ON THE ROLE OF SCENARIOS IN OBJECT-ORIENTED SOFTWARE DESIGN

Kai Koskimies, Tatu Männistö, Tarja Systä
and Jyrki Tuomi

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1996-1

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1996-1, JANUARY 1996

ON THE ROLE OF SCENARIOS IN OBJECT-ORIENTED SOFTWARE DESIGN

Kai Koskimies, Tatu Männistö, Tarja Systä
and Jyrki Tuomi

University of Tampere
Department of Computer Science
P.O.Box 607
FIN-33101 Tampere, Finland

ISBN 951-44-3920-1
ISSN 0783-6910

On the Role of Scenarios in Object-Oriented Software Design

Kai Koskimies[1], Tatu Männistö[2], Tarja Systä[1], Jyrki Tuomi[1]

[1] Department of Computer Science, University of Tampere, Box 607, FIN-33101 Tampere, Finland

[2] Laboratory of Software Engineering, Tampere University of Technology, Box 526, FIN-33101 Tampere, Finland

Abstract

Scenario diagrams are a graphical notation for describing the interaction of a set of collaborating objects. We study the relationships between scenarios and other models used in object-oriented software development, in particular dynamic model and static object model. These relationships are significant for improving various consistency checks between the different models and for developing automated support for model synthesis.

1 Introduction

A fundamental problem of software development is how to derive design information from the requirements of a software system. Object-oriented design methods address this problem by identifying conceptual objects in the system's specification, and by representing these with classes. The different aspects of objects are captured with special notations; in particular, the static relations of the classes are described using some variant of entity-relationship diagrams, the dynamic behavior of (active) objects is described using finite state machines, and the information flow of the system is presented with data flow diagrams. A popular object-oriented design method is OMT [12] which we will use as the basis in this paper as far as the notation is concerned.

Although the object-oriented approach supports the shift from analysis to design by providing a common paradigm for analysis and design phases, the step from externally observable properties of the application to the description of software units (classes) remains problematic. This is particularly reflected in the difficulties to specify the functionality of objects. As noted e.g. by Rumbaugh [13], the conventional dynamic description vehicle — a state machine — is relevant only for a minority of the objects, since most objects do not undergo significant state changes. Instead, usually the behavior of the object should be specified as a set of operations (messages) the object accepts. A common weakness of most object-oriented methods — including OMT — is that they do not clearly express how such sets of operations can be found.

An attractive approach to object-oriented design is to consider descriptions of the expected responses of the system to certain logical event sequences caused by the environment (e.g. user actions), and to analyze these descriptions to find out various properties of the objects involved. Such descriptions are usually presented in a graphical notation called an event trace diagram [12] or an interaction diagram [7]; we will use here the term *scenario* (diagram). Design methods making extensive use of this idea include OMT++ [1] and OOSE [7]. The advantage of such a method is that the required functionality of the objects can be observed in the scenarios and expressed either as a set of operations or as a state machine.

A scenario diagram describes the interaction of a set of objects in terms of message passing. For each participating object there is a vertical line. A message sent from one object to another is presented as a horizontal arrow from the sender object to the receiver object. Time flows from top to bottom. An example of a scenario is shown in figure 1.

There are many variants and extensions of this basic form (see e.g. [4, 7, 9]). When

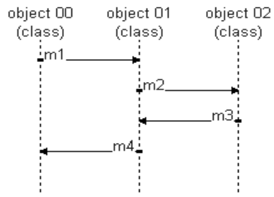

Figure 1: A scenario diagram

scenarios are used as an essential part of the design methodology, their relations to other design documents and techniques become important. These relations play a crucial role when some design documents are derived (possibly automatically) from the others, or when the consistency of the various design documents should be preserved (possibly automatically). Hence these issues are relevant for the design of any kind of CASE tool intended to support an object-oriented software development method relying on scenarios.

In this paper we will consider the relations of scenario diagrams to other design documents used in the OMT method. With the exception of choosing OMT as the notational basis, we will analyse the relations of the various design documents without making assumptions of the nature of a design method or of a particular tool making use of the relations. In the next section we discuss the relations between scenarios and state machines, which obviously are the strongest and also the most useful. In section 3 we will study the relationships between scenarios and the static object model. In section 4 we show how the implementation of the operations can be derived from scenarios. Finally in section 5 we present some concluding remarks.

We assume familiarity with the OMT graphical notation. This work is part of the SCED project aiming at automated support for dynamic modeling in object-oriented software development. Many of the ideas discussed here have been exploited in the current SCED system [9, 11].

2 Scenarios and dynamic model

2.1 General relationships between scenarios and state machines

Since both scenarios and state machines describe the dynamic aspects of a system, they necessarily share much in common. In its basic form a scenario describes a particular event sequence occurring among a set of objects, wheras a state machine desribes the complete behavior of a single object in terms of responses to received events. Hence neither a scenario is completely implied by a state machine nor a state machine is completely implied by a scenario. If all objects are assumed to have a state machine, a scenario represents a trace of a collection of collaborating state machines. The events related to a particular object in a scenario represent a trace through the state machine of the object.

