SCED: A TOOL FOR DYNAMIC MODELLING OF OBJECT SYSTEMS

Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1996-4

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1996-4, JULY 1996

SCED: A TOOL FOR DYNAMIC MODELLING OF OBJECT SYSTEMS

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 951-44-4003-X
ISSN 0783-6910

SCED: A Tool for Dynamic Modelling of Object Systems

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

[1]Department of Computer Science, University of Tampere
Box 607, FIN-33101 Tampere, Finland
{koskimie@cs.uta.fi}

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

Abstract

Dynamic modeling of object-oriented software makes use of scenario diagrams, i.e. descriptions of particular uses of a system in terms of message flow between the objects belonging to the system. Such diagrams help the designer to specify the general behavior of objects as state machines or as collections of methods. Several techniques are discussed for building automated tool support for the dynamic modeling aspects of object-oriented software development. The discussed techniques include synthesis of state machines and method descriptions on the basis of scenario diagrams, constructing scenario diagrams with the support of existing state machines, visualizing the run-time behavior of an object system, extracting state machines of objects from running systems, consistency checking between scenario diagrams and state machines, automated simplification of state machines using OMT notation, and automated layout for state machines.

1 Introduction

The basic problem of any software design is how to derive executable software components from the requirements specification, and how this process could be supported by the computer. The object-oriented approach provides a common paradigm throughout the software development process from analysis to implementation. This allows smooth shift from one phase to another and makes it possible to use wide-spectrum tools for software development.

Current object-oriented CASE tools support drawing various graphical notations for modeling the application from different perspectives, consistency checking between the models, storing the models in a repository, and generating documents and code from the models. Although these tools facilitate the construction of graphical models and the transformation of these models into code, the level of built-in automation is relatively low as far as the actual development process is concerned.

In object-oriented analysis and design, dynamic modeling aims at the description of the dynamic behavior of objects using some variant of a finite state machine. For example, in OMT [Rum91] the dynamic model is one of the three models employed in software development; the other models are the object model describing the static relations of objects and the functional model describing the data flow between the processes of the system.

In this paper we study raising the level of automated support for the dynamic modeling part of object-oriented software development. We use the OMT method as a guideline and notational basis, but in principle our approach is not tied to any particular design methodology. All the facilities discussed here have been implemented in a working tool called SCED [MST94a, MST94b]. This system is running under MS-Windows and implemented in C++.

We assume that the requirements concerning the dynamic behavior of a system are expressed as scenarios, describing how the system responds to a particular sequence of external events (e.g. user interactions). A scenario is close to the concept of a use case of Jacobson [Jac92]: a use case is a specific way of using a system to accomplish an identifiable task, consisting of possibly several scenarios. A *scenario diagram* (event trace diagram, message sequence chart, interaction diagram) is a graphical formulation of a scenario, specifying how objects communicate with each other and external actors during the scenario. Each object participating in a scenario is represented by a vertical line; an event is shown as a horizontal arc from the sender object to the receiver(s). Time flows from top to bottom.

SCED demonstrates how ideas discussed in this paper can be incorporated in a practical tool. SCED consists of two conventional CASE components, a scenario editor and a state machine editor, and of a more intelligent component integrating scenarios and state machines with various mechanisms. The main principles of the latter component are the central topic of this paper. The scenario notation employed in SCED has been extended from that of OMT for reasons discussed in the sequel; the state machine notation is taken from OMT (which is in turn strongly influenced by Harel's Statecharts [Har87]). The overall logical structure of SCED is depicted in Fig. 1.

We proceed as follows. Section 2 introduces our graphical notation for scenario diagrams. Next two sections discuss techniques for transforming a set of scenarios into a state machine, and, reversely, for using existing state machines to generate new scenarios. Section 5 discusses some implications useful for reverse engineering. In section 6 we briefly study the problem of keeping scenario diagrams and state machines consistent. In section 7 we discuss simplification algorithms for state machines making use of the special graphical notation of OMT. Finally, in section 8 we discuss automatic layout algorithms for state machines. Some concluding remarks are presented in section 9.

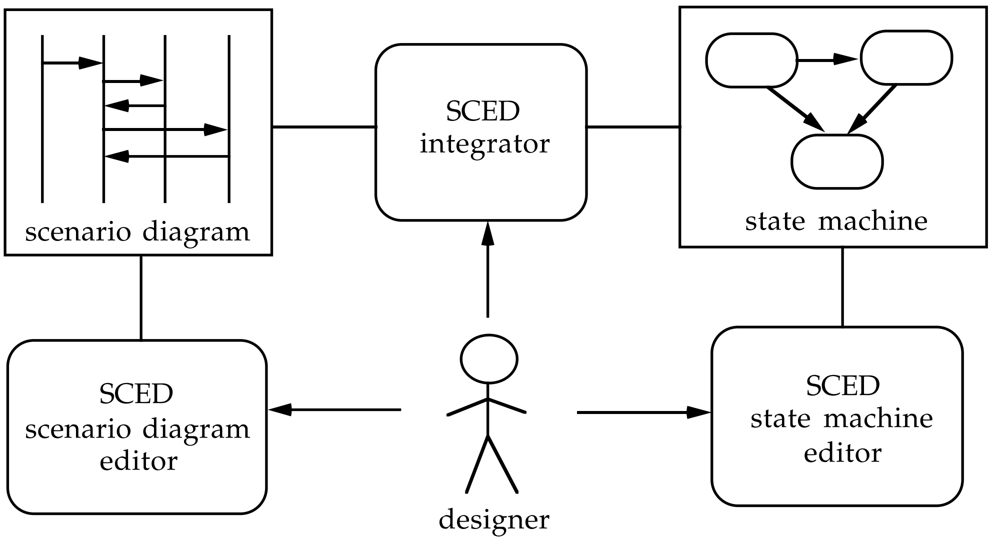

Fig. 1. General structure of SCED

2 Scenario diagram notation

For practical purposes we have extended the rather rudimentary scenario diagram notation of OMT in various ways (see figure 2). First, a comment can be drawn as a rounded box stretching over (and concerning) selected partipants.

It is often necessary to present other actions than events: an object may perform arbitrary computations without sending messages. For such actions we use an *action box* drawn at the vertical line of the object executing the action. Some techniques dicussed later require that the designer can express conditions that are known to hold at certain positions in a scenario for a particular object. Normally such a condition is given in terms of the attribute values of an object. This kind of a condition is drawn as an *assertion box* with a form following the CCITT scenario notation standard [CCITT92] (see the symbols in figure 2).

We introduce also a third type of box associated with a single participant, a *state box*. A state box gives a name to a particular situation in a scenario from the point of view of a certain participant; i.e. the name of the state of an object at that situation. Although this is needed primarily for technical reasons rather than as a design aid, it is sometimes convenient for the designer to express her assumption that an object should be in an identifiable state in a particular time position of a scenario. Technically, state boxes are necessary for expanding conditional and repetition constructs discussed below. Note that a state box is different from an assertion box: a condition may hold in many states.

