOSIS 36
3.9.6 Storyboarding 36
3.9.7 Business processes (“Geschaeftsvorfaelle”) 37
3.9.8 Extending OOSE by use case levels 38
3.9.9 Use case maps 39
3.9.10 M.E.R.o.DE 39
3.9.11 Behavioural models of Kowal 39
3.9.12 Further approaches 40
4 Examples of diagrams for modelling global behaviour 41
5 Conclusions 60
6 References 60

List of figures

Figure 1: External versus internal views 2
Figure 2: Technical and conceptual events for different abstraction levels 3
Figure 3: Using notifications for the external and requests for the internal view 5
Figure 4: Modelling scenario instances and scenario types 8
Figure 5: Determining the end of a scenario 9
Figure 6: Transitions between states versus interactions between objects 11
Figure 7: The dynamic model of a system: life-cycles of the objects or scenarios of the system functions 12
Figure 8: Overview of the concepts for modelling global behaviour 16
Figure 9: Object model with communication associations (arrows with no labels) in OOSE [Jacobson92, page 189] 41
Figure 10: Description of the use case 'customer withdrawal' in OOSE [Jacobson92, page 349] 41
Figure 11: Modelling the relationship between abstract and concrete use cases in OOSE [Jacobson92, page 343] 42
Figure 12: Interaction diagram in OOSE (example with two interface objects) [Jacobson92, page 381] 42
Figure 13: Interaction diagram in OOSE (input and output signals) [Jacobson92, page 218] 43
Figure 14: Event trace diagram in Fusion, part of the analysis model [Coleman94, page 47] 43
Figure 15: Life-cycle expressions in Fusion, part of the analysis model [Coleman94, page 33] 43
Figure 16: Scenario diagram of extended Fusion showing a use case instance with a sub use case [Coleman95] 44
Figure 17: System interaction graph in Fusion, part of the extensions for requirements modelling [Coleman95] 44
Figure 18: Schema of a system operation in Fusion, part of the analysis model [Coleman94, page 31] 44
Figure 19: Decomposition of an object interaction graph in Fusion [Coleman94, page 79] 45
Figure 20: Object interaction graph in Fusion, part of the design model [Coleman94, page 75] 45
Figure 21: Event list in Modern Structured Analysis [Beringer92] 46
Figure 22: Context diagram in Modern Structured Analysis [Beringer92] 46
Figure 23: Context model, message table and template for task cards in FORAM [Graham94a] 47
Figure 24: Event trace in FORAM [Graham94a] 47
Figure 25: Event charts and scenario charts in BON [Walden95, pages 253-255] 48
Figure 26: Object Scenario in BON [Walden95, page 259] 48
Figure 27: Object Scenario in BON (with grouping of objects into several subtasks) [Walden95, page 114] 49
Figure 28: State chart as used in Booch [Booch94, page 207] 49
Figure 29: Interaction diagrams in Booch [Booch94, page 218] 49
Figure 30: Object diagram for one mechanism [Booch94, page 402] 50
Figure 31: Description of a use case in OMT [Rumbaugh94a] 50
Figure 32: Add-relationships between use cases in OMT [Rumbaugh94a] 51
Figure 33: Two different kinds of event trace diagrams in OMT [Rumbaugh95a] 51
Figure 34: Event flow diagram in OMT [Rumbaugh95a] 51
Figure 35: Object interaction diagram showing the design of the operation 'redisplay' [Rumbaugh95b] 51
Figure 36: Concurrent object interaction diagram in OMT [Rumbaugh95b] 52
Figure 37: Operation specification in OMT [Rumbaugh95b] 52
Figure 38: Object event table and structure diagrams in M.E.R.O.DE [Dedene94] 52
Figure 39: Scenario tree and grammar of the user view of the 'caller' [Hsia94] 53
Figure 40: Scenario tree and conceptual state machine of the user view of the 'callee' [Hsia94] 53
Figure 41: State chart for one scenario in [Glinz95] 54
Figure 42: Composition of scenarios in [Glinz95] 54
Figure 43: Scripts in OBA [Rubin92] 55
Figure 44: Collaboration graph in Responsibility Driven Design [Wirfs90, page153] 55
Figure 45: Class diagram with main collaboration in [Wilkinson95, page143] 56
Figure 46: Class card in [Wilkinson95, page115] 56

Figure 47: Scenario description in [Wilkinson95, page146] 56
Figure 48: Textual description of a scenario [Kilberth93, page 99] 56
Figure 49: Action diagram with brackets, used for the specification of a scenario [Kowal92, page292]..... 57
Figure 50: Event-response list and sample storyboard in OORD [Umphress91] 57
Figure 51: Graphical representation of a “Geschaeftsvorfall” in [Mueller93] 58
Figure 52: Bergen interaction diagram, captures also the inheritance hierarchy of the objects [Baklund95] .. 58
Figure 53: Description of a group of interacting objects using path expressions (PEG) [Adam94] 59
Figure 54: Use case map for a design pattern [Buhr96] 59

1 Introduction

Investigating the current notations for modelling global behaviour can be very confusing. On the first glimpse, some of the notations only differ in their graphical symbols and in the additional information they represent. But if there were only these differences, then all these diagrams should be easily transformable into each other. This is not the case because there are also divergences in the basic concepts that are used. The result are similar notations for totally different concepts, and different notations for the same concepts. To make it even more difficult, the same terms are used for different concepts in different methods. This is possible because there does not yet exist any definitions and classifications of the different terms, concepts, approaches and notations that are generally accepted.

In this technical report we discuss the underlying concepts of various notations. We also describe what aspects of global behaviour can be reflected by these notations, and how these aspects can be combined into one diagram. The starting point of the investigation were those approaches that use interaction diagrams or event trace diagrams to model sequences of interactions between a system and its agents, or among its internal objects, and also those notations providing textual descriptions of such sequences (e.g. use cases or use scenarios). Because the same content can also be expressed with regular expressions or state charts, we also included such techniques. The emphasis, however, was put on interaction diagrams and use case descriptions.

We do not provide a comparison or evaluation[1] of different methods, nor do we present a general metamodel[2]. We focus on the various concepts found in techniques that model global behaviour, we give an overview on these concepts and on their relationships in chapter 2, and we provide a short description of various methods in chapter 3. Examples of diagrams are found in chapter 4. They are added to this report in order to facilitate the reading.

2 Modelling global behaviour

2.1 The global behaviour of an object-oriented system

Berard [Berard93] defines an object-oriented system as a *system of interacting subsystems or objects*. The *structure* of such a system is given by how each component is composed of and connected to other components. The *behaviour* is given by the interactions within the system and by the interactions between the system and its environment. The behaviour specifies how an object or system acts and reacts, in terms of its state changes and interactions, and it shows its outwardly visible activity [Booch94, page86].

2.1.1 Global versus local behaviour

In any system of interacting objects we can look at the behaviour of the individual objects or at the behaviour of the whole system. When describing the *local behaviour*, we focus on one single object and how it interacts with its environment under certain conditions (e.g. by specifying the class interface with the contracts for all the services the object offers and for all the services the object needs from other objects). When describing the *global behaviour*, the whole system of interacting objects comes into the focus, and we concentrate on the behaviour that is provided by these objects together.

1. A quantitative comparison of object-oriented methods, mainly focusing on notational issues, can be found e.g. in [Stein93, Stein94]. There 4 macrocomponents (or characteristics) are distinguished, each being divided into about 40 microcomponents. For each method these microcomponents are rated with a value between 0 and 2 (no support, average support, good support). The interpretations of this data are very delicate, because they cannot take into account the underlying concepts, orientation and philosophy of a method. The conclusions are purely quantitative, and in my opinion do not really do justice to those methods that do not do well in respect to the implicitly chosen evaluation criteria.
2. There does not yet exist a commonly accepted metamodel that could be used for a comparison. OMG has tried to get all different object-oriented methods described on the basis of a common metamodel [OMG92b]. Yet as far as I know, this effort has not continued.

2.1.2 Internal versus external view of the global behaviour

Modelling global behaviour of a system can denote either an internal or an external view of the system (see figure 1). In the external view, the model shows the services that the system as a whole offers to its environment, without considering how this behaviour comes about internally. The system itself appears as one single object; we only describe the interactions between the system and its environment. The environment may be further detailed into different agents[3]. In other words, the external view shows the interface of a system or subsystem. Yet the interface definition does not only mention all the services offered but also all the services needed from other systems in the environment. An example for an external view of the global behaviour are the use cases of OOSE or the interface model of Fusion.

A special case of modelling the external view is by using user views. This is e.g. the case in the scenario trees of [Hsia94]. There, the behaviour of the system is modelled from the perspective of one single agent, i.e. only the interactions between the system and this agent are taken into account, interactions with other agents are only shown in the user views of these other agents (see figures 39 and 40).

An internal view not only contains the interactions with the environment but also the interactions among the internal objects of the system. The internal model thus shows how these objects work together to provide a certain global behaviour. Examples are the object scenarios in BON and the object interaction diagrams in OMT, Fusion and OOSE.

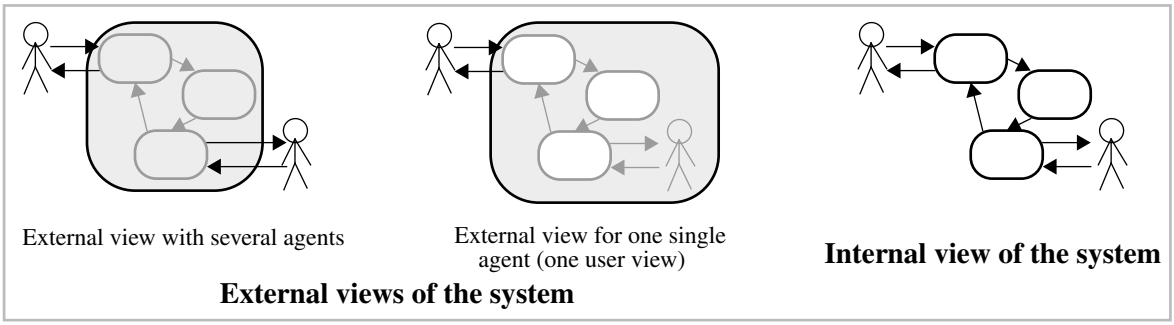


Figure 1: External versus internal views

2.1.3 Conceptual versus technical events

When modelling the interactions between objects (among internal objects or between the system and the agents), we can do this on different abstraction levels. Therefore we introduce here the distinction between *technical, conceptual and purely conceptual* events or interactions.

- *Technical:* A technical event or interaction corresponds to a certain software construct in the final software system. This may be a procedure call, an entry in a dialogue window, a menu item, or a hook in an UIMS. Examples are the method calls in the interaction diagrams of Fusion, OOSE and OMT.
- *Conceptual:* A conceptual event or interaction needs not to correspond to any software construct, it may be more abstract. Yet such an event can always be expressed by the vocabulary of the problem domain. Examples are the transactions in the use cases of OOSE, or the events in the event lists of Modern Structured Analysis, M.E.R.O.DE or OORD.
- *Purely conceptual*: Conceptual events that are not technical events are purely conceptual events. If in the analysis or high-level design an interaction is a purely conceptual event or not can sometimes only be determined from a retrospective view, i.e. when the design is completed.

Of course within the technical events and within the conceptual events there are again arbitrary many abstraction levels. E.g. a technical event provided by a UIMS can again be detailed into the single key-

3. An agent may be anything (a human being, a software system, a hardware component) that is outside of the system and can send or receive events to or from the system. The various agents are normally distinguished by their roles they have in respect to the system.

board entries that are necessary to generate that event. When building an information system, the lowest necessary abstraction level to be considered will be the events provided by the UIMS. Yet when building an UIMS, we will also model more low-level events. The same is true for the conceptual events. Moreover, a certain abstraction level is conceptual as well as technical, whenever the actual events found in the software system correspond to concepts of the problem domain (e.g. system operations in Fusion). When modelling the global behaviour, any possible abstraction level can be chosen. Figure 2 shows twice the same scenario, once on a purely conceptual level and once on a technical level.

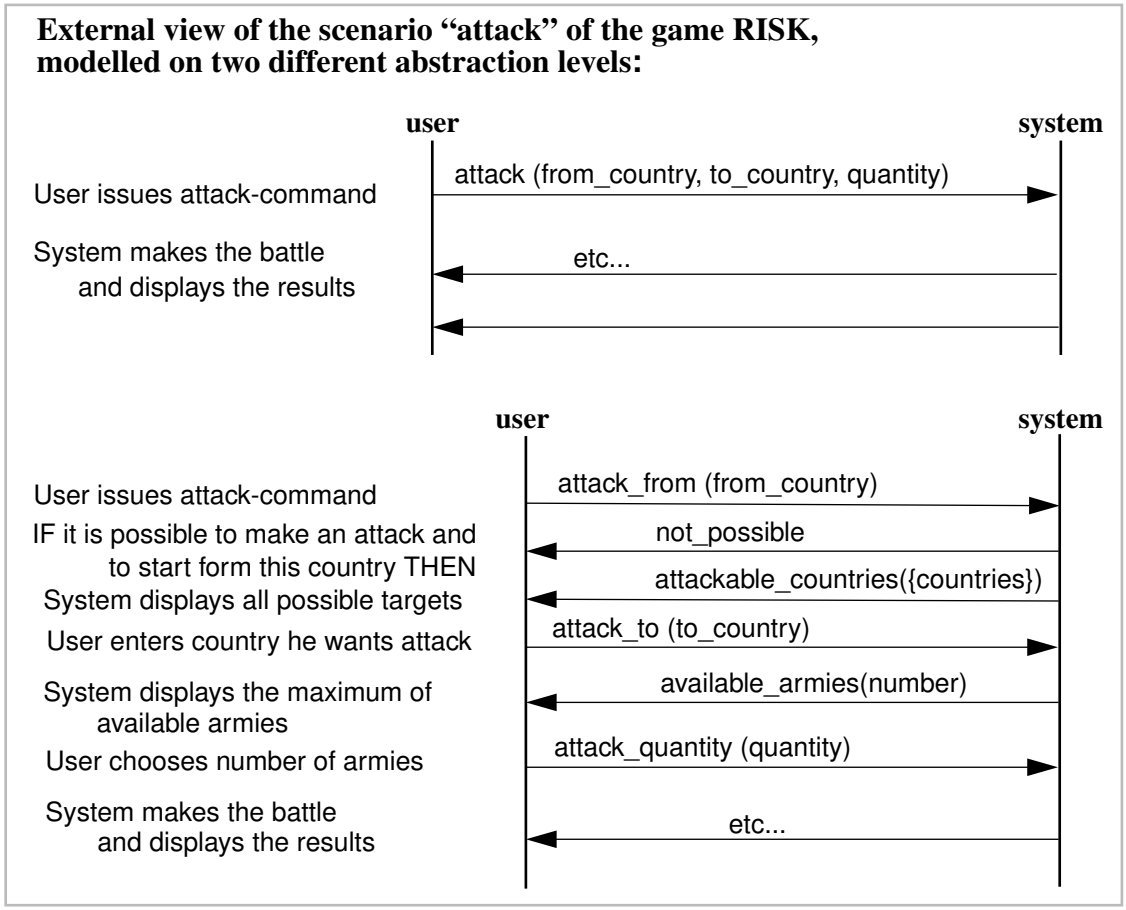


Figure 2: Technical and conceptual events for different abstraction levels

An often heard criticism concerning certain object-oriented analysis methods is the weak connection between the static model showing the object types and their operations and the dynamic model being a set of state transition diagrams, when the state transition diagrams show conceptual events that are on a higher abstraction level than the events calling the object operations shown in the static model. Unfortunately the explicit mapping between these two levels of events is often missing.

2.1.4 Requests versus notifications

Besides the decision for a specific abstraction level there are two further points to decide upon concerning the interactions: Do we consider the objects as being active, each having its own thread of control, or as passive? Do we consider the interactions being one-way events (notifications) or being requests?

If we decide for passive object and thus for only one thread of control in the whole system, the flow of control coincides with the flow of interaction. We have *intraprocess* interactions. Sequential programs only have intraprocess communication, and also languages like Smalltalk and C++ or methods like Booch [Booch91], first generation OMT [Rumbaugh91] and Fusion [Coleman94] only support intraprocess interactions. In contrast to the intraprocess communication, *interprocess* communication takes

place between two active objects, each of them having its own thread of control. Interprocess communication is the basic concept for all concurrent systems, for the models of workflow management systems and for languages like CSP and Occam. Interprocess communication is also that type of communication we encounter in daily life between people, between organisations and between machines.

An interaction between two objects can be a simple *notification* (or *one-way event*), or it can be a *request*. In case of a notification, one object notifies another one by sending him a message with some information. No return event takes place after the notification. In case of a request, a client object requests a service from a server object and waits for its reply. Requests imply a client-server- (or uses-) relationship between two objects. Object-oriented languages for sequential systems only support requests.

When modelling object-oriented systems, we thus get four possible forms of interaction, though only two of them play a major role when discussing the modelling of global behaviour:

	interprocess	**intraprocess**
notification	**signal, notification**	**"goto"**
request	**remote procedure call**	**procedure call**

Sometimes the same notations are used for models showing notifications as well as for models showing requests. An example for this is the event trace diagram of OMT in figure 33: the left diagram shows notifications, the right one shows requests using two arrows for each call. In contrast to this, the concurrent object interaction diagram of OMT (figure 36) uses slightly different symbols for interprocess notifications and for intraprocess requests (procedure calls). Also in the interaction graphs of Fusion, arrows can denote notifications (in the system interaction graphs) or procedure calls (in the object interaction graphs). Though the interpretation of the diagram is quite different, in both cases the same symbols are used, which may be quite confusing.

Often the external model of analysis represents the system on a more abstract level and thus uses notifications for the interactions between the active system and the active agents, representing purely conceptual events. E.g. the input and output events of the interface model of Fusion are notifications. They may have parameters but have of course no return values. Figure 14 shows an event trace, figure 17 a system interaction graph with notifications. When moving to the design, the perspective then often changes, because in fact it is the system that prompts the user for some command or information. The interaction with the user is a procedure call from some controller or problem domain object to an interface object which acts as mediator. Figure 3 shows the transition from notifications to requests when changing from the external view to the internal view.

In a sequential system, the interactions among internal objects are always modelled as requests. For an example see the object interaction graphs of Fusion (figure 20). Yet when describing the interactions among internal objects as well as the interactions with the environment of the system in one interaction diagram, we often would like to show the effective procedure calls for the internal interactions, and at the same time to abstract away the interface mechanism, thus modelling the user inputs as notifications[4]. Examples for this are found in the interaction diagrams of OOSE (figure 13) where even different kinds of arrows are used for the notifications and the requests (procedure calls), and in the object scenarios of BON (figure 26).

2.1.5 Design versus analysis

Modelling the global behaviour of a system may be part of the analysis as well as of the design. What view of the global behaviour is recommended in which phase is different from method to method. Some methods model technical events only during design, others already during analysis. Some determine the

4. For the user interactions in sequential programs, the transition from notifications to procedure calls often is deferred to the implementation phase. There an active object is introduced that handles the interactions between the problem domain objects and the user interface components (e.g. by having an event loop). This object may also control the system life-cycle (e.g. as suggested by Fusion).

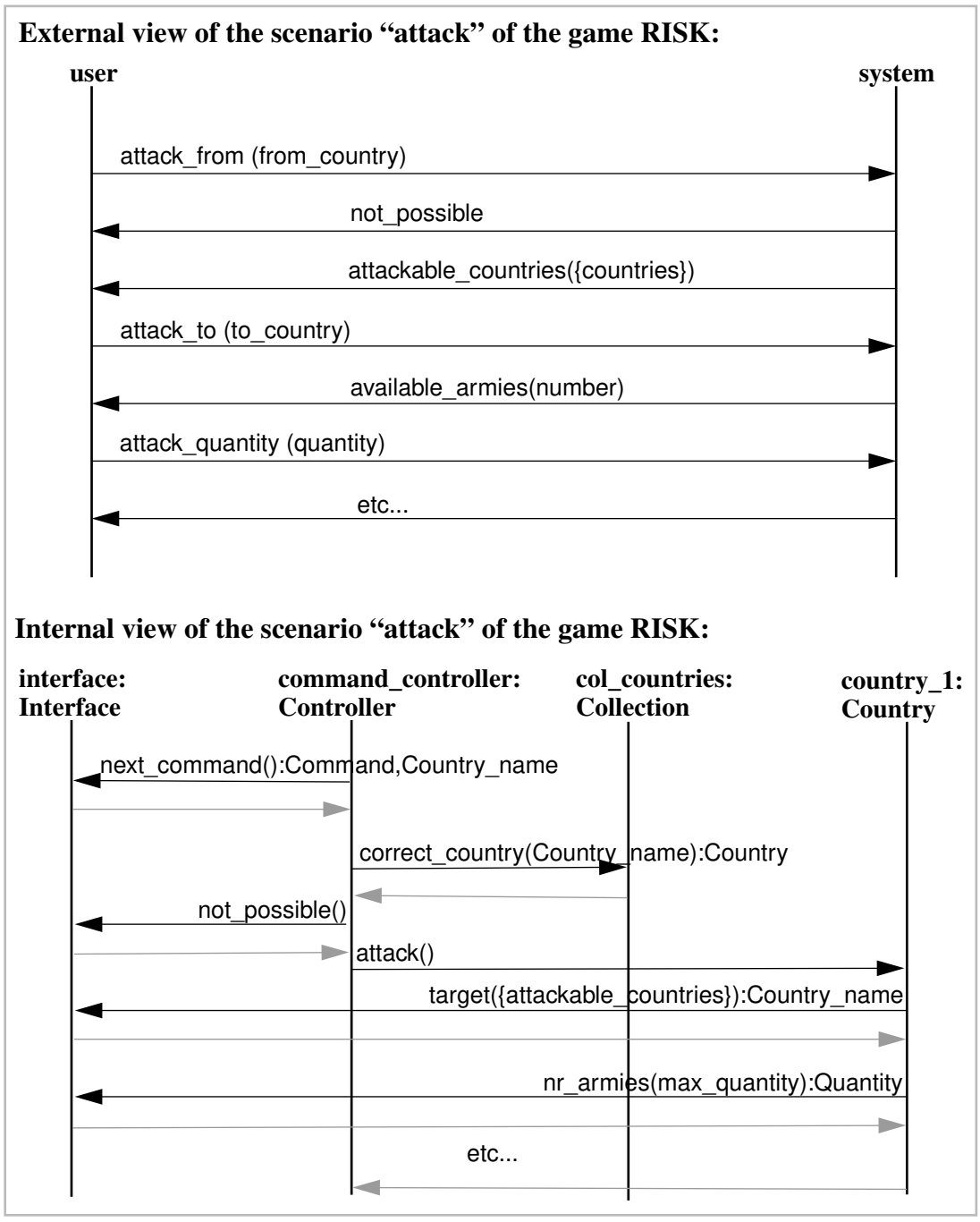


Figure 3: Using notifications for the external and requests for the internal view

internal view only in the design, for others it is part of the analysis. In some methods certain notations like state charts or interaction diagrams are used during analysis, in other methods they belong to the design. Furthermore, there are also many methods that do not know a clear distinction between analysis and design at all.

In this overview we primarily want to discuss modelling techniques for the analysis. Yet due to the often quite arbitrary distinction between analysis and design, we also consider certain design notations.

2.2 Modelling global behaviour by modelling scenarios

We use here the following definition for the term scenario: "A *scenario* is a sequence of events or interactions. These events provide together some service or system function. The sequence is triggered by an initial event, which is also called the *stimulus* of the scenario."

Many methods use scenarios that fulfil above very general definition. But the scenarios have different names, different syntax and semantic, and are used for different purposes. Examples for scenarios are: use case (OOSE), scenario and system operation (Fusion), script and use scenario (OBA), scenario, user case and top-level system operation (2nd-generation OMT), use case, scenario and function point (Booch), scenario (Kowal), essential activity (MSA), business transaction (MSA-GfAI). For more details see chapter 3.

2.2.1 Scenarios of one object versus scenarios between several objects

In above definition of scenarios we have not distinguished between the scenario models that show the interactions among several objects and those that show the interactions of one object with its environment. Yet there are some essential differences between these two kind of models.

In the first case, *the scenario between several objects*, we have several objects, subsystems or systems which can all interact with each other. These interactions can be either notifications in the case of a concurrent system or of several interacting systems, or requests in the case of a sequential system. In those methods that focus on the development of sequential systems, all models showing interactions between several objects show requests; of course this delimits the possible interaction mechanisms.

In the second case, *the scenario of one object*, we show the interactions between this object, subsystem or system with its environment, the agents. Direct interactions between the agents are not considered. This kind of models are mainly used for two purposes: to model the local behaviour of one single object which is part of a larger system (e.g. state charts showing object life-cycles), or to model the global behaviour of a system or subsystem from an external point of view (e.g. interface model of Fusion). As there is only one object we are focusing on, we can divide up the interactions into input (from an agent or a client to the object) and output events (from the object to an agent or a client). One input event causes only one output event if it is a request (see e.g. the operation schema of an internal method in Fusion), or may cause several output events if it is a notification (see e.g. the schemas of the system operations in Fusion).

A scenario that shows an external view of the system only contains for one specific system function all the interactions between the system and its environment. It can be refined into a scenario that shows also the interactions between the components of the system (internal view of the system, external views of the components). For each component, the external view showing input and output events can again be refined into an internal view showing also the interactions among the components of this component. Most methods do not have a recursive refinement of this kind, but stick to the two levels system and objects. They only differentiate between an external model of the system as a whole and the internal model on the level of objects and their interactions. There are two reasons for it:

- Somewhere during the refinement process we switch from purely conceptual events to technical events and from notifications to requests. It is most easy to make these switches between the external view of the whole system, and the internal view on the level of objects. Often also the modelling technique is changed (see e.g. OOSE and Fusion).
- Most methods do not really support the elaboration and the modelling of subsystems that are black-box encapsulations[5].

2.2.2 Scenario instances versus scenario types

When modelling interactions we can either show only one possible *scenario instance* or we can show a whole *scenario type*. A scenario type represents, depending on the chosen classification criterion, a smaller or larger set of possible scenario instances (see figure 4).

5. BON uses object groups in its object scenarios; these serve only for simplifying the diagrams and have no further meaning within the system.

Modelling scenario instances

Diagrams of scenario instances show snapshots of the system behaviour. They are suited for the communication with people not used to abstract thinking, for finding the boundaries of the system or for comprehending a complex internal mechanism. An example for such a diagram is found in figure 33. There nothing is parameterized. Even all the parameters that do not influence the course of actions are replaced by concrete values or object instances. Of course, diagrams showing only scenario instances are not suited to acquire a complete behavioural model. Therefore, much more often scenario types are modelled. Data types or object types are used as parameters in the event types, and several possible courses through the scenario are described.

Algorithmic versus declarative specification of scenario types

When making a complete model of a scenario type, we have to specify all possible orders of events. We can do this in an *algorithmic* way, annotating the diagram with *sequence information* that shows iteration, alternation and concurrency (see e.g. scripts for interaction diagrams of OOSE and Booch in figures 13 and 29, dotted lines for iterations in the event trace diagrams of Fusion in figure 14, sequence labels for interaction graphs of Fusion in figure 20 and operators in the life-cycle expressions of Fusion in figure 15 or in the path expressions of PEG in figure 53). Not all notations allow to model any kind of algorithms, e.g. the event traces of Fusion only know iterations. Others have symbols for alternative, optional and iterated events. Some even offer a syntax for concurrent or interleaving sequences of events.

In contrast to the algorithmic description, we can specify the reactions on the initial event in a *declarative* way. In this case, only the state changes and all the interactions that took place are specified, but not their order. An example are the system operations in Fusion (see figure 18).

