SYNTHESIZING OMT STATE DIAGRAMS

T. Männistö, T. Systä and J. Tuomi

(Tampere, Finland)

Abstract. OMT is a well-known object-orientated software development methodology. Various possibilities to support the automated construction of OMT state diagrams are considered. The proposed facilities are included in a prototype environment whose basic components are a scenario editor, a state diagram editor and a state diagram synthesizer. We discuss automated support for generating a state diagram on the basis of a set of scenarios and generating concepts of OMT notation for an existing state diagram.

1. Introduction

The use of object-orientated approach to software development has become popular during last years. This is partly explained by the fact that there are various techniques and languages avialable supporting this approach. However, most of the CASE tools supporting object-orientated programming are hardly more than specialized graphical editors giving tools for creating and editing the specifications of certain modeling techniques. The support of these tools is often limited to consistency checking and code generation. SCED has been developed for improving automated support for dynamic modeling in object-orientated software construction.

OMT (Object Modeling Technique [14]) has become a popular analysis and design method in object-orientated software development. Its virtues are relatively precise and rich notation, and systematic development steps (at least when compared to some of its competitors). OMT has been adopted as the basis of software development also in many industrial environments.

In OMT dynamic modeling is based on a variant of a finite state automaton in which both states and transitions can be associated with actions. The OMT variant of a state automaton is called a *state diagram*. A state diagram basically

consists of *states* and *transitions*. A state is an abstraction of attribute values of an object. In a system objects stimulate each other by sending and receiving events. A transition is a change of state caused by an event. Various other additional notations (such as entry, internal and exit actions of states, actions attached to transitions, nested states and concurrency notations) are allowed in state diagrams to make the description more expressive, compact, readable and precise.

The state diagram is composed after analyzing the behaviour of the system using *scenarios*, i.e. sequences of events occuring during a particular example run of the system. A scenario is presented formally as a diagram in which the participating objects are drawn as vertical lines, and events sent from one object to another are drawn as horizontal arcs between the object lines; here we will use the term scenario to refer to this particular representation.

Scenarios are a natural and effective medium for thinking in general and for design in particular. There are four sources of argumentation supporting a scenario-based approach to object-orientated design. These correspond to the four disciplines or areas of research and development work, that underpin object-orientated design more broadly: cognitive science, usability engineering, software engineering and experience applying object-orientated design [4]. In cognitive science, e.g. artificial intelligence, stories are characterized as a basic representation for descriptions and explanations of events. Scenarios suit well for giving these sequential examples. In usability engineering it is important to make the communication between users and designers possible and both ways understandable in the early stages of design. Users need not understand the underlying design or implementation in order to provide highly specific change requests when these requests are given in form scenarios. Because of their clear and simple representation scenarios are easy to use and understand. In software engineering system designing can use interaction scenarios for designing user training and documentation as well as usability tests. Scenarios seek to be concrete; they focus on describing particular instances of use, and on a user's view of what happens, how it happens, and why. Scenarios are often open-ended and fragmentary: they help developers and users pose new questions, to question new answers, and open up possibilities [4]. Scenarios can be used in different levels of abstraction, and they can easily be refined. In object-orientated design scenarios and use cases are widely used for showing object integration.

SCED uses the OMT methodology, presented by Rumbaugh et al. in [14], as a guide-line, although the resulting system could be useful for other methods as well, in particular for methods with a scenario-driven approach. The software consists of three main components: a scenario editor, a state diagram generator (synthesizer) and a state diagram editor. State diagrams can

be synthesized automatically on the basis of scenarios by giving instructions for the synthesizer. Finally, the user can automatically or by using the state diagram editor add advanced OMT notations to state diagrams. Support for checking the consistency between state diagrams and scenarios is also available.

State diagrams and scenarios are widely used not only in some other object-orientated modeling techniques (e.g. OOSE method developed by Jacobson [8]), but also in software engineering as general. Hence the use of SCED is not limited to the OMT method or even to object-orientated software construction.