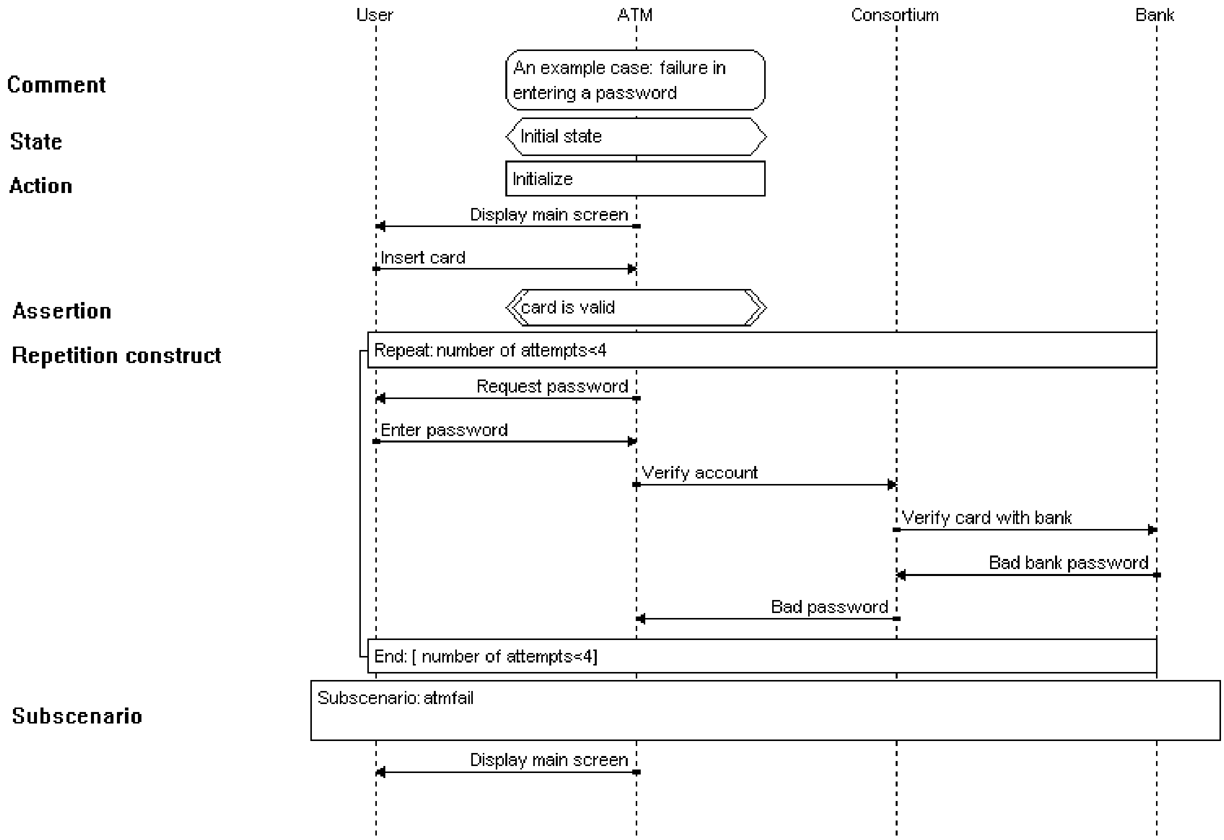

Fig. 2. Elements of SCED scenario notation

To support the presentation of a use case as a single scenario diagram, the scenario notation must be extended with algorithmic constructs like conditionality and repetition. We call a scenario diagram having such constructs an *algorithmic scenario diagram*. Note, however, that a certain object may be involved in several use cases; hence algorithmic scenarios are full specifications for use cases but (usually) not for objects. In general, algorithmic scenario diagrams can be used as full specifications of *multi-object functions*, i.e. functions defined in terms of several interacting objects. A restricted form (conditional construct) of algorithmic scenario notation has been applied e.g. in [Por95]; Jacobson [Jac92] associates textual pseudocode to ordinary scenario diagrams to achieve the same effect.

SCED supports algorithmic scenario diagrams by providing structured graphical notation for conditionality and repetition (see figure 3) with arbitrary nesting. Algorithmic scenarios can be interpreted as sets of ordinary scenarios. The interpretation is shown in figure 3. Note that in the case of repetition the number of iterations (and therefore the number of scenarios) is potentially infinite, but the repetition construct can nevertheless be represented by two scenarios making use of the state box: after evaluating the loop expression and executing the body, the object will be in the same state as before entering the loop; hence the object will be able to re-execute the loop infinitely.

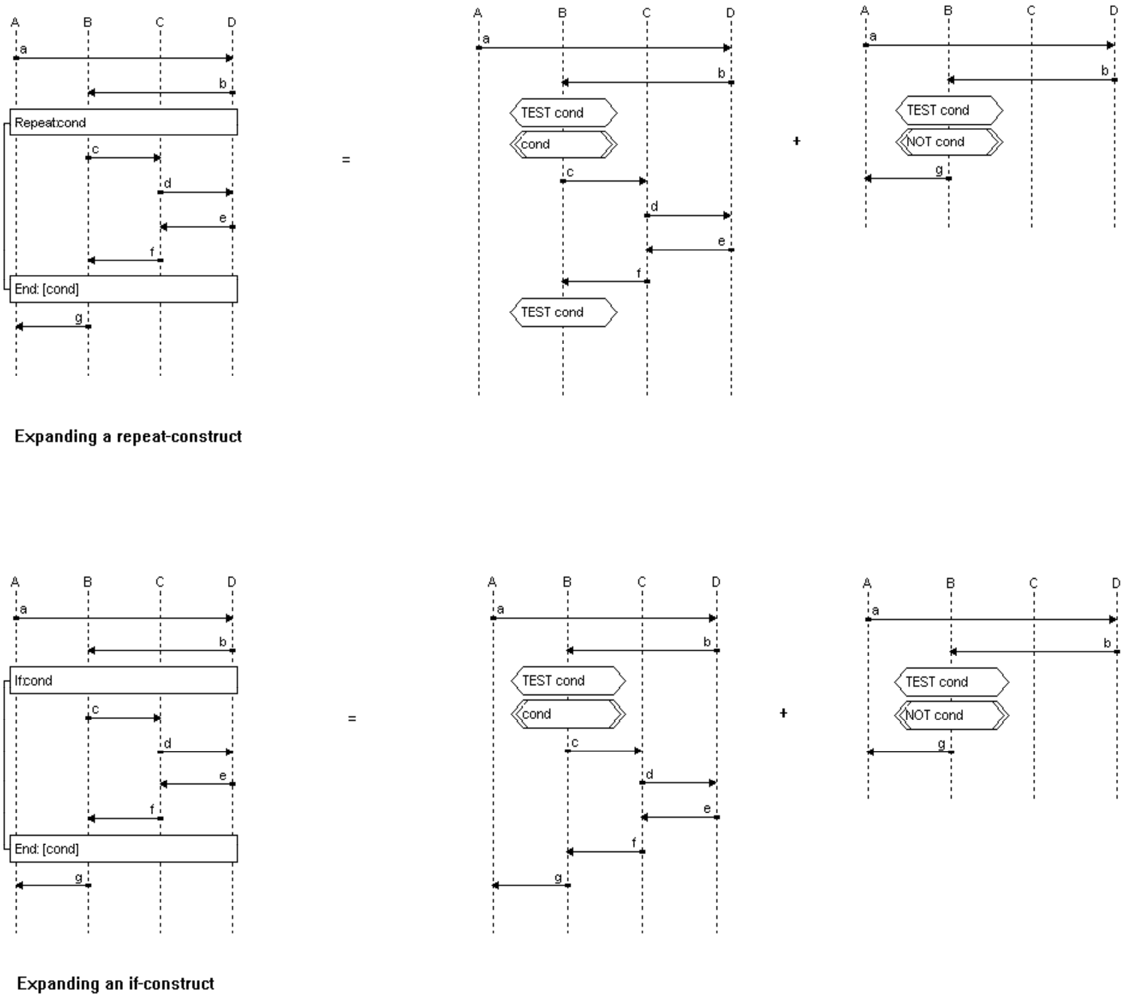

Fig. 3. Dissolving if- and repeat-constructs into simple scenarios.

Analogously to subroutines, a scenario may consist of parts that have their own aims and characterizations. For instance, a scenario for using an ATM might include event sequences like "checking a valid card" or "giving a correct password" for which one can give separate scenarios. To make the scenario diagrams easier to both read and write we have adapted the notion of a *subscenario* in SCED: a rectangle stretching over all the participants denotes a named subscenario diagram given elsewhere. To obtain a complete scenario diagram, all the subscenario boxes are simply considered to be replaced by corresponding scenario diagrams. The participants of the subscenario can be different of those of the host scenario. Hence a subscenario is especially useful if the subscenario requires objects that are not needed for the rest of the scenario: then the host scenario becomes smaller both in vertical (event sequence) and in horizontal (participants) direction.

3 Synthesizing state machines: design-by-example

Since both scenarios and state machines describe dynamic aspects of a system, they necessarily share common information. However, it should be emphasized that a scenario is not an instance of a state machine: it is an instance (trace) of a set of collaborating state machines. A state machine gives the complete behavior of a single object, while a scenario gives a single behavior (trace) of a complete set of objects. Consequently, a scenario contains information not included in a state machine, and vice versa. Hence, scenarios and state machines are complementary notions, and they should be constructed in concert, rather than one after the other. In this and the following section we will show how the fact that they share same information can be exploited in developing design tools.