2.2.3 Scope of a scenario, classification schemes

There are several possibilities how we can group scenario instances into scenario types. There are also several possibilities how we determine when a scenario ends and the next one begins. All the instances that belong to a scenario type as well as the length of these instances determine the *scope* of a scenario type. In the following we present two sets of possible classification criteria (classification schemes) for determining the scope of a scenario type.

Possible criteria for classifying scenario instances into scenario types

One scenario type may include:

1) all scenario instances with exactly the same sequence of event types; only the values of the system state and of the event parameters may differ,

2) some scenario instances with the same initial event type (stimulus) and a subset of all the possible sequences of further event types,
  2a) the scenario type is determined by the content of the input events,
  2b) the scenario type is determined by the system state at the time of the initial input event,

3) all scenario instances with the same initial event type (stimulus); the rest of the scenario may differ arbitrarily,

4) all scenario instances that have certain common parts or handle similar tasks.

In most of these criteria, the specification of the initial event types (i.e. the classification of event instances into event types) determines how the scenario instances are grouped into scenario types. These criteria presuppose well defined event types, the classification of the scenarios depends on the specification of the events.

The first criterion is only used in partial scenario models. Due to the vast number of possible different reactions on an initial event it is not feasible to model the whole system in this way (see e.g. the use of scenario diagrams in Fusion and [Kilberth93], figures 14 and 48). The fourth possibility is the other extreme: a very rough classification covers all possible scenarios, yet the types cannot be modelled in depth concerning alternative courses, intermediate interactions, event parameters and other details (see e.g. BON, where only for some scenario types a high-level diagram is made; a more detailed modelling of scenarios is considered as being too complex [Walden95, page 170]). The third possibility is chosen

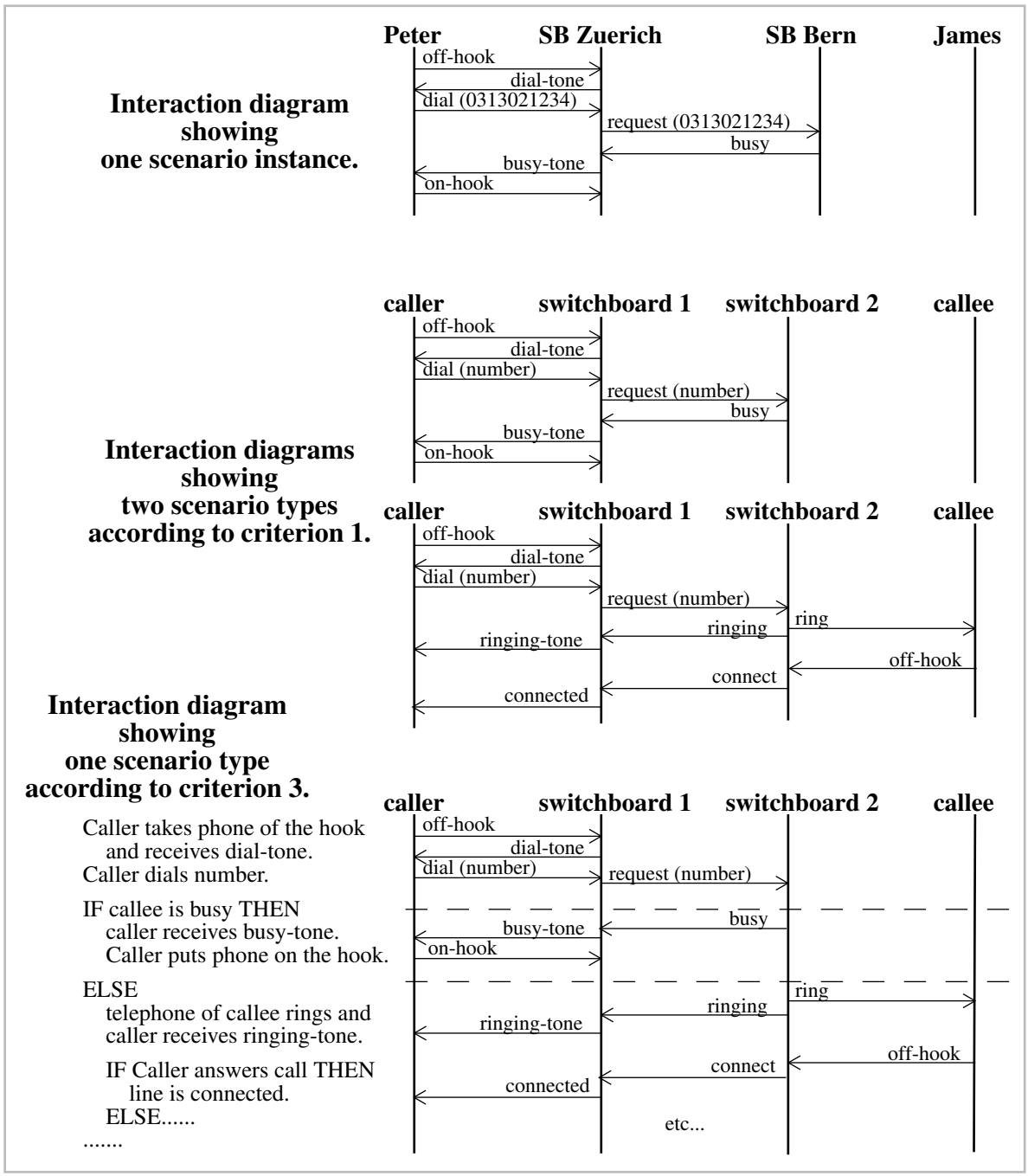


Figure 4: Modelling scenario instances and scenario types

by methods like Fusion and MSA, where a scenario contains no or only restricted intermediate interactions. So one scenario type contains all the scenario instances which start with the same input event type, but different output event types are possible. When a scenario may contain longer sequences of interactions, the third possibility may lead to very few, yet very complex scenario types (OOSE mentions as an example a telephone system, where most scenarios or use cases start with the event “handle lifted”). Therefore, methods like OOSE prefer the second possibility for the classification of scenarios. They leave it to the judgement of the developers to decide which scenario instances are considered as variants of one and the same scenario type, and which ones are handled as separate scenario types.

Possible criteria for the end of a scenario

The classification schemes differ also in their criterias concerning the end of one scenario. Of course, we can say that one scenario type models one system function or all the immediate reactions of the system to an initial input event. Yet there are different possibilities as how to define when these reactions of the system end and when the next system function starts. Possible criteria are:

a) One scenario consists of one input event and zero, one or several output events. Intermediate interactions between the agents and the system are not allowed (see e.g. system operations of Fusion or essential activities in [McMenamin84]).

b) A scenario ends as soon as a new essential system state is reached, i.e. a system state which is used in the system life-cycle model (see e.g. scripts in OBA).

c) Intermediate interactions are allowed as long as further input events cannot trigger themselves scenarios of their own (see e.g. business transaction in MSA-GfAI).

d) The whole course of input and output events that is perceived as belonging together, as being part of one and the same overall task, is modelled as one scenario (see e.g. use cases in OOSE).

e) The whole life-cycle of an object or a system, i.e. the sequence of events from the initialisation until the shut-down, is considered as one large scenario.

The only really precise criteria for the end of a scenario is the first one. It also satisfies the notion of having perfect technology within the system and thus no duration for the execution of scenarios. Yet its disadvantage is that it necessitates that the input event has in its parameters all necessary information that the system needs for this system function from the agent, because several interactions between the system and the agent for obtaining further informations would result in several scenario types. Also the system boundaries towards further agents influence the classification, as is shown in figure 5.

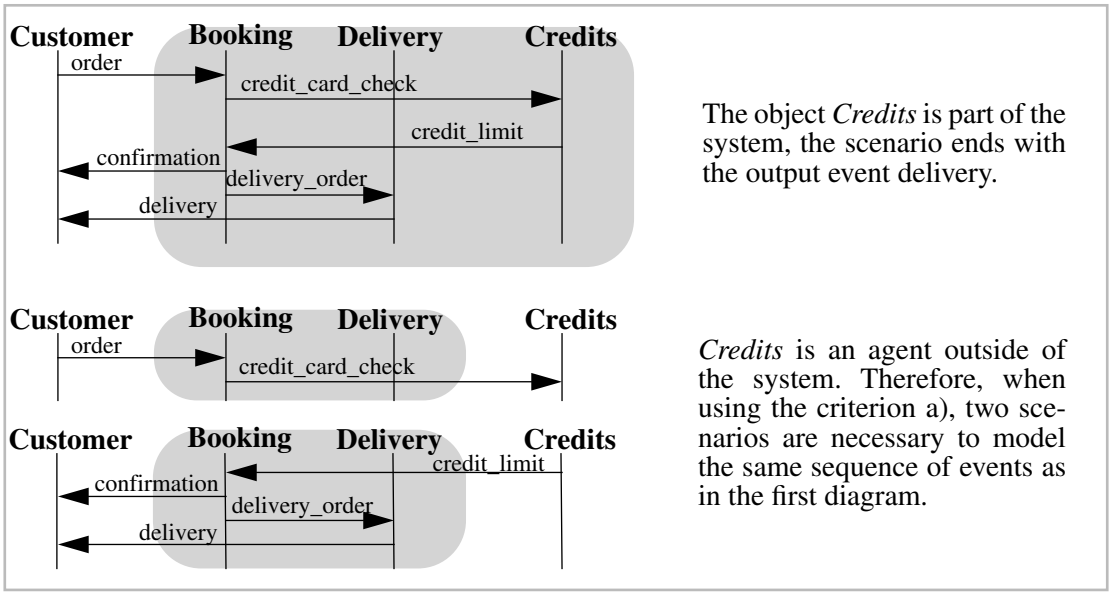


Figure 5: Determining the end of a scenario

2.3 Notational limits in modelling interacting objects

When modelling interactions, we would like to show many different aspects all at once. We name only a few of them:

- the names of the system, agents or objects involved in the interactions
- the dimension of time, i.e. the order of events
- all possible sequences in which the events can occur together with the corresponding conditions

- the events with their parameters and possible return values
- the different states of the objects or of the system and the state changes

Yet any diagram is limited to two dimension. Of course, we can decompose diagrams. We can also use annotations, maybe even supported by a hypertext tool. But the restriction remains. We always have to choose a few main aspects we want to focus on; the others need to be neglected or dealt with by annotations. In the war for the 'best' method or 'best' tool, it gets often forgotten that depending on the type of system we are modelling, on the momentary intent behind the model, and on personal objectives, certain techniques are better suited than other ones. Moreover, some views can even be automatically transformed into other ones.

In the following paragraphs we are going to discuss possible aspects one might like to describe in models showing object interactions, and we will show which modelling techniques focus on which aspects.

2.3.1 Modelling the dynamics versus modelling the statics

According to standard IEEE 610.3, a *dynamic model* shows the change that occurs in a system in which there is change, such as the occurrence of events over time or the movement of objects through space, whereas the *static model* shows those aspects that do not change over time. Also when modelling the global behaviour of a system, we have to distinguish between those models that show the static aspects of interactions and those models that show the dynamics of the behaviour.

As a *static interaction diagram* we define an interaction diagram that shows all possible interactions between its objects; it does no contain any information concerning the sequence of these interactions or their preconditions. Collaboration graphs (see figures 44 and 45), event flow diagrams of OMT listing all possible events between certain objects (see figure 34), and the communication associations in the object model of OOSE (see figure 6, the arrows denote that over these associations events can be sent, the events themselves are not listed) are examples for static diagrams showing interactions.

A *dynamic interaction diagram* shows one or several possible sequences of interactions between several objects (see e.g. figures 20 and 13). State transition diagrams also belong to the dynamic models; they show all possible sequences of interactions with one specific object.

Often in the static model object types are used whereas the dynamic model shows object instances. Yet these object instances are often rather used as prototypes of their object types, the concrete values of these object instances are not important. Considering the prototypical nature[6] of the objects in the object diagrams, the distinction between object instances and object types gets blurred. The formulation or the symbol for "object x" is interpreted as "an object of type x" when interactions are concerned, and as "object type x" when e.g. the inheritance hierarchy is specified.

2.3.2 Dynamic models: transitions between states versus interactions between objects

An event in an object-oriented system has two aspects:

- it is an interaction between two objects, modelled by interaction diagrams or equivalents,
- it is a transition between two states of the object that receives the event, modelled by state transition diagrams or equivalents.

We can compare this to looking at a coin from different angles of vision (see figure 6). At each angle there are certain parts we see better and other parts we see less. If we look only at one side (e.g. at a two-dimensional object interaction diagram), this side is very clear to us, yet the other side (the different states of these objects) is totally hidden. If we hold the coin very near to our eyes with the edge towards us, we can get a glimpse of both sides, yet none is very clear, only the edge (events) is really visible (this is somehow the case in the life-cycle expressions of Fusion).

6. See also object-oriented languages like SELF [Ungar91], which do not differentiate between object types and object instances; see also diagrams like in figure 30, which show interactions between instances as well as further associations between the corresponding types.

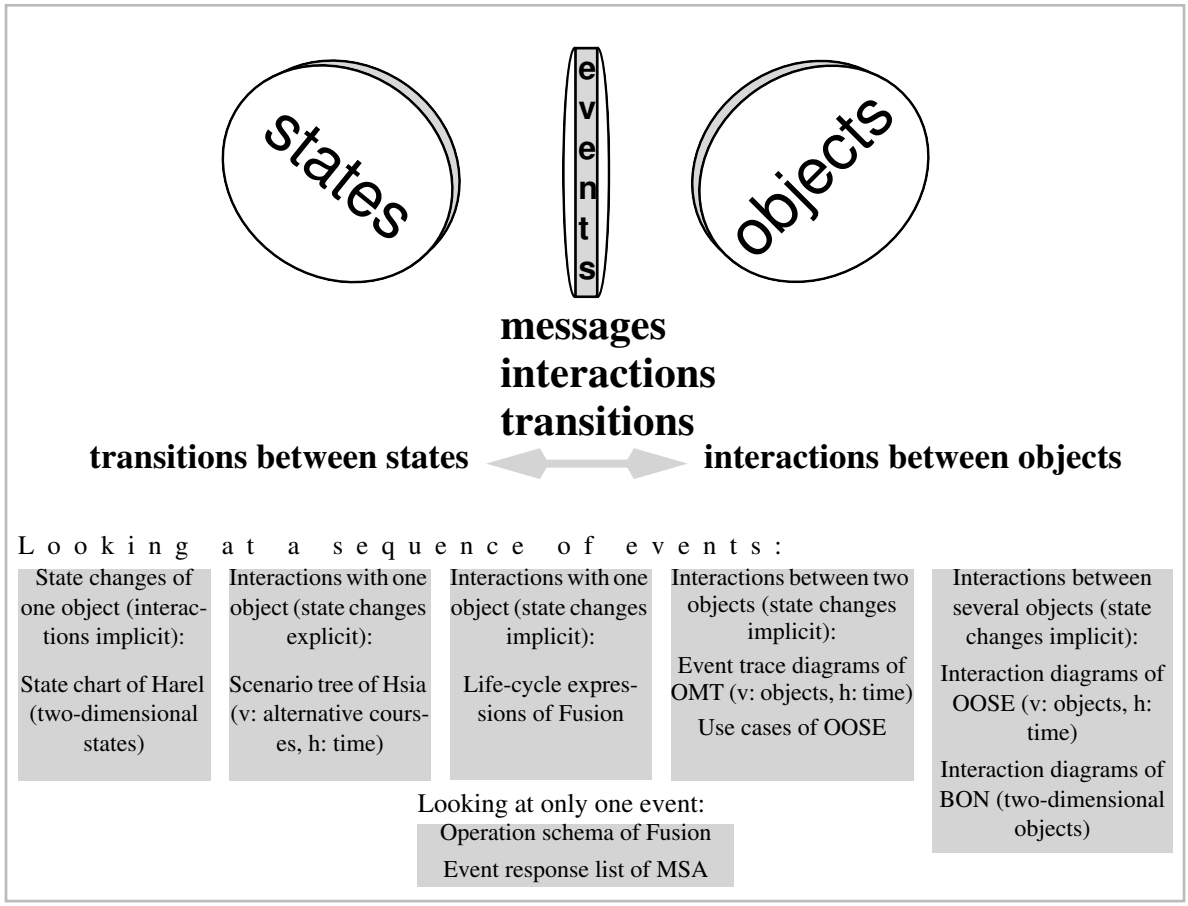


Figure 6: Transitions between states versus interactions between objects

The two counterparts

State transition diagrams show the whole life-cycle of one single object, i.e. all the events the object can receive in all its states. If the diagram is also annotated with the events the object sends and the recipients of these events, all the state transition diagrams together with a list of external events define the whole dynamic behaviour of a system. Interaction diagrams that show the interactions caused by a specific external event can be derived from a complete set of annotated state transition diagrams.

When we make for each external event an interaction diagram that shows all necessary interactions between objects, and annotate this diagram with the preconditions that must be fulfilled by the objects involved, we also get a complete model of the dynamic behaviour of the system from which we can derive the state transition diagrams of the individual objects. Figure 7 shows the two possible views: life-cycles and scenarios.

For system development both views are necessary. If we design the classes, we need state transition diagrams. If we focus on the system functions and the overall design of the interactions, we prefer interaction diagrams. In some methods, both views are explicitly modelled; yet often one view does not contain all the information necessary to derive from it the other view. Other methods combine the different techniques, e.g. Fusion uses life-cycle expressions (for the life-cycle of the system as a whole) and preconditions (referring to the life-cycles of individual objects) to specify the possible sequences of system operations.

Another reason why changing from one view to the other one is often not possible is that sometimes only the most important events are included in the state transition diagram of an object (e.g. leaving out all events that can occur in any state) or in an interaction diagram (e.g. leaving out all events that an object sends to itself). Also often the object states in the preconditions of an interaction diagram are not described in the same manner as in the state transition diagrams.

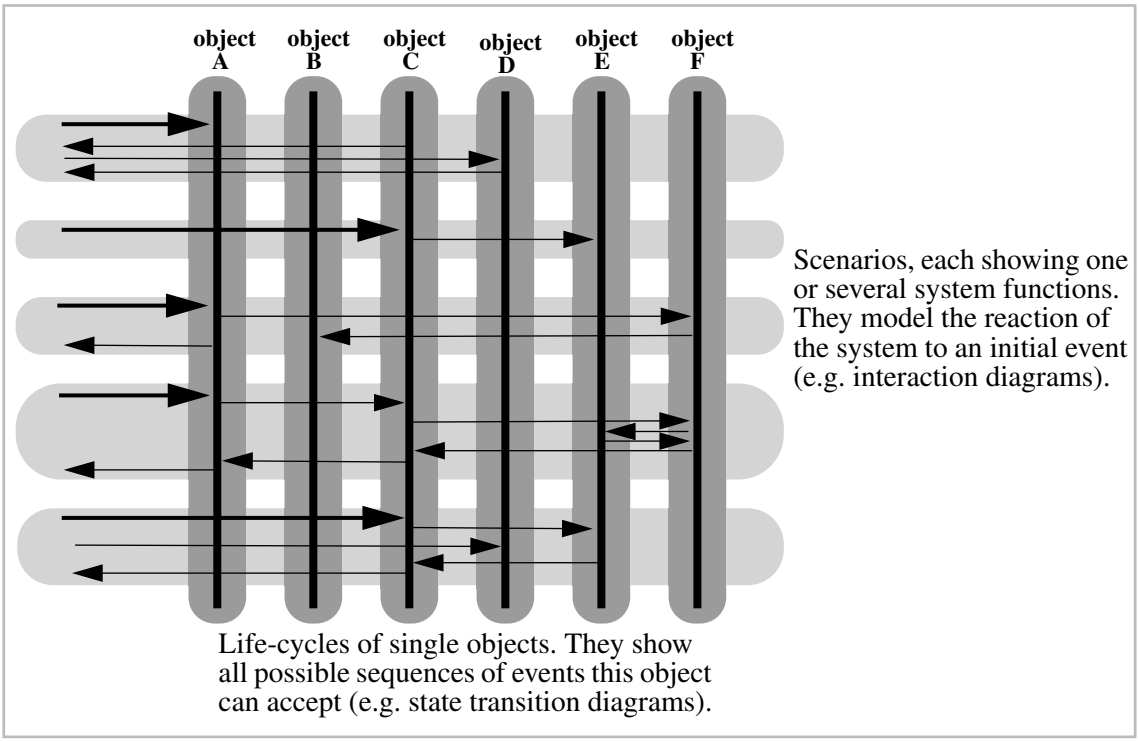


Figure 7: The dynamic model of a system: life-cycles of the objects or scenarios of the system functions

Figure 6 gives an overview on the different possibilities to combine objects and states, and it mentions also examples of modelling techniques.

The system life-cycle

For the special case where the modelling effort focuses on the interactions between a system and its environment, both, an interaction diagram that is enlarged to include all interactions (in the form of an event trace diagram with script or in the form of regular expressions) and a state transition diagram of the system as a whole can contain the identical information and can show the same view, namely the whole system life-cycle. For this reason several methods define automateable transformations between corresponding notations, e.g.:

- life-cycle expressions --> state transition diagram (Fusion)
- scenario trees --> regular grammar --> state machine ([Hsia94])

Hsia uses the transformations to verify the consistency of the dynamic model of analysis, whereas in Fusion the transformation is one of several steps for implementing the system life-cycle, which is specified by regular expressions.

Assertions and rules

Instead of using state transition diagrams or interaction diagrams, the dynamic behaviour of a system of interacting objects can also be specified by using rules and assertions. This is done in FORAM, where for each object type not only the operations are defined, but also any business rules that affect this object. In the domain of information systems, business rules are often already formulated and can easily be allotted to the rulesets of the object specifications; also they are better understandable to the users verifying the models than a set of state transition diagrams. During design, assertions on the object operations are then derived from the rulesets and any interaction diagrams. Of course, also other methods than FORAM use assertions in the object specification during the design, often replacing earlier dynamic diagrams by assertions.

We may mention here also the approach of Odell: the relationships in the class diagram may be annotated by different types of business rules, some of them also expressing dynamic aspects [Odell93].

2.3.3 Interactions between objects: time-line diagrams versus object diagrams

Time-line notations

Time-line diagrams

Time-line diagrams are two-dimensional graphs where the vertical axis is dedicated to the time and on the horizontal axis the different objects are listed (see e.g. figures 13, 14, 29 and 33). Sequence information is found in two places: a basic sequential ordering of the interactions is given by the order of the arrows in respect to the time axis. Iterations and alternations are shown by additional symbols and/or by pseudo-code that is written to the left of the diagram, also called script (not to be confused with the scripts of OBA). Historically this notation comes from the telecommunication industry (message sequence charts as specified by the ITU-T Z.120 standard).

Time-line diagrams have the advantage that they emphasize the time related aspects and concentrate only on dynamic aspects. The possible sequences of events can be seen at a glance. Furthermore, reading a text from top to bottom, even with iterations and alternatives, is very familiar not only to engineers but also to users coming from other domains. The disadvantage of time-line diagrams is that for very complex scenarios, involving either many objects or many different courses, the diagram becomes easily confusing. Also most methods do not provide any decomposition for time-line diagrams.

Tables

OBA uses scripts (not to be confused with the annotations to time-line diagrams often also called scripts) to model scenarios. These are tables, the rows show the interaction and the objects involved, the vertical axis shows the sequence of time (see figure 43). OBA-scripts normally only show one possible sequence, so no further information for additional courses is necessary.

Only text

When the diagram only shows the interactions between a system and its environment, then the graphical part can be replaced by a textual description. This text may be in structured English, or even in plain text as e.g. in the use cases of OOSE (figure 10). Yet still the vertical axis shows the flow of time.

Object diagrams

Object diagrams represent the objects as two-dimensional symbols and not as mere lines. Therefore they use both dimensions to show the objects and interactions involved in a scenario. The dynamic aspects are only described by numbering the arrows and by additional text. The information concerning the different possible sequences of interactions is found in the textual description. Fusion duplicates some of the sequence information by having a more complicated numbering scheme, yet these numbers can only represent simple dynamic behaviour (see figure 20). Fusion also enforces one textual description for each method, which results in further redundancy between the graphical and the textual part of the diagram, whereas most other notations suggest to have only one textual description for the whole interaction diagram (e.g. BON, figure 26).

The advantage of object diagrams is that they look very similar to class diagrams, so people do not have to get accustomed to two very differing notations. Also the grouping of objects (e.g. figure 27) or the decomposition into several diagrams (e.g. figure 19) is quite easy. Yet the dynamic information is not very obvious. It is mostly in the textual part. There, of course, are no limits to describing the dynamic aspects into details.

Modelling further associations between objects

Object diagrams can be used for both, for showing the dynamic and the static aspects of behaviour. These aspects can also be mixed. E.g. we can make an object diagram that shows all interactions involved in one specific scenario, but also all further associations between the objects involved. So also notations for showing visibility can be integrated into the interaction diagrams (see e.g. object diagrams of Booch, figure 30).

Transformation into time-line diagrams

Time-line diagrams emphasise the *call-sequence* and thus the dynamic aspects of the system. This is in contrast to the object diagrams which emphasize the *call-structure* and thus the static aspects of the system. These two types of diagrams can be converted automatically into each other, if they are limited to show the objects, interactions and possible sequences, and if no further aspects are mixed in. There is much arguing as to which kind of interaction diagram is the better one. At a closer look, these arguments stem from a different weighting of the disadvantages, from differing backgrounds and from differing tastes.

2.3.4 Some further aspects of interaction diagrams

Implicit versus explicit return events, flow of control

Whenever we model requests, we have to choose between:

- modelling the return event from the request explicitly as an arrow for itself (e.g. some event traces of OMT, figure 33),
- including the return event in the call event (e.g. the interaction diagrams of OMT and Fusion, where the label of the message includes always both, the name of the operation to be called with its parameters as well as any return values),
- neglecting the return event at all (e.g. object scenarios of BON).

When the return event is modelled explicitly, the *control flow* and the *scope or duration* of the operations can be deduced from the order of the events, as long as it is clear that requests and interprocess interactions can be assumed. When the return events are modelled implicitly, then the flow of control respectively the scope of the operations need to be shown explicitly. OOSE shows the scope of operations by rectangles (figure 13), OMT uses the same symbol to show not the scope of the operation but the flow of control (figure 33), and Fusion uses a special numbering scheme for the sequence labels to indicate with them the flow of control and the order of the return events (figure 20). When we model concurrent systems using notifications, then the flow of control and the scope of the operations can no more be deduced from the sequence of the events.

Great confusion may arise, when it is not clear, if a diagram is overspecified because it shows both, return events and control regions when only requests are allowed, or if it just uses both, requests and notifications. The interpretation of interaction diagrams may also get difficult, if it is not clear, if a sequential or a concurrent system is modelled.

Inheritance hierarchies of objects

If time-line interaction diagrams are used for the detailed design, it is possible to add annotations concerning the object inheritance hierarchy. For this the lines of the objects are replaced by rectangles. If an object inherits operations, its superclass is shown by a rectangle inside its own rectangle, and the messages get adorned with the symbols “i” for implementation and “d” for declaration. Depending on where the arrow points at (inner or outer rectangle) and where the adornments are placed, we know where the operations are declared and where they are implemented [Baklund95]. Figure 52 shows an example.

Real-time constraints

When interaction diagrams are used for higher level models or if time is not a delimiting factor, real-time aspects are not reflected in the interaction diagrams. All possible sequences of interactions are modelled, but the effective time used up by an interaction or an operation is not considered. However, it is possible to adorn the interaction diagram with the worst-case assumptions concerning the execution time of operations and their operation calls.

Modelling all possible courses versus focusing on the main course

Modelling with interaction diagrams all possible courses of all scenario types is often not feasible. Therefore, in order to avoid too many or too complex diagrams, certain methods recommend to concentrate on the most important or most complex scenario types, or to show only the main course of execution and maybe some important exceptions (e.g. object scenarios in BON, scripts in OBA, interaction diagrams in OOSE). In contrast to that, a complete model of the global behaviour containing all scenario types and all possible courses of execution is made in Fusion.

