DESIGN OF STATE DIAGRAM FACILITIES IN SCED

Tatu Männistö, Tarja Systä and Jyrki Tuomi

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1994-11

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1994-11, DECEMBER 1994

DESIGN OF STATE DIAGRAM FACILITIES IN SCED

Tatu Männistö, Tarja Systä and Jyrki Tuomi

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

ISBN 951-44-3703-9
ISSN 0783-6910

DESIGN OF STATE DIAGRAM FACILITIES IN SCED

TATU MÄNNISTÖ, TARJA SYSTÄ, AND JYRKI TUOMI

ABSTRACT. Various possibilities to support the automated construction of state diagrams of the OMT method are considered. The proposed facilities are planned to be included in a prototype environment whose basic components are editors for scenarios and state diagrams. The considered support covers automatic means for synthesizing a state diagram on the basis of scenarios, for making the state diagram as compact as possible, for determining a satisfactory layout for the state diagram, and for maintaining consistency between scenarios and state diagrams. The prototype environment — called **SCED** — currently supports editing of scenarios and automatic synthesis of state diagrams.

1. Introduction

OMT (Object Modeling Technique [19]) has become a popular analysis and design method in object-oriented 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 industrial environments.

In OMT, dynamic modeling of software 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 will be called a *state diagram*. Various other additional notations are allowed in state diagrams to make the description more expressive, compact, readable, or precise. The state diagram is composed after analyzing the behavior of the system using *scenarios*, i.e. sequences of events occurring 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. A state diagram specifying the behavior of an alarm clock is shown in figure 1, and a scenario is given in figure 2.

In this paper we present techniques for automatically deriving a state diagram representation of the behavior of a particular object belonging to a system under design. The resulting state diagram should be comparable to a hand-written one with respect to size and readability. Further, there should be mechanisms guaranteeing consistency between scenarios and state diagrams, even though one or the other is changed. The facilities described here will be part of a prototype environment for developing dynamic models of object-oriented software [13]. The project is carried out in co-operation with Finnish industrial partners.

In the following section we briefly describe the basic synthesis algorithm for state diagrams. This part of the work has already been implemented, and is discussed in more detail in [12]. In section 4 various techniques for simplifying a state diagram using certain OMT notations are described. Section 5 discusses layout algorithms for state diagrams and in section 6 we study the consistency problems between scenarios and state diagrams. Finally some concluding remarks are presented in section 11.

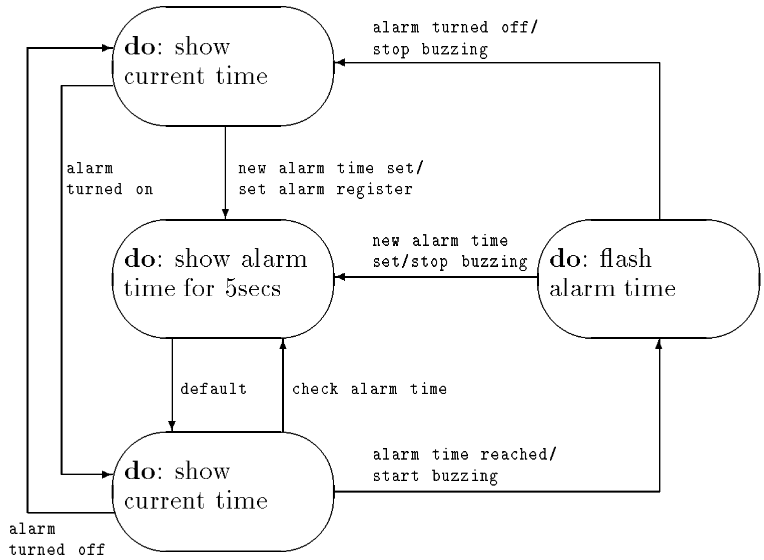

FIGURE 1. The state diagram for an alarm clock control unit

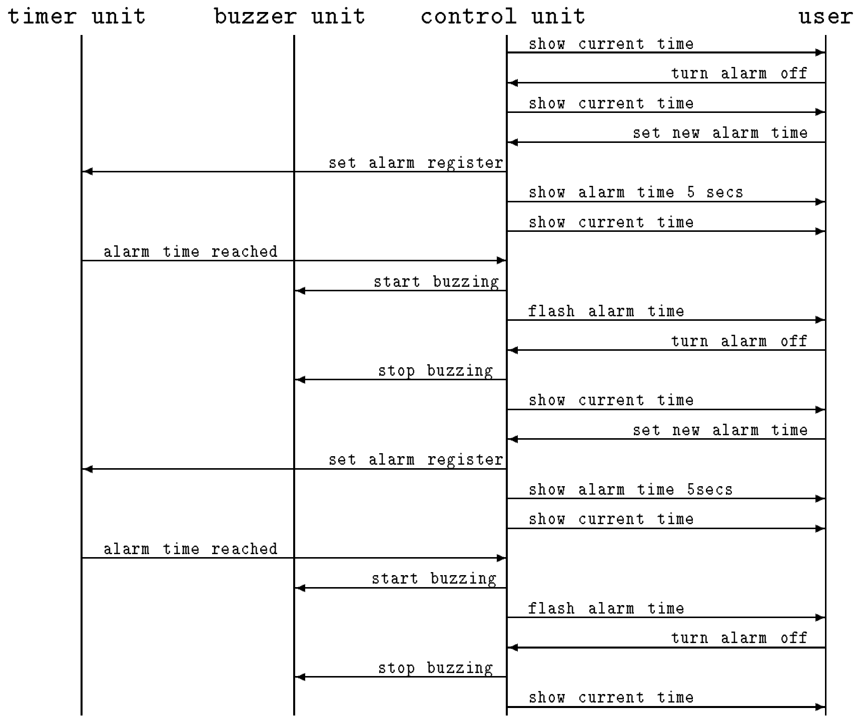

FIGURE 2. A scenario for an alarm clock

2. NOTATIONS AND CONCEPTS

In this section we shall briefly introduce the notations and concepts which will be used in this paper.