In [BiK76], Biermann and Krishnaswamy presented an algorithm for synthesizing programs from their example traces. The algorithm was used in an actual system that was able to synthesize programs on the basis of examples of sequences of primitive actions (like assignments) and assertions given by the programmer using a partly graphical interface. For example, the programmer could give examples of sequences of actions and assertions for a sorting algorithm, and the system generated the complete code for the algorithm. The assertions were employed by the algorithm as conditions for branching the execution.

We have applied the Biermann-Krishnaswamy (BK) algorithm for synthesizing state machines from scenarios. Since both the algorithm and its adaptation have been discussed in detail already in an earlier paper [KM94], we will here only summarize the main ideas. A trace is extracted from a scenario by selecting an object (i.e. the object for which the state machine will be synthesized) and traversing the vertical line of that object from top to bottom. Each received event is regarded as an assertion ("event e has occurred") and each sent event is regarded as a primitive action ("cause event e") in terms of the BK-algorithm. First, a lower bound N for the number of states of the resulting state machine is required. For this we use the number of different actions: since each state can have at most one action, this is clearly a lower bound. The algorithm maps actions to states and assertions to transitions, starting from the beginning of the trace. If a nondeterministic state results, the algorithm backtracks to a previous position where there was some freedom in associating an action with a state, and takes another untried choice. If at some point N+1 states are needed, the algorithm backtracks again. If backtracking is no more possible, a state machine with N states cannot be achieved. Then the whole process is repeated for the allowed number of states N+1 etc. The algorithm is completed when all actions have been associated with states. The BK-algorithm works incrementally, too: a scenario can be fused into an existing state machine using the algorithm.

It follows directly from a theorem given in [BK76] that the algorithm produces a minimal (with respect to the number of states) state machine capable of acting in the role of the selected object in the scenarios. Due to potential backtracking, the algorithm has exponential time complexity in the worst case, but this is not a problem in practice: real-life state machines seldom require heavy backtracking. In vast majority of the practical examples we have encountered, the synthesis time is less than a second. In fact, the automatic layout algorithm discussed in section 7 is usually more time-consuming although it has linear time complexity.

The original BK-algorithm makes use of assertions that are known to hold between primitive actions of the trace. When this algorithm is applied to scenario diagrams, some aspect of the diagrams must be interpreted as those assertions. Above we viewed the received events (with respect to a selected object) as assertions, which is appropriate for active objects using incoming events as the basis of the control. However, scenarios are useful (and used) also for passive objects whose dynamic behavior is characterized by a set of operations rather than as a state machine. In the sequel we show how operation descriptions can be automatically synthesized from scenarios.

First, note that the algorithm of an operation can be presented as a state machine, when transitions are associated with guards (conditions) rather than with events - this is in fact close to a conventional flow chart. In each state, the guards associated with leaving transitions must be nonoverlapping and cover all possible cases (since the algorithm must be able to continue deterministically in all cases). For passive objects, event arcs in a scenario diagram correspond to operation calls and returns.

Consider a particular object in a scenario. An operation call for this object is shown with an arriving call arc. All the leaving arcs between the call arc and the corresponding leaving return arc are internal calls of operations of other objects, and all the arriving arcs are returns of these calls (for simplicity we ignore the possibility that the same object is called during the execution of its own operation). Hence the trace of the operation call consists of the internal calls shown by the leaving arcs. The arriving (return) arcs within the call are insignificant (indeed, return arcs are omitted in some scenario diagram variations, e.g. [Gam95]).

Since received events cannot act as assertions when synthesizing operations for passive objects, the control information must be provided by assertion boxes. Other primitive statements than operation calls are presented as action boxes. Both assertion boxes and action boxes can be incorporated in the synthesis algorithm described above in a simple way: an assertion is regarded as an event without a sender, an action is regarded as an event without a receiver. This results in the natural representation of an assertion as a label for a transition, and of an action as a state activity.

The other extensions to the scenario diagram notation cause no major problems for synthesizing state machines, either. A state box is taken into account by using the state identifier as a name for the corresponding state in the state machine. Further, the algorithm refuses to join two states that have different names. A subscenario is simply expanded before synthesis. Condition and repetition constructs are likewise dissolved according to the rules in figure 3.

In SCED, state machines can be synthesized both for active objects and for the operations of passive objects. In the former case the designer selects a participant of an open scenario diagram; in the latter case she selects a call arc of the operation in the scenario diagram. When the synthesis command is given, the system generates a state machine either for the object or for the operation, showing it in a separate window. The designer can determine whether the traces are extracted from the active scenario, from all open scenarios, from all scenarios of the current project, or from selected scenarios (containing the object in question).