Abstracting away low-level details

To make a diagram more easily readable we may neglect the low-level interactions and objects that are not considered as essential. This may be done in two ways:

- by leaving out certain interactions, i.e. we do not show all the interactions with further objects or agents a specific object or system would need in order to respond to a certain event,
- by using conceptual events, i.e. we abstract away effective interaction mechanism be leaving out mediator objects like interface objects and collection objects; the effective technical events are replaced by a more abstract conceptual event that is not found like that in the implementation and may connect objects that do not interact directly in the implementation.

It is not always clear if a specific modelling technique or diagram shows all the interactions, or if some interactions are neglected for the sake of simplicity. Also it is not always clear if the interactions shown are on a technical or on a purely conceptual level.

Furthermore, some or all of the event parameters may be omitted. This is done in most notations using regular expressions (e.g. life-cycle expression in Fusion, path expressions in PEG), but also in some interaction diagrams (e.g. object scenarios of BON, event traces of OMT). Of course, this necessitates that a complete specification of the events is found somewhere else.

Structuring of diagrams

To avoid too heavily overloaded diagrams we have three possibilities:

- we choose a classification scheme that avoids too complex scenarios,
- we deliberately aim at incomplete interaction diagrams,
- we have some notational aids for structuring and decomposing a diagram,
- we have several models which represent the interactions on various abstraction levels.

Fusion e.g. allows to have multiple operations schemas for one system operation, each of the schema describing the system response for another set of preconditions. The interaction diagrams may be decomposed into arbitrary many diagrams, one for the system operation itself, the other ones for the more complex methods. Another approach is chosen in BON: object instances may be grouped into object groups; these are used to simplify the object scenarios. The last one of the possibilities mentioned above is not supported by any of the investigated methods.

2.4 Summary

An overview on the different concepts and their most essential relationships as presented in this chapter is given in figure 8. For the sake of simplicity, not all the details mentioned in the text are represented in the graphic.

Figure 8: Overview of the concepts for modelling global behaviour

3 Overview on various methods and modelling techniques

3.1 Introduction

The following overview summarizes various methods that provide modelling techniques for describing global behaviour. To make such an overview is not a trivial task, as the essential differences often lie in the details. Especially the definitions of terms like trigger, motive, event, message, stimulus, action, service and operation are different from method to method. Also the mapping between these terms, i.e. how they relate to each other, is not the same at all. One consequence of this are similar looking diagrams not modelling the same facts and similar notations not having the same expressiveness and precision. Another consequence is that the overview suffers from the following weaknesses:

- No simple classification or direct comparison as is it is often done in overviews on static models has been possible. Concerning the static aspects of object modelling their is much more common agreement than concerning global behaviour and dynamic aspects. Here, we not only have differences in the symbols (as e.g. different symbols for modelling inheritance), in the terms (like using class instead of object type), and in the semantic elements offered (like offering link-classes or not). Much worse, the basic concepts of the different methods are often so distinct that they cannot be compared directly and that the terms cannot be mapped onto each other.
- No general meta-model and no meta-models of the individual methods are provided. Most publications of methods do not offer a meta-model of the basic modelling concepts. Not that meta-modelling would not have been known for quite awhile[1], but most methods[2] are only described quite vaguely by text, some definitions and examples. Due to the missing meta-models, there are often inconsistencies and inexactitudes in the concepts and terms; these then necessitate personal interpretations and therefore any meta-models created by someone else than by the authors of the method only reflect these personal interpretations. Having no meta-models of the individual methods, we also did not make a generic meta-model for the overview. An attempt for such a generic meta-model can be found in [OMG92b], though not many methods have been described based on this meta-model[3].
- No complete description of the goals of the different models has been included in the overview. Though the goals of the different models were of great importance, not all the methods mention them explicitly, or they describe them only very vaguely or even contradictory. Most often, the methods mainly offer a set of notations useable for various purposes.

In the first chapters we summarize some of the well-known methods that have found acceptance also in industry. To round up the overview, we add some further interesting notations not mentioned in the previous chapter. Not all of the following methods claim to be object-oriented, yet they all offer modelling techniques that are of great interest in the context of object-oriented analysis. This is an overview, not a reference manual. Therefore certain details are omitted or simplified.

3.2 OOSE

The summary is based on [Jacobson92] and [Jacobson95], though these sources are in their description of use cases and interaction diagrams quite vague and in many aspects incomplete.

3.2.1 Terms

Actors: Actors are objects in the environment of the system which interact with the system. They define the roles users can play.

1. Many companies have made meta-models of their modelling techniques and of published well known methods in order to get a common agreement on their interpretation of the method, as e.g. [Briod93].
2. An exception will be the unified modelling language (UML) of Booch et. al. Its notation will be based on a meta-model (see [Booch95]).
3. An exception is FORAM as described in [Graham93].

Stimuli: The communication between objects is modelled by stimuli. A stimulus (equivalent to what in many methods is called an event) is sent by one object to another object in order to stimulate some behaviour in this object. It is either a **signal** (interprocess) or a **message** (intraprocess).

Use case instances: A use case instance is a sequence of related transactions performed by an actor and the system in a dialogue. A use case thus concentrates on the interactions observed at the user interface. It is an external model of the system as a whole. Though internal objects may be mentioned, the use case does not model or elicit the internal interaction structure.

Transactions: A transaction starts with a stimulus from an actor and finishes when its use case instance waits for the next stimulus from the actor. Transactions in OOSE could be compared to the system operations in Fusion.

Use case classes: A use case class consists of similar use case instances and contains a specification of the possible transactions of the use case. The whole set of all use case classes specifies the complete functionality of the system and gives a dynamic and black box view of the system.

3.2.2 Mappings and classifications

Grouping of use case instances into use case classes: Several use case classes may start with the same stimulus. As a consequence, it may happen that the class of a use case instance can only be determined after its complete execution. Which execution paths are grouped together into one use case class is up to the developer.

End of a use case: There are no general rules when a use case ends. If a sequence of transactions is modelled as one or as several use cases, is decided by the developer. Complexity is one possible criterion.

Common parts of use cases: Common parts of use cases can be extracted into **abstract** use cases.

Aggregation of use cases: In order to avoid any bias towards functional decomposition, OOSE does not offer any aggregation of use cases (in contrast to OMT that knows an add-relationship between use case instances).

Use cases and subsystems: Objects can be grouped into a hierarchy of subsystems, yet in the references mentioned above no precise description is given how the use case model is used together with subsystems.

More abstract use case models: OOSE suggest to make for each abstraction level a complete use case model and to have traceability between these levels. A table is used that shows which use case from one level corresponds to which use cases of the other level. However, the different use case models can not be integrated into one model.

Transactions versus system state transitions: A use case can also be looked at as an incomplete state transition graph where each stimuli performs a state change of the system. The transactions of the use case correspond to the transitions of the state transition graph. These state transition graphs are implemented by controller objects.

Use cases and the user interface: It is assumed that the sequence of the external stimuli and the classification of the use cases does not change between requirements analysis and design. The initial use case model is already supplemented by a UI-prototype. The interaction diagrams and all other models of OOSE are based on the use cases of the requirements model.

3.2.3 Notations

The use case model

The use case model consist of a narrative description of each use case class and of a diagram that shows the relationships between the use cases.

Possible relationships between use cases are:

- **Extends**: for modelling rare variants, subcourses or optional parts of a course. The extend-use case inserts itself into the original use case at instantiation time when certain conditions are fulfilled.[4]
- **Uses** (see example in figure 11): for extracting parts that are similar into **abstract** use cases.

These cannot be triggered directly from the user, they are only executed as part of a concrete use case. The **concrete** use cases insert abstract use cases at instantiation time.

The narrative description contains the **basic course** of transactions, and in several additional paragraphs the **alternative courses** of transactions (see example in figure 10). Alternative courses show variants and contain error handling. The basic course contains more than only one possible sequence of transactions; some of the transactions may well be conditional or repetitive. What is expressed in the basic course and what is considered as alternative course is up to the developer. For each transaction, the user actions (stimuli) as well as the reactions of the system on these stimuli are recorded. Often the transactions are numbered. In the description of the transactions the same terms may be used as in the object model. Object names may be emphasised; but this is not a necessity. In the case of abstract or extending use cases, the details concerning where and how this use case may be inserted into another use case may also described in a narrative way.

Interaction diagrams

Each use case is refined with interaction diagrams. These show the object and object services involved in that use case as well as the detailed user interactions with the interface objects. The objects need not correspond to the final implementation classes, most often they are larger blocks. Each stimulus in the diagram corresponds to exactly one operation of a block, even if internal to the block the operation may be dispatched onto different object operations in the case where the object is state controlled and not stimuli controlled.

There is one diagram for the basic course and one for each of the important alternative courses. Not all possible courses are modelled. Thus one single diagram only has a small amount of optional or repetitive parts. The interaction diagram is a time-line diagram with the following elements (examples see figures 12 and 13):

- *Blocks or objects*: These can be entity, interface or controller objects, yet no low-level objects. Objects of the same type can be drawn with several lines or collapsed into one line.
- *System border*: All external actors are collapsed into one special line. The communication with the actors is only by signals. Most interaction diagrams start with a stimulus from an external actor. There may be an arbitrary number of signals crossing the system boundary on one interaction diagram.
- *Stimuli*: Messages (requests) and signals (notifications) are distinguished by different symbols. Each stimulus is labelled with its name.
- *Parameters:* The parameters of the stimuli are normally listed together with the name of the stimuli.
- *Operations:* Rectangles on the lines denote the operation that is triggered by the stimuli and show the duration of the operation. Return messages may (e.g. to emphasize a return parameter) but need not be explicitly shown.
- *Textual description*: A short description (structured english or pseudo-code) of the algorithm is provided to the left of the diagram. This description also contains pseudo-code constructs for *repetition* and *condition*.
- *Probes*: Probes are special symbols in the interaction diagram that denote where another use case can be inserted (for the extend-relationship between use cases).

State transition diagrams

State transition diagrams are recommended to help implementing the controller objects. Any notation may be used.

3.2.4 The use of the models in the development process

To define the requirements and to discuss them with the customer, a use case model is made together with a domain object model and a first interface prototype. For finding the use cases, also storyboarding

4. Some critics doubt the usefulness of this modelling construct and of the distinction between extends and uses, see e.g. [Firesmith95], [Rumbaugh94b] and [Booch95].

techniques may be used. These do not concentrate on general use case classes but on individual instances. The use cases are also used to define the responsibilities of the individual objects in the analysis object model. This is derived from the problem domain object model and extended by interface and controller objects. During design, the objects or blocks are then further specified by the definition of the operation signatures which are determined by the interaction diagrams. The goal of the interaction diagrams is to define the protocol of the blocks and the parameters for each stimulus. The interaction diagrams are also an important tool to stabilize and rework the architecture of the system before any detailed internal design of the blocks and their classes is done. The use case model is also reworked during design because the interaction diagrams lead to a homogenization of the original description of the use cases. It is thus not assumed that the use case model is stable at the end of requirements analysis.

Use cases reflect the user's view, whereas the objects reflect the developer's view and are derived from the use cases. Though the identification and definition of the objects is said to be straight forward, it is also mentioned that during robustness analysis and interaction diagram design the objects have to be harmonized and homogenized so that they support all and not only some use cases in a reusable way (reusable objects are the *entity* and *interface objects*, use case specific objects are the *controller objects*). The use cases drive the development activities, from delimiting the system to testing the application. Use cases are used for finding the initial requirements as well as for defining the user interface, they neatly fit into the user-centred design. Yet for all this, the underlying assumption is that those use cases defined at the beginning of the project will be the same as those of the final high-level view, and that already the initial use case model contains a high-level specification of the user interface.

3.3 Fusion

The summary is based on [Coleman94].

3.3.1 Terms

Events: "An *event* is an instantaneous and atomic unit of communication between the system and its environment. An *input event* is sent by an agent to the system; an *output event* is sent by the system to an agent."

Scenario: "Sequence of events flowing between agents and the system for some purpose".

Agents: Agents are active entities in the environment with which the system communicates. Agents model human users, or other hardware or software systems. The system itself is nothing else than the agent that is being analysed.

System operations: "An input event and the effect it can have are called a system operation". In other words, these are the services offered by the system as a whole. They are atomic. Because only sequential systems are taken into account, at any point in time only one system operation can be active. A system operation is always triggered by an input event from an agent. It can neither be triggered by another system operation, nor by any active object internal to the system (e.g. a timer).

Methods: These are the services offered by individual objects.

States: The term state is used in two different contexts: for all possible system states (values of all variables) and for the states used in the state transition diagram that controls the correct sequences of the events.

3.3.2 Mappings and classifications

There is a 1:1-mapping between input events and system operations, and between the system operations and the methods of the controller objects. The parameters of the input events are attributes and objects of the object model or data elements only defined in the data dictionary. Each system operation is described by one operation schema (in order to simplify the postconditions these can be broken up into several schemas with differing preconditions). Each system operation becomes a method of an object (which serves as controller object) in the design and has one interaction graph which can be decomposed.

Output events are mentioned in the operation schemas of the corresponding input events, but they need not correspond directly to any methods or return values in the interaction graphs.

One system operation includes all those sequences of events that start with the same input event type. A system operation contains only one input event, the triggering event, and ends as soon as all internal state changes and corresponding output events have occurred. Longer sequences of events are described in the system life-cycle. The regular expression used for its specification can be decomposed into several expressions.

3.3.3 Notations

The interface model of analysis consists of the life-cycle model specified by regular expressions and the operation model specified by operation schemas. Furthermore event trace diagrams are used to sketch certain scenarios in order to help finding the system operations. The event trace diagrams are not suited to model the whole system life-cycle and do not belong to the final system documentation. In the design, global behaviour is modelled by the interaction graphs. In the implementation, a state machine is used to implement the system life-cycle as specified in analysis.

Event trace diagrams for scenarios

These diagrams show one possible sequence of input and output events between the system and its agents. Iterations can be represented (see figure 14), but not interleaving or alternative courses.

Regular expressions for the system life-cycle

The life-cycle model is a regular expression (example see figure 15), consisting of:

- *input event names*, *output event names*, *local names* of regular expressions (the expressions can be decomposed into arbitrary many regular expressions, making the specification more readable),
- *operators* for concatenation x.y, alternation x|y, repetition x* or x+, optional [x], interleaving x‖y and grouping (x).

The most important rules are that output event must not be interleaved with input events and that interleaved regular expressions must not start with the same event.

Operation schemas for the operation model

One operation schema describes one system operation and contains the following information:

- *operation name* and textual *description*
- list of all those objects, attributes and relationships of the object model that are *read* or *changed*
- list of all those objects that are *created*
- list of all those objects, attributes and data elements that are *supplied* as parameters of the system operation
- list of all *output events* including the names of the receiving agents
- *preconditions* and *postconditions*

All the objects, attributes and relationships mentioned in the operation schema must be defined in the object model of analysis.

Common parts of system operations cannot be factored out, they are repeated in each system operation. System operations that can be triggered by an agent as well as by another system operation (e.g. producing a status report), are modelled twice.

Interaction graphs for the design model

One interaction graph represents all possible sequences of method calls necessary for the execution of one system operation. The graph contains the following information (examples see figures 19 and 20):

- *objects* and *collections of objects* (a message to a collection means that the method is invoked for each object of the collection, a collection object is modelled as an ordinary object),
- *messages*: method calls with parameters and return value,

- *sequence numbers*: show the flow of control and give a vague idea on possible sequences (the precise information on the order of method calls is found in the description of the methods),
- *method descriptions*: for each method the algorithm is described including all method calls to other objects.

The interface objects are modelled for the output events yet not for the input event. The detailed user interactions as well as the mechanisms to check the life-cycle are not part of the design model.

State transition diagrams for the implementation

The state machine that controls the sequence of the events and decides if an input event is accepted or rejected is already specified by the life-cycle expressions. These expressions are transformed into a state transition diagram which is implemented by specific controller or interface objects.

3.3.4 The use of the models in the development process, weaknesses

The analysis model turns an initial requirements document into an unambiguous and precise set of models and defines the intended behaviour of the system. Requirements determination itself is not part of the Fusion process. The analysis model is expected to abstract away implementation details and to focus on the problem and not on the software solution. All external behaviour is modelled exhaustively. This enables consistency and completeness checks, yet makes latter changes that exceed minor details impossible and presupposes that the first choice of system operations is the best and final one.

The interaction graphs are used in the design to help allocating the methods to the classes and to document the implementation of each system operation. Basically it is assumed that the "external" view of the analysis model can directly be enhanced by the internal view of the design and that only minor changes in the external view are necessary. This gives a very straight forward development process with a high degree of seamlessness. Yet the method ignores some inherent limits to seamlessness (see also [Beringer94] for a deeper discussion of the limits of seamlessness).

In the following we list some weaknesses of the Fusion method. Most of them are due to the fact that the analysis model predetermines many important design decisions concerning the internal software structure. Yet these decisions cannot be taken properly when modelling the external behaviour on a conceptual level.

Internal state information in the operation schemas

In order to define the pre- and postconditions, internal state information needs to be modelled in the object model that is never used as input or output information and is hidden from the user of the system. This state information appears also in the read and change clauses of the operation schema and must be assigned to classes. This leads to complications in the transition to the design for two reasons:

- The final object model should satisfy the quality criterias for good encapsulation of operation and data (as they are even mentioned by [Coleman94] itself), yet the assignment of the internal state information to classes is made before the responsibilities and methods of these classes are determined. Where which state information is stored influences heavily the possible structure of the interaction graphs and limits their optimization. Of course, it is possible to rework the whole object structure during the design and to redo the analysis model, yet this is not very feasible.
- The postconditions of the operation schemas describe the reactions in a declarative way. When moving to a procedural specification, not necessarily the same attributes are optimal in the algorithms of the methods as have been used in the declarative descriptions.

Moreover, there is an imbalance between the treating of the system states used in the life-cycle model and of the system states used in the pre- and postconditions. Whereas the first ones are not defined and considered in the object model until implementation, the internal information necessary to define the system state used in the operation schemas must already be modelled in the object model of analysis.

No further user interactions during one system operation

Fusion assumes that the input events defined in analysis (conceptual events) correspond exactly to procedure calls (technical events) invoked by a user interface which is not further modelled in analysis. When the design models already exist or some people have already experiences with many similar sys-

tems, it is of course possible to make an appropriate analysis model with input events that correspond to object methods in the design. Yet in normal forward engineering, this normally is not the case; as a consequence, decisions that influence heavily the possible software solutions and the system architecture are taken when considering only an external high-level view of the system.

Most often we want to design user interfaces where the user can enter all the parameters of a system operations interactively. Yet for this the following problems arise:

- At any time only one system operation can be active. For multi-user systems (like information systems) this would mean that one user blocks the whole system until he has entered all parameters.
- If a wrong value of a parameter should not lead to the rejection of the whole input event, then already the interface objects need to check them. For this they either have to interact with the corresponding problem domain objects, or they need a copy of their content, before they send a message to the controller object that itself interacts again with the corresponding problem domain objects. Both solutions do not give an optimal system architecture, and worse, do not appear in the models of the design.

Life-cycle model

The recommended transformations of the system life-cycle into a state machine targets only at sequential systems. Distributed and concurrent systems cannot be implemented by the suggested approach. System operations may never execute in parallel and only one single life-cycle for the whole system is provided.

Those cases where the acceptance of an event is determined by both, the event types and their parameters (like the name of the agent that sent it or the object instances on which the events will have an effect), are very difficult to model appropriately by the system life-cycle and the preconditions. A certain amount of redundancy is unavoidable. This redundancy is carried on into the implementation, because:

- different objects are responsible for checking the preconditions and the values of the input parameters and for controlling the state machine,
- the states mentioned explicitly in the preconditions and the states derived from the life-cycle expressions are treated in different phases of the development process and are never homogenized.

No symmetry between input and output events

Output events most often correspond to individual objects or attributes without any parameters. In the operation schema there is no corresponding keyword to ‘supplied’. This inconsistency between input and output events continues in the interaction graphs where output events do not become the return values of the system operations. They are only parameters of some new methods of some interface objects. This lacking symmetry is due to the fact that the system life-cycle of analysis views the input and output events as notifications, whereas in the design the input events become requests.

Extended Fusion for requirements modelling

In [Coleman95] use cases as introduced by OOSE are integrated into Fusion. The use cases are modelled by

- a context diagram showing also the relationships (extend, uses) among the use cases,
- a textual description for each use case class defining its name, stating its goal and listing all the transactions involved in the use case,
- scenario diagrams showing use case instances.

The uses relationships allows to factor out common parts of use cases into full fledge sub use cases. In the scenario diagram, sub use cases appear as shaded boxes (see figure 16). The extend relationships allows to weaken assumptions made for a use case by treating special cases. The inserted transactions are no full fledge use case. [Coleman95] recommends the usage of the extend relationship for defining system increments for the EVO-Fusion development process [Cotton95]; first only the basic use cases are implemented, later on the system is enhanced to treat also the various extends.

The use case description does not specify the exact ordering of the transactions. This is done during analysis, when use cases are subdivided into system operations, and the system life cycle gets defined.

3.4 OBA

OBA stands for Object Behaviour Analysis. The method focuses mainly at determining the scope and the functional requirements of the system and at finding suitable objects for an object-oriented analysis model. OBA can be used in combination with other methods like OMT and Booch, which then provide the syntax for the analysis model and also a process for the design. OBA is described in [Rubin92].

3.4.1 Terms, mappings and classifications

An **event** occurs whenever an object invokes a service in another object. Yet during analysis, only those events are taken into account that cause one or more objects to experience a state change. Also only those state changes are considered that consequently affect the behaviour of the system[5]. Events thus reflect essential occurrences in the system or in its environment.

A **scenario** (also called use scenario) is a sequence of service requests and activities in order to accomplish some overall task. The developer is quite free which sequence of service requests he considers to be one scenario. For event-driven systems, a good starting point is to consider each external event type as one scenario. The only restriction is that there may be no noteworthy or essential state changes within one scenario, i.e. after each such state change a new scenario starts. Essential states are all those states that are modelled in the dynamic model. Thus each scenario equals to one transition or event in the state transition diagrams of the dynamic model.

Each scenario is documented by one **script** and contains a reference to a core activity area; these areas can be used to group scenarios. The possible sequences of scenarios are first defined by pre- and post-conditions (added to all the scripts) and then also shown in the dynamic models of the objects (state transition diagrams).

The participants and initiators in the scenarios are called **parties**. These are either outside of the system or correspond to one or several objects within the system. A party may correspond to a group of objects. Due to the differentiation between parties and objects, it is possible to keep these groups of objects even after the definition of a finer and more complex object model.

The behaviour of the parties that can be contracted for use by other parties are the **services**. There are four main categories of services: accepting notification, accepting information, providing information, providing a service. For providing a service a party performs several **actions**; these may invoke services of other parties.

3.4.2 Notations

Scripts

A script is a table with four columns:

- the party that is the *initiator*,
- the initiator's *action* that necessitates a service from another party,
- the *participant* that provides the service,
- the *service* (its name or a brief declarative description).

The reaction of the participant on a service invocation is expressed on the next line of the script, if it is considered as important enough to be listed at all. There exist two versions of scripts: scenario instances with concrete values for parties and objects as in figure 43, and scenario types annotated by *control flow annotations*. These are either textual annotations to the left of the scripts or are expressed by additional diagrams. They may show concurrency, repetition, selection and options.

5. A **state** represents a condition of an object during which certain physical laws, rules, and policies apply. Only those states that influence the behaviour of the system differently (i.e. not the same scripts are applicable in these states) are considered to be different in analysis.

Preconditions express what must be true in order that the script is applicable. *Postconditions* denote those state changes that have an influence of the applicability of future scripts. Pre- and postconditions are expressed in terms of state descriptions of objects, e.g. "overdrawn (account)".

References to goals, objectives, core activities and other scripts are listed as *traces*.

Glossaries

Glossaries are used for the description of *parties* (they specify also the objects that a party consists of), of *alias names*, of *attributes* (used for the definition of states and the description of services), of *services*, of *states*, of *reorganizations* and so on. They evolve during the scripting process and during object modelling. They enable to find gradually the optimal object structure without redoing all other scripts. Together with the scenario scripts and the object definitions, glossaries are an integral part of the analysis documentation of OBA.

Dynamic model

The dynamic model contains for each object a *glossary* of the essential states of this object and, if applicable, an *object life-cycle*. The glossary lists for all states their names, their definitions (boolean function over attributes and values), their textual description and their traces to the scripts. The definitions are derived from the pre- and postconditions and are expressed in terms of logical expressions over tuples of object and state description. An object may well have several states in parallel. The Object life-cycles are modelled by petri nets or Harel state charts.

3.4.3 The use of the models in the development process

The goal of analysis is to construct a model of the problem domain, i.e. a static and a dynamic model of the domain objects with traceability back to goals and objectives of the system. First the analysis context is set by identifying business goals, objectives and the core activity areas of the system. Typical scenario instances are chosen and scripted. Glossaries of parties, services and attributes evolve. Only in a second step are the objects determined and specified by using object modelling cards and recording the results in the glossaries. Building inheritance hierarchies and identifying relationships involves also the reorganisation of the object definitions which also include a list of all contracted services. The last step is modelling the system dynamics: state glossaries are derived from the pre- and postconditions of the scripts, state charts or petri nets are made for all important objects, state glossaries are updated. If necessary, the scenario scripts are restructured so that each scenario corresponds to one event in the dynamic model. Missing states and missing scripts are added. Also, the scripts may now be enhanced by control flow information.

The scripts have thus a twofold purpose during the development process: they support the learning process at the beginning of the project and help to find the problem domain objects, and they document a high-level view of the behaviour of the final system.

3.5 FORAM

FORAM stands for Financial Object-Oriented Rapid Analysis Method. The strength of FORAM lies in its process for determining the systems goals and in its approach to domain and system analysis. It uses task scripts developed in a workshop to capture requirements, and it integrates very well the users and managers during the whole analysis process. For the object model, the SOMA notation is used. FORAM is described in [Graham93] and [Graham94], SOMA in [Graham94a], business process modelling using basically the same modelling techniques is described in [Graham95].

3.5.1 Terms and notations

FORAM distinguishes the following three models:

- **Task object model** (TOM): *context model*, *message table*, and *task cards* (see figure 23). The objects are tasks and not business objects! At the beginning, the task object model is an external view of the system. But after the decomposition process, it also shows an internal view, yet not based on objects but on internal tasks. These are then used to find the objects and their methods. It is assumed that the task model reflects both, the business and the software

system alike.

- **Business object model** (BOM): *object model*, *class cards* (including rules and collaborators), *event traces* and *state diagrams*.
- **Implementation object model**: detailed object model of the code. It is programming language dependant and includes also low-level objects.

Whereas the task object model is mainly for capturing requirements and analysis, the business object model incorporates also the first steps of logical design.

Context model

The context model shows the system object, the external objects and the message flows between them; it may also show part of the task model. External objects are either actors (users adopting a particular role to interact with the system for some purpose), external systems, or something else that interacts with the system. The system itself can be considered as an actor being under closer consideration.

Message table

The message table lists the external messages of the system. The table has one message row for each interaction on the context diagram. Each row lists the *message name*, the name of the *trigger event* (may be a temporal or an external event), the *source* of the message (an external object), the *target,* the *information* sent or received (parameters of the message), the *expected result* (what the initiator expects to get or to happen), and the *goal* of the message.

Messages are dataflows exchanged between the system and the external objects. The message is from the initiator to its recipient, even if the actual dataflow is the other way round. The mapping between events, messages and tasks is very vague; messages appear only in the message table and not in the task specifications. The **goal** gives the characteristics of a message. It specifies the desirable states to be achieved and shows why the message has been sent. Goals complete the description of the system's scope and are the starting point to discover the tasks.