In Section 2 we will study the relationships between scenarios and a state diagram, and the basic idea and properties of the algorithm used for the state diagram synthesis. In Section 3 we show how the synthesized state diagram can be optimized using advanced OMT notations. In Section 4 the functionality of SCED is briefly introduced. Finally, in Section 5 we present some concluding remarks.

2. Automatic synthesis of state diagrams

In OMT scenarios are usually given first for "normal" cases and then for different kinds of "exceptional" behaviour. When a sufficiently complete set of scenarios exists, they are transformed into a state diagram, describing an example path in the state diagram; a state diagram is the union of all possible (usually infinite) scenarios.

A basic observation behind SCED is that the construction of scenarios, i.e. the sequences of evets occuring during a particular execution of a system, and the fusion of these scenarios into a state diagram can be supported by automatic tools far more than what is done by current systems. Biermann presents a method for synthesizing programs from their traces [2]. The method is originally used in an "autoprogramming" system which automatically constructs computer programs from example computations executed by the user. The system is presented by Biermann et al. in [2, 3].

The idea is that the user specifies the data structures of a program and describes (graphically) the expected behaviour of the program in the case of an example input in terms of primitive actions (like assignments) and conditions that hold before certain actions. Essentially, the user gives traces (i.e. sequences of such actions and conditions) of the expected program, and the algorithm produces the smallest program that is capable of executing the given example traces. Moreover, after giving some finite number of example traces, taken from a program, the algorithm produces a program that can execute

exactly the same set of traces as the original one - that is the algorithm learns an unknown program.

Roughly, Biermann's algorithm works as follows. First the minimal number of states n is estimated for the state diagram. Each action item in the trace is then associated with a state one after the other. If a nondeterministic state results (i.e. different actions will be performed after the state for the same condition), 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 diagram with n states cannot be achieved, n is increased by one, and the whole process is repeated.

Biermann's algorithm defines a minimum labeling for actions. This means that the number of different instances of actions in the resulting program, i.e. the number of nodes in the directed graph illustrating resulting program, is minimized.
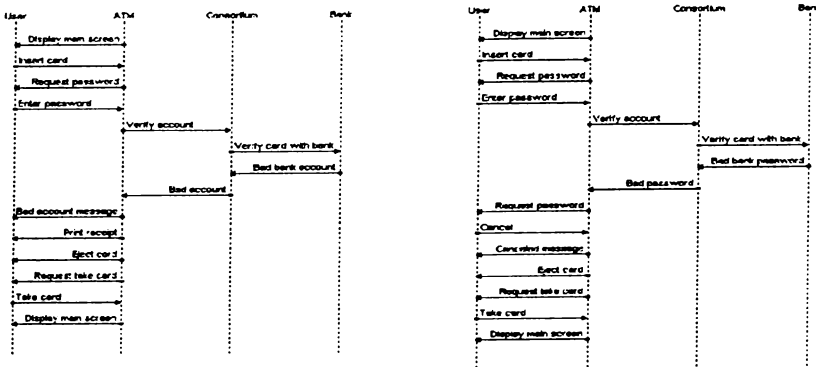

Fig.1. Two ATM scenarios: one for describing a case of bad account, and one describing cancellation after entering a wrong password

Due to the similarity between the concepts of a program (in Biermann's sense) and a state diagram, Biermann's algorithm can be applied to state diagram synthesis as well. A program trace corresponds to a vertical object line in a scenario: each outgoing event arc corresponds to an action (sending the event), and each incoming event arc corresponds to a condition (the event arrives). Hence, an event arc is interpreted as an action from the sender's point of view and as an event from the receiver's point of view. A condition corresponds simply to the arrival of a particular event. Together with some

relatively straightforward conventions (see [9]) Biermann's algorithm therefore synthesizes state diagrams from a set of scenarios.