Sometimes the scenario diagram notation is extended with algorithmic constructs like conditionality and repetition ([1, 9, 3]). In figure 2, the notation for conditionality used in [9] is shown. With special notations for conditionality and repetition, a scenario

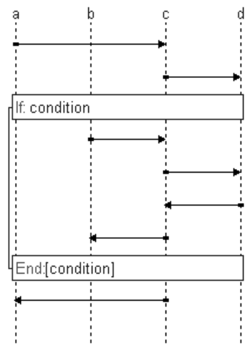

Figure 2: Conditionality in SCED scenario

diagram has the full power of an algorithmic language, and it can be used to describe a complete function with multiple executing objects, rather than only one example trace. We call such descriptions *multi-object functions*. Clearly, a conventional scenario is an instance of a multi-object function. Still, one object can take part in several multi-object functions; therefore multi-object functions have in principle the same relation with state machines as conventional scenarios. One can view a multi-object function as a shorthand notation for a (possibly infinite) set of scenarios. The relationships between the various concepts are depicted in figure 3.

2.2 Deriving state machines from scenarios

As demonstrated in [8], it is possible to synthesize automatically a minimal state machine which is able to execute all the given scenarios with respect to a certain object. This algorithm essentially realizes the instance-of relation (1) in figure 3: a state machine is generated from a set of scenarios taking into account only the role of a particular object in the scenarios. We will not discuss this algorithm here in detail (see [8]).

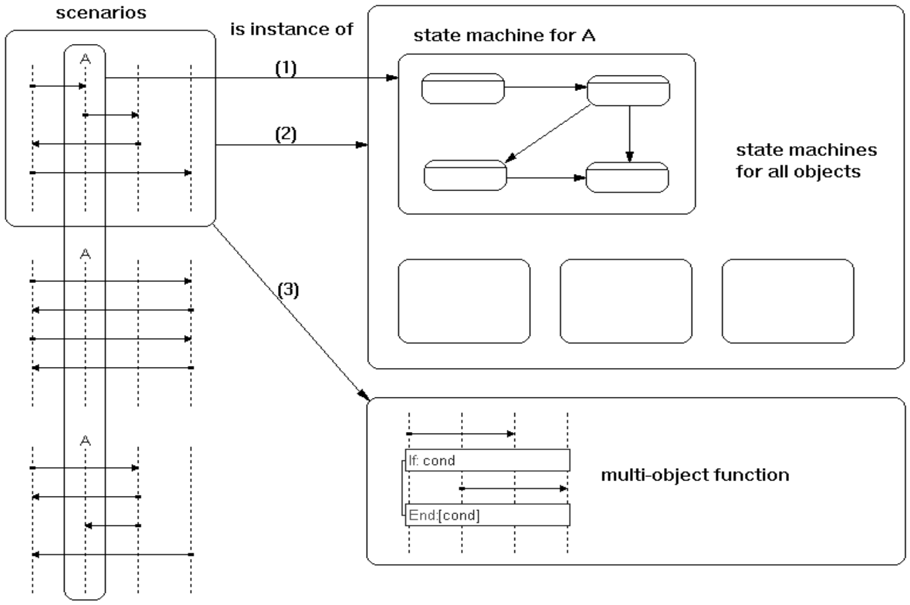

Figure 3: Relationships between the concepts. The arrows (1), (2), and (3) denote "instance-of" relations.

The use of multi-object functions as the basis of state machine synthesis gives rise to some additional problems. A conditional structure in a scenario can be simply interpreted as a shorthand for two variants of the scenario: one in which the condition holds and one in which it does not hold. Hence a set of scenarios containing conditional structures can be expanded into an equivalent (but larger) set of scenarios without conditional structures. However, the presenting of the condition in the resulting "pure" scenarios is less obvious. A natural representation of a conditional structure in a state machine is a state with two outgoing transitions associated with guards (OMT term for a condition associated with a transition). This effect is achieved if the controlling object receives an event labelled with the condition in a scenario. Since this kind of an event has no actual sender object, an expanded scenario should include special "condition" boxes representing events without sender. This kind of special event will be necessary also for operation code generation, as discussed in section 4.

However, it is not clear *which* object should take the responsibility of checking the condition; i.e. which object will have the condition box in the expanded scenario. This problem is avoided if the notation for conditionality in a scenario specifies the responsible object, like in [3]. On the other hand, especially in the analysis phase it is useful to allow condition expressions which do not need to specify the responsible object; this has been the motivation for the more general notation of SCED in figure 2.