Task scripts

The task scripts (described by **task cards**) specify the individual task types. They include the following items: *name* of the task, *supertasks*, *component tasks*, *task body*, *associated tasks*, *exceptions*, *side-scripts*, *rulesets*. Task scripts are decomposed into **component scripts**. **Sub-scripts** show the specialisation of tasks. Exception handling and special cases are treated by **side-scripts**.

Tasks are first class objects in the task domain. Tasks are carried out in order to achieve the goals of the messages. They are considered as objects due to their prototypical character and because they can participate in specialisation, composition and usage structures. For the external view of the system, there is at least one task script for each message in the message table. Tasks are decomposed until **atomic tasks** are reached. Which tasks are considered as atomic tasks is determined by the developers and may change during the process. The process of task decomposition is called **task analysis**. Task decomposition is considered as being object-oriented, in contrast to functional decomposition combined with ERD-modelling. It is well suited for RAD-workshops, and it mirrors well business processes, based on communicating actors. The decomposition activity is purely in the task domain, which is orthogonal to operations and objects.

A **task** is an equivalence class of use cases, a **use case** is an equivalence class of **scenarios**. Task scripts, use cases and scenarios stand for three different abstraction levels.

Class cards and object diagrams

Class cards specify the interface of the objects and contain *attributes* (responsibilities for knowing), *operations* with *assertions* (responsibilities for doing), *relationships* (only one-way pointers out of the object!), *server objects*, and a *ruleset*. The object model models interface, domain and application objects. Any notation is possible, though the SOMA-notation for the object diagram is preferred, because it also shows rules and allows to model composite objects, so called layers.

For each object type, **rules** are defined which may be inherited like other features of an object. Rules need not be as precise as the **assertions** of the operations; these are derived during the design from the rules. Rules have the advantage that they model in a very understandable and readable way the business

rules and dynamic behaviour of an information system and that they reflect human reasoning. Furthermore, they are not bound to a particular logic and to the restrictions of this logic, and they can be fuzzy whenever the state of knowledge acquisition does not yet allow more details. Yet they are not suited to express the dynamics when formal correctness is already an issue in analysis. Rules can show global control, business rules (relating several attributes), dependencies between attributes and operations (triggers) and integrity rules, i.e. they can contain any kind of second order information.

Event traces and state charts

The dynamic aspects are completely specified by the rules, assertions and constraints on the class cards. Event trace diagrams (example see figure 24) and state charts are only used as a help to identify and specify the objects and their operations. Event trace diagrams may be annotated with comments concerning temporal constraints, decision processing etc.

3.5.2 The use of the models in the development process

In contrast to many other methods, FORAM includes also object-oriented requirements determination. This is mainly done in workshops at the beginning of the project where the *task object model* and the *preliminary business object model* are developed. The following steps are carried out in the workshops:

- **Defining primary business objectives:** determining and prioritizing the primary business objectives, defining measures for each objective. The objectives are needed to determine the system boundaries and later on to decide on the time-boxes for the RAD-development cycles.
- **Context modelling**: defining the system boundaries and developing the context model and message table. The goals and expected results become clear.
- **Task analysis**: decomposing and describing the tasks that lie behind the goals.
- **Building the object model**: retrieving candidate objects and methods from the task-scripts using text analysis. The task-scripts may be further refined, until no further objects fall out. Furthermore, the tasks-scripts may be decomposed until the level of the effective user interface is reached. The objects are recorded in class cards and in an object diagram. To verify the object model and to find the collaborators for each class, event traces are modelled and role-plays are carried through.

All the models developed during the workshop are further revised and refined when developing the implementation object models and the prototypes in the time-box cycles.

3.6 BON

The method BON is documented in [Walden95].

3.6.1 Terms

Events: An event or system event is “a stimulus that can change the course of action of a system and make it react”, “something to which a system will respond with a certain behaviour”. An **external event** “is triggered by something in the external world over which the system has no control”. An **internal event** “is triggered by the system itself as part of its reaction to one ore more external events”.

Scenarios: A scenario is “a script of a possible system execution showing the objects involved, which other objects they call, and the temporal order of these calls”.

States: The **system state** is the sum of all information stored in a system, it reflects the history of events. An **object state** is that part of the system state that has an effect on the future behaviour of one particular object. It may but needs not correspond to some attributes of the object, the data making up the object state needs not be stored in the object itself.

Messages: A message is an invocation of an operation on an object (feature call, message passing).

3.6.2 Mappings and classifications

Events

The system events of the final system are very low-level (e.g. command selection, sensor inputs, commits of transactions). As scenarios and event charts are only used to give a high-level overview of the

system and to detect problem domain classes, only those events that represent a more abstract stimuli are considered in the models of analysis. Yet there exists no precise concept of event abstraction.

Scenarios

A scenario is stimulated by either an external or an internal event, normally such an event is listed in one of the event charts. There may be further inputs from the user during a scenario not mentioned in any event chart. The name of a scenario may or may not be similar to the name of the event by which it is stimulated. There is also no direct correspondence between the events in the event chart and the message relations in the diagrams. Theoretically there would exist a client relation in the static model for any two classes that have a message relation between any of their instances in the dynamic model. Yet for the sake of simplifying the static model, not all client relations are shown, the less important ones only appear in the class interface specifications.

Because the dynamic model only contains diagrams for a few examples of scenario instances, their are no rules concerning the classification of scenario instances into scenario types. Also no guidelines are given when a scenario should end and the next one begin. A decomposition of scenario diagrams is possible by using object groups.

Object groups

Objects can be compressed into groups. An object group is defined as "a set of objects treated as a unit in some message passing context". It may group objects

- that receive all the same or a similar message (mirroring an inheritance structure in the system and incorporating polymorphism in the dynamic model),
- which are called in a sequence to do a more abstract step (see figure 27),
- of which all than one serve only as data containers.

An object may belong to different group hierarchies. The rules for the compression of message relations (omitting message relations, applying one message relation to a whole group) are analogous to the rules for client relations in the static model. Though BON emphasizes the possibilities for zooming and compressing, these concepts are only applied for classes, objects and relations, but not for scenarios and events, because the dynamic diagrams are only considered as auxiliary models which are neither exhaustive nor complete and only help to develop the class specifications.

3.6.3 Notations

Charts

The **event chart** is a list of important system events. For each event, its name (which may well be a sentence) and a list of possibly involved object types are recorded. Only a subset of all system events is considered, namely:

- *input events*: external or internal events which trigger essential types of system behaviour,
- *output events:* internal events that are triggered by special system states and are not directly related to an external event; other internal events of interest.

In the **scenario chart** all those scenarios are listed that illustrate important aspects of the overall system behaviour. For each scenario its name and a short textual description (only a few sentences) are listed.

For an example of the above charts see figure 25. There exist also charts for classes, clusters, systems and object creation. For more complex systems, separate event and scenario charts may be made for different subsystems or tasks.

Dynamic diagrams

Only a representative set of scenarios is further detailed in dynamic diagrams, these are called object scenarios (see figures 26 and 27). One object scenario describes the call structure of only a single execution path. Alternative courses are not included and are deliberately neglected in the dynamic model.

A dynamic diagram is the graphical representation of one object scenario. It contains:

- *Objects*, *sets of objects* and *object groups*.
- *Message relations*: no message names, no parameters or return values. Message relations

signify a potential call from a client object to an operation of some server object.

- *Sequence numbers*: represent time (only one possible sequence of potential message relations) and are at the same time the references to the entries in the scenario box.
- *Scenario box*: contains the name of the scenario plus a list of the most important steps of the scenario execution. One entry summarizes several message relations, no details are included.

The dynamic diagrams of BON are on a very high abstraction level. For the reason of simplicity, neither the diagram nor the scenario box show any conditional control or event type information (parameters, return messages).

External events like the stimuli of the scenario and other user inputs are denoted by special arrows and do not signify a call to an operation. Output events (respectively the invocation of interface operations) are not included in the diagram. Also return messages are normally not shown. If and when a return value or the control is passed back is too low a detail for these diagrams. Interface objects are not included in the diagrams, therefore the user interactions are not modelled into details. User inputs are notifications, all other message relations are requests.

An arrow to a set of objects means sending a message to one object of this set (in contrast to the semantic in Fusion). The intermediate steps of getting the object handle from the set object and the set object itself are not shown on the dynamic diagram. They are represented on the static diagram as (compressed) client relations and in the class interface specification as references to sets.

3.6.4 The use of the models in the development process

BON divides up the whole development process into 9 *tasks*. The three tasks for gathering the analysis information are: *delineate system borderline*, *list candidate classes*, *select classes and group into clusters*. The three tasks for describing the gathered structure are: *define classes*, *sketch system behaviours*, *define public features*. The three tasks for designing a computational model are: *refine system*, *generalize*, *complete and review system*. For each task the input sources, the deliverables and the acceptance criteria are defined. The order of the tasks may change, depending on the concrete circumstances of the project. Also not in every project all the tasks are needed. Furthermore, various *activities* are distinguished: *finding classes, classifying, clustering, defining class features, selecting and describing object scenarios, working out contracting conditions, assessing reuse, indexing and documenting, evolving the system architecture*. These activities appear in all the tasks though with differing weight.

For finding scenarios, BON recommends to look for *user tasks* (not to be confused with the tasks of the development process). A user task may then be broken down into *user actions* which become the scenarios. Yet this hierarchical structuring is not reflected in the dynamic models. The scenarios are used for the following purposes:

- delineating system borderline,
- finding classes and basic concepts, determining object operations, refining the system structure,
- documenting the final system.

Reversibility and seamlessness are leading principles in BON. Furthermore, constructing metaphors and contracting are emphasised in order to get a high quality software system that suits the needs of the user. The class interface specifications are the basic model, all other charts and diagrams are only additional aids to develop and to understand the contracts of the individual classes. The high-level abstractions of analysis are chosen in such a way that they can be used all the way down to the design and the code, thus simplifying the propagation of changes throughout the different models. In order that the dynamic model always represents the actual system, it is subject to continuous improvements until the final user metaphors and the best abstractions are found. Therefore, at no point in the development all the dynamic characteristics of the system will be modelled, the effort is concentrated on those aspects that are felt to be the most important ones and are needed in order to evolve the contracts of the classes. The documentation always reflects the actual system and not historical aspects of the different iterations in the development process (faking the ideal process). Thus most charts and scenarios produced during development are discarded at the end; the final documentation contains only those few scenarios that are considered as essential for helping other people to understand the implementation.

3.7 Booch and OMT

Booch and OMT both have a vague distinction between analysis and design, their notations can be used throughout the development process. The main focus of the methods lies on the static object model. Modelling the global behaviour is only done as far as it is necessary to find the static model. Both methods offer a large kit of notations with many optional details. It is up to the developer which notations he uses to what extent and for which purposes, and how he integrates the resulting models. The following overview does not go into the various details of the methods concerning modelling the dynamics and the global behaviour, we only mention the most important notations. The references are [Booch94], [Rumbaugh91] and [Rumbaugh94]. For experience reports on the use cases as defined by OMT see also [Loenvig95] and [Hansen95].

3.7.1 Notations

State transition diagrams of Booch

The diagrams are derived from Harel state charts and from the dynamic model of OMT. Each diagram describes the behaviour of one object type by specifying its essential state changes (see figure 28).

Object diagrams of Booch

Object diagrams contain information concerning associations, visibility and messages between objects. They represent a snapshot in time of an otherwise transitory stream of events. They show all messages and objects needed in a certain context. e.g. for a complicated interaction mechanism or for a system function. In analysis they are used to indicate the semantics of primary and secondary function points, in the logical design to enlighten certain key mechanisms. It is up to the developer what he models with the object diagrams and how many different static and dynamic elements he mixes into one diagram. An example is found in figure 30.

Interaction diagrams of Booch

The interacting diagrams do not contain additional information to the object diagrams, yet they represent the execution of a scenario as a time-line diagram. These may be used to express one single scenario instance or a whole scenario type. Further details like scripts to express different execution courses or symbols for showing the focus of control are optional (see figure 29).

Use cases in OMT

"A use case describes the possible sequences of interaction among the system and one or more actors in response to some initial stimulus by one of the actors. It is not a single scenario (a specific history of specific events exchanged among system and actors) but rather a description of a set of potential scenarios, each starting with some initial event from an actor to the system and following the ensuing transaction to its logical conclusion" [Rumbaugh 94b]. Use cases describe the system as a black box and focus on the externally visible behaviour. In one use case all those transactions are grouped together that are similar in nature. Within one use case arbitrary many interactions with several actors may occur.

The use cases are described in natural language (see figure 31). The description contains a *main case* and various alternatives or *subcases*. Common parts of use cases may be extracted by the *add-relationship* (see figure 32). The relationships are annotated with their cardinality (one, optional, many). The add-relationship replaces the uses- and the extends-relationships of OOSE.

The functional model of OMT

Any *operation* can be looked at from a black-box (external) viewpoint, or from an internal viewpoint. The external viewpoint is modelled in a declarative way by operation specifications which are similar to the operation schemas of Fusion (see figure 37). The internal view is documented by object-oriented data flow diagrams and object interaction diagrams (see figures 35 and 36).

In the system analysis model, only the top-level *system operations* are considered; these are invoked by interactions with outside actors.

The dynamic model of OMT

The complete dynamic model is specified by state charts of all those objects that undergo different essential states, and by tables for modeless objects. In the design the events correspond to procedure calls. In the analysis they can be more abstract.

Other diagrams like the event trace diagrams (see figure 33) for showing a particular sequence of interactions among objects in a single execution (scenario), or the event flow diagrams (see figure 34) only support the better understanding of the dynamic model.

3.7.2 The use of the models in the development process

After the identification of the core requirements of the future software system and of the system boundaries, the desired system behaviour is modelled by use cases or business functions (also called function points). Booch differentiates the behaviour into a primary behaviour (also called *key* or *fundamental behaviour*) and a secondary behaviour (behaviour under exceptional conditions). Whereas during analysis all primary behaviour is modelled by scenarios, only some of the secondary behaviour is looked at in order to insure that no essential patterns of behaviour are missed. These use cases or business functions are then used together with other sources like domain models to define the responsibilities[6] of the individual objects, i.e. all the services an object provides.

Use case descriptions and scenario diagrams are not expected to make up a complete model, they are only aids to determine the object-structure. The complete behavioural model is given by the class specifications and the state diagrams of the individual objects.

Though the processes described by Booch and OMT are not identical, they both target at developing models that show the static and the dynamic aspects of the individual objects. Booch uses for this the class diagrams, the state transition diagrams, the object diagrams, the interaction diagrams, the module diagrams and the process diagrams. OMT subdivides the model into an object model, functional model and dynamic model. All these models evolve during analysis and design, getting more and more detailed and precise.

3.8 Modern Structured Analysis

as described by [Yourdan89] and [Brantschen91].

3.8.1 Terms

Terminators: objects in the environment of the system that interact with the system.

Input and output dataflows: interactions between the system and its terminators.

Business transactions or **essential activities:** abstractions of processes triggered by a specific event.

Events: trigger the business transactions. Different types of events are distinguished:

- *flow event*: the business transaction is triggered by a dataflow coming from a terminator.
- *time event*: the business transaction is triggered by some time constraints (e.g. “once a week”).

3.8.2 Mappings and classifications

There is a 1:1-mapping between event types and business transactions. A business transaction ends when the reaction of the system is completed. Yet there are two different criteria that can be applied to determine when the reaction of the system is completed:

- The business transaction ends as soon as the next input from a terminator is awaited. Thus every input dataflow is a new event. The concept of having perfect technology and business transactions with zero duration can be applied to such an analysis model.
- Especially in the realm of information systems, the first classification results in a huge amount of different business transactions and in cutting apart business procedures into independent

6. The responsibility of an object includes two key items: the knowledge an object maintains and the actions an object can perform [Wirfs90, page 61] and [Booch94, page90].

business transactions which belong together from the viewpoint of the user and sometimes even cannot be executed independently. Therefore all reactions of the system can be said to belong to one business transaction until a more conceptual business procedure is ended or the system is in a "stable" state. If intermediate interactions with terminators are necessary, these is done by input and output dataflows that are not considered as events. Though this second classification results in simpler models, there are many unresolved details around it (like vagueness, inconsistencies, concept of perfect technology).

Modern structured analysis does not provide any further structuring or grouping of events or business transactions. Some adaptations of the method have introduced a third type of events, the *internal events*. These help to extract common parts into a separate business transaction. This business transaction is not triggered by a terminator but by some other business transaction.

3.8.3 Notations

Environmental model

The **event list** is a table with the following rows:

- name of the *terminators* causing the business transaction
- name (and parameters) of the *input dataflow*
- *type of the event* (flow, time, internal)
- *conditions* for the different reactions
- *reactions* (name and parameters of the output dataflows)
- name of the *terminators* receiving the output dataflows

The table contains one entry for each business procedure (example see figure 21).

The **context diagram** (see figure 22) is a dataflow diagram that shows the system, the terminators and all input and output dataflows. A textual description of the dataflows specifies all the data entities, attributes and relationships transmitted by this dataflow.

All the data that flows into or out of the system is modelled in an **entity relationship diagram**.

Behavioural model

For each business transaction a **dataflow diagram** is made. These are further decomposed according to the rules of dataflow modelling. The lowest level processes are described by **minispecs**. These describe textually or with pseudo-code the algorithm for processing the input into the output data. Furthermore the dataflow diagrams can be enhanced by control flows and by control processes specified by state transition diagrams.

The correlations between the data entities and the business transactions are shown in a **CRUD-matrix**, which can be automatically derived from the minispecs and the entity relationship diagram.

3.8.4 The use of the models in the development process

Modern Structured Analysis is used at the beginning of a project to get a technology independent, essential model of the problem. The main goal is defining the problem, and thus having an aid in discussing and documenting it. This is mainly achieved by the environmental model. A complete behavioural model as described by the method is often omitted in practice, because it does not add any new insights into the problem and the structure given by low-level dataflow diagrams often cannot be used for designing the software system. The gap between the analysis model and the design model is quite large, and the two goals, having a technology independent model and having a smooth transition to the design, often proof to be irreconcilable.

3.9 Further notations for modelling global behaviour

3.9.1 Path Expression Groups

Path Expressions Groups (PEGs), introduced in [Adam94] and [Adam92], are used for the description of essential interaction patterns within a system and between a system and its environment.

The interactions of a group of cooperating objects are described by path expressions. A **group** contains several object instances that cooperate to accomplish a task. Groups are orthogonal to classes. A **path** defines a set of permissible sequences of such a group, i.e. it captures the essential interaction patterns within the group and specifies its global behaviour.

Notation

Paths are described by regular expressions over object operations and other paths (see figure 53). The possible orders are defined by the symbols ";" for sequence, "|" for alternation, "*" and "+" for repetition and "[ ]" for optional paths. Interleaving or parallel object operations or paths cannot be modelled. The typical initiator of a path (most often external to the group for which the path is specified) is modelled by the symbol ">>" in front of the path expression. Arguments passed to the operations are not specified in the paths for the reason of simplicity.

Groups are considered as first class objects. Their declarations contain the following elements:

- *name* and *textual description,*
- *inheritsFrom:* a group inherits the path expressions from its super-group,
- *primary participants*: the objects which are components of this group,
- *secondary participants*: other objects that interact with this group as clients or as servers,
- *invariants*: important dependencies between the objects,
- *behaviour*: high level behaviour defined by a path expression composed of detailed behaviours; thus any client of the group can quickly grasp the typical behaviour,
- *paths*: typical/essential interaction patterns, detailed behaviour,
- *exception paths*: atypical interaction patterns.

Each group may contain other groups, and thus its behaviour is a composition of the behaviour of its components. Also each path expression can be composed by other path expressions which may also be path expressions over component groups. Thus composition of groups and composition of path expressions goes hand in hand. When taking all those detailed behaviours that are typically initiated by the user, the external visible functionality of the system can easily be derived.

Using the PEG notation

PEG is a notation for capturing and describing object interaction patterns. It is not a system development method but may be used as enhancement to traditional notations[7]. Adams considers path expressions as being far more usable for documenting object interaction than state charts, which focus only on one object type, or event traces, which capture only one single sequence of messages, though latter can be easily deducted from path expressions.

3.9.2 Scenario trees

Scenario trees are used for modelling user views in a formal way. They are described in [Hsia94].

Scenarios and scenario schemas: A scenario is a "possible way to use the system to accomplish some function the user desires", a "sequence of event types that accomplishes a functional requirement". Scenario schemas are the descriptions of scenario types, scenario instances are instantiations of scenario schemas. A scenario schema corresponds to exactly one path in a scenario tree, each alternative course of event types results in a different scenario schema. Events that are optional or iterated are not possible within one scenario schema, several scenarios are necessary to model them. One of the possible scenario schemas of the scenario in tree in figure 39 is the following list of input and output events:

7. BON recommends to use PEGs for complex dynamic systems where a full dynamic model is necessary and thus object scenario diagrams are not sufficient [Walden95].

```
Off_H, Not9, digit, digit, digit, Busy, On_H
```

Each scenario starts in the initial state of the system and ends, when this initial state is reached again. The states are modelled in such a manner that one scenario corresponds to one functional requirement.

User views and scenario trees: A user view is "a set of specific scenarios as seen by a certain user group or by users that employ the system in a like manner". A scenario tree "describes and represents all the scenarios for a particular user view". Examples for scenario trees are found in figure 39 and 40. A scenario tree consists of states (nodes) and events (transitions). Only those input and output events are mentioned, that cause a state transition. The attributes of the events are not specified in the scenario trees. The root of the tree is the initial state, each leaf is again the initial state. The graph has the form of a tree because the same states can be mentioned several times.

Event and event types: Events are "specific stimuli that change the system state, trigger another event, or do both. An event... can be both an input and response, internal and external to the system." An event can have attributes (parameters). An event type "defines all possible events with similar attributes".

Assumptions for the use of scenario trees:

There are several assumptions that limit the use of scenario trees and of this kind of scenario analysis:

- The system must be sequential, no concurrent stimuli (input events) and responses (output events) are allowed. Only a single response can be taken into consideration for each stimulus.
- Each scenario is described individually for each agent, i.e. one scenario does not show the interactions with several agents but only between one single agent and the system. Thus the scenarios are strictly modelled for one single user's viewpoint. In this aspect, the scenario trees of Hsia differ from most other methods (usually a scenario contains input and output events from and to different agents).
- The starting and ending state of the system must be the same for each scenario. Scenarios for system initialisation cannot be modelled. Further the behaviour of the system under consideration ideally consists of a flat structure of independent scenarios, each scenario having a well defined sequence of events without any alternative courses.
- Concurrent or interacting user views cannot yet be checked against each other for deadlocks.

The goal of using scenario trees in the requirements specification process is to *generate, analyse and validate dynamic requirements in a systematic and formal way.* It is only applicable to requirements that can be easily expressed as a sequence of fine-grained events normally on a technical level (like input of individual digits), where inconsistencies and modelling errors are a great risk for the project (the damage caused by errors justifies high prevention costs), where the sequences of events are very complex (as it is typically the case in transmission systems like PBX or network protocols) and where the requirements are known at the time of requirements determination and will only change in minor details.

Process for developing scenarios:

The process for the identification, formalization, verification and validation of scenarios contains the following steps:

- *Scenario elicitation:* Building for each user group a scenario tree. All scenarios for one single agent are grouped into one user view and a scenario tree is created.
- *Scenario formalization:* Converting each scenario tree into a *regular grammar* and a *conceptual state machine*, one per user view (see figures 39 and 40). A conceptual state machine is a deterministic finite state machine that has exactly one initial and one terminating state which are identical.
- *Scenario verification*: Automated checking the grammar for internal incompleteness, redundancies and inconsistencies. If errors are detected, the previous steps are repeated.
- *Scenario generation, prototype generation and scenario validation*: All possible scenario schemas are extracted, a prototype is built automatically from the conceptual state machine, and the user validates the scenarios with the aid of the schemas and of the prototype.

The user view is first written down in the form of a scenario tree and not directly in the form of a state chart because scenario trees match better the way how a user sees its interactions with the system, and because the sequence of the events is directly visible. The scenario trees are not necessarily consistent;

but possible inconsistencies are detected with the regular grammars and the conceptual state machines. The user is involved heavily in the scenario elicitation and scenario validation, and also to some part in the formalisation and verification.

3.9.3 Composition of scenarios based on statecharts

In [Glinz95] a behavioural model showing the external view of a system is made. A formal notation is used in order to enable various consistency checks and to allow simulation or automatic prototyping of the behavioural model.

A **scenario** is defined as a sequence of interactions between a user and a system. All the scenarios in a scenario model must be disjoint. In case some scenarios are overlapping, they must first be transformed into several disjoint scenarios or into one single scenario. Each scenario is modelled by a state chart, this state chart is closed, i.e. it has exactly one entry and one exit state (see figure 41). The transitions are triggered by external or internal events and have actions. These can again be interpreted as internal events and can trigger other transitions. Events are broadcast to all state transitions. A transition takes an arbitrarily small time interval, thus the model is quasi-synchronous. This avoids certain nondeterministic behaviour that could otherwise occur.

Disjoint scenarios can be composed and integrated into a complete model of the system behaviour (example see figure 42). The following four compositions are possible: **sequence**, **alternative**, **iteration** and **concurrency**.

Various consistency checks are possible, even if some scenarios are not yet specified by a decomposition or a closed state chart (these scenarios are replaced by dummy scenarios). Deadlocks can be detected, the reachability of states can be checked, the required mutual exclusions of scenarios can be verified, inconsistencies in the names of events and actions can be detected (by executing the model) and the model can be checked for completeness (missing specifications of certain parts, missing behaviour when executing the model).

3.9.4 CRC cards

CRC cards, introduced by Cunningham [Beck89] in order to help students to learn the essentials of object-oriented programming, have become widely used in many methods (e.g. also by Wirfs-Brock, who has made them widely known by her book on responsibility driven design [Wirfs90]). Here we base our summary on the description of CRC-card by [Wilkinson95]. In this book not only the cards themselves are introduced, but also an informal development process that uses the cards for analysis and design is described.

Basically, a CRC card is an index card. There exists one card for each class. On its front side the card has the name of the class as well as a table of responsibilities (left) and collaborators (right). In each row, one responsibility offered by this class is listed, together with the names of the collaborating classes this class has to contact in order to fulfil this responsibility. Classes, responsibilities and collaborators are considered as the most essential concepts of an object-oriented model. During analysis, only the aggregate responsibilities are listed; these are the responsibilities as they are needed by a user. Also only the name of the collaborating classes are listed. A short textual description of the class is added to the back of the card. In the design, subresponsibilities (corresponding to the various steps the object has to take to provide the desired results) are added. The collaborators are annotated with the name of the responsibility that is needed (see figure 46). Further details like the class attributes are added to the back of the card.

The model is developed in CRC card sessions. There a small group of people (developers and users) develop the model. The sessions are often driven by scenarios. These are used in walk-throughs to detect new classes and to verify the model. A **scenario** is defined as a detailed example of a function of the system. A function is considered as a single, visible, testable behaviour. In analysis, these functions are analogous to system requirements, seen from a high level user point of view. For each function, a set of scenario exists that explores what would happen given different parameters."

The static view of the resulting model is documented by a class model that shows the collaborations and subtyping relationships between the classes (see figure 45), and class descriptions that correspond to the information on the class cards. The dynamic view is given by scenario descriptions. These are tables

that show for each step of the scenario the client, the server, and the responsibility demanded of the server[8] (see figure 47). A scenario can also call another scenario. In this case, instead of the responsibility name, the scenario name is entered in the table. There is no rule as to what makes up one scenario, any activity of the application that requires collaboration can be considered as being a scenario. Yet in order to avoid redundancy, short scenarios are preferred. Whenever a scenario may be initiated by several client objects, the name of the client object is left open.