Two example scenarios are shown in Figure 1. A synthesized state diagram for object "ATM" is shown in Figure 2. For practical state diagrams of reasonable size the algorithm is fast (the synthesis takes only fractions of a second), but unfavourable state diagrams may take tens of seconds to synthesize. This is due to exhaustive search used in the algorithm. Biermann et al. have discussed techniques for speeding up the algorithm in [3]. Comparing to some other inference algorithms for synthesizing programs from their traces Biermann's algorithm has an advantageous possibility to use partial traces, i.e. traces not starting from an initial state or ending at a final state. Specially when applied to state diagram synthesis, it would be sensible to require that a scenario begin at an initial state for all the participants.
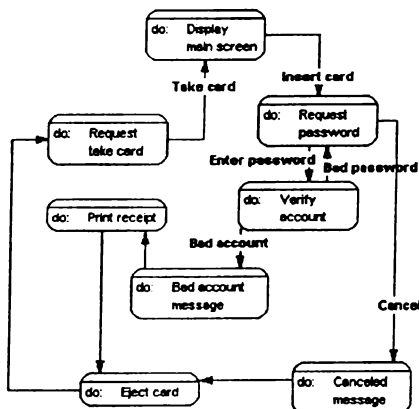

Fig.2. State diagram for class ATM

The main problem in applying Biermann's method for state diagram synthesis is that the programs are more "complete" than state diagrams: there is usually a valid continuation for every possible combination of variable values in every point of a program (except after halt statement), but there is usually not a valid transition for every possible event in every state. For this reason the learning results of Biermann do not necessarily hold for state diagrams. The main cause for this difference is the fact that in the case of state diagrams the traces (scenarios) do not give sufficient information, since the forbidden transitions are not represented. In most cases this yields to a synthesized state diagram that overgeneralizes the given scenarios. Usually this is exactly the desired effect, but in some cases the result is not what the user expects. As an example of overgeneralization consider the state diagram in Figure 2. It

accepts both event traces specified in Figure 1. However, the state diagram accepts many other event traces as well, e.g. traces in which the user enters a bad password several times. In fact, it accepts an infinite number of event traces. There is no way to tell which event traces are "real" ones specified in scenarios and which are not.

This problem can be solved in several ways. We could simply require that the scenarios should also cover forbidden transitions - that would give the algorithm sufficient information to avoid undesired overgeneralization. However, this would be rather inconvenient and unnatural for the user. A slightly better way to exploit user-given information is to give the user a possibility to give descriptive labels to certain actions in the scenarios: if two actions that could be merged into same state by the algorithm have different labels, the algorithm can keep the actions in separate states. This possibility is offered in SCED: the labels are used as state names in the state diagram. This approach does not always seem to be sufficient or very convenient, since events corresponding to actions to be separated might appear in several scenarios. In SCED tools for splitting and merging states after the synthesis are offered. In addition, scenarios can be desynthesized out of the state diagram. Hence detecting possibly overgeneralized states before or during the sythesis becomes less important. It also means that the responsibility of finding such states and separating them is left to the designer. This seems to be the most versatile approach to handle the overgeneralization problem.

3. Generating OMT notations

A synthesized state diagram consists of states, actions in states and transitions that usually are labeled. Due to Biermann's method each state can have at most one action [2]. We call this kind of state diagram a *plain state diagram*. In this section we discuss how an OMT-type state diagram can be generated on the basis of a plain state diagram.

In an OMT state diagram more information is attached to states and transitions than in a plain diagram. There are several ways to attach information to states. E.g. states may have *entry actions*, i.e. actions that are executed immediately after entering a state. Correspondingly, *exit actions* are executed immediately after leaving a state. Furthermore, an event can cause an action to be performed without causing a state change. Such actions are called *internal actions*. Actions can also be attached to transitions: an event corresponding to a transition may cause an execution of actions. Such

actions are called *transition actions*. All these *OMT actions* are instantaneous operations: executions of them have duration insignificant to the resolution of the state diagram. When optimizing a state diagram by generating OMT notations, some *normal actions* of a plain state diagram (associated with a keyword "do") are moved and used as OMT actions. Hence, also all normal actions are regarded as intantaneous operations. Continuous and sequential activities that take time to complete should not be changed when optimizing a state diagram. Allowing them requires a way they could be distinguished from instantaneous actions.