If the system is based on fully synchronous message passing (i.e. there is always exactly one object controlling the execution), a condition is sensible only if it is checked by the object having the control at the point of the conditional structure. In particular, this is the case if the events correspond to conventional method calls. However, in a system consisting of concurrently executing objects there may be several objects which could in principle be responsible for checking the condition. Even in such a case there is one object which is most likely the responsible one, namely the object which is the sender of the first event within the if-construct. Only in that case it is possible that the condition forks the future execution; otherwise the condition could serve only as an assertion. The handling of an if-construct is depicted in figure 4.

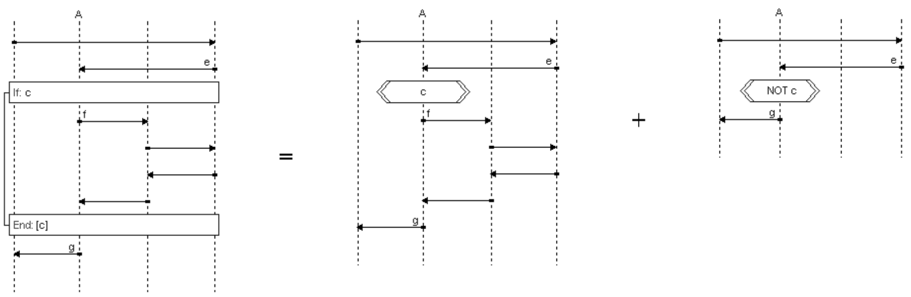

Figure 4: Expanding conditional construct.

To allow the user to clearly specify the object which is responsible for checking the condition, the "structured" if-construct of SCED should be augmented with the possibility to mark one of the objects as the "owner" of the condition. If the mark is missing, the default owner should be the sender of the first event within the if-construct.

SCED also allows a repetition construct which has appearance similar to the if-construct, except that the keyword "If" is replaced with "Repeat". However, the handling of a repetition construct is somewhat different, because it is not possible to unravel this construct into a finite number of "pure" scenarios. Hence a repetition must be viewed as a state machine fragment rather than as a set of scenarios. The fragment contains a loop state which has two outgoing transitions: one associated with a guard identical to the condition, leading to the body of the repetition construct, and the other, identical to the negation of the condition, leading to the continuation of the construct. Nevertheless, the problem of finding the object responsible for controlling the loop is exactly the same as for the if-construct, and it can be solved using the same principles.

2.3 On the consistency of scenarios and state machines

Since any new scenario can be fused with an existing state machine using the algorithm of [8], it is not possible to write a "wrong" scenario with respect to an existing (so far incomplete) state machine. However, if scenarios and state machines can be edited independently of each other, various kinds of inconsistencies can be created.

Assume first that an existing scenario is edited, and that there is a state machine of an object that is able to execute the object's role in that scenario. Since the scenario is edited explicitly by the user, we must assume that the new version of the scenario is correct information. Hence the state machine should be updated according to the new version of the scenario.

A non-incremental update of the state machine would be trivial, if the state machine has been generated automatically from the scenarios using e.g. the algorithm of [8]: one can simply regenerate the state machine using the revised set of scenarios. However, it may be possible that the state machine has been manually extended after the generation, so that the state machine contains genuine new information in addition to the information synthesized from the scenarios. That is, the state machine may execute traces that cannot be inferred from the scenarios, and this ability should be retained. A regeneration would destroy manual extentions of the state machine: it would be in general hard to restore the effects of editing in the new state machine. Moreover, a non-incremental approach for updating the state machine has one important difference when compared to an incremental approach: a non-incremental method gives no opportunity for the user to examine how changed scenarios influence the state diagram scenario by scenario. If several scenarios have been changed, it may be useful at least to have the possibility not to update all changes at the same time.

An incremental update of the state machine can be obtained, if one keeps track of the scenarios contributing to a state transition. This can be implemented simply as follows: each transition is associated with an integer giving the number of scenarios using this transition; we call this the *scenario counter* of the transition. When a scenario is changed, it is removed from the state machine: the scenario is run through the state machine, and each transition it uses is marked. Then the scenario counters of the marked transitions are decremented by one. If some scenario counter becomes zero, the corresponding transition is removed. If some state becomes in this way isolated from the other states, it is removed as well. After the effect of the original scenario is removed from the state machine, the new version of the scenario can be added to the state machine using the (incremental) algorithm in [8].

Assume now that an existing state machine is edited, and that there is a set of scenarios contributing to this state machine. As long as one adds new states and transitions to the state machine, no inconsistencies may be created: these changes cannot prevent the state machine from executing the scenarios. However, if a state or a transition used by a scenario is removed (modification can be viewed as removing and adding), or new actions are attached to it, a conflict arises: a scenario cannot be run through the state machine any more. Transitions whose scenario counter is zero (i.e. they have been added by direct state machine editing) can be edited without problems, as well as states which are associated only by such transitions. We call the other transitions and states *scenario-sensitive*.