3.9.5 Requirements scripts in OSMOSIS

OSMOSIS is an experimental CASE-tool that supports also the modelling of scenarios in the form of requirements scripts [Berteaud95]. It is assumed that requirements are gathered in the form of manuals, work descriptions, workshop notes or interviews. These are called sections and are together the document of informal requirements. When building the requirements model, in a first step **requirements scripts** (RS) are derived from the sections. In a second step, the scripts are then reworked in order to avoid redundancies and incompatibilities. Also the actors for the major scripts (not all scripts have an external actor) are identified, the relations between these scripts are defined, the values of the attributes (characteristics or aspects of the RS) are defined, the objects (data entities) used by the scripts are identified, and where necessary the scripts are decomposed into further scripts.

The OSMOSIS CASE-tool provides browsers to view the resulting semantic network. This network has the nodes document, section, requirements script, actor and object as well as links among these concepts and between the concepts and textual attributes (which are of type String). The following list shows all the possible links, a * denotes a set of links:

writer [RS, String]	performance [RS, String]	partOf [RS, RS]
date [RS, String]	security [RS, String]	next* [RS, RS]
status [RS, String]		previous* [RS, RS]
	what[RS, String]	inheritFrom [RS, RS]
identifiedBy [RS, String]	preConditon[RS, String]	composedOf* [RS, RS]
fromDocument [RS, Document]	postCondition[RS, String]	
fromSection [RS, Section]	who*[RS, Actor]	etc.
	toScripts*[Object, Script]	

The link “what” describes the objective of an RS (normally just one sentence). The detailed description of the behaviour is given by the decomposition of the RS into further RS’ and not by the link “what”. The focus of the modelling process is always “on what the system realizes for its users rather than how it does it.”

3.9.6 Storyboarding

as proposed by [Umphress91]

OORD (Object-oriented requirements determination) [Umphress91] uses storyboarding, together with concepts maps and event-response lists, to determine the requirements. The notations are deliberately vague, soft and fuzzy, their strength lies in their intuitiveness, understandability and simpleness. OORD knows the following notations:

- **Concept maps**: semantic layered network of information. All concepts (not necessarily data elements or objects) and links (not only static relationships) that make sense to the user are noted; there are no further modelling rules.
- **Event-response list**: a list of events, responses, constraints and maybe some conditions for the events (see figure 50). One event with its responses makes up one scenario. The event-response list shows all those interactions of the system with its environment that are visible to the user. For technical systems the events may be very low-level, yet the developer is free to use the list for any appropriate abstraction level.
- **Storyboard**: like a prototype on paper (see figure 50). The pictures show how the user interface or system state could look like and change its appearance during the scenarios. The event-response list

8. These tables are very similar to the scripts of OBA, though begin and end of scenarios are defined differently.

describes the behaviour from one frame or picture of the storyboard to the next one. Whereas the event-response list should be determined as complete as possible, storyboards are only drawn for some selected scenario instances.

Object-oriented requirements determination is a front-end process to traditional object-oriented development methods. The only goal of OORD is to find and establish candidate needs and to provide a conceptual model of the problem in a manner that is closer to the object-oriented paradigm than a functional description. The notations make the communication between developers and clients easy and provide also the basis for the decisions concerning the automation boundary of the target system.

The reconciliation of the candidate needs and their checking for completeness and consistency is left to the process of requirements analysis, which will then result in the requirements specification document. In the requirements analysis the concepts of the concept maps become potential classes. The storyboard gives hints for the allocation of operations to the classes. The events and responses help to validate the completeness of the behaviour.

as proposed by [Zorman95]

[Zorman95] describes the tool REBUS that allows to capture and document scenarios in a storyboard like representation. Each scenario consists of various pictures (called frames), and for each scenario various concepts and relations can be specified. These so called *within* scenario relations are objects, measures, spatial elements, temporal elements and behavioural elements. The scenarios are considered as being a natural mean of representing and capturing domain knowledge during requirements envisaging. The definition for scenarios is taken from [Benner93]: “Scenarios are partial descriptions of system and environment *behaviour* arising in restricted *situations*." Partial description means that they need not completely specify all the states that comprise a behaviour, nor need they completely specify all the attributes of any given state, only what is relevant for the chosen representation. The concepts behaviour and situation are clearly distinguished, but both, situation and behaviour are a partial ordering of states or transitions. The focus can be either on the sequence of states, or on the sequence of transitions.

3.9.7 Business processes (“Geschaeftsvorfaelle”)

In certain problem domains, especially banks and insurance companies, the global behaviour is modelled by business processes (business transactions, banking operations, “Geschaeftsvorfaelle”). These models are well suited to define how the enterprise works and what services it offers, but the question arises, how these models are combined with object-oriented software development. In [Mueller93] it is discussed how these business processes can be mapped onto an object-oriented analysis model.

Characteristics of business processes in [Mueller93]

A “Geschaeftsvorfall”, we translate it here by business process, is an autonomous process or activity within the enterprise. It may be triggered internally or externally. A business process consists of certain building blocks, also called actions or business processes (“Geschaeftsprozesse” in contrast to “Geschaeftsvorfaelle”). These may be manual or automated. The business processes are modelled on a conceptual level that does not consider any software system related aspects. The building blocks are logically and temporally related, but they cannot be triggered on their own. The duration of a business process is not limited in time, it may also be interrupted to call other business processes, or to get further information. In a software development project, the model of the business processes is made together with a semantical data model as part of the first step of analysis. They show the functionality of the system from the user’s viewpoint. But models of business processes may also be used outside of software development. An example for a graphical description of a business process is shown in figure 51.

Mapping business processes onto classes of an object-oriented system

There are two possibilities how business processes are mapped onto classes. It is assumed that all the persistent information is modelled by entity classes. The first possibility is to allocate each business process as an operation to a specific entity class. The second possibility is to model each business process as a class itself (so called process classes). The first mapping handles well all those business processes that are very short, cannot be interrupted and have no persistent data that concerns the business process itself. But if for a certain business process there may exist at the same time several instances

that concern the same entity instance, the business process must be modelled as a process class. If it were allocated as an operation to the entity class, then the operations would have to be re-entrant. Furthermore, whenever several business processes have the same operations and attributes, the business processes could be modelled by an inheritance hierarchy in order to allow reuse. Because inheritance is only possible between classes and not between the operations of a class, also in this case the business processes are modelled as business classes.

3.9.8 Extending OOSE by use case levels

the approach of [Armour95]:

In [Armour95], use cases are modelled on several abstraction levels. The highest level consists of the **high level use cases** that are grouped into **functional areas**. These functional areas organize the system behaviour by functionality. A high level use case is described by its business event, which is initiated by an actor and reflects a responsibility of the system. Each high level use case is detailed by an **expanded use case**. Expanded use cases contain a complete use case descriptions as known from OOSE, with the exception that normally only the basic course is shown. Also pre- and postconditions are added. Each expanded use case is once again detailed into one or several **detailed use cases**. There may also be several levels of detailed use cases. On the level of detailed use cases also alternative and exceptional courses are specified, redundant parts of the use cases are factored out into abstract use cases, and conditional logic is used in the description. Furthermore, **use case dependency diagrams** show the dependencies between the use cases. They are used for verifying the pre- and postconditions, for deriving work flow models, or for giving a better understandability for newcomers to the use case documentation.

Use case modelling is advocated in [Armour95] for software system development as well as for business modelling and business reengineering. The process of use case modelling is subdivided in various steps, starting with the context diagram and the high level use cases, ending with the refinement of the detailed use cases. When modelling the software system, then the high level use cases are meant to be understandable to the users, whereas the most detailed use cases may contain the minute details of a user interaction by a GUI interface.

the approach of [Regnell96]:

In [Regnell96], a hierarchical use case model is suggested providing three levels of use cases, each level having its own notations. The highest level is the **environment level**. Use cases are identified and associated with services, actors and goals. “A service is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have. ... A use case models a usage situation where one or more services of the target system are used by one or more users with the aim to accomplish one or more goals. A use case may either model a successful or an unsuccessful accomplishment of goals." Furthermore, use cases can be organized into packages according to the services. The use cases are described by context diagrams, which show use cases, actors, and packages.

At the **structure level**, use cases are divided up into episodes, using sequencing, alternatives, repetitions, exceptions and interrupts. The diagram is like a flow chart, showing for each use case its pre- and postconditions as well as its episodes. Repetition, exception and interrupts are shown by annotations to the episode symbol. Each episode can again be decomposed into further episodes, for this a Jackson like structure chart is used.

Events and scenarios only come in at the **event level**. For each use case an interaction diagram in timeline notation is made. It shows the events between the system and its actors. Repetition etc. are shown by rectangles drawn around the whole part of the diagram that is repeated. There exists also a rich set of symbols for activating and deactivating timers. And **event** is defined as being either a stimuli (a message from a user to the target system), a **response** (a message from the target system to a user), or an action (an event inside the system). A **scenario** is defined as being “a realisation of a use case described as a sequence of a limited number of events with linear time order." Two different grades of scenario instantiation are distinguished: scenarios with formal parameters and scenarios with specific parameter values.

The proposed notation is an extension to the telecommunication norm ITU-T MSC for message sequence charts.

3.9.9 Use case maps

In order to bridge the gap between the use cases of the requirements level and the interaction and visibility graphs of the detailed design, Buhr proposes use case maps [Buhr95]. Use case maps model large-grained behaviour patterns. They can be used for detailing use cases as well as for abstracting design patterns. Use case maps target at a system view and are a tool for reasoning and documenting the high-level design. They show chains of causally-related responsibilities. Details like the interfaces of components or effective interactions among components cannot be shown.

The graphical part of a use case map (example see figure 54) contains one or several paths through a use case. These paths are cause effect chains through the system. Along the path responsibilities are shown. These are coarse-grained units of activity. The responsibilities may be bound to components. Components are only characterized by the responsibilities, no interfaces are defined. In an unbound use case map, responsibilities are not yet allocated to any components, and components are not yet shown.

The textual part of a use case map contains among other things the names of the responsibilities, the names of the input parameters of the stimulus of the use case, and pre- and postconditions for interpath coupling.

3.9.10 M.E.R.o.DE

The goal of M.E.R.o.DE (Model-driven Entity Relationship object-oriented DEvelopment) [Dedene94] is to provide a formal object-oriented notation for requirements specification in order that the specification models can be verified concerning their consistency and that the implementation can be validated in respect to its specification.

Models

The **business model**, which models the exact functioning of the business by business entities, business constraints and business rules, contains the following schemas:

- for the static aspects: *entity relationship models*, *dependency graphs* and *abstract data types specifications*,
- for the dynamic aspects (see figure 38): one *object event table* for the whole system and a *structure diagram* for each object.

The object event table shows for each relevant event type (common events which may act on several objects), which object types it may create (C), modify (M) or destroy (D). The structure diagrams, which are JSD-diagrams, describe the sequence restrictions each object type imposes onto the event types which trigger methods of this object. Structure diagrams specify the life-cycles of object instances. Based on the dynamic models, the object types are then specified as abstract data types by their state vector (containing all attributes) and their event-vector (one method for each event in which the object participates).

Further models of M.E.R.o.DE are the **scope model**, the **design model** and the **technology model**.

Consistency checking

Because the schemas of the business model can be represented by formal languages and because the schemas are strongly interconnected, consistency checking is possible. The models are checked for deadlock by deriving the global behaviour from the individual schemas: the structured diagrams are transformed into regular expressions over common event types, including operators for sequence, iteration and selection. In order to guarantee consistency, object types that have common event types need to allow the same sequence of event types in their life-cycle.

3.9.11 Behavioural models of Kowal

The behavioural models of Kowal are described in [Kowal92].

Terms

A **motive** is the cause or reason why a source sends one or several **stimuli** to the system. Stimuli are input dataflows. An **external event** is the action of presenting a stimulus to the system, the system reacts with a response, i.e. output data flows and/or internal activities. A **scenario** consists of a set of input and

output dataflows, control flows and behaviours. A **behaviour** consists of one or more units of activity. Behaviours are e.g. inputs, outputs, verifications, activation of another behaviour, calculations, deductions etc. Behaviours use stored data and input data flows and they achieve some inspectable results, either by internal state changes or by output dataflows. Beside the external events there exist are also **temporal events** and **anomalous events** (an occurrence of an anomaly that is recognized by the system and that requires action to preserve or restore the system to a normal condition). To show the possible orders of events, the following relationships between events can be specified:

- *Administrative relationship*: All those events that serve the administrative purposes of the same data stores (creation, maintaining, securing...) depend on each other.
- *Aggregate relationship:* The result from one event provides the data for another event. The two events accomplish a single objective.
- *Directive relationship*: An event issues a control flow and/or data flow that initiates an otherwise independent event.

Notations

The system behaviour is modelled on three different abstraction levels:

- **Operational view:** A *context diagram* shows the dataflows between the system and the agents and defines the problem boundaries. *Event diagrams* show for each event the information flows, the event process and the data stores. Each event process may have several input dataflows. Also the relationships between events are modelled. All the data of the dataflows is modelled in an *entity relationship diagram*.
- **Architectural view:** *Scenario diagrams* are data flow diagrams that show the processors, controlflows, dataflows and data stores involved in a scenario; there may be one or several scenario diagrams for one event. They are annotated with additional descriptions of constraints. *Scenario specifications* describe the interface and the technical requirements for each processor of a scenario. The interface definitions are done by action diagrams that show the sequence of behaviour patterns, i.e. list all inputs and outputs for each processor.
- **Behaviour view:** *Behaviour diagram***s** are dataflow diagrams that show for each behaviour pattern the processes involved. They are supplemented by the *behaviour specifications* which use action diagrams with pseudo-code to give a low-level specification of the behaviour patterns. Moreover for reactive systems *state transition diagrams* are used to specify finite state processors.

In the action diagrams graphical brackets and pseudo-code are used to model the structure of the logic which is used to transform input data into output data. The brackets can express sequence, selection, repetition and concurrency (see figure 49).

Behavioural models of Kowal and structured analysis

The method of Kowal is a further development of structured analysis overcoming many of its weaknesses. Kowal introduces dependency between events, traceability throughout all the abstraction levels, flexibility for the divergencies between the views of the problem and the structure of the final software solution, and allows a complete and consistent specification of the software system. The handling of the many different diagrams and concepts is possible, because the relations between them are well defined.

3.9.12 Further approaches

The methods mentioned above are by no means all methods having modelling techniques for showing the global behaviour. A more informal use of scenarios is also found in [Reisin90] or [Holbrook90]. Syntropy [Cook94] also models the system as a stimulus response system and shows the responses of the individual objects onto the external events which are interpreted as being broadcasted across the system. [Martin95] explicitly models the specialisation hierarchies of event types and their connection with the object operations by event diagrams. A more formal approach to modelling the behaviour of composite systems is proposed in [Dubois93].

4 Examples of diagrams for modelling global behaviour

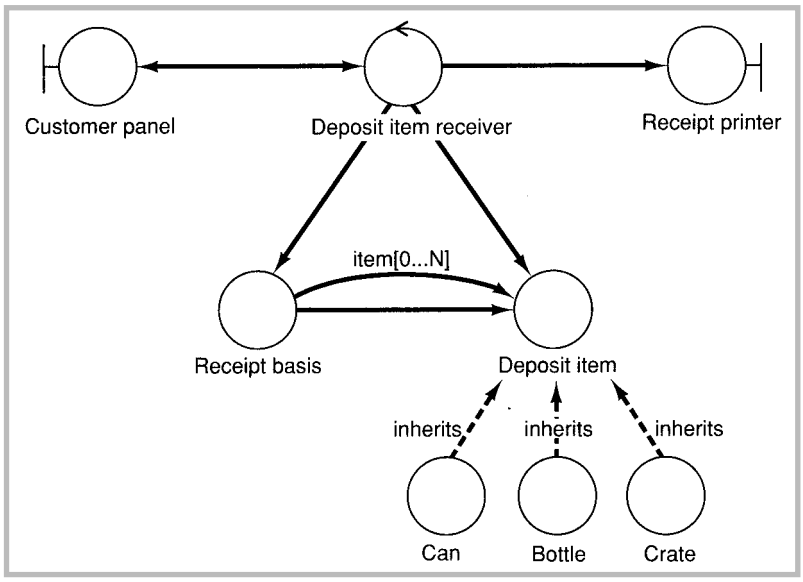


Figure 9: Object model with communication associations (arrows with no labels) in OOSE [Jacobson92, page 189]

(1) The office personnel insert a request for a customer withdrawal at a certain date and a certain warehouse. The window in Figure 13.7 is shown to the office personnel when giving the command issuing a customer withdrawal.

(2) The information filled by the office personnel is Customer, delivery date and delivery place. The name of the office personnel person is filled from the beginning but it is possible to change it. The office personnel person selects items from the browser and adds the quantity to be withdrawn from the warehouses. The browser can here show only the items of the current customer.

(3) The following criteria are checked instantly:

(*a*) The customer is registered,

(*b*) The number of items ordered exist in any warehouse.

(*c*) The customer has the right to withdraw the items.

(4) The system initiates a plan to have the items at the appropriate warehouse at the given date. If necessary transport requests are issued. The items are reserved three days in advance. (The longest possible time for a warehouse redistribution has been measured to take three days.)

(5) At the date of delivery a warehouse worker is notified of the withdrawal.

(6) The warehouse worker issues requests to the forklift operators to get the items to the loading platform. The forklift operator executes the transportation.

(7) When the customer has fetched his items the warehouse workers marks the withdrawal as ready. The items are removed (decreased) from the system.

Alternative courses

There are not enough items in the warehouses

The office personnel are notified and the withdrawal cannot be executed.

Customer has no right to withdraw an item or Customer is not registered.

Notify the office personnel. The withdrawal cannot be executed.

Figure 10: Description of the use case 'customer withdrawal' in OOSE [Jacobson92, page 349]

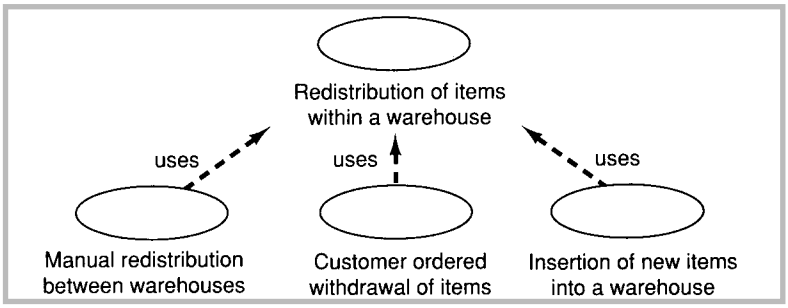


Figure 11: Modelling the relationship between abstract and concrete use cases in OOSE [Jacobson92, page 343]

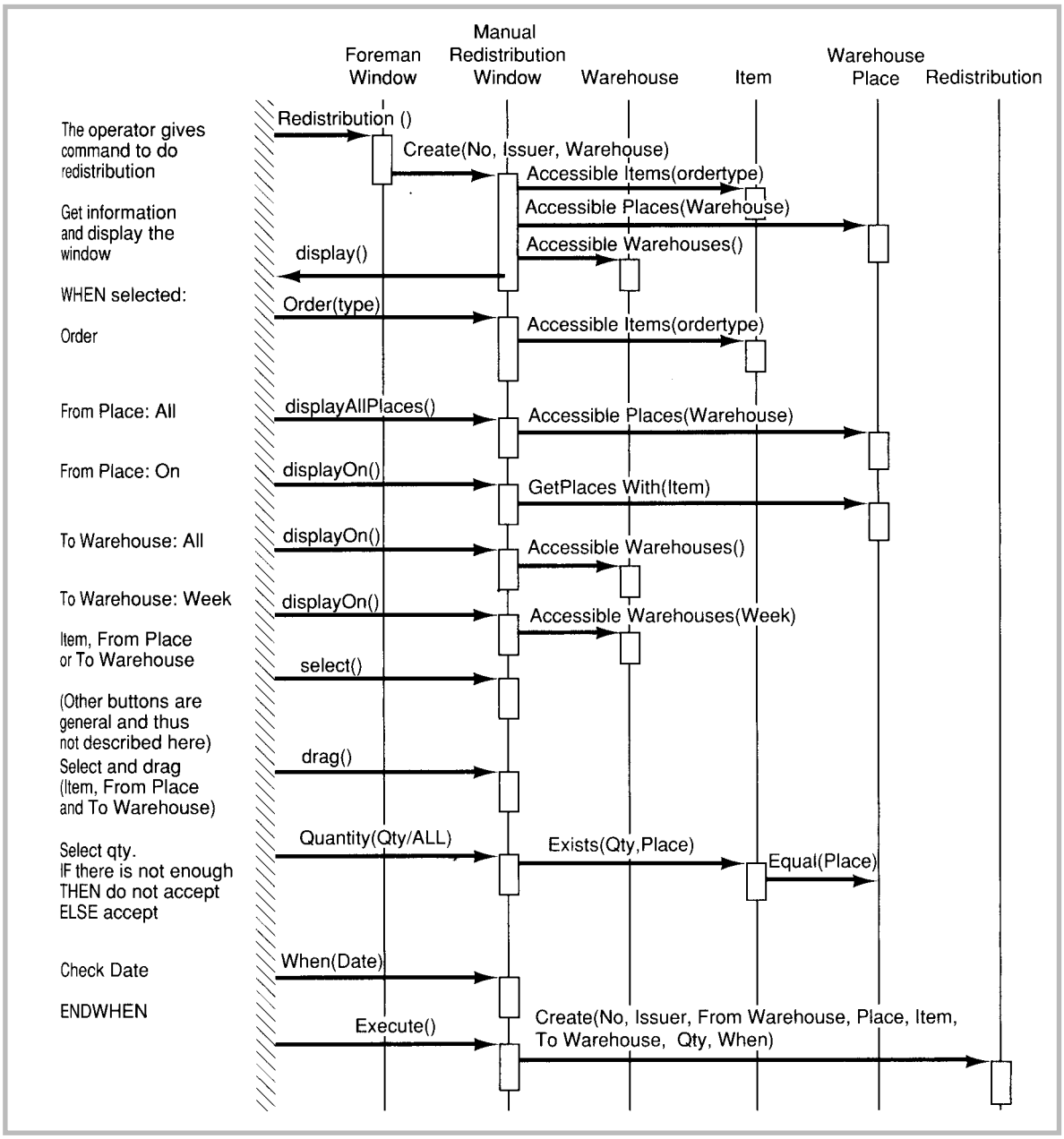


Figure 12: Interaction diagram in OOSE (example with two interface objects) [Jacobson92, page 381]

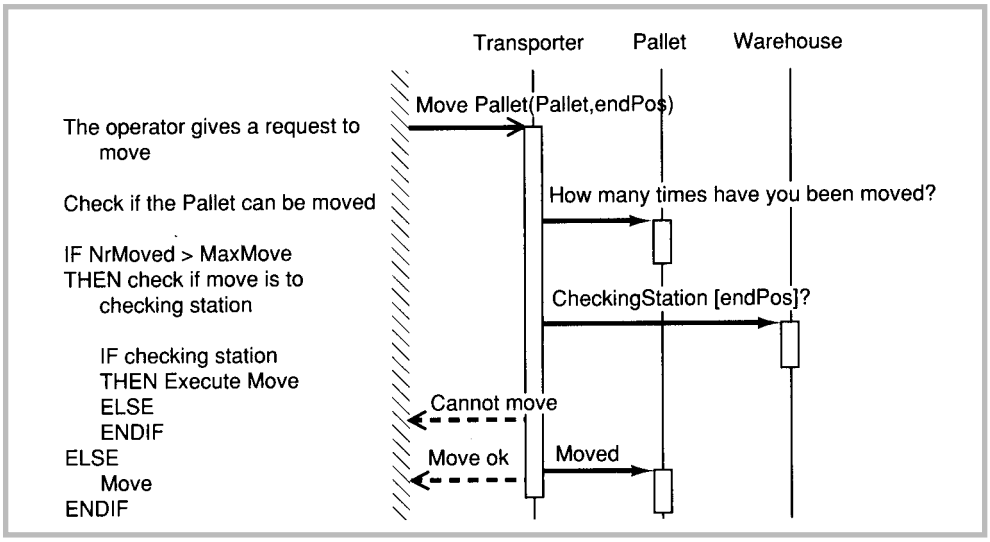
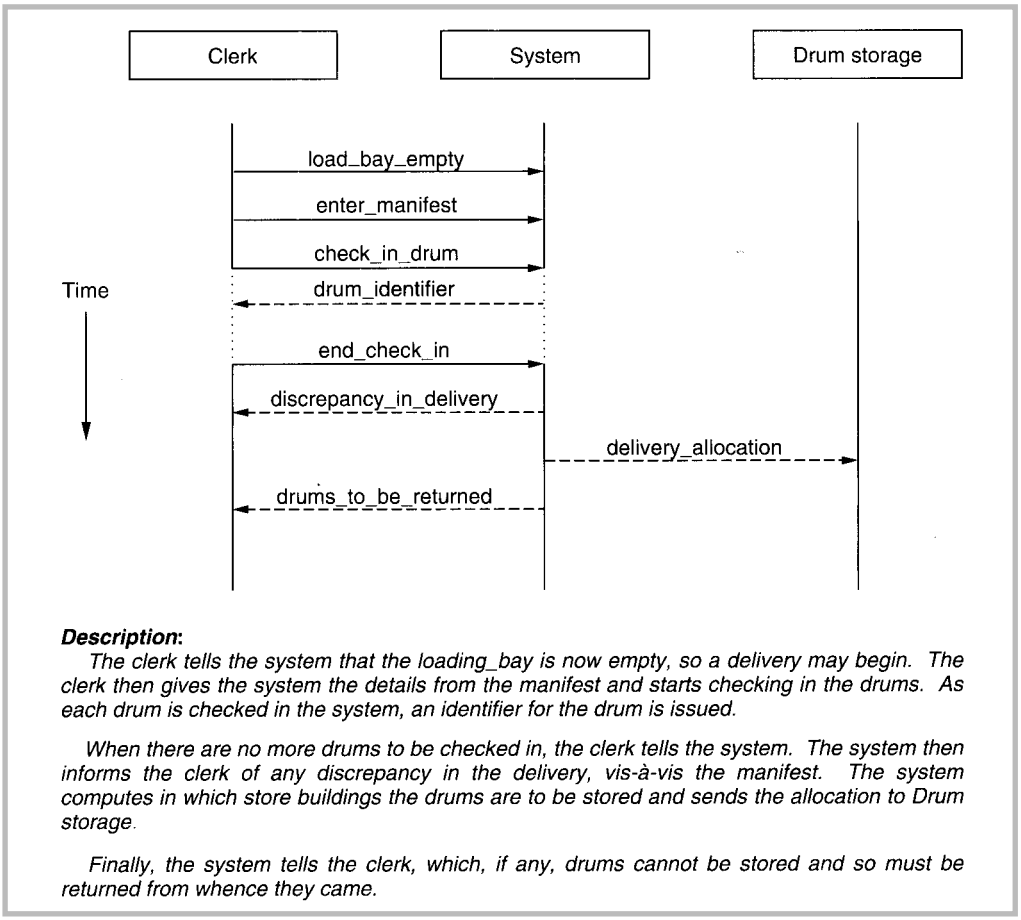


Figure 13: Interaction diagram in OOSE (input and output signals) [Jacobson92, page 218]



***Description*:**

The clerk tells the system that the loading_bay is now empty, so a delivery may begin. The clerk then gives the system the details from the manifest and starts checking in the drums. As each drum is checked in the system, an identifier for the drum is issued.

When there are no more drums to be checked in, the clerk tells the system. The system then informs the clerk of any discrepancy in the delivery, vis-à-vis the manifest. The system computes in which store buildings the drums are to be stored and sends the allocation to Drum storage.

Finally, the system tells the clerk, which, if any, drums cannot be stored and so must be returned from whence they came.