Using OMT-style notation makes it possible to reduce the number of needed states and transitions. While generating OMT concepts, we aim to minimize the number of needed states and transitions so that the information content of the state diagram is preserved: both the original plain state diagram and the resulting OMT state diagram should accept exactly the same scenarios.

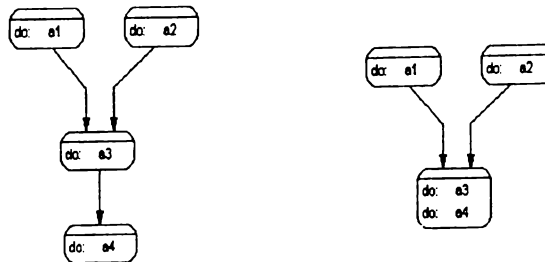## 3.1. Gathering several actions into a single state

Fig.3. Gathering actions

A state diagram may include transitions that fire automatically after the activity associated with the source state is completed. Such transitions are called *automatic transitions*. In a plain state diagram several actions can be gathered into a single state by removing automatic transitions and states they enter. This can be done for the longest sequence of normal actions which is approved by all the paths in a state diagram on the right. The reversal operation to action gathering is straightforward: needed states and automatic transitions are added.

3.2. Reducing the number of states and transition by combining

In this section we introduce a way to add internal, entry, exit and transition actions into a state diagram. Adding these concepts not only makes a state

diagram more compact but it also emphasizes the similar behaviour of different paths running through a state diagram.
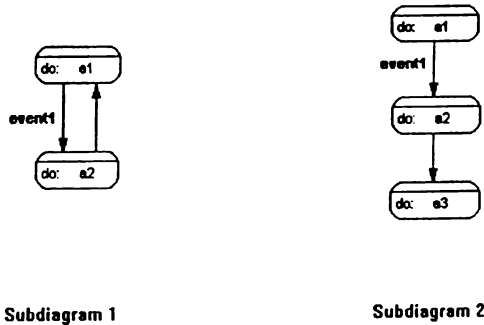

Subdiagram 1   Subdiagram 2

Fig.4. Subdiagrams that could be replaced by OMT concepts

It can be seen [16] that a plain state diagram may include two kinds of subdiagrams which could be shown in terms of these concepts. These subdiagrams are shown in Figure 4. In Figure 5 an example state diagram on the right is generated from a plain state diagram on the left by adding OMT actions.
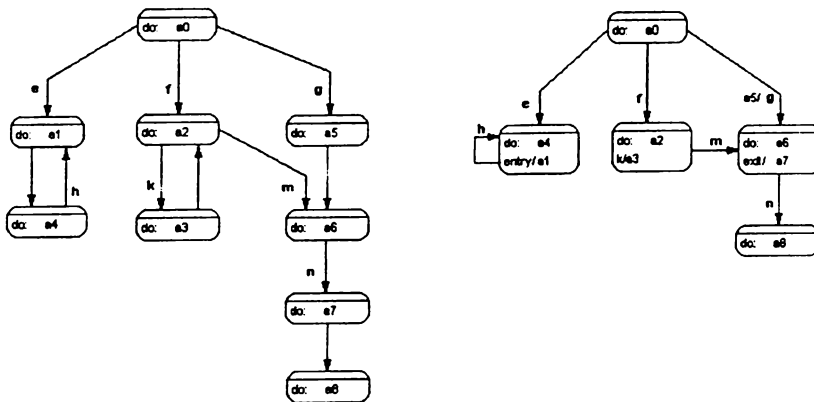

Fig.5. A plain state diagram on the left and a state diagram with OMT concepts on the right