The problem of having invalid scenarios after editing the state machine is more difficult than the reverse problem. Note that the reverse problem was relatively easy to solve because there was an incremental algorithm for updating the general description (state machine) with the information contained in a new instance (scenario). However, we feel that revising scenarios automatically on the basis of state machine editing is much more questionable: scenarios can be viewed as requirements that should hold after any state machine modifications, rather than implications of the state machine. A straightforward approach to solve this problem is therefore simply to consider changing the scenario-sensitive parts of a state machine illegal. Since the scenario-sensitive parts can be recognized using scenario counters, this is easy to implement. An actual system might either give a warning or refuse to perform the editing action. In all cases it would be useful to show the conflicting scenarios to the user. To implement this, all the scenarios must be executed by the state machine, and those scenarios making use of the removed transitions are marked.

However, sometimes the user might actually want to edit the scenarios "automatically" through the state machine. That is, the user may wish to enforce a desirable general behavior, and to adjust the existing scenarios according to this view. Hence the principle of considering scenario-sensitive editing of a state machine simply illegal should perhaps be relaxed: an optional mode of operation allowing the automatic adjusting of the scenarios could be provided. Note that we consider here only scenario-sensitive editing: modifying a state machine otherwise causes no revisions in the scenario set.[1] For instance, editing names of transitions, states, and actions globally through a state diagram might be desirable. The implementation of such glogal editing property is straightforward.

Assume that the user removes a scenario-sensitive transition in a state machine. This can be regarded as a collective editing action: "remove all events causing this transition in all scenarios". Accordingly, this can be implemented as follows: we again run all the scenarios through the state machine, and mark all events in scenarios that traverse along the removed transition. All scenarios that have marked events are removed from the state machine, as described above. Then the marked events are removed from the scenarios, and the revised scenarios are added to the state machine using the incremental algorithm of [8].

However, in spite of its logical basis this approach has a serious flaw from the practical point of view. The problem is that the state machine obtained as the result of the above algorithm is not the same as the state machine that would appear as the result of the editing action itself. For example, assume that the only transition connecting two states is removed. If this transition was caused by an event in a scenario, this event will be removed. This in turn implies that the situation after the event automatically follows the situation before the event. Consequently, a so-called automatic transition will appear in the place of the removed transition — an effect the designer hardly expects.

Hence any sensible collective editing of scenarios through the state machine should guarantee that the total effect on the state machine is equivalent to the effect of the pure editing action. Some special kinds of editing actions have this property. For example, one can edit transitions by detaching their target or source states. Sensible updating

[1] A possible approach would be to require full consistency in the sense that a state machine is always exactly the result of the synthesis algorithm, when applied to the current set of scenarios. However, this approach seems unrealistic because it would imply that some artificial scenarios are generated after each adding of new transitions or states. Such scenarios would only distract the user since they have no other meaning except that of preserving the full consistency.

of the scenarios is in this case easier: If the new target/source state lies on the path of a scenario, events between corresponding points in a scenario are replaced by a single arriving event named as the transition. If the new target/source state doesn't lie on the path, events after/before the transition are all removed and events corresponding to the new target/source state are added to replace them. Effects of an addition of new actions to scenario-sensitive parts of a state diagram can also be updated to scenarios without causing any side effects as long as the receiver object of each action is defined.

2.4 Using state machines for creating scenarios

Animation is a powerful way to analyze, debug, and visualize interactions between objects. A scenario is a natural and descriptive way to document animation steps. Several CASE tools have been developed for animating the dynamic behavior of systems composed of objects. In [14] Salmela introduces a framework in which animation techniques to visualize object communication, dynamic instantiation, dynamic binding and execution of both functional and dynamic models are described. The animation framework is based on OMT notation and uses highlighting, token placing, and token moving as visualization techniques. The highlighting technique is also applied, e.g. in Statemate [6], in which it is used in animation of statecharts (see [5]) and activity charts. In Scene [10], object-oriented programs are visualised using scenarios produced automatically during the execution of a target system.

We will discuss here the generation of (parts of) scenarios using existing state machines during the design of the dynamic model. This is in fact close to animating a system represented by a set of state machines and displaying the result as a scenario. However, in contrast to conventional animation systems one can add new behavior to the system during the animation process. This kind of technique could be characterized as *design-by-animation*.

When synthesizing a state machine for a participant on the basis of information given in scenarios, sent events are interpreted as actions and received events as transitions ([8, 11]). The same interpretation applies to interacting state machines used when synthesizing a scenario from a state machine set. We will call the process of synthesizing a scenario using existing state machines *tracing*.

For example, let SD_1 and SD_2 be two state machines being in states A and B, respectively. If state A has an action, say *act*, and state B has an outgoing transition also labeled *act*, then execution of action *act* of state machine SD_1 is interpreted as event *act* sent by SD_1 and received by SD_2. Further, it causes state machine SD_2 to change states from B to the target state of the transition with label *act*. All actions in SCED state machines are considered to be instantaneous events, sent to one object only. In addition, they are regarged to be synchronous; next action won't be executed before the previous action is completed.