Figure 14: Event trace diagram in Fusion, part of the analysis model [Coleman94, page 47]

```
lifecycle Banking_system: Initialization . ((Transaction | Enquiry)* || PrivEnquiry*)

Initialization = open_account. #account_number . Enquiry* . deposit_money
Transaction    = withdraw_cash . (# dispense_cash | # insufficient_funds) |
                 deposit_money
Enquiry        = check_balance . #current_balance
PrivEnquiry    = authorize. (#confirm | #deny).
                 (inquire. #amount)* . finish
```

Figure 15: Life-cycle expressions in Fusion, part of the analysis model [Coleman94, page 33]

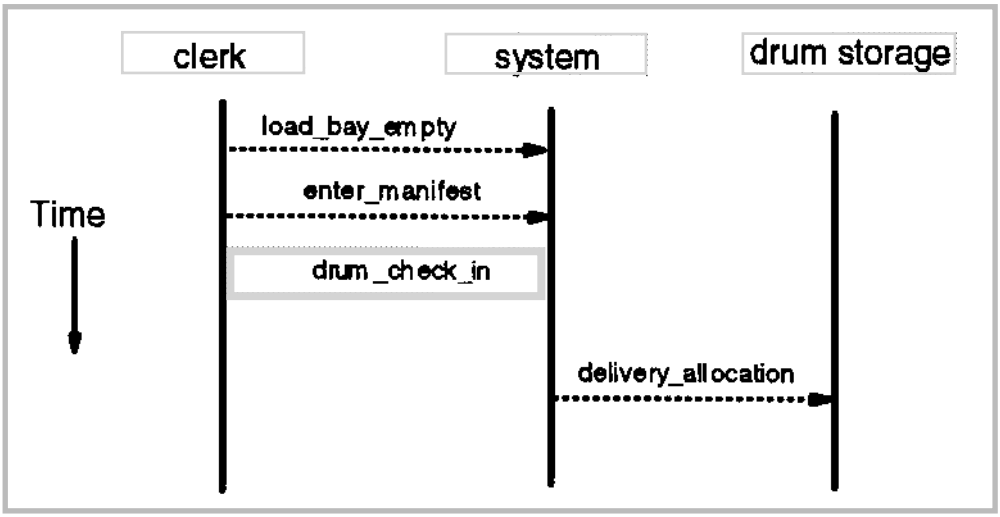


Figure 16: Scenario diagram of extended Fusion showing a use case instance with a sub use case [Coleman95]

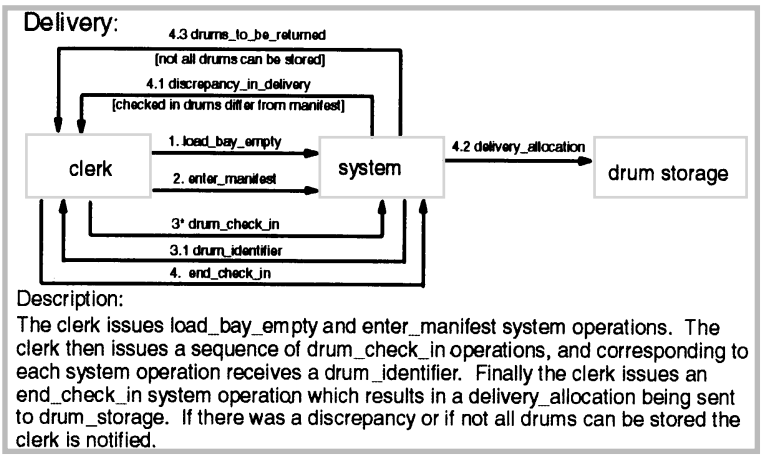


Figure 17: System interaction graph in Fusion, part of the extensions for requirements modelling [Coleman95]

Operation: withdraw_cash

Description: Requests an amount of cash to be taken from a given account. Cash is dispensed only if account has sufficient funds.

Reads: **supplied** acc_number:accountnumber, **supplied** request:money

Changes: account with account.number equal to acc_number (*i*)

Sends: customer: {insufficient_funds}, cash_dispenser: {dispense_cash}

Assumes: acc_number is a valid account number, and the account is not overdrawn.

Result: If (Initial account.balance ≥ request) then account.balance has been reduced by request, and dispense_cash amount has been sent to cash_dispenser. Otherwise, insufficient_funds has been sent to customer. The account is not overdrawn. (*ii*)

Figure 18: Schema of a system operation in Fusion, part of the analysis model [Coleman94, page 31]

Operation: space_for_drum

Description: Find out if the store can accommodate a new drum

---

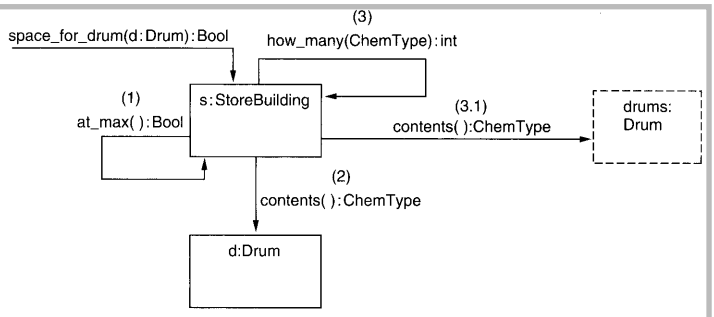
Reads: *supplied* d : Drum, drums in store, max capacity of store

Changes:

Sends:

Assumes:

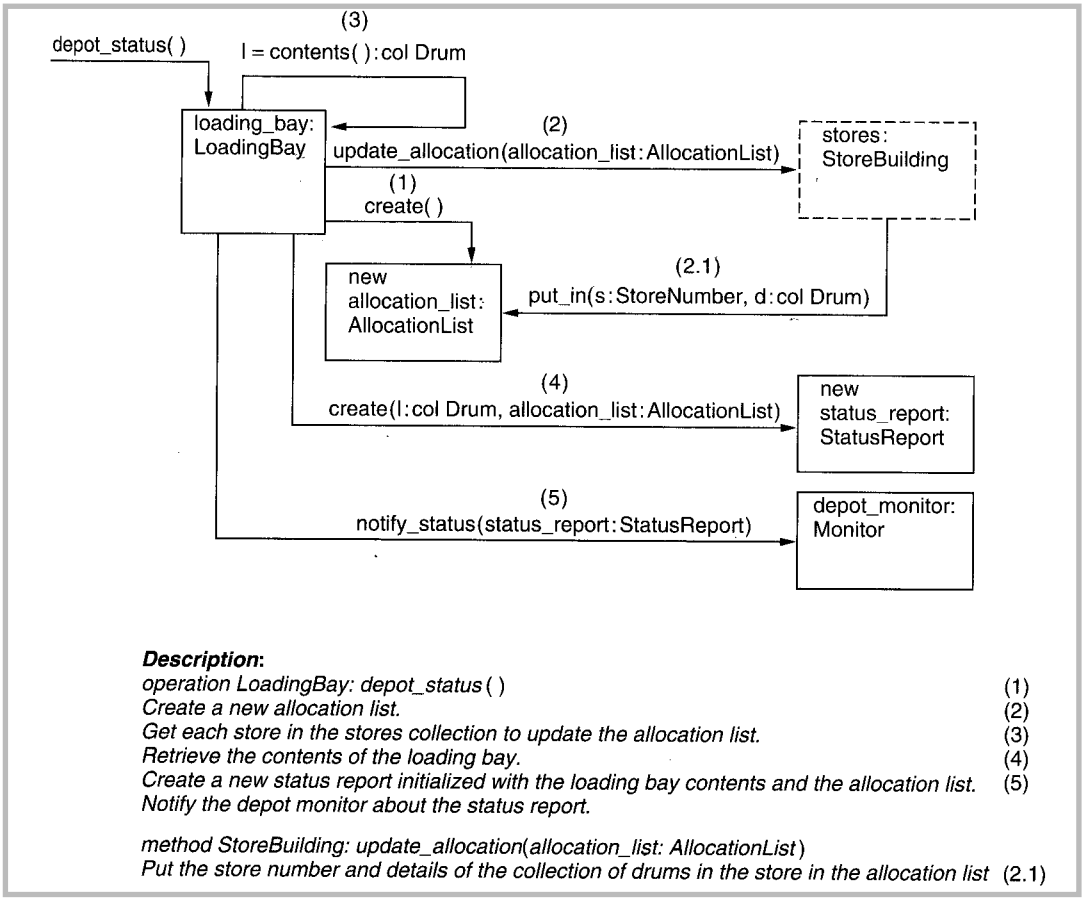
Result: Return true if the addition of d does not exceed the maximum capacity of the store and it is safe to store chemicals of the type in d. Return false otherwise.



Description:
operation Store Building:space_for_drum (d:Drum:Bool
Check to see if the store is at maximum capacity. (1)
If not then
Find out the chemical type of the drum. (2)
Check if it is safe to add chemicals of this type to the store.
(find out how many drums of each chemical type are stored), (3)
// type 1 not to be stored with type 2
If safe return true otherwise return false
Else
Return false

method Store Building: how_many (ChemType) :int
Find out how many drums in the store contain the ChemType. (3.1)

Figure 19: Decomposition of an object interaction graph in Fusion [Coleman94, page 79]



Description:
operation LoadingBay: depot_status ( ) (1)
Create a new allocation list. (2)
Get each store in the stores collection to update the allocation list. (3)
Retrieve the contents of the loading bay. (4)
Create a new status report initialized with the loading bay contents and the allocation list. (5)
Notify the depot monitor about the status report.

method StoreBuilding: update_allocation(allocation_list: AllocationList)
Put the store number and details of the collection of drums in the store in the allocation list (2.1)

Figure 20: Object interaction graph in Fusion, part of the design model [Coleman94, page 75]

N	TYP	EREIGNIS	BEDINGUNG	ANTWORT
1	(F)	Kunde verlangt Auskunft über Hotels, Flüge, Destinationen		Auskunft über Destinationen, Flüge, Hotels, Hotelbelegungen
2	(F)	Kunde bucht Angebot		Bestätigung an Kunde u. Buchhaltung Flug buchen, Hotel reservieren
3	(F)	Zahlungsavis trifft ein	Ticket vorhanden	Reiseunterlagen u. Ticket an Kunde
			Ticket nicht vorh.	intern
4	(F)	Kunde annuliert Reise		Flug u. Hotel annulieren, Annulation an Buchhaltung
5	(F)	Adressänderung eines Kunden		intern
6	(F)	neue Angebote, Adressaenderungen von Hotels, Fluggesellschaften		intern
7	(F)	Ticket trifft ein	Zahlung eingegangen	Reiseunterlagen u. Ticket an Kunde
			Zahlung nicht eing.	intern
8	(T)	Kunde zahlt nicht		Flug u. Hotel annulieren Annulation an Buchhaltung
9	(T)	alte Buchungen, Kundendaten		intern

Figure 21: Event list in Modern Structured Analysis [Beringer92]

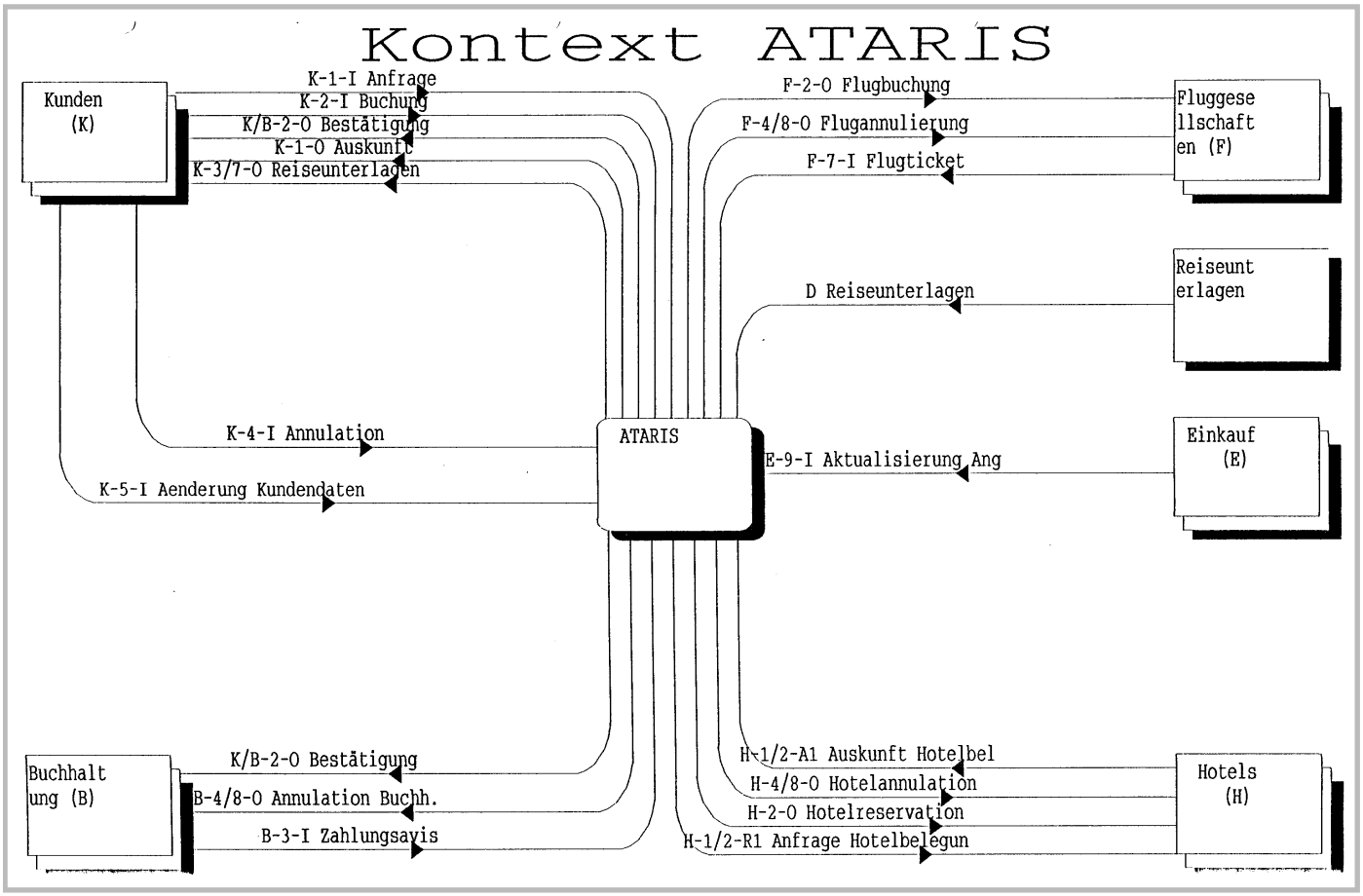


Figure 22: Context diagram in Modern Structured Analysis [Beringer92]

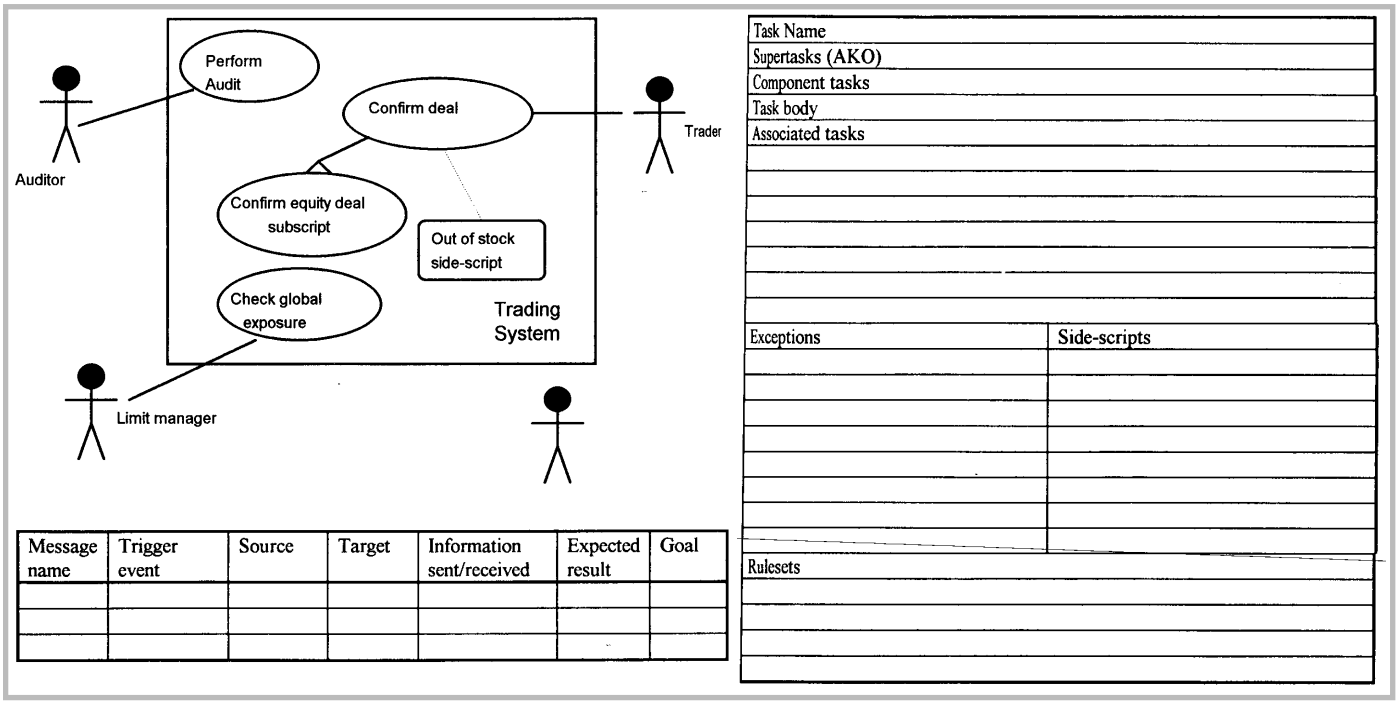


Figure 23: Context model, message table and template for task cards in FORAM [Graham94a]

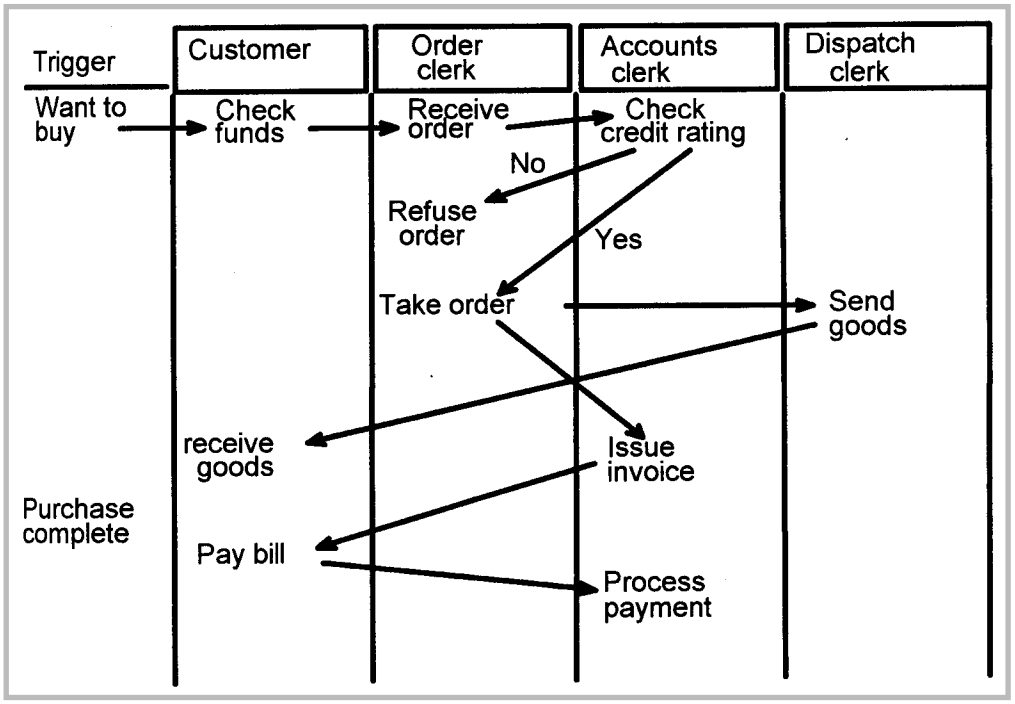


Figure 24: Event trace in FORAM [Graham94a]

SCENARIOS	*CONFERENCE SUPPORT*	**Part:** 1/1
COMMENT Scenarios selected for study.	**INDEXING** **keywords:** paper evaluation, notification, registration, badge printing	
Input review results and compute score: The scores found in the formal review reports for a submitted paper are entered into the system, and an average score is computed for later use at the program committee review meeting.		
Accept or reject a paper and notify authors: A submitted paper is selected and an acceptance or rejection date is entered; a notification letter is sent to the first author.		
Register attendee: An attendee is registered with his/her address, and selected tutorials are recorded.		
Print attendee badge: An attendee is selected, and the corresponding badge is printed in the appropriate format.		

Scenario chart

EVENTS	*CONFERENCE MANAGEMENT SYSTEM*	**Part:** 1/2
COMMENT Typical incoming events triggering interesting system behavior.	**INDEXING** **keywords:** external events, incoming stimuli	

External (incoming)	Involved object types
A contribution is registered	*CONFERENCE, PROGRAM_COMMITTEE, PRESENTATION**
The formal review results of submitted paper are entered	*CONFERENCE, PROGRAM_COMMITTEE, PAPER, REVIEW*
A contribution is accepted or rejected	*CONFERENCE, PROGRAM_COMMITTEE, PRESENTATION*, STATUS*
An accepted contribution is attached to a session	*CONFERENCE, PROGRAM_COMMITTEE, PROGRAM, SESSION, PRESENTATION**
A session chairperson is selected	*CONFERENCE, PROGRAM_COMMITTEE, PROGRAM, PAPER_SESSION, PERSON*
An attendee is registered	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON*

Incoming events

EVENTS	*CONFERENCE MANAGEMENT SYSTEM*	**Part:** 2/2
COMMENT Typical outgoing events.	**INDEXING** **keywords:** internal events, outgoing responses	

Internal (outgoing)	Involved object types
Call for papers is sent	*CONFERENCE, ORGANIZING_COMMITTEE, PERSON, MAILING*
Invitations are sent	*CONFERENCE, ORGANIZING_COMMITTEE, PERSON, MAILING*
A paper is sent to referees	*CONFERENCE, PROGRAM_COMMITTEE, PAPER, STATUS, REVIEW, PERSON*
Warning issued for exceeding tutorial session capacity	*CONFERENCE, REGISTRATION, TUTORIAL*
An author notification is sent	*CONFERENCE, PROGRAM_COMMITTEE, PERSON, PRINT_OUT*, LETTER_FORM*
An invoice is sent to a prepaying attendee	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, INVOICE, INVOICE_FORM*
A badge is printed	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, BADGE, BADGE_FORM*
The conference attendee list is printed	*CONFERENCE, ORGANIZING_COMMITTEE, REGISTRATION, PERSON, ATTENDEE_LIST, LIST_FORM*

Outgoing events

Figure 25: Event charts and scenario charts in BON [Walden95, pages 253-255]

Scenario 3: Register attendee	
1–3	*A new registration is created*
4–5	*Attendee references are recorded*
6–7	*Selected tutorials, if any, are registered*

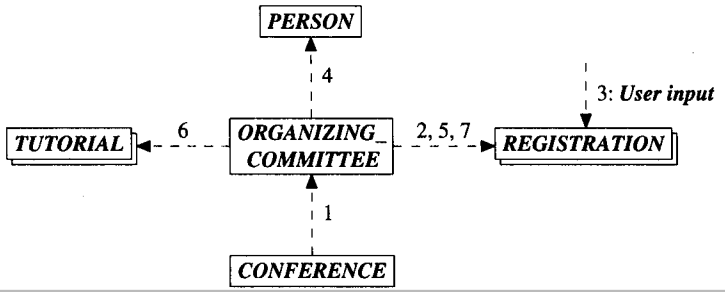


Figure 26: Object Scenario in BON [Walden95, page 259]

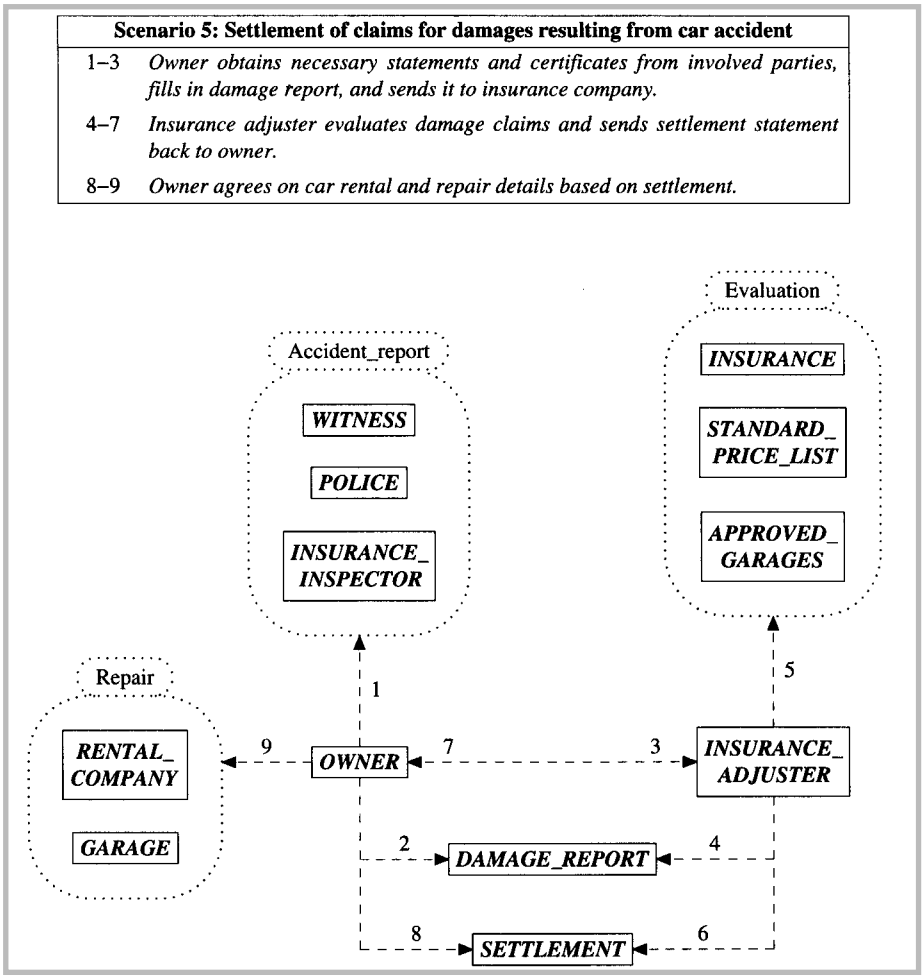


Figure 27: Object Scenario in BON (with grouping of objects into several subtasks) [Walden95, page 114]

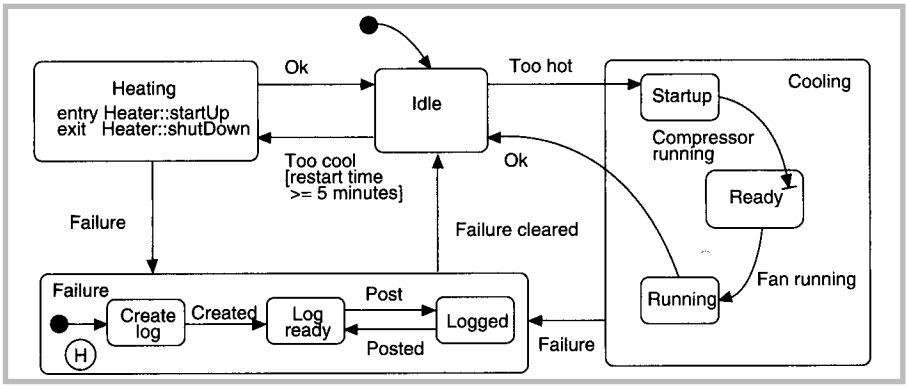


Figure 28: State chart as used in Booch [Booch94, page 207]