The information associated with a state should be easily seen. If entry, exit or internal actions are used, all the essential information connected to the state is written inside the state box, the state itself shows what has to be done just before entering the state, immediately after leaving the state or when a

certain event is received which does not cause a state change. Therefore we prefer combining information into states to combining it into transitions. Hence internal, entry and exit actions are formed before transition actions. Entry and exit actions can be formed for a state even if it already has internal actions. An internal action can be seen to contain more specific information than entry or exit actions, firing an internal action depends on a single event while firing entry and exit actions may depend on several events. For these reasons we form internal actions before entry and exit actions. Hence generation of these OMT concepts is done in order: internal, entry/exit and transition actions. The following list contains cases in which these concepts can be adopted. The states in subdiagram 1 in Figure 4 are called "state 1" and "state 2" in top-down order. Correspondingly, the states of subdiagram 2 are called "state 1", "state 2" and "state 3". All such subdiagrams are handled separately.

1. Internal actions

Subdiagram 1: All transitions entering state 2 have to be leaving transitions of state 1.

2. Entry actions

Subdiagram 1: In addition to the automatic transition there cannot be any other transition entering state 1. While forming an entry action all transitions entering state 2 have to be changed to enter state 1.

Subdiagram 2: In addition to the automatic transition there cannot be any other transition entering state 3. While forming an entry action all transitions entering state 2 have to be changed to enter state 3.

3. Exit actions

Subdiagram 1, subdiagram 2: All transitions leaving state 1 have to be entering a state with actions "a2". In addition all transitions entering these states have to leave state 1. While attaching an exit action to a state all transitions leaving the states to be removed (like state 2) have to be changed to leave state 1.

4. Transition actions

Subdiagram 1, subdiagram 2: "event1" has to be the only entering transition attached to state 2.

3.3. Nested states

Rumbaugh's nested state diagram is a form of generalization on states. Generalization is an "exclusive-or relationship". An object in a high-level diagram must be in exactly one state in a nested diagram. The states in the nested diagram are all refinements of the state in the high-level diagram [14].

In OMT state diagrams states may have *substates* that inherit the transitions of their *superstates*. Rumbaugh uses a contour, i.e. a large rounded box enclosing all of its substates, as the notation for a superstate. This notation is similar to Harel's notation for clustered states [7]. Nested contours are good for conveying the intuitive feel that the general case includes all its specialized varieties but awkward to draw if nesting exceeds two or three levels [14]. For our purposes adopting superstate concept to SCED state diagrams is desirable because of its powerful way to outline the structure of state diagrams and to decrease the number of transitions needed.
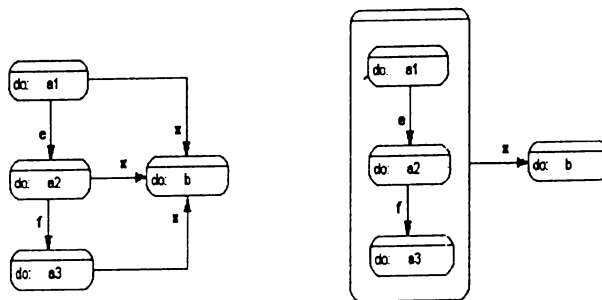

Fig.6. Two state diagrams for the same state machine

Superstates can be formed in a state diagram if there are several states with similarly labeled leaving transitions, all of them entering the same state. When forming a superstate we replace all such transitions with a single, similarly labeled transition. The new transition is drawn from the superstate contour. We say that this transition is a *leaving transition* of the superstate. Figure 6 shows two state diagrams for the same state machine. The transitions with label "x" in the upper state diagram are replaced with one leaving transition of the superstate in the lower state diagram. The number of needed transitions decreases from five to three. A state diagram may have mutually disjoint or nested superstates. Hence two superstates can be formed for the same state diagram if one of the following conditions is satisfied:

1. the sets of substates are mutually disjoint;
2. one set of substates is included in the other and the labels of leaving transitions in corresponding superstates differ.