The designer can start to trace a scenario after constructing state machines for the objects she wants to take part in the scenario. By guiding the tracer to select desired paths through these state machines, the designer gets an example sequence of interactions between the corresponding objects, i.e. a scenario. This tracing process is basically handled in the following way: the designer selects either a state or a transition at a time when guiding the tracer. As a result, the selected state (or the target state of the selected transition) becomes the current state of the state machine. The possible event sending action associated with the state is executed. This means that all the other state machines are examined to find out if one of them is able to respond to the event in its current state (that is, there is a leaving transition for the current state with the event). If such a state machine is found, the event is sent to this machine which changes its current state accordingly. The possible event sending action in the new state is again executed etc.

Proceeding in this way the tracer activates states and transitions automatically, showing the activated paths in the state machines by highlighting, until it cannot find an explicit path to continue. If the underlying collection of state machines is a complete system, the fact there is no path to continue can only mean that the system expects input from the user of the system. Automatic tracing continues as soon as the designer has made it possible by guiding the tracer with her state/transition selections specifying the user's response. A log of the animation is shown in a form of a scenario. The scenarios thus synthesized do not make use of the more advanced SCED concepts (e.g. comments, subscenarios, if- and repeat-structures etc.), but otherwise they are normal scenarios that can be further edited if desired.

Note that OMT kind of state machines can have several actions attached to states and transitions. The role of these actions has to be taken into account. E.g., if a state is activated, the tracer goes through entry and normal actions of that state. When a transition is selected, the effects of exit actions of the source state as well as actions attached to the transition have to be enforced before activating the target state.

Above we discussed the animation of a complete state machine system and the presentation of the log of the animation as a scenario. However, this technique becomes more interesting when we assume that the system to be animated is not a complete one, i.e. there is one object whose behavior is unknown (besides the user of the system). It is then the task of the designer to act in the role of the unknown object during the tracing process. That is, when the animated set of state machines is not able to continue because it is waiting for an event sent by the unknown state machine, the designer intervenes and specifies the response of the unknown object by showing which transition should be activated next. As an example, such an unknown object might be a new GUI component making use of existing ones with known behavior. Note that it must be possible to distinguish the situations requiring the response of the user from those requiring the response of the unknown object.

More precisely, in the current SCED the tracing with an unknown object proceeds as follows. The tracer gives the unknown object name *Untitled*. A scenario with participants for all open state machines, and with participant *Untitled* is created. The names for the other participants are taken from the file names of the corresponding state machines. Object *Untitled* is considered to take part in the tracing process as a sender or a receiver of an event if no other (predefined) object can do it. Hence, this object is regarded as a sender of an event every time the designer has to help the tracer to continue by selecting a transition, and as a receiver of an event when the designer activates an action for which no responding state diagram can be found. If there are names of sender participants (written after a keyword "FROM") attached to transitions or receiver participants (written after a keyword "TO") attached to actions in a state machine, the sender or receiver participants are determined accordingly. These participants can also be manually inserted ones. It should also be noted that the designer may freely edit the traced scenario any time during the tracing process.

The first scenario traced in this way can be used for synthesizing a skeleton of a state machine for the object *Untitled* using the normal state machine synthesis algorithm. This state machine is added to the state machine set and can be used to make the subsequent tracing process more powerful: some of the events that were added as a result of the designer's selections can be automatically inserted in the next scenarios. In other words, paths need to be constructed only once; after that they can be followed automatically. The cycle of tracing scenarios and synthesizing them to the existing state machine *Untitled* is repeated until a satisfying scenario set (or state machine) has been achieved.

While tracing a scenario there might be several objects that could respond to a sent event. In that case the designer may continue the tracing process after selecting the right (desired) object from a given dialog box.

While being in a certain state, the object may receive events which may be:

1. *accepted* causing a change of state and/or an action to be performed or
2. *ignored*.

Received events may be ignored, because:

1. the object does not recognize the event or
2. the object recognizes the event but it is not allowed to respond to it while being in the current state

Constructing scenarios correctly requires that the designer knows which events can be accepted by an object and when. To prevent the designer from using events that should be ignored, the former case is not a problem in OMT methodology: the designer can check from the object model which events can be recognized by the object. However, the latter case is more difficult to ensure.

It is much easier for the designer to see from a state machine if the current state is allowed to accept the event or not, than from a scenario. Assume that the designer has a set of correctly constructed state machines, perhaps after some modifications, and she starts to trace a scenario. As far as she selects transitions or states attached to the currently active state, it is impossible for her (unless she makes manual modifications) to trace a scenario in which a participating object accepts an event even though it should be ignored. Hence the support for constructing scenarios correctly is much better than if scenarios are constructed from scratch; in addition to avoiding events that ought to be ignored, the designer can be more certain that objects react to received events correctly.