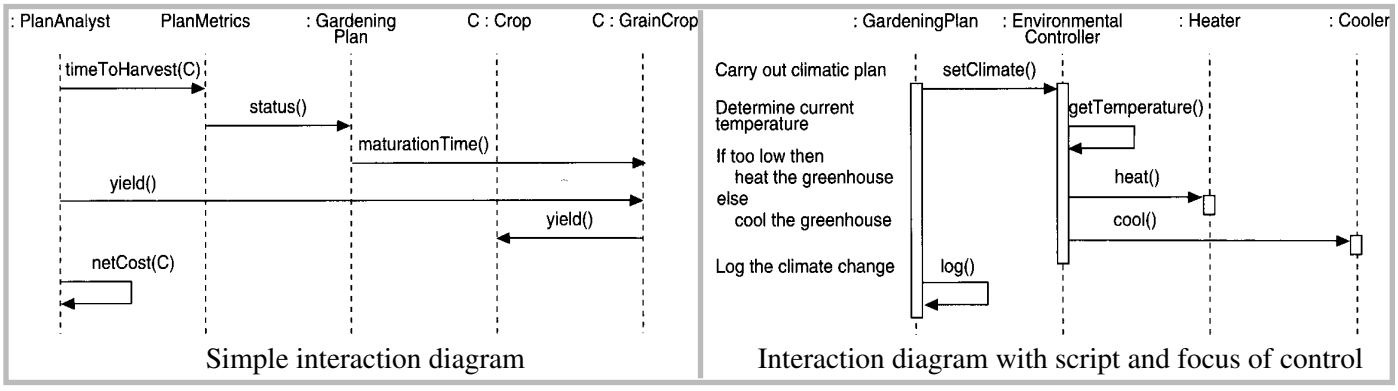


Figure 29: Interaction diagrams in Booch [Booch94, page 218]

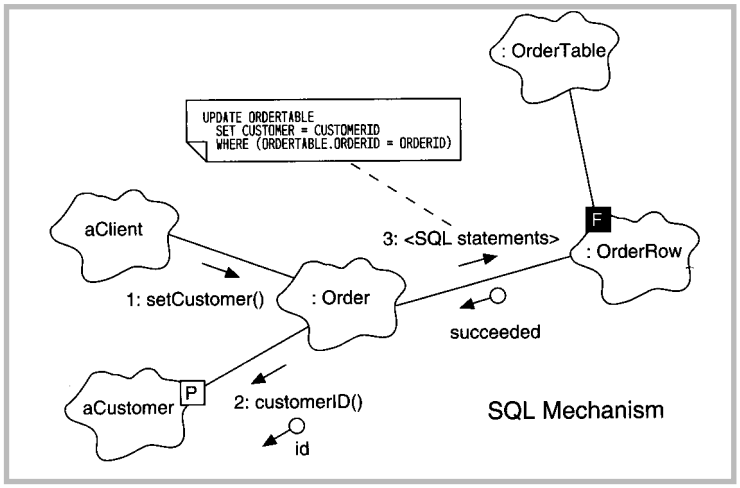


Figure 30: Object diagram for one mechanism [Booch94, page 402]

Use Case:	Assign Seat
Summary:	A passenger with a reservation on a flight requests a seat assignment. The system obtains information from the passenger and then attempts to make an assignment. The assignment is given to the passenger or the passenger is told that no assignment is presently available.
Actors:	Passenger
Preconditions:	The passenger has a reservation on a flight by the given airline
Description:	A passenger requests a seat assignment on a flight. (This may be implicit as part of checking in or may be an explicit request by the passenger.) The system (in the form of the agent) asks for the date of the flight, the flight number, departure airport, and passenger name. The passenger supplies the information. Instead of name, the passenger can supply frequent flyer number with the airline. [Exception: Too early to make assignment.] The system finds the reservation. [Exception: No reservation found.] If the reservation already has a seat assignment, it is given to the passenger who is offered the opportunity of changing the assignment. If there is no assignment or the passenger wants to change, the system requests seat preference (window, aisle) and smoking preference (yes, no). The system uses the information, including frequent flyer level, to try to find a suitable seat assignment subject to previous assignments and the policies of the airline. If necessary, the system asks additional questions: would the passenger accept a bulkhead seat, would the passenger accept a seat in an emergency exit row? The system proposes a seat assignment. If all the passenger's preferences cannot be satisified, then the system proposes the best match it can find. [Exception: No assignment possible.] The passenger can accept the assignment or ask for changes, in which case another assignment is attempted. The use case concludes when an assignment is made and accepted.
Exceptions:	Too early: Raised if the current date is too early, based on an airline algorithm that includes fare category and frequent flyer level. The passenger is advised when seat assignments will be possible and the use case terminates. No reservation found: Raised if no reservation can be found. The information is rechecked and searched if necessary for a partial match. If still no reservation can be found, then the passenger is advised to obtain one and the use case terminates. No assignment possible: Raised if no assignment is possible based on an airline-dependent formula that includes number of unassigned seats, days until departure, fare category, and frequent flyer level. The passenger is advised to obtain an assignment on check in. If this is part of check in, then the passenger is placed on standby in the order of arrival. The use case terminates.
Postconditions:	If this is part of check in, then the passenger either has a seat assignment or is on the standby queue in order of arrival. Otherwise none (the attempt may fail).

Figure 31: Description of a use case in OMT [Rumbaugh94a]

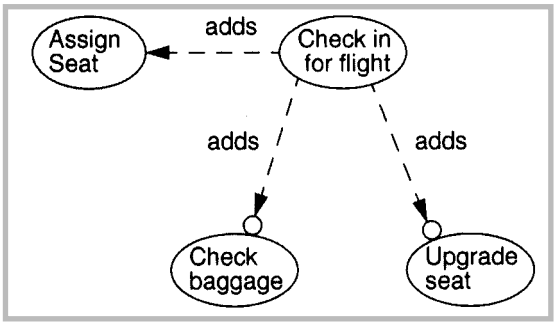
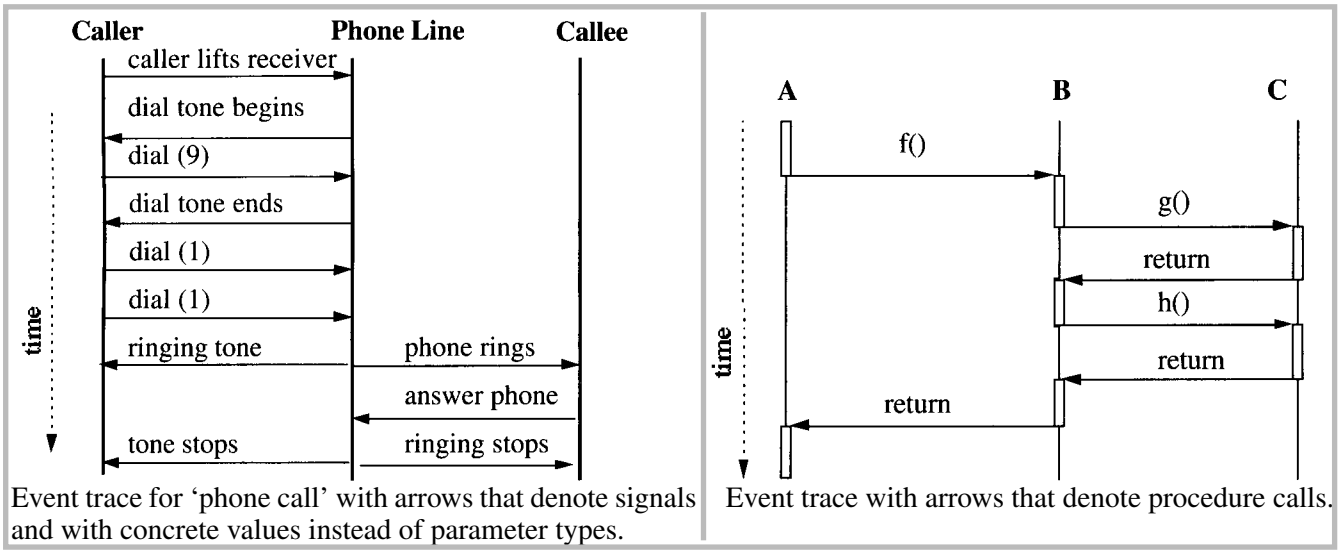


Figure 32: Add-relationships between use cases in OMT [Rumbaugh94a]



Event trace for 'phone call' with arrows that denote signals and with concrete values instead of parameter types.

Event trace with arrows that denote procedure calls.

Figure 33: Two different kinds of event trace diagrams in OMT [Rumbaugh95a]

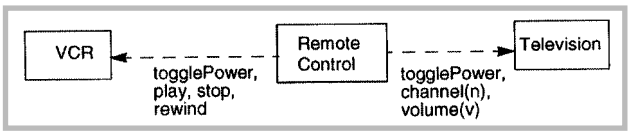


Figure 34: Event flow diagram in OMT [Rumbaugh95a]

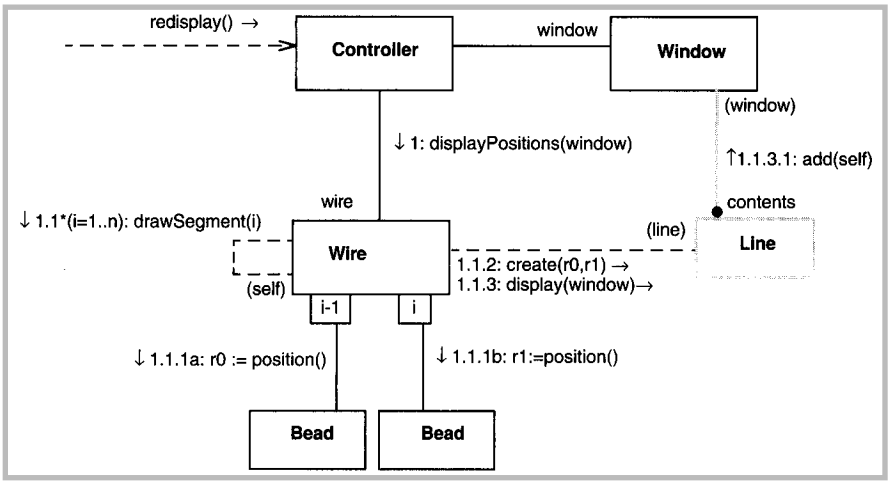


Figure 35: Object interaction diagram showing the design of the operation 'redisplay' [Rumbaugh95b]

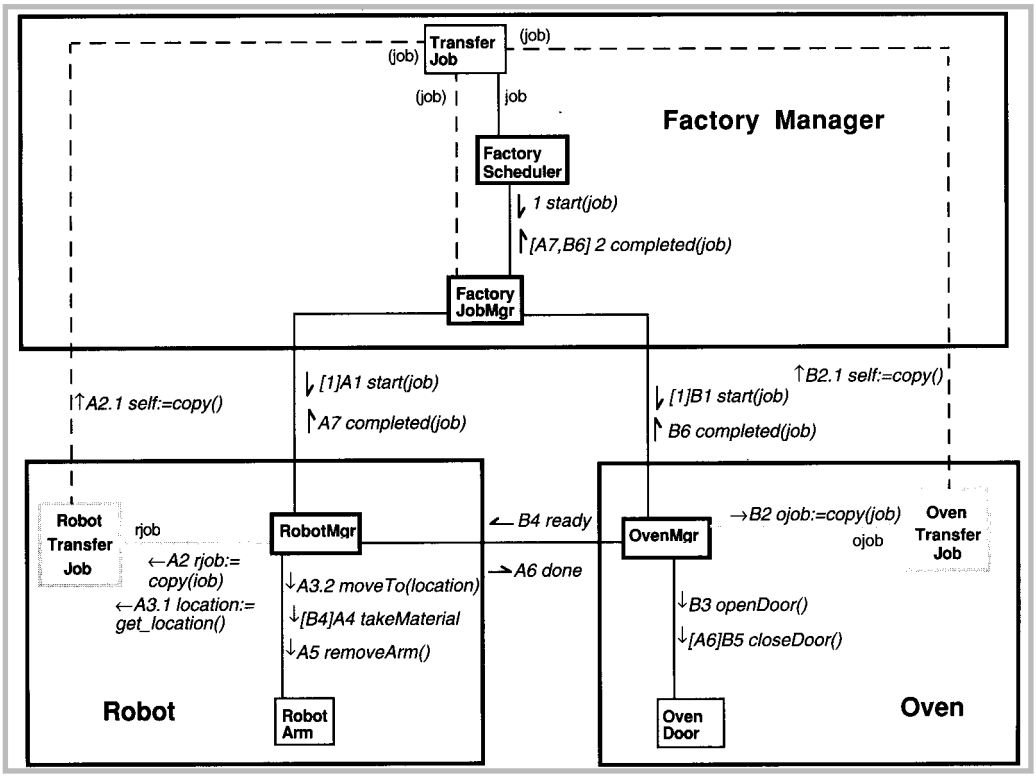
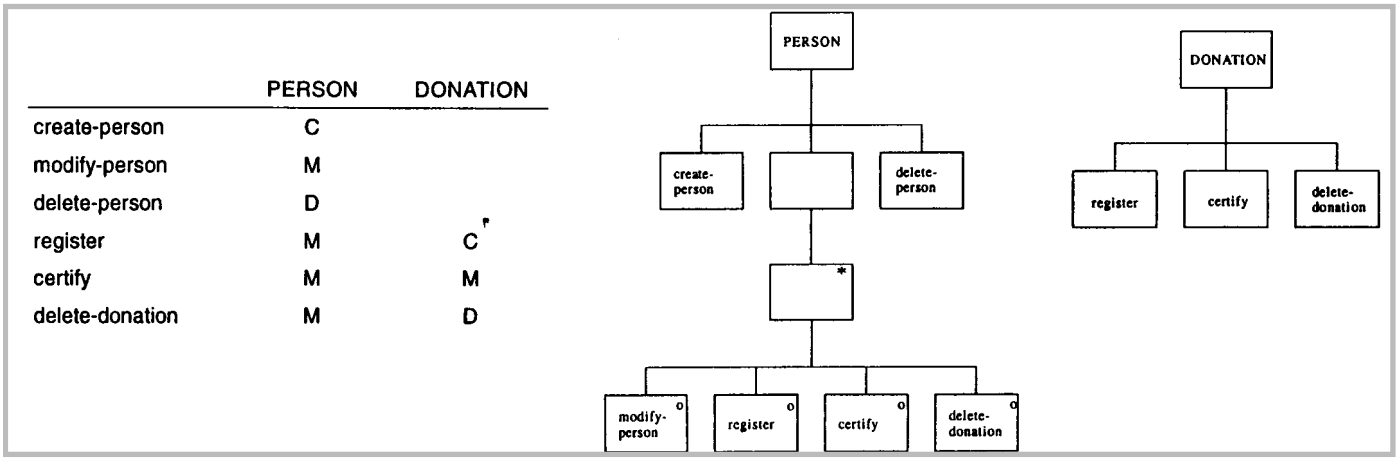


Figure 36: Concurrent object interaction diagram in OMT [Rumbaugh95b]

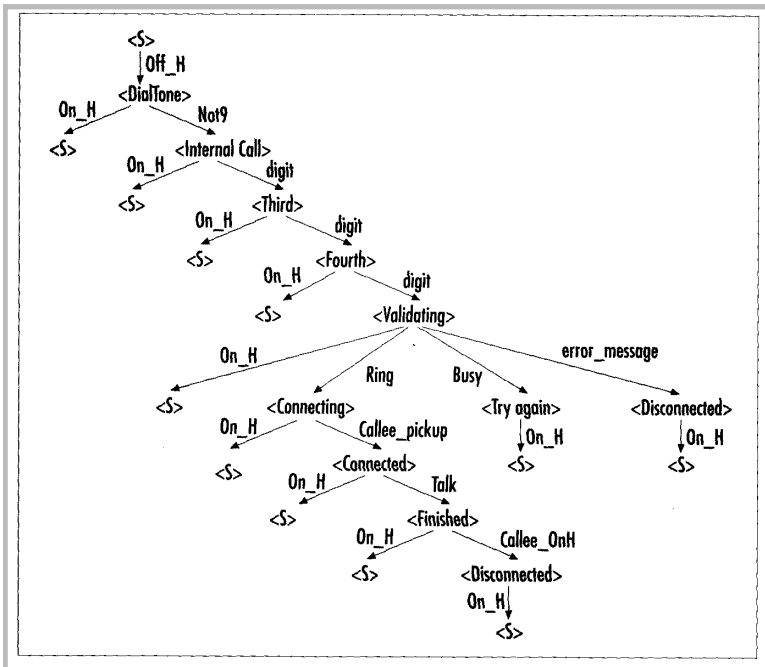
Operation:	sort (elements: Array of T)
Responsibilities:	organizes the elements of the array (in place) so that they are in increasing order
Inputs:	elements – an array of objects of type T
Returns:	none
Modified objects:	elements
Preconditions:	All of the elements must be of type T or a subtype. Type T has an operation compare(t1:T) returning {LT, EQ, GT} which compares two elements and returns whether the first is less than, equal to, or greater than the second. The operation must define a total order on the values of T. Duplicate values of T are allowed.
Postconditions:	The elements are ordered according to the comparison function. The array contains exactly the same number of occurrences of each value of T as before. If two elements compare as equal, then they have the same relative order before and after the operation.

Figure 37: Operation specification in OMT [Rumbaugh95b]



	PERSON	DONATION
create-person	C	
modify-person	M	
delete-person	D	
register	M	C
certify	M	M
delete-donation	M	D

Figure 38: Object event table and structure diagrams in M.E.R.O.DE [Dedene94]

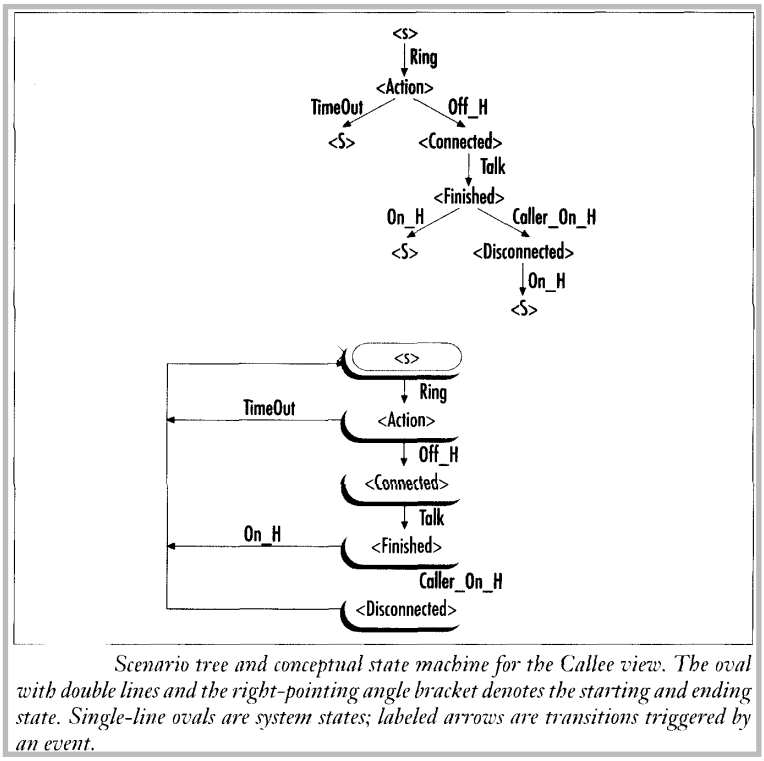


Scenario tree for the Caller view. <S> designates a node, which represents a system state. The arrows denote edges, which represent events. Off_H is phone off hook, On_H is phone on hook, and Not9 signifies that a digit was used that was not a 9. This is significant because Not9 indicates an internal call within the PBX system's service domain.

Gv = (NT, S, R, A)

S = {Off_H, Not9, d, R, Cpu, talk, Chup, On_H}
NT = {Caller, <FD>, <Int>, <Third>, <Fourth>, <Action>, <Talk>, <Fin>}
A = Caller
R = {Caller → Off_H <DialTone>,
<DialTone> → Not 9<InternalCall>,
<Int> → digit<Third> | On_H
<Third> → digit<Fourth> | On_H,
<Fourth> → digit<Validating> | On_H,
<Validating> → Ring<Connecting> |On_H | Busy<TryAgain> |
<Disconnected>,
<Connecting> → Callee_pickup <Connected> | On_H,
<Connected> → Talk <finished> | On_H,
<Finished> → Callee_On_H <Disconnected> | On_H,
<Disconnected> → On_H}

Figure 39: Scenario tree and grammar of the user view of the 'caller' [Hsia94]



Scenario tree and conceptual state machine for the Callee view. The oval with double lines and the right-pointing angle bracket denotes the starting and ending state. Single-line ovals are system states; labeled arrows are transitions triggered by an event.

Figure 40: Scenario tree and conceptual state machine of the user view of the 'callee' [Hsia94]

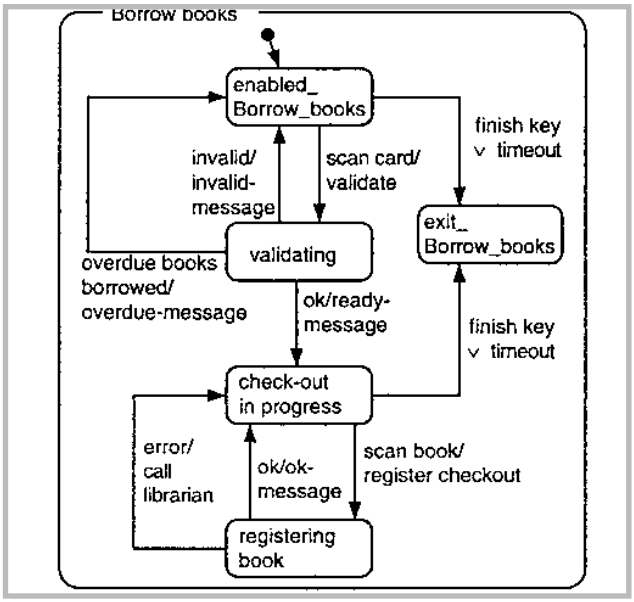


Figure 41: State chart for one scenario in [Glinz95]

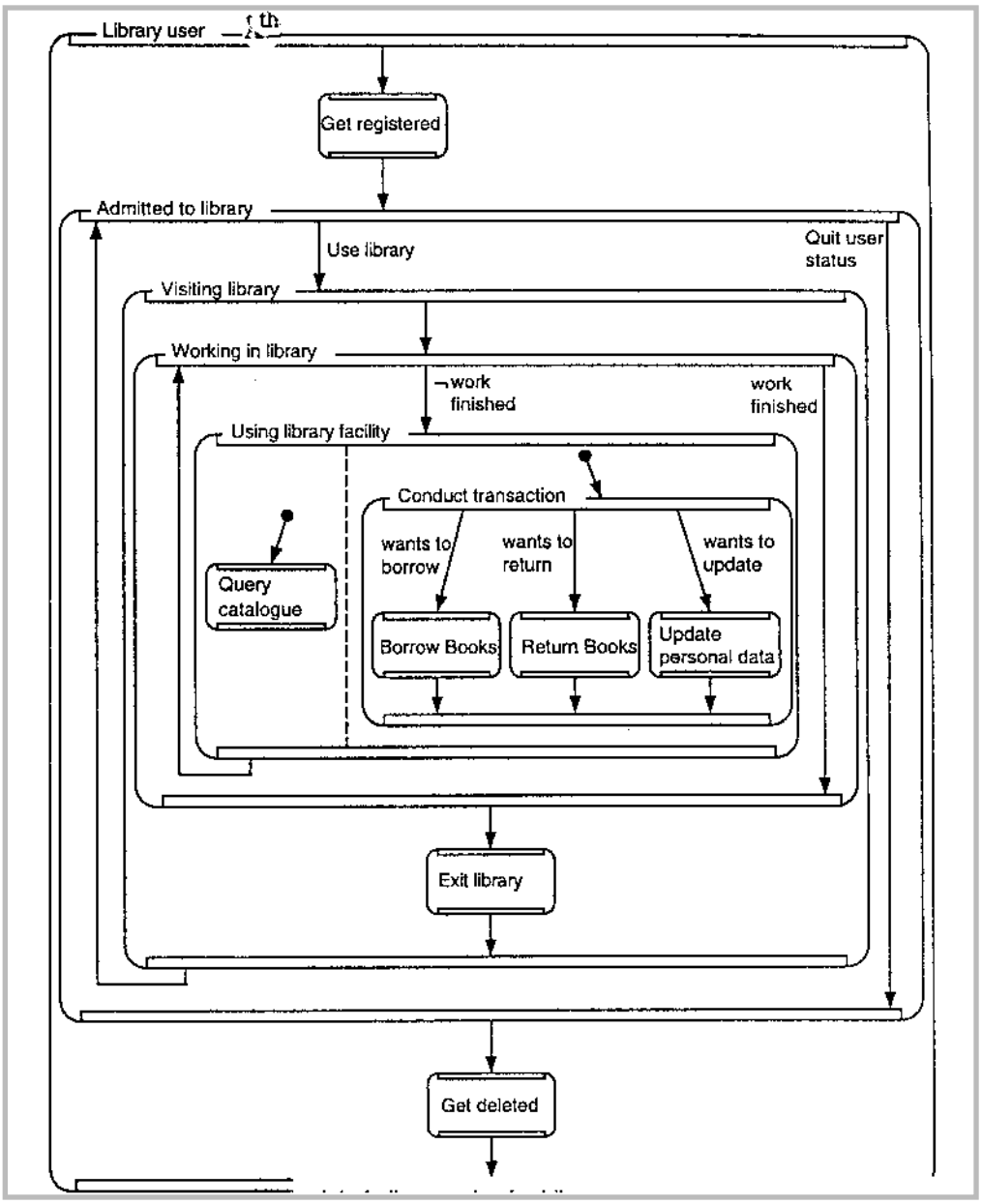


Figure 42: Composition of scenarios in [Glinz95]

Script notation:

Initiator	Action	Participant	Service
thing1	notifies	thing2	thing2 can be notified
thing1	provides info to	thing2	thing2 can accept info
thing1	requests info from	thing2	thing2 can provide info
thing1	requests service from	thing2	thing2 can provide service

Example for a user script:

Script Name Modification.1.example
Author Donna
Version 1.0
Precondition exists (Spreadsheet), displayed (Spreadsheet)
Postcondition modified(Spreadsheet)
Trace Core Activity—Modification

INITIATOR	ACTION	PARTICIPANT	SERVICE
User	select D1	Spreadsheet	select a cell
User	type text NEW	D1	set content to text
User	set text style to bold	D1	set text style to bold
User	select A2	Spreadsheet	select a cell
User	type text NAME	A2	set content to text
	(repeated select and type text for example)	B2, C2, D2, A3 through A10	
User	select Row 2	Spreadsheet	select a row
User	set text style to bold	Row 2	set text style to bold
User	extend row height to 34 pixels	Row 2	resize height
User	select A12	Spreadsheet	select a cell
User	type text TOTALS	A12	set content to text
	(repeat select and type)	A13, A14	
User	select A12:A14	Spreadsheet	select vertical collection of cells
User	set text style to bold	A12:A14	set text style to bold
User	select B3	Spreadsheet	select a cell
User	type number 55000	B3	set content to number
User	choose format $xx,xxx	B3	set format to currency
User	select B3:B10	Spreadsheet	select vertical collection of cells
User	copy first cell into rest of cells	B3:B10	fill down
User	select B4	Spreadsheet	select a cell
User	replace 55 by 60	B4	set content to text
	(repeat rest of changes)	B5:B10	
User	select Column A	Spreadsheet	select a column
User	contract column width to 30 pixels	Column A	resize width

Figure 43: Scripts in OBA [Rubin92]

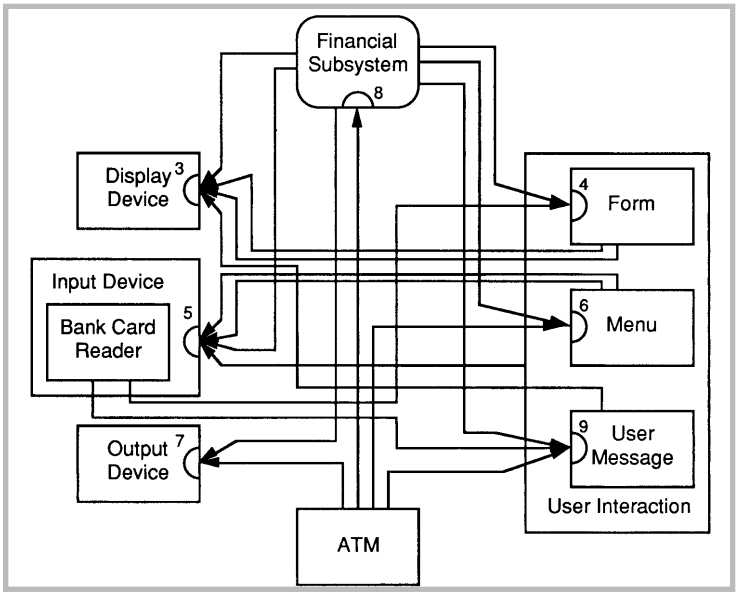


Figure 44: Collaboration graph in Responsibility Driven Design [Wirfs90, page153]

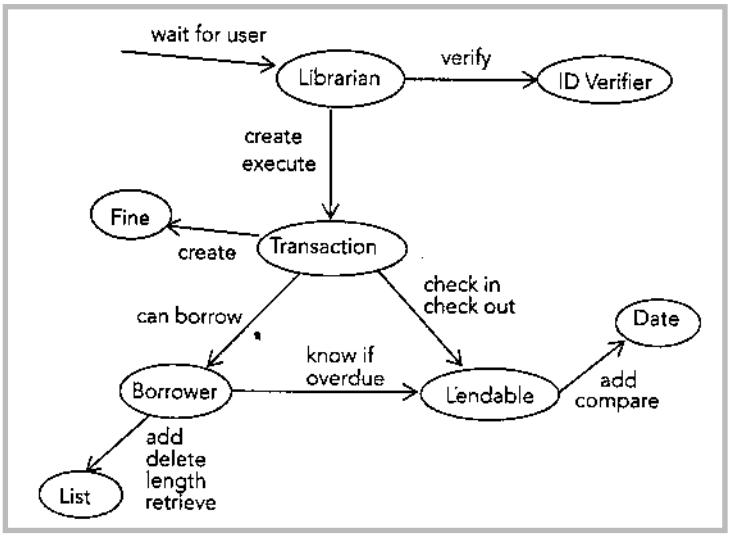


Figure 45: Class diagram with main collaboration in [Wilkinson95, page143]

Librarian	
check out Lendable	
check Borrower status	Borrower: can borrow
check it out	
update Borrower	Borrower: know Lend.
check in Lendable	Lendable: check out
check if overdue	Lendable: know overdue
check it in	Lendable: check in
update Borrower	Borrower: know Lend.
record fine	Borrower: know fine
search for Lendable	Collection: retrieve
display message	UI Subsystem
get info from user	UI Subsystem

Figure 46: Class card in [Wilkinson95, page115]

Scenario 3: Someone Gets a Borrower Object.