Otherwise we say that superstates are *mutually contradictional*. There can be several, also mutually contradictional, possibilities to form superstates. While constructing superstates automatically, we cannot make a difference between these possibilities on the basis of semantics. Hence, we have to specify another criteria for comparing two alternative superstate combinations. Even

though maximizing the reduction in the number of transitions is in accordance
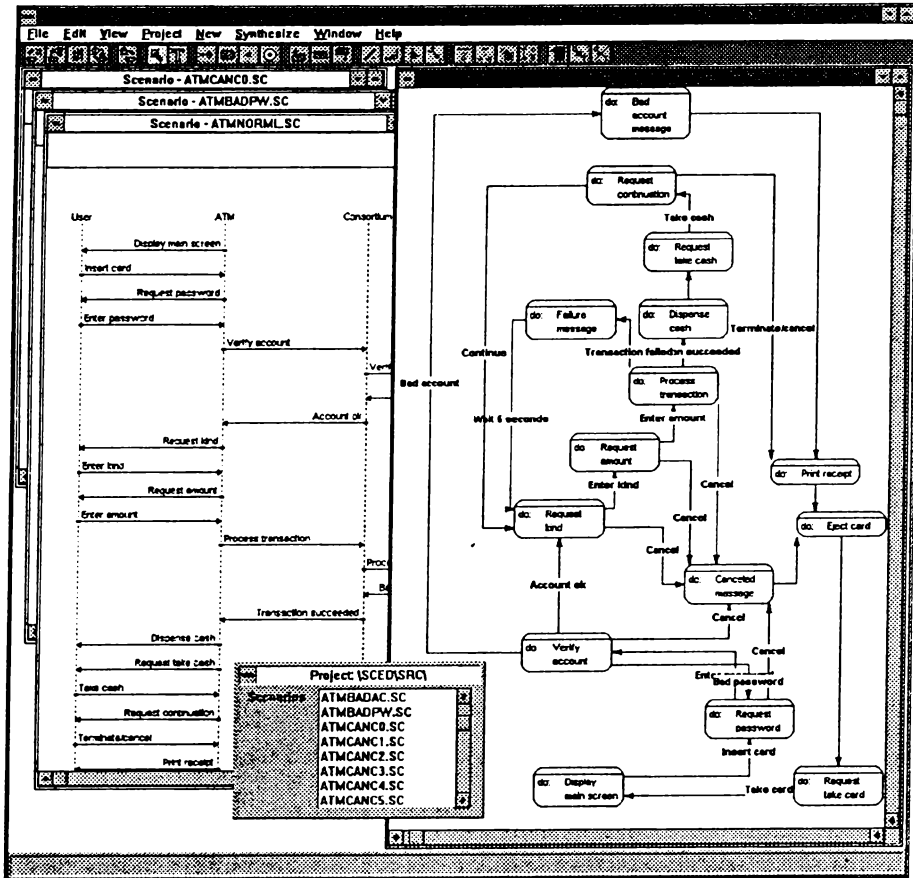with our principles for generating other OMT concepts it may not always result

Fig. 7. The user interface of SCED

in a state diagram best suited for user's needs. If there are many transitions
between a certain substate and states outside the superstate, the state diagram
does not probably meet the user's intentions: the exceptional behaviour of
the object in substates is more common than the similar behaviour of the
object in these substates giving a reason to form a superstate. Transitions
that represent the exceptional behaviour, i.e. transitions between a substate
and states outside a superstate, are called *critical transitions*. The problem is

that we cannot minimize the number of transitions needed and the number of critical transitions at the same time.

If we decided to minimize the number of transitions needed we would still have an NP-complete problem to solve [16]. So far, superstates are generated in the following way: the largest set of states, in which all states have a similar transition, forms the first superstate. The second largest set forms the next superstate if it does not contradict to the already formed ones, and so on. Although this greedy method minimizes neither the number of transitions needed nor the number of critical transitions, it gives satisfying results in most reasonable cases. The advantage of this method is that it is fast.

4. SCED

A central idea of SCED is to support a design-by-example approach for OOA/D. Similar direction is taken also by Jacobsen et al. [8], although in their method the use of scenarios is less systematic: we expect that many steps from scenarios to running software can in fact be automated.