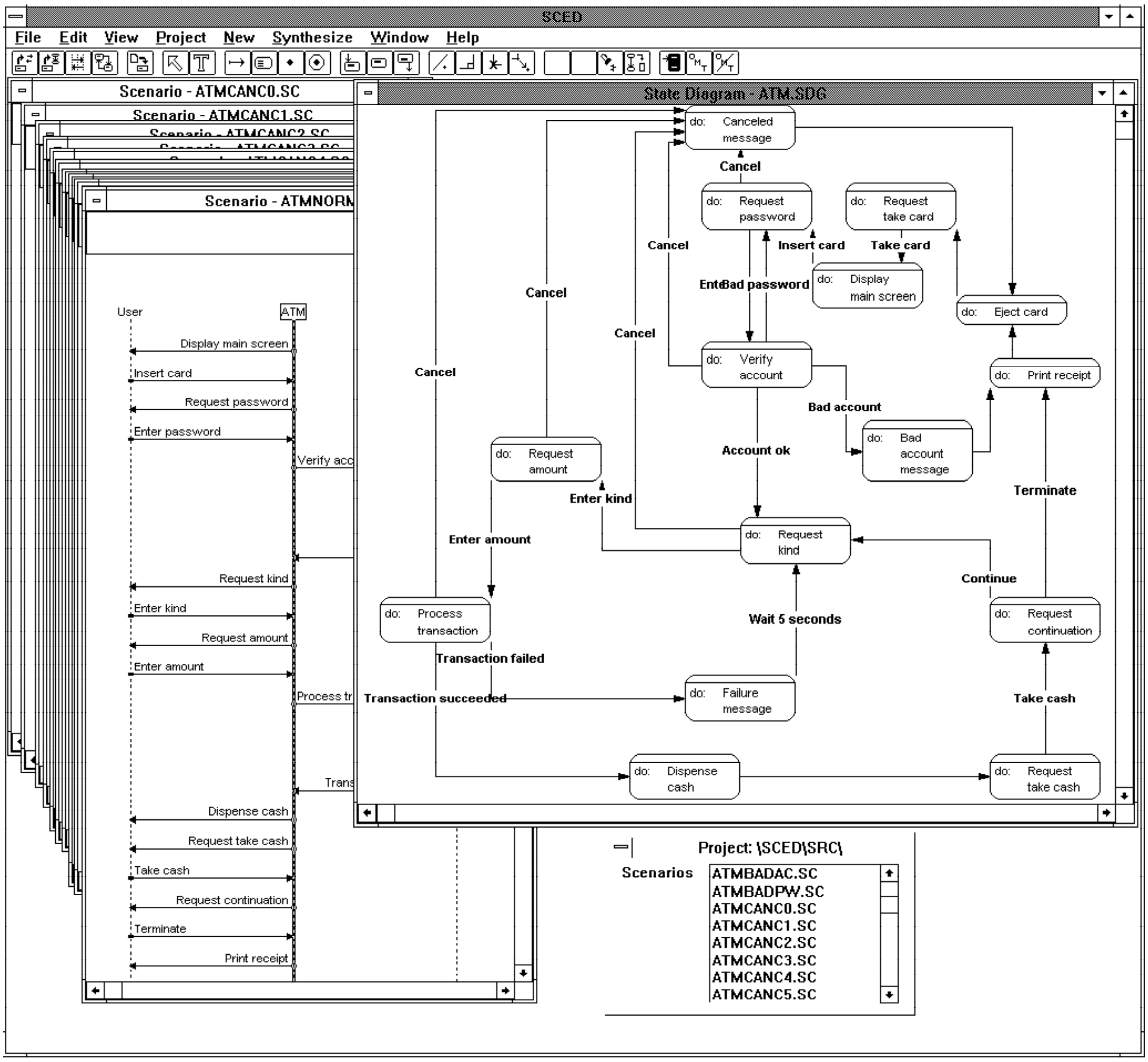

Fig. 4. Scenario diagrams and a synthesized state machine in SCED

The synthesis algorithm is the cornerstone of the design-by-example approach supported by SCED. The designer can describe the expected behavior of the application in some typical example cases using scenario diagrams, and synthesize automatically state machines for interesting objects. If these state machines are not satisfying, the designer can draw a few other scenarios and fuse them (automatically) into the existing state machines, and finally tune the state machines by hand using the state machine editor. The same technique can be used for operations of passive objects: with a single command the designer can see the synthesis of various executions of a particular operation appearing in scenarios in the form of a state machine. This synthesis can be used as a basis for implementing the operation, for checking purposes, or simply as a summary of the operation's requirements (as expressed by the scenarios) so far.

Figure 4 illustrates the synthesis of a state machine. A set of scenarios have been given describing the use of an automatic teller machine (ATM), and a state machine for the control unit has been synthesized. The shown state machine is exactly in a form generated by SCED (including the layout).

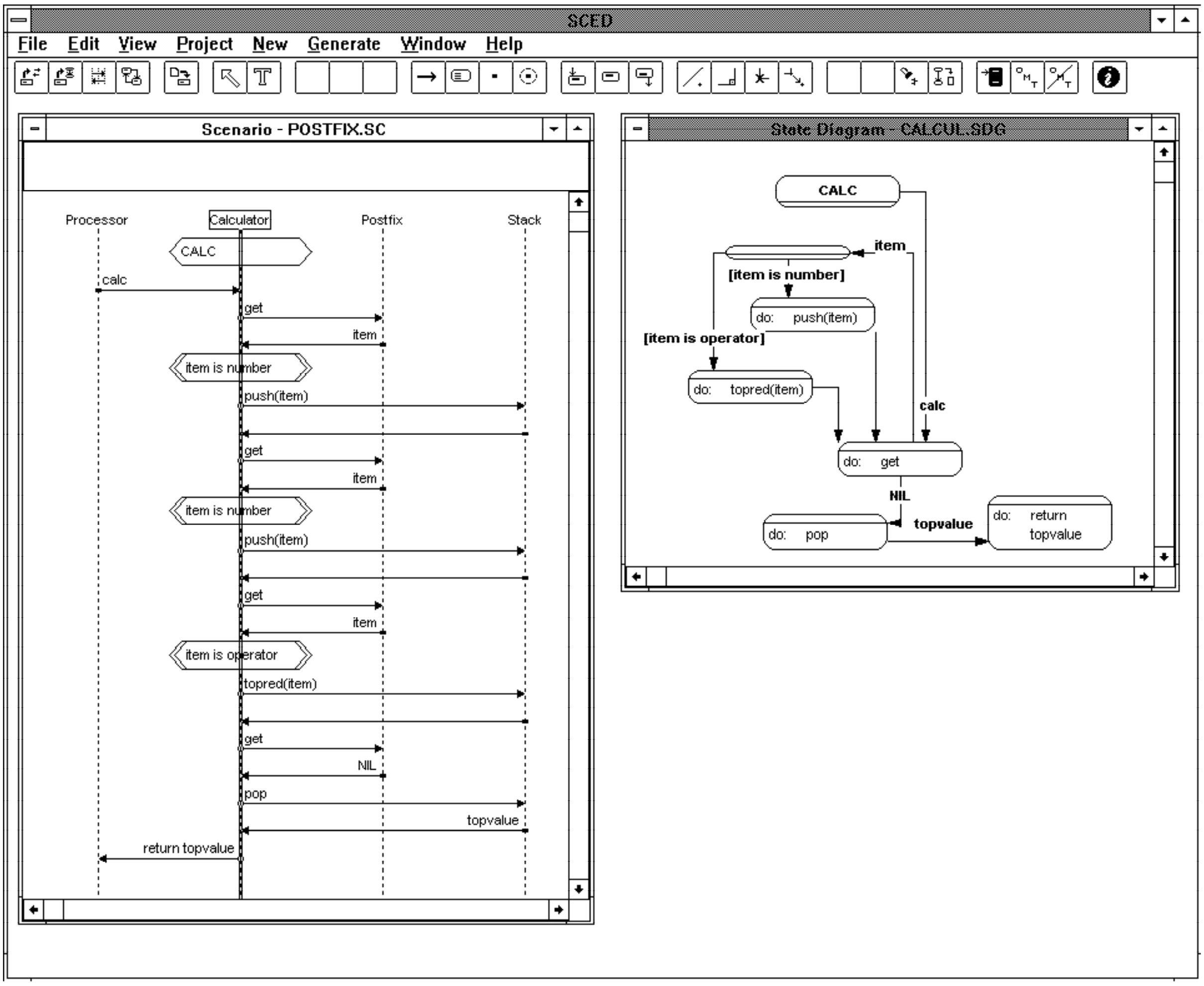

Fig. 5. Synthesizing operation descriptions from scenarios.

The generation of a description of an operation from its traces is demonstrated in a simple case in figure 5: the operation computes the value of a postfix expression using a stack in the conventional way, assuming that the operands and operators are provided by object Postfix through operation get(). The shown state machine on the right is produced by SCED on the basis of the example call appearing in the scenario on the left. The layout of the state machine is also automatically produced. The scenario is in this case the smallest one that can produce the correct state machine; further readings of operands and operators can be added to the scenario without affecting the resulting state machine. Note that a special initial state has been generated whose name is the same as the name of the operation.

Another type of synthesizing information obtained from scenarios is an *event flow diagram* [Rum90] in which nodes and arcs represent classes and events, respectively. The fact that an instance of class A sends an event e to an instance of class B is represented by an arc from A to B. An event flow diagram gives a useful global view of the possible interactions between classes, without referring to particular executions of the system. Hence an event flow diagram is a synthesis of all participants in a set of scenarios, while a state machine is a synthesis of a particular participant only. SCED produces an event flow diagram automatically by request.

4 Generating scenario diagrams: design-by-animation

In this section we show how existing state machines can be used to support the construction of scenarios, and hence - using the methods of the previous section - the construction of new (or more complete) state machines. We call this approach *design-by-animation*, because it is based on animating a partial specification of a system consisting of a set of state machines, employing scenarios as the runtime system representation. Using scenario diagrams for visualizing the behavior of running object systems is an idea proposed recently by several authors ([LN95], [EiW96], [KM96]).

Assume that there is a set of state machines describing a complete system. As long as someone provides the required external stimuli to the system when needed, the state machines can simulate the behavior of the system, sending events to each other and changing states according to received events. The result of the execution can be shown as a scenario diagram.

However, suppose that one of the state machines of the objects in the system is either incomplete or missing, and that the objective is to specify this state machine. We call the corresponding object *Unknown*. Consequently, at some point the execution of the system is stuck because some state machine is waiting for the response of the state machine of *Unknown*. The designer can now act in the role of *Unknown*: she can select the object that should react upon the response of *Unknown*, and indicate which state transition will be applied for the former object. After that the execution proceeds normally until *Unknown* is again needed. As a result, a complete scenario diagram with *Unknown* can be generated, and the designer can ask the system to either synthesize the first approximation for the state machine of *Unknown*, or augment the incomplete state machine of *Unknown* with the new scenario. The design process can continue making use of the resulting state machine, too: only if some new behavior (i.e. a new path in the state machine) is required for *Unknown*, the designer has to intervene.

This method can be generalized to an arbitrary number of unknown objects if the designer can specify which object is the receiver of an unidentified message sent by some state machine. Then the designer can start with an empty set of state machines, construct the first scenario diagram by hand, synthesize the initial state machines on the basis of this scenario, construct a new scenario with the support of these state machines, fuse the scenario into the state machines etc. until satisfactory state machines are obtained.