The most important advantages of the scenario tracing property are:

1. scenarios are easy to construct; only few (if any) events need to be drawn and labeled manually, other events will be constructed either automatically or as a result of selecting a state or a transition;
2. support for avoiding the use of events which are recognized by the object but not allowed in the current state;
3. support for describing objects' reactions to received events correctly; object's reaction may depend on its state;
4. support for animating the behavior of the system in example cases;
5. early visibility of intended features;
6. changing the dynamic modeling process smoothly from the "water fall" type of modeling (first scenarios, then state machines) to more spiral way of modeling.

Scenario tracing property seems especially useful in following cases:

1. modeling the use of predefined classes, e.g. GUI library classes;
2. checking the correctness of modeled behavior of objects.

If the object to be modeled (*Untitled*) inherits a predefined class, the object is able to use the methods and variables (in C++: public and protected ones) of the class. In that case it may be possible to form a skeleton of the state machine on the basis of the inherited class before tracing any scenarios.

Some problems arise in this approach, too. First, visualizing animation steps is not a trivial problem. Currently, the tracing process is visualized by highlighting the current states and automatically followed paths after previous designer's selection in all the involved state machines. However, state machine windows are typically large. This makes it difficult and in most cases impossible to show all the state machines at the same time; if these windows do not overlap, the designer is able to see only a small corner of them and is at least forced to scroll them a lot.

Perhaps the biggest problem of scenario tracing results from the fact that all events for which a sender/receiver object cannot be found are attached to the *Untitled* participant. In a traced scenario there might be several manually inserted participants with no corresponding state machine, e.g. participants that represent the system border. Events that should be attached to them can be concluded during the tracing process only if the receiver names are used with actions (e.g. *SendOkMessage() TO User*) and sender names with transtions (e.g. *PushButton() FROM User*). However, that is not always the case. If one is not careful, there is a high possiblity that some events will be attached to the *Untitled* participant although they should be attached to some other participants. The same problem will be faced if the state machines used are not complete. It should be emphasized that the tracing process is beneficial wrt security and usefulness only if there is at most one participant taking part to the example run with no corresponding state machine. Otherwise, participant *Untitled* represents a fusion of all such participants.

3 Scenarios and object model

3.1 Classes and operations

Although the participants of scenarios are objects rather than classes (since a scenario describes a run-time event sequence), in practice the objects in scenarios usually represent their classes. Hence the normal practice is to denote the participants with class names. If there is a need to have more than one instance of a certain class in the same scenario, the class name should be augmented with an appropriate characterization of the individual object. In any case we conclude that each participant of a scenario should have a class in the object model.

An event in a scenario is something an object reacts upon. For a passive object this can only be an operation call. For an active object (i.e. an object having its own logical thread of control) an incoming event can be some external stimulus (e.g. from the mouse), a message or signal sent from another object, or a remote operation call activated by another object. We assume that the distinction between passive and active objects can be made by the designer, and possibly marked in the scenarios using some appropriate notation. In the sequel we focus on passive objects. Recall that Section 2 deals with specifying the dynamic behavior of active objects.

Hence, an event p from object (class) A to object (class) B corresponds to a public operation p of B called by A. If an object calls its own operation (some scenario notations allow events from an object to itself), the operation is private unless it is also called publicly.

The signatures of operations should also be derivable from the scenarios. Events can have parameters, which naturally correspond to parameters of the operations. If the types (or classes) of the parameters are known, the parameter part of the signature can be determined.

In the case of function call the return value is denoted usually as the label of the return event. If there is no return event, the identifier denoting the return value is usually associated with the call event. In every case the fact that the call returns a value is visible in the scenario. If the type of the return value is known as well as the parameter types, the full signature of the operation can be determined.

Since the types of parameters or return values are usually not included in scenarios, an automatic tool deriving object model information would fail in determining the signatures of operations. A practical solution would be to assume that every identifier X in a scenario denoting a value (or object) has the type (class) TypeOfX. Such artificial type identifiers are assumed to be replaced by the actual type identifiers manually at a later stage.

Since the participants of scenarios are objects rather than classes, the classes appearing in scenarios are normally concrete. Hence all the abstract classes should be found in some way on the basis of the concrete ones appearing in the scenarios. If the intersection of the sets of operations and attributes of two participating classes turns out to be nonempty, an abstract class consisting of the intersection features could be constructed. If the intersection is exactly one of the classes, no new abstract class needs to be created, but a subclass relation can be established directly between the classes. This procedure is based on the names (and signatures) of operations and attributes only, and may lead to undesirable results.

3.2 Attributes

Consider again the parameters of events in a scenario. As noted above, these correspond to the actual parameters of operation calls. If the actual parameter is given as a single identifier, there are two possibilities: either the identifier denotes an attribute of the caller or another parameter passed to the caller by an enclosing call.