The facilities described in this paper are part of a prototype environment for developing dynamic models of object-orientated software [16]. The project has been carried out in the University of Tampere and in the Tampere University of Technology in cooperation with Finnish industrial partners, which have given us useful feedback about their usage patterns of scenario editing tool and provided valuable commentary during the development of SCED. The name of the software "SCED" (SCenario EDitor) was originally given to the scenario editor component, but later adopted for the whole environment. A more detailed description of the functionality of SCED is given in [15] and [16].

SCED consists of two independent editors: a scenario editor and a state diagram editor. Typically most of the user interaction is concentrated on use of these editors. To large extent, both of the editors can be manipulated directly avoiding the use of several dialogs.

The scenarios that can be edited and processed with SCED are somewhat more elaborated than the scenarios used with OMT methodology described in [14]. SCED offers various other concepts for scenarios to make the description more compact and precise.

SCED state diagrams lack a notation to describe concurrency. Otherwise they are similar to the ones in OMT. The automatic layout facility in SCED for state diagrams is based on a layout algorithm for orthogonal drawings by

Nummenmaa [11]. An implementation of this algorithm for ER-diagrams has been described in [12]. One of the features of this algorithm is that the nodes in the drawing need not be of the same size, actually the sizes are completely arbitrary. In the state diagram notation used in SCED nodes (i.e. states) of different sizes occur naturally as the states may or may not contain additional such as internal actions, entry/exit actions, etc.

The basic layout algorithm has been modified to better support the layout of state diagrams and enhanced with a number of optimizations (e.g. node alignment, node packing and bend elimination) to improve the readability of the resulting layout.

The third part of SCED in addition to scenario and state diagram editors is a generator which is not a user-visible component of SCED. At any time during scenario editing the user can select one participating object and ask the generator to synthesize a state diagram automatically for this object. The synthesis can be done for one scenario only or for a specified set of scenarios. In the latter case the generator synthesizes each scenario that includes the selected object. Scenarios can also be synthesized to an existing state diagram. The resulting state diagram is editable using the state diagram editor. Further, the generator can be asked to add advanced OMT concepts to state diagrams or to remove them, preserving the information. SCED can also be asked to generate an event flow diagram on the basis of the current scenario set. Event flow diagrams are shown in a matrix form and they are not editable. Finally, support for checking the consistency between a state diagram and scenarios is available. Various services are offered by SCED for analyzing state diagrams with respect to scenarios.

SCED views a set of related scenarios as a project. The project window, which shows the names of scenarios belonging to the project, can be used to select scenarios for which a desired operation is done. The user interface of SCED is shown in Figure 7.

SCED has been developed in - and for - the Microsoft Windows operating environment. The tools that are being used for the development work have been selected so that porting to Unix with OSF/Motif should be possible with moderate effort. These tools are:

1. Borland C++ - C++ language compiler [6].
2. LEDA - *Library of Efficient Data types and Algorithms* [13]. Portable across wide range of platforms, MS-DOS and several Unix systems.
3. wxWindows - GUI library. Portable between MS-Windows, Windows NT, Motif, Open Look.

Borland C++ provides an object-orientated layer to insulate the applications programmer from direct access to MS-Windows API. This layer is called

the Object Windows Library (OWL) and it is included with the Borland C++ and Application Frameworks package. However, the OWL was directly tied to Borland C++ environment, because a non-standard method is used to associate C++ functions to MS-Windows message. OWL is not currently available with other GUI environments besides MS-Windows and OS/2. Furthermore, OWL does not obviate the need to communicate directly with the MS-Windows API.

To support the porting of SCED to Unix/X-Windows environment and - more significantly - to simplify the SCED's actual development a portable object-orientated GUI library - wxWindows - has been used in the current SCED implementation instead of OWL.

5. Conclusions

SCED has been in use at NCS (Nokia Cellular Systems) during the SCED project since 1993. The software has been actively used by almost 100 designers and engineers. The response among users has been favourable especially when compared with commercial tools and their support for creating and editing event traces or scenarios.