The significant advantage of this approach is that the designer need not recall the behavior of the existing state machines: those parts of the scenario concerning known behavior are produced automatically. Furthermore, it is guaranteed that these parts of the scenario are consistent with the existing state machines. The approach works well if the design is based on existing components with known behavior (say, GUI library components), but also if the design is made from scratch. The more complete the designed system gets, the less the designer needs to intervene.

Note that the external interface (e.g. end-user) is normally represented by an "unknown" participant in a scenario. The design-by-animation approach is particularly attractive when the interactions concerning such a participant can be given through a simulated graphical interface (instead of drawing events in a scenario): the software designer can explore all the possibilities of the actual interface, observe the response of the system in the form of a scenario, and extend the behavior when necessary.

In SCED, design-by-animation is supported by so-called *tracing mode*. In this mode scenario diagrams are constructed with the support of existing state machines. First, the participating state machines are opened by the designer. Activating the tracing mode creates a scenario with participants bound to these state machines. If desired, the designer can add new participants (with unknown behavior). The designer can enter events normally, but whenever the state machine bound to the target object is able to respond to the event in its current state, a state transition takes place and the action part of the new state is executed, causing possibly a new event to be sent to another object. This may in turn cause a state transition in the state machine bound to the new target object etc. As a result, a cascade of events is created, and these events are immediately shown in the scenario diagram.

The generation of a scenario gets stuck in two situations: either a state machine has sent an event but there is no unique receiver, or an event appears with a target object whose (incomplete or missing) state machine is unable to recognize the event. In the former case the designer can select the receiver simply by clicking on the receiver object in the scenario. Those objects that are able to recognize the event (at least two) are shown with a hollow circle; those objects that do not recognize the event but whose behavior can be extended (i.e. they are not defined to be frozen objects) are shown with a black spot (see figure 6). As a result of selecting one of these objects, the event is drawn, and the receiver may respond to the event, causing new events to appear in the scenario. If the receiver is unable to respond, the latter case applies. In the latter case the designer can see the events each object is able to receive in its current state by clicking on the object bar in the scenario. Selecting one of these events from the popup menu creates an event arc from the currently active object to the selected object, labeled with the event. Consequently, the receiver object can respond to the event, change its state, and execute the possible event sending actions in the new state, thus continuing the scenario.

The execution may also get stuck when waiting for external stimuli, e.g. the response of the end-user of the animated system. In that case the designer can either give the end-user's reaction as a normal event (manually or selected from the popup menus giving the currently recognizable events), or use a simulated interface. To facilitate the latter, SCED provides an interface for other applications to insert events using the Windows clipboard as a buffer: SCED polls this buffer regularly and automatically appends found events to the end of the scenario. If the end-user is the only participant whose behavior is not known (as a state machine), applying the tracing mode of SCED essentially means animating and testing the existing dynamic model of the system.

Note that any intervention of the designer in the animated execution of the system implies new behavior not exhibited by the current state machines. For some participants, a state machine is not even interesting or sensible (e.g. an end-user of the system). For others, the new behavior should be added to the existing state machines after an appropriate scenario diagram has been produced. This can be done in SCED simply by selecting the desired object in the diagram and giving a synthesis command. As a result, the new scenario is fused into the state machine of the object using the BK-algorithm (see section 3).

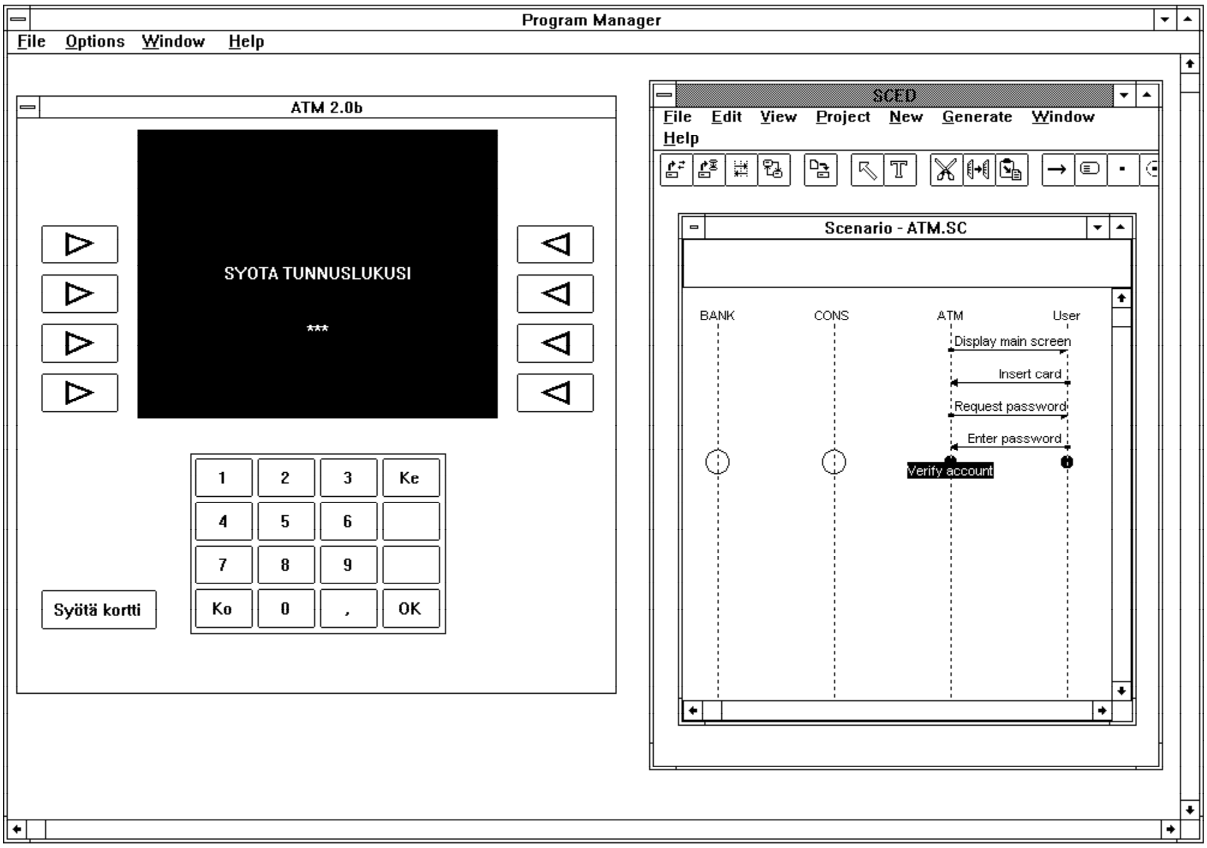

Fig. 6. Design-by-animation: specifying the behavior of ATM with the support of a simulated ATM interface and existing state machines.

In figure 6, SCED is used in concert with a simulated graphical user interface of an ATM. The latter simply emits user input events (in ascii form) to the clipboard, to be picked up by SCED. The designer has already constructed state machines for Consortium and Bank, and frozen them (freezing is shown with coloring that is not visible in the figure). For ATM only a preliminary state machine exists, and the designer strives for the specification of its complete behavior. The figure corresponds to a situation in which (1) the designer has inserted a card through the simulated interface, (2) the (so far incomplete) ATM state machine has responded by requesting a password, (3) the designer has given a password (through the graphical interface), and (4) the ATM responds by sending "verify account" message. However, since by accident both Consortium (CONS) and Bank are able to recognize this message, the receiver is not unique; hence the designer must select the desired receiver among the potential receivers indicated by blinking circles and spots. In this case only the first and the third event in figure 6 are generated automatically, but in principle generated event sequences can be arbitrarily long.