We assume that the return point of an operation call can be seen in the scenario; this can be shown with a special return arc or with a control bar ([7, 4]). Then the nested operation calls can be easily recognized in a scenario. This makes it possible to locate the enclosing call whose body contains the call with a parameter identifier. If the enclosing call contains the same identifier as a parameter, the parameter in the nested call can be ignored — it simply passes on the previous parameter. Otherwise the parameter denotes an attribute of the caller.

Besides as parameters, identifiers denoting values (or objects) may appear as return values of functions. Such an identifier may denote an attribute of the owner of the function (i.e. the receiver of the call event). The attribute may either be defined in the owner class or inherited from its superclass. This is the case if the purpose of the function is to read one of the attributes of the host object; such functions are typical in OO programming because of the principle of data abstraction. A call of such a reader function can be recognized with some certainty by examining the events occurring inside the call: if there are no internal events, the function is likely a reader function. A shorter expression for successive call and return events might be useful. E.g., the return value could be written after the operation name separated with a semicolon or written under the event arc. In addition to making scenarios shorter, it would help distinguishing attributes from other events.

3.3 Associations

The fact that one object sends a message to another object implies that there is an association between the objects. At the implementation level, the caller must know the identity of the callee, which usually means that the association is implemented by a link from the caller to the callee. On the other hand, from the functional point of view, an association is necessary only if it is used by some operation (assuming that objects are manipulated only through their operations). Hence an association that is not used is at least highly suspect. We conclude that if the caller-callee relationships are known (which can be recognized from scenarios), the association relationships between classes can be approximated with reasonable preciseness. It may be, however, that these associations differ from the analysis level associations.

The names and kinds (aggregate or general, multiplicity) of associations are more difficult to extract from scenarios. A system which creates associations automatically on the basis of scenarios might give the associations artificial initial names which are assumed to be changed later by the designer. A possible name could be e.g. “usesForP”, where P is the name of an operation exploiting the association. If there are different operation calls between the same pair of objects, the name of the association could be chosen on the basis of the most frequent operation. If there are different operation calls between the same pair of classes (but between different objects), they should give rise to separate associations.

To distinguish a general association from an aggregate one could look for propagated operations, i.e. chains of nested successive calls of the same operation in the scenarios. The existence of such a chain hints that the callees are parts of the callers (excluding the first call). On the other hand, a non-nested repetition of the same operation with the same caller and callee classes but with different callee objects suggests that the corresponding association between the classes should allow multiplicity at the callee end (black circle in OMT). However, deciding whether objects are in a general association relationship or one is an aggregate of the other, is very difficult by examing only their interactions. Therefore, conclusions based on nested operation calls shouldn’t be made automatically. Instead, such calls should be seen as a hint to the designer.

4 Scenarios and operation code

Let us assume that all events concern operation calls, and that both the call events and the return events are shown in the scenarios. Then the slice of the scenario starting from the call event and ending at the return event corresponds to the execution of the operation. Since this is essentially a trace through the operation, and since there may be several such traces for the same operation in the same scenario or in other scenarios, it seems sensible to conjecture that the implementation code of the operation could be synthesized in the same way as a state machine can be synthesized for an entire object using the algorithm of [8] (see section 2).

Consider the call of operation p of object x in a scenario. The portion of the line of x between the call of p and the return of p containing all leaving call arrows is called the *call trace* of the call of p. Note that the execution of a nested call may call another operation of x, but the call trace of this operation will not be included in the call trace of p. The call trace of p represents the actions executed directly in the body of p. Figure 5 illustrates the concept of a call trace. In fact, the algorithm [8] is a variant of another algorithm [2] which was originally developed for synthesizing programs rather than state

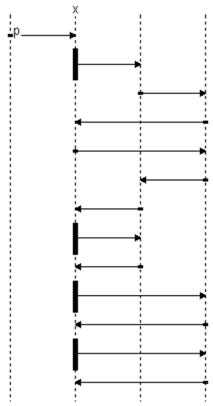

Figure 5: The call trace for p of x is shown with a thick line.

machines. Using a convenient interpretation of an event as a condition this algorithm could be applied to state machines. The problem is that if events in a scenario correspond to operation calls, they cannot act in the role of conditions any more. A return event arriving at the object line has no control aspects at all as far as the receiving object is concerned; it cannot represent a condition. This means that an essential part of the original algorithm, conditions, is completely missing.

Another problem is that operation calls are usually not sufficient as primitive actions of operation implementations. Primitive internal actions like computations or assigments are either ignored or sometimes briefly described as comments in scenarios.

On the basis of [2], it is clear that operations cannot be synthesized using call traces only. For sensible operation synthesis, primitive actions and conditions must be added to call traces and therefore to the scenario notation. This need not introduce substantial modifications: primitive actions are in fact included already e.g. in [9] as boxes in object bars (action box). A condition can be represented by a similar box; the condition expression is assumed to be given in terms of the attributes of the object and parameters of the enclosing operation.

Condition boxes are essential: they replace the incoming events in the conventional state machine synthesis algorithm (recall that a call trace has no incoming events). In a sense a condition box is a dual concept with respect to an action box: an action box can be viewed as an event without a receiver, a condition can be viewed as an event without a sender.