Client	Server	Responsibility	Comment
someone	DB	get Borrower	
DB	DB++ classes	get info from DBMS	get info from Borrower table
DB	Borrower	create	
Borrower	List	create	create empty set of lendables
DB	DB	Scenario 2	get lendables for Borrower's set
DB	Borrower	add lendable to set	add each lendable

Figure 47: Scenario description in [Wilkinson95, page146]

„ Der Berater holt dann seinen *Beratungsordner*, *sucht* das *Produkt* über ein *Register* und *öffnet* den Ordner an der *entsprechenden Stelle*. Zum Beratungsordner gibt es noch einen *Formularordner*, in dem *Standardformulare* (z.B. *Vertrag zugunsten Dritter*) *abgelegt* sind und einen *Musterordner*, in dem *Ausfüllhilfen* für Verträge und *Schlüsselblätter* *abgelegt* sind.“

Ausschnitt aus einem Szenario

Figure 48: Textual description of a scenario [Kilberth93, page 99]

Behavior Model page 31

SCENARIO SPECIFICATION

Scenario 6.3 - Shopper wants information about the merchants who sell a brand-name product.

Input REQUEST-B from shopper.
Search INDEX for BRAND-NAME.
If PRODUCT-TYPE is found,
  Input INDEX entry for PRODUCT-TYPE,
  Output INFO to shopper,
Else Search PRODUCT for similar PRODUCT-TYPEs,
  Output ALTERNATES of likely PRODUCT-TYPEs to shopper,
  Input CHOICE from shopper,
  Input PRODUCT-TYPE entry for PRODUCT indicated by CHOICE.
  Output INFO to shopper.
Input CHOICE of BUSINESS-NAME from shopper.
  Input MERCHANT entry for BUSINESS-NAME.
  Output INFO to shopper.
  If number of MERCHANT entries > 1,
    Output ALTERNATES to shopper.
    If shopper Input CHOICE,
      continue loop,
    Else Wait 30 seconds.
      Output WELCOME to shopper.
End.

Processor Requirements: See Technology section of System Description.
Processing Priorities: Shoppers' requests are high priority.
Performance Requirements: Some type of unique screen display must be made within 3 seconds. This includes messages to shoppers such as "Searching for MERCHANT," "Still searching for MERCHANT," etc.
Capacity Requirements: System must be able to perform all requested activities and provide unique screen displays to a maximum of six shoppers at a time.
Time Relationships: N/A.
Interface Standards: N/A.
Security Requirements: N/A.
Fail-Safe Requirements: N/A.
Safety Requirements: N/A.
Other Requirements: Similar PRODUCT-TYPE, BUSINESS-NAME are names that have reasonable likeness to the input provided by the shopper.

Figure 49: Action diagram with brackets, used for the specification of a scenario [Kowal92, page292]

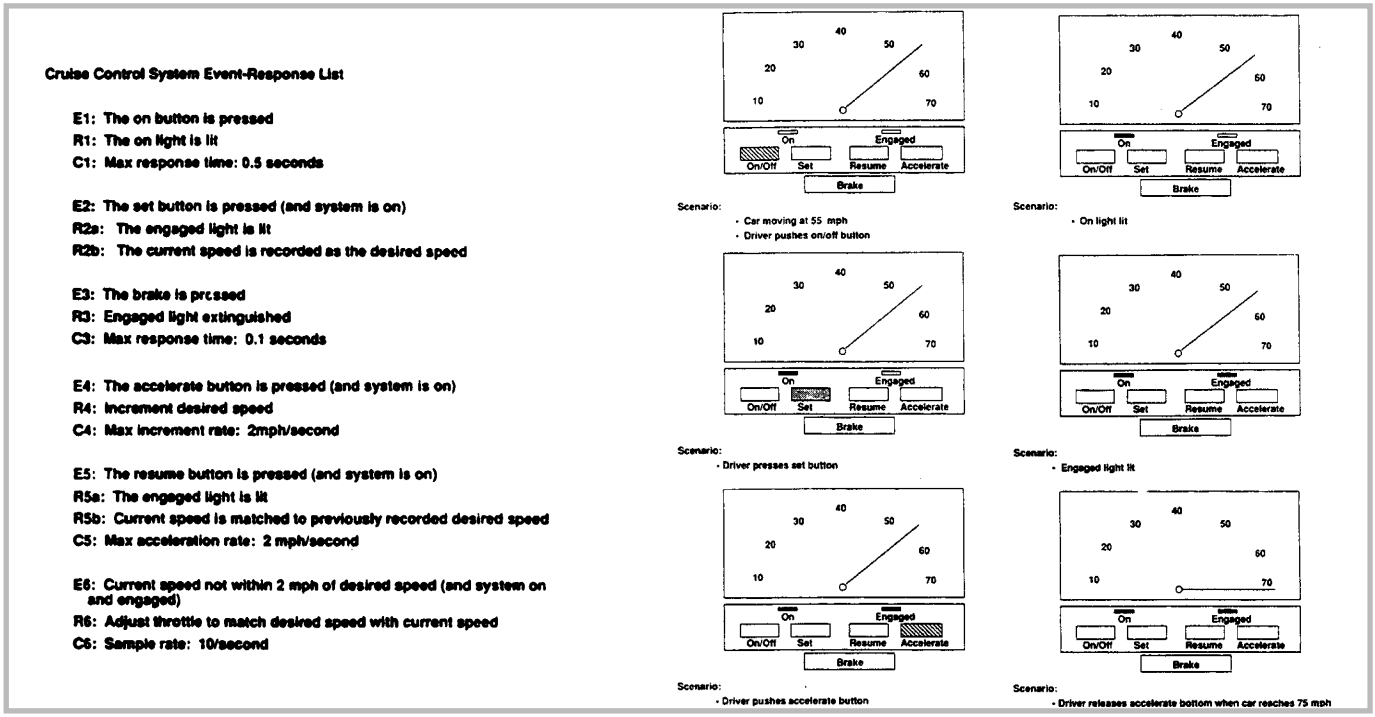
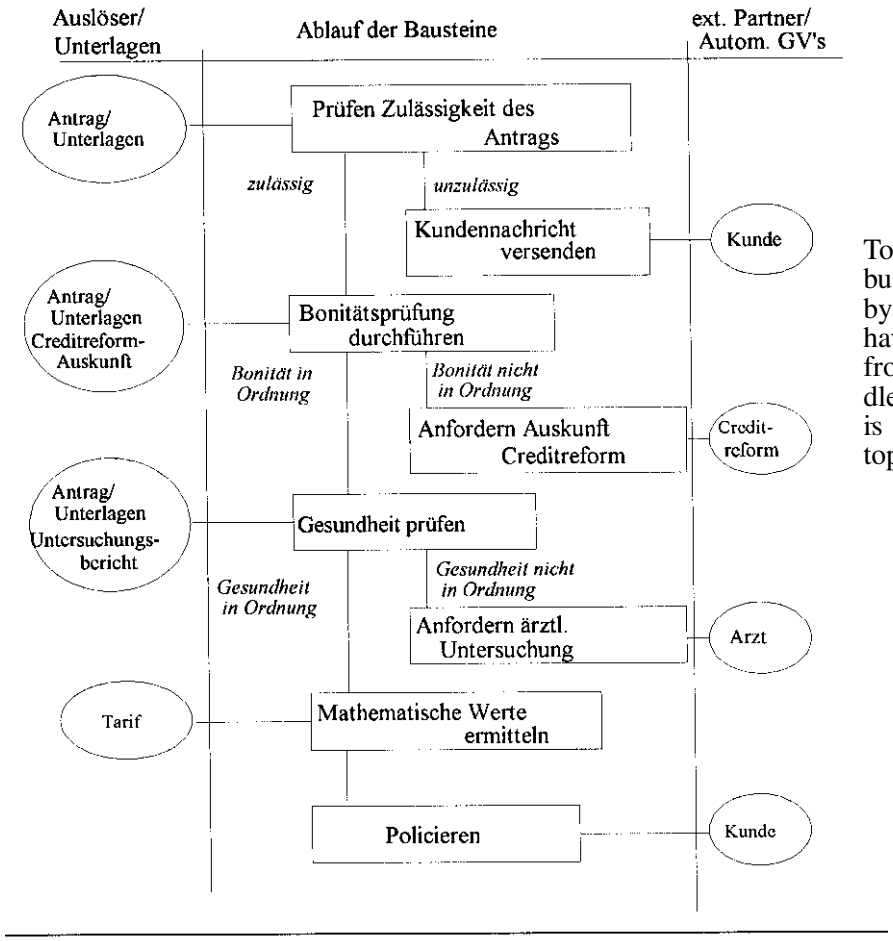


Figure 50: Event-response list and sample storyboard in OORD [Umphress91]

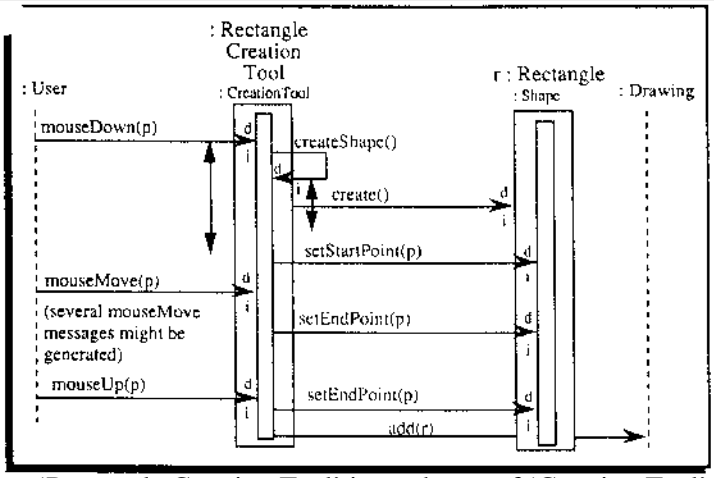
Neuantrag einer Lebensversicherung

Abbildung 1



To the left we have the trigger of the business process and documents needed by the business process. To the right we have external partners that get messages from the business process. In the middle, the sequence of the building blocks is displayed. The diagram is read from top to bottom and left to right.

Figure 51: Graphical representation of a "Geschaeftsvorfall" in [Mueller93]



- 'Rectangle Creation Tool' is a subtype of 'Creation Tool'
- the horizontal arrows point to the objects which are referenced in the operation call
- 'd' shows in which type the operation is declared
- 'i' shows in which type the operation is implemented
- the vertical arrows indicate the durations of the operations

Figure 52: Bergen interaction diagram, captures also the inheritance hierarchy of the objects [Baklund95]

Part of the description of the framework Dispatcher/Pane:

```
behaviour:
    create ;
    open ;
    (windowActivating;
       (update*;
        cursorEntersPane;
          (update | userInput)*;
        cursorLeavesPane
       )*;
     windowDeactivating
    )*
    terminate.

 paths:
    create:
"path to create the group"
        model  >>
           pane model:;
"specifies the model"
           [pane name:];
"specifies the pane's name"
           [pane paneMenuSelector:].
"specifies a message that can be sent to the
model to obtain the menu"

    terminate:
"path to terminate the group"
        close.

    open:
       topPane  >>
          pane reframe:;
"determine rectangle for the pane's area"
          pane open.

    close:
       topPane >>
          pane close ;
          model release.
"break connections between the model and the
pane"



update:
        model >>
"update the displayed information"
           refresh
           | (pane update: ; refresh)
           | (pane update:with: ; pane perform:)
           | (pane update:with:with: ;
                    pane perform:with:).
    cursorLeavesPane:
       USER >>
          dispatcher processInput ;
          dispatcher deactivate .
    cursorEntersPane:
       topDispatcher >>
          dispatcher processInput ; dispatcher

activate ;
    windowActivating:
       topPane >>
          pane displayWindow .
    windowDeactivating:
       topPane >>
          pane graySelection .
    userInput:
"handle user input"
        USER >>
          dispatcher processInput ; dispatcher
processKey ;
          (menuRequest | keyInput | ctrlKeyInput |
             functionKeyInput).
    menuRequest:
          dispatcher processFunctionKey ;
          (topPane performMenu
          | (pane performMenu;
                  (topPane performMenu
             | model perform:))).
    functionKeyInput:
          (dispatcher select
          | (dispatcher processKey: ;
              dispatcher processFunctionKey:)).

 exception paths:
    none
```

Figure 53: Description of a group of interacting objects using path expressions (PEG) [Adam94]

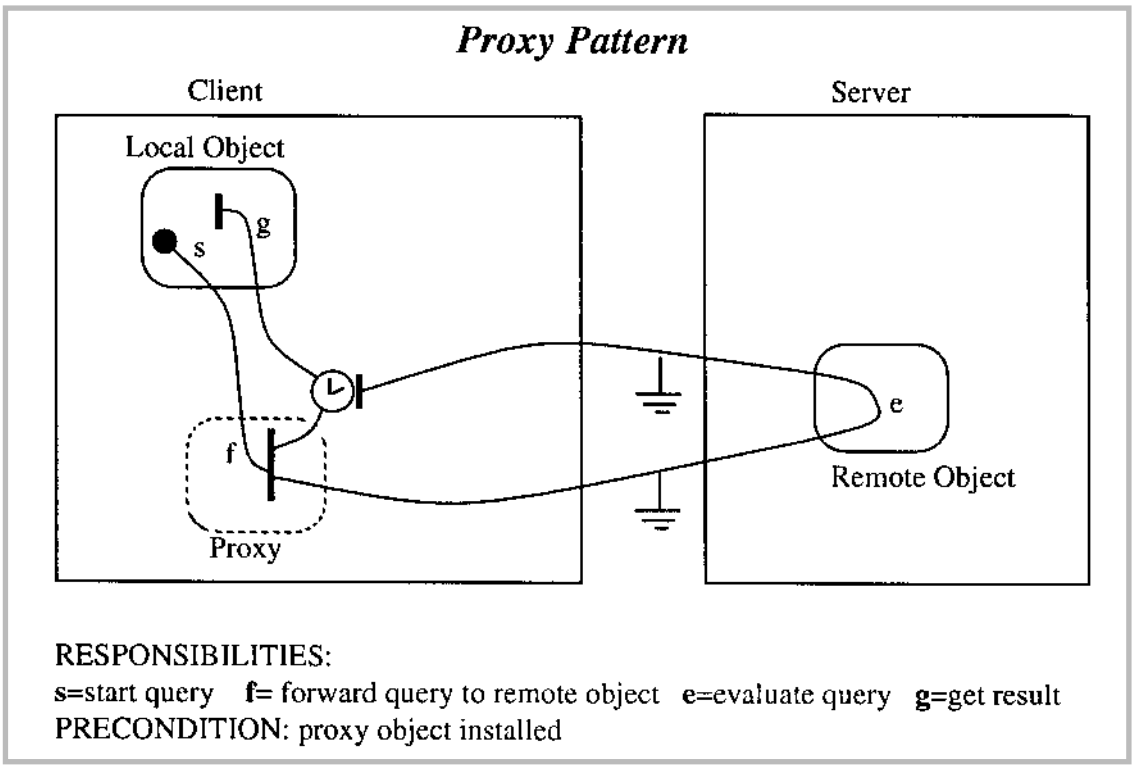


Figure 54: Use case map for a design pattern [Buhr96]

5 Conclusions

We have investigated various methods that offer notations for modelling global behaviour and have shown some of their underlying concepts and the relations between these concepts. We have seen that no single technique can show all interesting aspects of global behaviour equally well. It would be often helpful to use several notations interchangeably, each one with its own strength. In order to make this feasible, we would need automated support for switching between the different representations. But there are two obstacles to this. Right now, a generally agreed and consistent set of basic concepts concerning the definitions of interactions, events, messages, operations and scenario types, and the mapping between these terms does not exist. Some methods even do not have a consistent set of definitions and concepts within their own notations. Because no simple set of consistent concepts would satisfy all possible variants of dynamic diagrams known right now, a choice would be needed. Further, a tool that really supports modelling scenarios must allow the user to play around with the different views of the system. This would imply very sophisticated view, change and transformation mechanisms, so that the tool takes over as much work as possible.

6 References

[Adams92] G. Adams; *Describing Groups of Interacting Objects Using Path Expressions*; master thesis, school of computer science, Carleton University, Ottawa, Ontario, 1992

[Adams94] G. Adams, J.P. Corriveau; *Capturing Object Interactions*; Proceedings of TOOLS Europe '94, Versailles, 1994

[Armour95] F. Armour et. al; *Use Case Modeling Concepts for Large Business System Development*;Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Baklund95] E. Baklund et. al; *Extending Interaction Diagrams*; ROAD July-August 1995

[Beck89] K. Beck and W. Cunningham; *A Laboratory for Teaching Object-Oriented Thinking*; Proceedings of OOPSLA 89l, 1989

[Benner93] K.M. Benner et. al; *Utilizing Scenarios in the Software Development Process*; Information Systems Development Process, edited by N.Prakash et. al, IFIP Transactions A-30, 1993

[Berard93] E.V. Berard; *Essays on Object-Oriented Software Engineering, Volume1*; Prentice Hall, 1993

[Beringer94] D. Beringer; *Limits of Seamlessness in Object-Oriented Software Development*; Proceedings of TOOLS Europe '94, Versailles, 1994

[Berteaud95] R. Berteaud, J. Bezivin; *Requirements Modeling in the OSMOSIS Workbench*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Booch91] G. Booch; *Object Oriented Design with Applications*; Benjamin/Cummings, 1991

[Booch94] G. Booch; *Object Oriented Analysis and Design*; Benjamin/Cummings, 1994

[Booch95] G. Booch, J. Rumbaugh; *Unified Method for Object-Oriented Development*; Documentation Set Version 0.8, Rational Software Corporation, 1995

[Brantschen91] S. Brantschen; *Essentielle Systemanalyse, Moderne Strukturierte Analyse*; seminar of the GfAI Gruppe für angewandte Informatik AG, 1991

[Briod93] P.A. Briod, S. Moser, G. Wanner; *BI-CASE/OBJECT V1.1 - A Modern Systems Development Methodology*; Bedag Informatik, Berne, Switzerland, 1993

[Buhr95] R.J.A.Buhr; *Use Case Maps: A New Model to Bridge the Gap Between Requirements and Design*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995; see also *Use Case Maps for Object Oriented Systems*; Prentice Hall, 1995

[Buhr96] R.J.A.Buhr; *High-Level Design with Use Case Maps*; Tutorial OOPSLA 96, 1996

[Coleman94] D. Coleman et al.; *Object-Oriented Development: the Fusion Method*; Prentice-Hall, 1994

[Coleman95] D. Coleman; *Fusion with Use Cases - Extending Fusion for Requirements Modelling*; presentation at the Fusion Users Meeting of OOPSLA 95; 1995

[Cook94] S. Cook, J. Daniels; *Designing Object Systems, Object-Oriented Modelling with Syntropy*; Prentice Hall, 1994

[Cotton95] T. Cotton; *Evolutionary Fusion: A Customer-oriented Incremental Life Cycle for Fusion*; Fusion in the Real World, edited by R. Malan, R. Letsinger and D. Coleman, Prentice-Hall, 1995

[Dedene94] G. Dedene, M. Snoeck; *M.E.R.o.DE: A Model-driven Entity-Relationship object-oriented DEvelopment method*; ACM SIGSOFT, vol 19 no 3, 1994

[Dubois93] E. Dubois et al.; *O-O Requirements Analysis: an Agent Perspective*; Proceedings of the ECOOP 93 in Kaiserslautern, 1993

[Firesmith95] D.G. Firesmith; *Use Cases: The Pros and Cons*; ROAD July-August 1995

[Glinz95] M. Glinz; *An Integrated Formal Model of Scenarios Based on Statecharts*; Proceedings of the 5th European Software Engineering Conference ESEC '95, Sitges, Spain, 1995

[Graham93] I. Graham; *Financial Object-Oriented Rapid Analysis Method (FORAM), Training Notes 1 to 4*; seminar given at the SBV in Basel

[Graham94] I. Graham; *Beyond the Use Case: Combining task analysis and scripts in object-oriented requirements capture and business process re-engineering*; Proceedings of TOOLS Europe' 94, Versailles, 1994

[Graham94a] I. Graham; *Object-Oriented Methods, 2nd Edition*; Addison-Wesley, 1994

[Graham95] I. Graham; *Business Process Modelling*; Object Expert, November-December 1995 and January-February 1996

[Hansen95] F. Hansen; *Experiences with OMT Analysis: "From the Real World to an Application"*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Holbrook90] H. Holbrook; *A Scenario-Based Methodology for Conducting Requirements Elicitation*; ACM SIGSOFT, vol 15, no 1, 1990

[Hsia94] P. Hsia et al.; *Formal Approach to Scenario Analysis*; IEEE Software, March 1994

[Jacobson92] I. Jacobson; *Object-Oriented Software Engineering*; Addison Wesley, 1992

[Jacobson95] I. Jacobson; *A growing consensus on use cases*; JOOP March-April 1995

[Kilberth93] K. Kilberth, G. Gryczan, H. Zuelligkoven; *Objektorientierte Anwendungsentwicklung: Konzepte, Strategien, Erfahrungen*; Verlag Vieweg, 1993

[Kowal92] J. A. Kowal; *Behavior Models, Specifying User's Expectations*; Prentice Hall, 1992

[Loenvig95] B. Loenvig; *Experiences with Requirement Capturing, Specification and OO Requirements Analysis*; Workshop on "Requirements Engineering: Use Cases and More", OOPSLA 95, 1995

[Martin95] J. Martin and J. Odell; *Object-Oriented methods: a Foundation*; Prentice Hall, 1995

[McMenamin84] M. McMenamin, J.F. Palmer; *Essential Systems Analysis*; Yourdan Press, 1984

[Mueller93] G. Mueller-Luschnat et al.; *Objektorientierte Analyse und Geschäftsvorfallmodellierung*; Proceedings of the EMISA-Fachtagung über objektorientierte Methoden für Informationssysteme, Universität Klagenfurt, 1993

[Odell93] J. Odell; *Using business rules with diagrams*; JOOP July-Aug. 1993

[OMG92b] *Object Analysis and Design, Volume 1: Reference model, Draft 7.0*; OMG, 1992

[Regnell96] B. Regnell et. al; *A Hierarchical Use Case Model with Graphical Representation*; Proceedings of ECBS'96, March 1996

[Reisin90] F.Reisin; *Kooperative Gestaltung in partizipativen Softwareprojekten*; Dissertation, Technische Universität Berlin, 1990

[Rubin92] K.S. Rubin, A. Goldberg; *Object Behavior Analysis*; Communications of the ACM, Sept. 1992

[Rumbaugh91] J. Rumbaugh et al.; *Object Oriented Modelling and Design*; Prentice-Hall, 1991

[Rumbaugh94] J. Rumbaugh; *Modeling & Design*; JOOP, issues of June 1994, Sept 1994, Nov/Dec 1994, Febr 1995, March/April 1995 and May 1995

[Rumbaugh94b] J. Rumbaugh; *Getting started, Using use cases to capture requirements*; JOOP September 1994

[Stein93] W. Stein; *Objektorientierte Analysemethoden - ein Vergleich,* Informatik-Spektrum (1993) 16:317-332, Springer-Verlag, 1993

[Stein94] W. Stein; *Objektorientierte Analysemethoden: Vergleich, Bewertung, Auswahl*, B.I. Wissenschaftsverlag, Mannheim, 1994

[Umphress91] D. A. Umphress, S.G.March; *Object-oriented requirements determination*; in JOOP Focus on A&D, edited by R. S. Wiener, SIGS Publications, 1991

[Ungar91] D. Ungar, R. B. Smith; *SELF: The Power of Simplicity*; Lisp and Symbolic Computation, vol4 no3, 1991

[Waldén95] K. Waldén, J.M. Nerson; *Seamless Object-Oriented Software Architecture*; Prentice Hall, 1995

[Wilkinson95] N. M. Wilkinson; *Using CRC Cards: an informal approach to object-oriented development*; Prentice Hall, 1990

[Wirfs90] R. Wirfs-Brock et al.; *Designing Object-Oriented Software*; Prentice Hall, 1990

[Yourdon89] E. Yourdon; *Modern Structured Analysis*; Prentice Hall, 1989

[Zorman95] L.A. Zorman; *Requirements Envisaging by Utilizing Scenarios (REBUS)*; PhD Thesis, USC/ISI USA,, 1995