The session in figure 6 could continue as follows: (1) The designer selects Consortium (as a result, event "verify account" is drawn from ATM to Consortium, and Consortium responds by sending the same event). (2) The designer selects this time Bank as the receiver of the event. The event is drawn, and Bank responds by sending event "invalid password" to Consortium. Consortium responds by sending event "account not ok"). Since event "account not ok" is so far not recognized by any object, the system again asks for the receiver. (3) The designer selects ATM, and an event is drawn from Consortium to ATM. (4) Since ATM is unable to respond, the designer inserts by hand event "invalid access" from ATM to the User. Note that steps 3 and 4 represent an extension to the behavior of ATM, and when the resulting scenario is synthesized into ATM's state machine, the extension becomes part of ATM's behavior.

5 Reverse engineering with SCED

Using the interface for externally generated events, it is also possible to employ SCED as an animator of any object-oriented system. The system to be animated must be instrumented with calls for generating textual event descriptions at the site of each event sending. Since the events are transmitted via ascii text (in the Windows clipboard), the system to be animated can be written in any language. One can run SCED and the (instrumented) animated system simultaneously and observe how new events appear as a result of using the animated system. This kind of program visualization comes close to Scene [KM96]. However, in contrast to Scene, we have not tried to solve the scaling problem with event compression techniques but instead we rely only on vertical and horizontal scrolling of a scenario. In practice this implies that the instrumentation should select a relatively small set of objects which give rise to events; otherwise the resulting scenario becomes too large for sensible examination. The call compression technique used in Scene is not applicable in SCED where events are not necessarily method calls.

When combined with state machine synthesis, the animation capability of SCED becomes a powerful reverse engineering tool. Recall that SCED is able to synthesize the general behavior as a state machine from any scenarios, independently of how the scenarios are produced. If the synthesis algorithm is applied to scenarios produced by running existing (instrumented) applications, one can - after running the application for a while - see the general behavior of some of the objects belonging to the application in the form of a state machine. With additional test runs, the state machines describing the general behavior become more and more complete. Hence, as long as an application can be instrumented for SCED, it is possible to extract the state machines for selected objects from executions of the application - a property which can be highly useful for understanding complex legacy systems. It is remarkable that this property is essentially a consequence of other features in SCED, rather than a separate facility.

The reverse engineering aspect of SCED can be demonstrated with the ATM example as follows. Assume that a simulator for the ATM system has been implemented from scratch. In addition to the ATM interface simulator (see above), the system simulates the behavior of the ATM control unit and the central bank computer. The system is instrumented such that each message sending between the user interface, the control unit and the bank computer is associated with a call of an operation that inserts the corresponding event descriptor to the SCED event buffer (Windows clipboard). This is sufficient for using SCED as program visualizer. The ATM interface can be used in arbitrary ways, and the resulting object interactions can be observed in the scenario displayed by SCED. If desired, new scenario windows can be opened, splitting the generated event sequence into meaningful parts. After a while the user can ask SCED to synthesize a state machine for the control unit. Depending on which parts of the ATM system have been used, a more or less complete state machine will be generated.

Various tools have been developed for supporting program comprehension; traditionally these tools rely on static or dynamic analysis of the source code.

Based on the result of such analysis, graphical representations of some relations of the source entities can be produced. Typical systems of this kind make use of program slicing, data dependencies, call graphs etc. (see e.g. [Oma90]). The essential difference with such tools and SCED-based reverse engineering is that SCED produces an *abstract* description of the behavior of an object (in terms of a state machine): this description is completely independent of the way this behavior is implemented in the source code. The drawback is that it is hard to tell when this description is complete, that is, when all the possible ways of using the system have been exercised. To accomplish this, the target system should be instrumented with dynamic statement execution counters.

6 Consistency between scenario diagrams and state machines

The fact that both scenarios and state machines can be edited independently makes it possible to create inconsistencies. Suppose that a state machine has been synthesized according to a set of scenarios, and that one of the scenarios is changed. Clearly the state machine may become invalid in the sense that it is no more able to execute the scenario. The same holds reversely: if a state machine is changed, there may be one or more scenarios that are no longer executable by the state machine. The system should maintain consistency in the sense that the state machine of an object can always execute all the scenarios in which the object participates.

Note that it would be possible to require even stronger consistency, namely that a state machine is always exactly the result of applying the BK-algorithm to the set of scenarios in which the object participates. However, this would have some unnatural consequences: if a state machine is edited e.g. by adding a transition, a small artificial scenario causing this transition should be generated. Eventually there would be a large set of such fragmental scenarios without any sensible context, which would confuse the designer rather than help her.

Assume that a scenario is edited after it has been synthesized into a state machine. We can update the state machine incrementally using the following technique. 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 in this way, the new version of the scenario can be added to the state machine using the (incremental) BK-algorithm.

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), a conflict arises: a scenario cannot be run through the state machine any more. In contrast, transitions whose scenario counter is zero (i.e. they have been added by direct state machine editing) can be removed without problems, as well as states which are associated only by such transitions. We call the other transitions and states *scenario-sensitive*.

Editing scenario-sensitive parts of a state machine gives rise to problems that are hard to solve in general. Note that the reverse problem was relatively easy to solve because there was an incremental algorithm for updating the general description (set of state machines) with the information contained by a new instance (scenario). However, 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 as plain implications of state machines.

SCED supports the consistency between scenarios and state machines according to the principles presented above. The designer can ask the system to remove a particular scenario from a state machine; after this the scenario can be edited and resynthesized into the state machine. The updating of the state machine could be easily made fully automatic, but we have preferred an environment in which the designer has a selection of useful commands rather than a fixed working model imposed by various implicit mechanisms. At any time the designer can check the validity of current scenarios by running them through the state machines.

7 Synthesizing OMT state diagram notation

The result of the BK-algorithm is a state machine consisting of states associated with at most one action and transitions usually associated with a label (unlabelled transitions are called *automatic transitions* in OMT: they fire automatically after the action of the state is completed). Such a state machine is called a *plain state machine*. However, OMT provides auxiliary means to make a state machine more compact and readable. These include entry actions, exit actions, internal actions, and transition actions[1]. Entry actions are executed when arriving at the state, exit actions are executed when leaving the state, internal actions are executed as a response to an event without changing the state, and a transition action is executed whenever the transition it is associated with fires. These extensions do not increase the expressive power of the formalism, but they serve to make the state machine simpler and more readable: in many cases one can reduce the number of states and transitions using these extensions. We will call state machines making use of these features *extended state machines*.

When state machines are produced automatically in OMT context, it would be highly desirable to generate directly extended state machines; otherwise the designer may have to edit the resulting state machine by hand just to improve the presentation. In the sequel we will discuss automated techniques to transform plain state machines into extended ones with fewer states and transitions. All the discussed techniques are implemented in SCED.

Usually a plain state diagram can be transformed to several mutually different extended states machines depending on the chosen extensions and the order they are introduced. The ones with minimal number of states and transitions are called *optimized state machines.* We consider here transformations aiming at optimized state machines. In practice it is often sensible to apply these transformations only locally, without the goal of global optimization; this is possible in SCED but not discussed here.

We require that the transformation of a plain state machine into an extended one is behavior-preserving, i.e. the transformed state machine responds in the same way to a particular sequence of events as the original one. Further, we do not allow the association of an action of a state to several states or transitions if the

---

[1]In addition, OMT allows nested states in the style of [Har87]. Since nested states are currently not fully supported by SCED, we ignore them in this discussion.

state is removed: an action may be moved but not copied. This rule guarantees that the state machine is not optimized at the cost of increased actions. Hence we have the following basic rules for the transformation:

Rule 1: The external behavior of the state machine is preserved.

Rule 2: No action is duplicated.

We say a state machine is *minimally deterministic* when there are no two states with the same action that could be joined without making the state machine non-deterministic. The characteristic property of the BK-algorithm is that the resulting state machine is minimally deterministic. In the following we assume that a state machine is minimally deterministic. We also assume that every state in a state machine is reachable from the initial state.

To prevent automatic transitions from causing nondeterminism we require that a state cannot have both automatic and labeled leaving transitions. Further, we exclude the possibility that the state machine contains a loop with automatic transitions only. Both of these situations are prevented in our adaptation of the BK-algorithm.