We illustrate the synthesis of operations with an example. Consider the scenario fragments on the left hand side in figure 6 describing possible event sequences during bank account withdrawal. These fragments are assumed to be parts of larger scenarios. Interpreting conditions as incoming events the result of the state machine synthesis algorithm is shown on the right hand side in figure 6. Note that the state machine in figure 6 could easily be transformed into pseudo code, if desired.

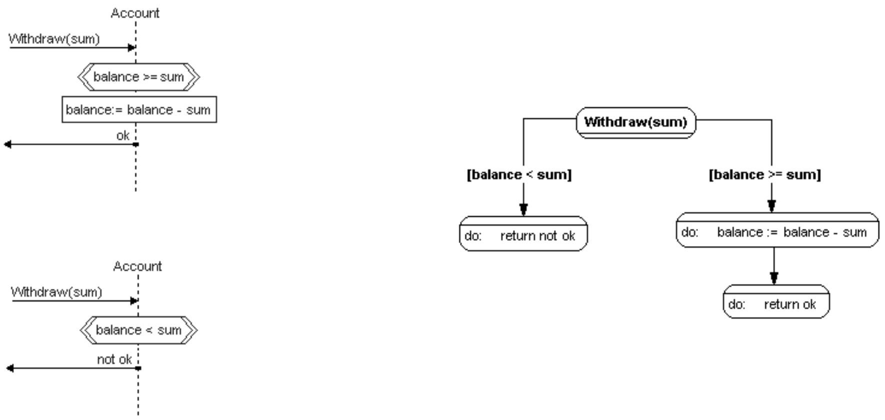

Figure 6: Two scenarios making use of condition (rounded rectangles) and action (rectangle) boxes on the left, and specification of the operation Withdraw on the right

5 Conclusions

We have studied the relationships of scenarios with other models used in object-oriented software development. Especially dynamic modeling can obviously benefit significantly from scenarios. If the dynamic behavior of an object can be modeled as a state machine, this state machine can be automatically derived from a set of scenarios in which the object is involved. It is possible to maintain sensible consistency between scenarios and state machines. We have also shown that scenarios and state machines can be constructed in concert, supporting each other.

The relationships between the static object model and scenarios is weaker, but nevertheless we could find several useful dependencies. From the point of view of tool development, these observations can be used either for automated consistency checking between the different models, or for generating at least partially the object model from the scenarios. In both cases the level of intelligence and automated support of the OO CASE tools can be improved.

From a methodological point of view these observations suggest that scenarios could be used as the basis of object-oriented design to larger extent than is usually done. Since the construction of scenarios is based on known objects, the first phase of the development method must be finding an initial set of objects. However, after that phase the design can proceed using scenarios as a central technique.

References

[1] Aalto J-M., Jaaksi A.: Object-Oriented Development of Interactive Systems with OMT++. In: *Proc. TOOLS* **14**, Prentice-Hall, 1994, pp. 205– 218.

[2] Biermann, A.W. and Krishnaswamy, R.: Constructing programs from example computations, *IEEE Trans. Software Engeneering*, SE-2, 1976, pp. 141– 153.

[3] Coplien J., Schmidt D.: *Pattern Languages of Program Design*, Addison-Wesley, 1995

[4] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Object-Oriented Software Architecture*, Addison-Wesley, 1995.

[5] Harel D.: Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, **8**, 1987, pp.231– 274.

[6] Harel D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Tauring, A., and Trakhtenbrot, M.: STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Transactions of Software Engineering*, Vol. **16**, no. **4**, 1990, pp.403–414.

[7] Jacobson, I., et al: *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[8] Koskimies, K. and Mäkinen, E.: Automatic Synthesis of State Machines from Trace Diagrams, *Software Practice & Experience*, **24**, **7**, 1994, pp. 643–658.

[9] Koskimies K., Männistö T., Systä T., Tuomi J.: SCED — An Environment for Dynamic Modeling in Object-Oriented Software Construction. In: *Proc. Nordic Workshop on Programming Environ ment Research '94*, Lund. Department of Computer Science, Lund Institute of Technology, Lund University, June 1994, pp. 217–230.

[10] Koskimies, K. and Mössenböck H.: Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In: *Proc. ICSE '96*, March 1996, Berlin. To appear.

[11] Männistö T., Systä T. and Tuomi J.: *Design of State Diagram Facilities in SCED*, University of Tampere, Report A-1994-11.

[12] Rumbaugh, J., et al: *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[13] Rumbaugh, J.: OMT: The dynamic model, *Journal of Object-Oriented Programming*, A SIGS Publication, vol. **7**, No. **9**, Feb 1995, pp. 6–12.

[14] Salmela M.: *A framework for graphical animation of object-oriented models of embedded real-time software*, VTT Publications 207, VTT, Espoo, 1994.