SCED has been mostly used inside NCS as a scenario editor, thus the state diagram related facilities have so far seen relatively little actual production use. The scenario facility of SCED has been used on many different levels of software design/implementation process, e.g. for describing the behaviour and interactions among dynamical objects in a subsystem (object as a participant), and for documenting the behaviour and connections between different subsystems (subsystem as a participant).

While SCED has proved itself as a viable alternative in dynamical object modeling it is clear that to become a more useful tool SCED should be integrated with the entire design process. It would be highly desirable to integrate SCED with the enterprise's object repository or database. This would enable SCED to make consistency checks e.g. to ensure that objects used in scenarios are defined in the object database and the events (and their visibility) correspond to defined methods in the object database. Adding an object database and object diagram editing facilities would make SCED very useful as an independent tool.

References

[1] **Aalto J.M. and Jaaksi A.,** Object-orientated development of interactive systems with OMT++, *Proc. TOOLS,* **14** (1994), Prentice Hall, 205-218.

[2] **Biermann A.W. and Krishnaswamy R.,** Constructing programs from example computations, *IEEE Trans. Software Engineering,* SE-2 (1976), 122-153.

[3] **Biermann A.W., Baum R.I. and Petry F.E.,** Speeding up the synthesis of programs from traces, *IEEE Transactions on Computers,* C-24 (1975), 122-136.

[4] **Carroll J.M., Mack R.L., Robertson S.P. and Rosson M.B.,** Binding objects to scenarios of use, *Intern. Journal of Human-Computer Studies,* **41** (1994), 243-276.

[5] **Coplien J. and Schmidt D.,** *Pattern languages of program design,* Addison Wesley, 1995.

[6] **Ellis M. and Stroustrup B.,** *The annotated C++ reference manual,* Addison Wesley, 1990.

[7] **Harel D.,** Statecharts: a visual formalism for complex systems, *Science of Computer Programming,* **8** (1987), 231-274.

[8] **Jacobson I. et al.,** *Object-orientated software engineering - A user case driven approach,* Addison Wesley, 1992.

[9] **Koskimies K. and Mäkinen E.,** Automatic synthesis of state machines from state diagrams, *Software Practice and Experience,* **24** (7) (1994), 643-658.

[10] **Koskimies K., Männistö T., Systä T. and Tuomi J.,** SCED - An environment for dynamic modeling in object-orientated software construction, *Proc. Nordic Workshop on Programming Environment Research '94, Lund,* Department of Computer Science, Lund Institute of Technology, Lund University, June 1994. 217-230.

[11] **Nummenmaa J.,** Constructing compact rectilinear planar layouts using canonical representation of planar graphs, *Theoretical Computer Science,* **99** (1992), 213-230.

[12] **Nummenmaa J. and Tuomi J.,** Constructing layouts for ER-diagrams from visibility representations, *Proc. of the 9th International Conference on Entity-Relationship Approach,* North Holland, 1991.

[13] **Näher S.,** *LEDA User Manual, Version 3.0,* Max-Planck-Institut für Informatik, 1992.

[14] **Rumbaugh J. et al.,** *Object-orientated modeling and design,* Prentice Hall, 1991.

[15] **Männistö T., Systä T. and Tuomi J.,** *SCED report and user manual,* University of Tampere, Report A-1994-5.

[16] **Männistö T., Systä T. and Tuomi J.,** *Design of state diagram facilities in SCED,* University of Tampere, Report A-1994-11.

T. Männistö
Software Systems Laboratory
Tampere University of Technology
P.O.B. 553
FIN-33101 Tampere, Finland
tm@cs.tut.fi

T. Systä
Department of Computer Science
University of Tampere
P.O.B. 607
FIN-33101 Tampere, Finland
cstasy@cs.uta.fi

J. Tuomi
Department of Computer Science
University of Tampere
P.O.B. 607
FIN-33101 Tampere, Finland
jjt@cs.uta.fi