The only way to reduce the number of states using special OMT actions in a plain state machine is to remove an automatic transition. Assume that the source state of an automatic transition is state 1 with action a1 and the target state is state 2 with action a2[2]. Further, assume that there is a transition e entering state 1. Under certain circumstances it is then possible to simplify the state machine by removing state 1 and (1) making a1 an entry action of state 2, (2) making a1 an exit action of the source state of e, (3) making a1 an internal action of state 2, (4) associating a1 with a new transition from the source state of e to state 2, or (5) merging state 1 and state 2 with action a1; a2.

The order in which the transformations (1) - (5) are applied in a plain state machine affects the result. Since the different transformations compete for the same automatic transitions, the various action types should be applied in the order of preference of these actions. Since we wish to emphasize the role of states rather than transitions, we attach as much information to states as possible.

[2]state 1 and state 2 are regarded as identifiers of states, rather than state names. Actions of states with real state names can only be moved to similarly named states.

Hence we allow transformations introducing transition actions (4) only when other transformations are not possible. On the other hand, we prefer transformations of type (3) to transformations (1) and (2) because the former are more specific: an internal action deals with a single event, while entry and exit actions are not directly associated with particular events. It can be shown that the order (3) -> (5) -> (2) -> (1) -> (4) - which satisfies the preferences above - results in an optimized state machine [Sys96].

For each type of transformation one can determine the conditions that must hold before the transformation can be carried out. As an example, consider internal actions. Assume again that there is state 1 with action a1, state 2 with action a2, an automatic transition from state 1 to state 2, and a transision e entering state 1. If state 1 had any entering transitions leaving a state other than state 2, action a1 couldn't be placed as an internal action to state 2 without violating either rule 2 or rule 1. Hence, we require that the source state of e must be state 2. If there were several transitions from state 2 to state 1, internal actions could be formed corresponding to each of these transitions but only by violating rule 2. Hence, we require that e is the only transition entering state 1. The fact that state 1 cannot have any other leaving transitions besides the automatic transition follows from our assumptions above. The synthesis algorithm also ensures that transition e must be labeled, and hence that also all other leaving transitions of state 2 have to be labeled (recall our restrictions on automatic transitions above). So, it can be concluded that other leaving or entering transitions of state 2 have no influence in forming an internal action 'e/a1' for state 2, since they cannot be attached to state 1 under our assumptions.

We can generalize this discussion and infer the smallest patterns (with respect to the number of states and transitions) which can be optimized by adopting OMT-type actions; this means determining the choices for the source state of e. As concluded above in the case of internal actions, the source state of e can be state 2. However, it cannot be state 1 according to the restrictions on automatic transitions. The third possibility is that it is some other state, say state 3. Noticing that state 1 and state 2 must indeed be different states (otherwise there would be a loop of automatic transitions), we conclude that there are two possible patterns in plain state machines that are potential candidates for simplification using OMT actions; these patterns are shown in figure 7. However, each type of action may introduce additional restrictions.

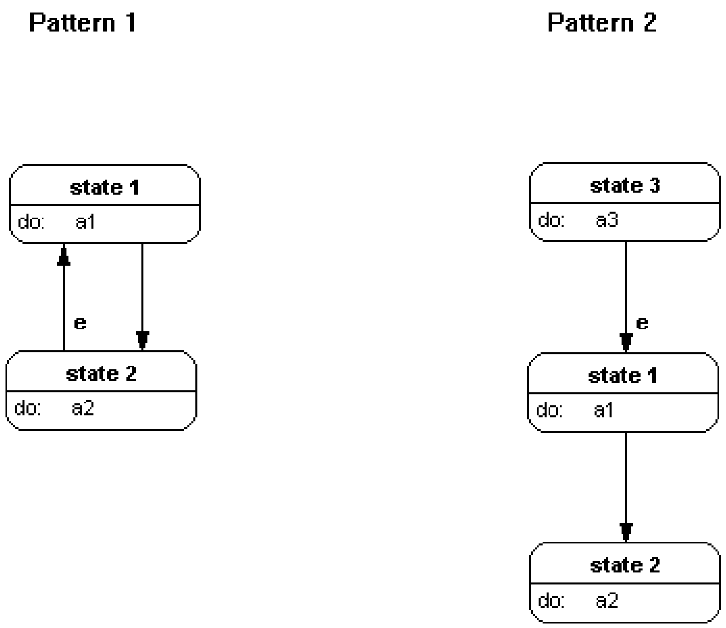

Fig. 7. Plain state machine patterns for special OMT-actions

The transformations discussed above are implemented in SCED. The designer can ask the system to optimize a plain state machine, and the system will produce an equivalent state machine in OMT extended notation, with (hopefully) less states and transitions. For example, when this command is applied to the state machine of figure 4, three states (and transitions) are removed. Another example of the transformation is shown in figure 8. In the upper window, a minimal plain state machine describing the behavior of a vending machine is given. In the lower window, the result of the optimization algorithm is shown; this form is roughly the same as the one given in the OMT book [Rum91, p. 96] (actually the algorithm has removed one more state (and transition) appearing in [Rum91]).

8 Automated layout for state machines

Producing the logical contents of a state machine is not enough: the result must be displayed in an aestethically satisfactory form. Even for a state machine of modest size, arranging the states and transitions manually into reasonable form is a laborious task - for a large state machine the task can be truly trying. A state machine should be visualized optimizing the layout with respect to alignment, symmetry, balance, crossing edges, number of bends of transitions etc.

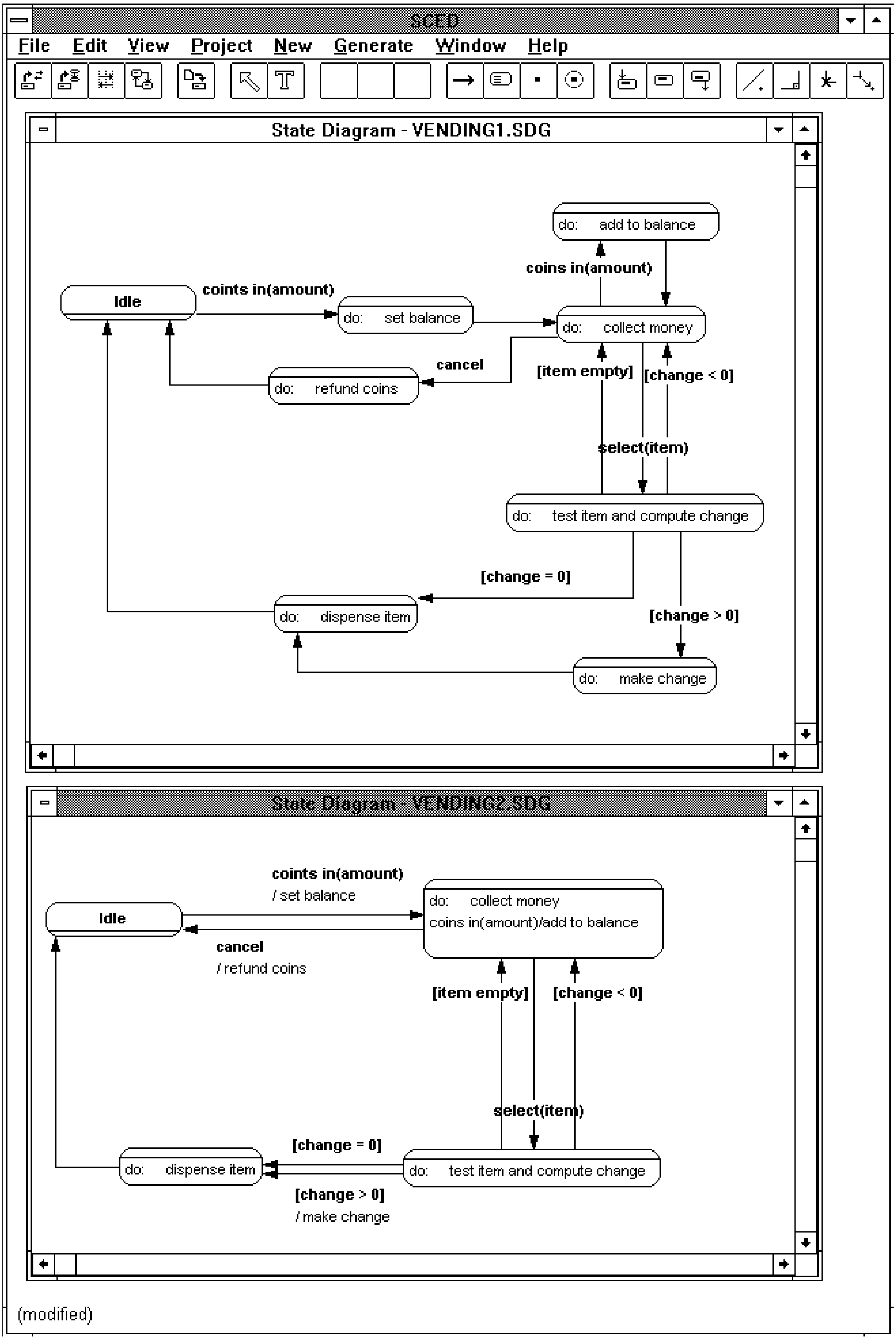

Fig. 8. Transforming a state machine into OMT notation.

We have applied an existing linear-time layout algorithm for directed graphs in SCED [Num91]: a state machine is interpreted as a directed graph in which states are the vertices and transitions are the edges of the graph. The advantage of this algorithm is its capability to handle arbitrary sized vertices. Although the states in a state machine are normally of small size, there are no fixed limits to the width and height of the state box: the actions written inside the box can take an arbitrary amount of space.

Unfortunately, the basic layout algorithm is far from sufficient, mostly because of the special features of OMT-type state machines. In addition to making some general improvements concerning e.g. horizontal centering of vertices, vertical alignment of vertices and edges, vertical packing of vertices, and bend elimination (for details, see [MST94b]), we had to solve some problems specific to state machines. These problems are discussed below.

When the designer adds manually parts of a state machine with automatically produced layout, or constructs the whole state machine from scratch, automatic layout should be applied very carefully: the designer would be completely lost if the whole state machine were rearranged after some editing actions. Hence the designer should be able to specify which parts of the state machine have satisfactory layout. To accomplish this, SCED allows the designer to select a rectangular area from the state machine and freeze it for future layout. During the layout process, the internal states of a frozen rectangular area are invisible: a frozen area is treated as a single state. After layout has been created for the state diagram, the transitions between internal states and the external states are routed properly.

The original layout algorithm accepts only simple graphs, that is graphs that do not contain multiple edges between any two nodes and self-loops. Since these are perfectly valid constructs in state machines, the layout algorithm has been modified accordingly. Multiple edges and self-loops are effectively hidden during the layout algorithm and added to the diagram while making final completing operations.

Often a practical state diagram has cyclic character: the behavior of an object can be described as a cycle of states and transitions, possibly augmented with short-cuts and auxiliary paths for exceptional cases. Since such a "main cycle" dominates the object behavior, it should be clearly visible in the state machine layout. In the current implementation we try to keep the main cycle relatively free of excessive bends. While doing basic layout optimization (horizontal and vertical alignment of vertices) those choices are preferred resulting in better visibility of the main cycle.

9 Concluding remarks

SCED has been in extensive use at Nokia, our industrial research partner, for about two years. Many of the facilities of SCED and extensions to the OMT scenario diagram notation were motivated by the needs emerging during this practical evaluation. Consequently, SCED fits nicely with the OMT++ design methodology [AJ94] developed at Nokia: this methodology relies heavily on scenarios as a basic instrument for extracting design information from requirements. However, some of the features discussed here (particularly the design-by-animation facility) have been implemented only recently and so far lack proper practical validation.

We feel that scenarios are a natural means to present expectations of the dynamic characteristics of a system at early stages of design, to be exploited throughout the software development process. As demonstrated by SCED, it is possible to apply existing machine learning techniques for inferring general dynamic specifications from scenarios; this opens up new directions not only for tool support but also for OO design methodologies. The tight coupling of scenarios and state diagrams becomes even more apparent when animated simulation of the dynamic model is combined with automatic state machine synthesis: this approach bridges the gap between an easily obtainable example and a full specification.

Automatic generation of state machines gives rise to new research problems. It is highly desirable to display the generated state machine in a form which is reasonably close to a hand-written one. Improving the representation of a state machine both in a syntactic and in an aesthetic sense is therefore necessary. We have shown that plain state machines can be systematically converted into (more compact) OMT notation, and that existing graph layout algorithms can be applied with some modifications for state machine layout.

Besides using SCED in real-life applications, our future work concerns relating the scenario-directed dynamic modeling technique supported by SCED with the broader context of object-oriented design. This includes methodological aspects as well as tool support. We feel that SCED should not be used as a separate tool, but rather as a part of an integrated OO development environment. We are currently building the functionality of SCED into a commercial integrated OO CASE tool (Stone AF).

SCED is available for free via anonymous ftp from cs.uta.fi, under directory /pub/sced.

Acknowledgements

SCED has been developed in a research project financed by the Center for Technological Development in Finland (TEKES), Nokia Research Center, Valmet Automation, Stonesoft, Kone and Insoft. Professors Ilkka Haikala and Erkki Mäkinen have significantly supported our work. Some parts of SCED have been implemented by Arto Viitanen, Juha Korhonen and Vesa Kauranen.

References

[AJ94] Aalto J-M., Jaaksi A.: Object-Oriented Development of Interactive Systems with OMT++. In: *Proc. TOOLS* 14, Prentice-Hall 1994, 205-218.

[BiK76] Biermann A.W. and Krishnaswamy R.: Constructing Programs from Example Computations, *IEEE Trans. Softw. Eng.* **SE-2** (1976), 141-153.

[CCITT92] CCITT: Document COM X-R 33-E, New Recommendation Z.120, Message Sequence Charts, July 1992.

[EiW96] Eick S.G., Ward A.: An Interactive Visualization for Message Sequence Charts. In: Proc. 4th Workshop on Program Comprehension, IEEE Computer Society Press, March 1996, 2-8.

[Gam95] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison-Wesley 1995.

[Har87] Harel D.: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8 (1987), 231-274.

[Jac92] Jacobson I.: *Object-Oriented Software Engineering - A Use Case Drive Approach*, Addison-Wesley, 1992.

[KM94] Koskimies K., Mäkinen E.: Automatic Synthesis of State Machines from Trace Diagrams. *Software Practice & Experience* 24,7 (July 1994), 643-658.

[KM96] Koskimies K., Mössenböck H.: Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In: Proc. International Conference on Software Engineering (ICSE '96), Berlin, March 1996, 366-375.

[LN95] Lange D.B., Nakamura Y.: Interactive Visualization of Design Patterns Can Help in Framework Understanding. In Proc. OOPSLA '95, Sigplan Notices 30, 10 (Oct 95), 342-357.

[MST94a] Männistö T., Systä T., Tuomi J.: SCED Report and User Manual. Report A-1994-5, Department of Computer Science, University of Tampere, February 1994.

[MST94b] Männistö T., Systä T., Tuomi J.: Design of State Diagram Facilities in SCED. Report A-1994-11, Department of Computer Science, University of Tampere, December 1994.

[Num91] Nummenmaa J.: Constructing Compact Rectilinear Planar Layouts Using Canonical Representation of Planar Graphs. *Theoretical Computer Science* 99 (1992), 213-230.

[Oma90] Oman P.: Maintenance Tools. IEEE Software 7, 3 (May 1990), 59-65.

[Por95] Portner N.: Flexible Command Interpreter: A Pattern for an Extensible and Language-Independent Interpreter System. In: Coplien J., Schmidt D. (eds.), *Pattern Languages of Program Design*, Addison-Wesley 1995, 43-50.

[Rum91] Rumbaugh J., Blaha M,, Premerlani W., Eddy F., Lorensen W.: *Object-Oriented Modelling and Design*, Prentice-Hall, 1991.

[SuM91] Sugiyama K., Misue K.: Visualization of Structural Information: Automatic Drawing of Compound Digraphs. IEEE Transactions on Systems, Man, and Cybernetics, SMC-21, 4 (1991), 876-892.

[Sys96] Systä T.: Synthesis of OMT State Diagrams. Internal Report, Department of Computer Science, University of Tampere, May 1996.