2.1. Scenarios. A scenario consists of participating *objects* and *events* which occur between these objects. Objects are shown as vertical lines and events as horizontal arrows extending from a sender object to a receiver object (see figure 2). In addition to these basic building blocks **SCED** scenarios use several other notations [13]. One of them is an *action* which represents a leaving event with undefined receiver object.

An *event trace* of an object is a sequence of events which are sent or received by a particular object, in the order of their appearance.

2.2. State diagrams. A *state* is an abstraction of attribute values of an object. In a system objects stimulate each other causing state changes by sending and receiving events. A state change caused by an event is called a *transition.* A system consisting of states of objects, events and transitions is described as a *state diagram.* In other words, a state diagram relates events and states. In addition to values of objects, states can also represent *actions.* An action is either a continuous activity or a sequential activity. A continuous activity ends when an input event causes a state change. A sequential activity ends when the operations are completed or an input event causes a state change. In both cases action starts immediately after entering the state. State diagrams are drawn as directed graphs in which nodes represent states and directed edges represent transitions. States are drawn as rounded rectangles including actions indicated within it and an optional name for the state separated with a horizontal line from the action part. Transitions are labeled with the names of the events causing them. An *automatic transition* which causes a state change immediately after the actions of a state are executed, is labeled with the word “default”. If event e causes a state change from state A to state B, the transition with label e is said to *fire.* We also say that this transition *leaves state* A and *enters state* B. A transition is drawn as an arrow from the state it is leaving to the state it is entering. Connections between scenarios and state diagrams are explained in more detail in [13].

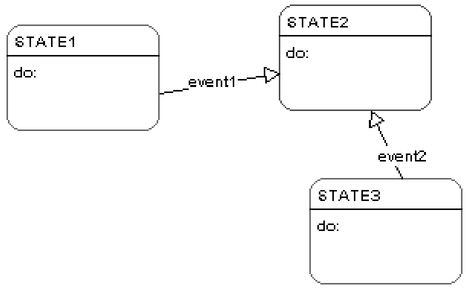

FIGURE 3. A state diagram with three states and two transitions

In the state diagram of figure 3 , the state “STATE2” has two entering transitions (labeled “event1” and “event2”) but no leaving transition. States “STATE1” and “STATE3” have both one leaving transition but no entering transition.

In **SCED** state diagrams all actions are considered instantaneous. Therefore, the placement of actions can be changed leading to the deletion of some states and transitions, requiring that the information content of the state diagram remains the same after these changes. This makes it possible to adopt most of the advanced modeling concepts of OMT to **SCED** state diagrams.

If an action is attached to a transition, it means that the action is executed right after the transition fires. In this case the name of the action is separated from the name of the transition with a “/” character. Such an action is called a *transition action.* There seems to be no inherent reason why there couldn’t be several actions attached to a single transition. In figure 1 there are four transition actions. An *entry action* and an *exit action* are shown inside the state box. An entry action is preceded by the keyword "entry" and a "/" character. Correspondingly, an exit action is preceded by the keyword "exit" and a "/" character (see figures 17 and 16). Whenever a state is entered (exited), by any entering (exiting) transition, the entry (exit) action is performed. An entry (exit) action is equivalent to attaching the action to every entering (exiting) transition [19]. An event can also cause an action to be performed without causing a state change. The event name is written inside the state box, followed by a "/" character and the name of the action. Such an action is called an *internal action* (see figure 14). Events causing the execution of internal actions do not cause the execution of entry or exit actions [19].

3. Basic synthesis algorithm for state diagrams

In [2] a method is presented for synthesizing programs from their traces. The idea is that the user specifies the data structures of a program and describes (graphically) the expected behavior 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.

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 [12]), Biermann's algorithm therefore synthesizes state diagrams from a set of scenarios. For practical state diagrams of reasonable size the algorithm is fast, but due to backtracking unfavourable state diagrams may take tens of seconds to synthesize.

The main problem in applying Biermann's method for state diagram synthesis is that 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. Suppose that scenarios are extracted from a given state diagram S. The algorithm then produces a state diagram S', but S' can be (and usually is) more general in the sense that it accepts scenarios not accepted by the original state diagram S. This will be true for any finite number of input scenarios. The point is that in the case of state diagrams the traces (scenarios) do not give sufficient information, since the forbidden transitions are not represented. The fact that the synthesized state diagram may overgeneralize the given scenarios is usually exactly the desired effect, but in some cases the result is not what the user expects.

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 require that the user gives 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. Another approach is to use some heuristic rules to identify "suspicious" state merging performed by the synthesis algorithm, and ask the user to either accept or reject the merging. This approach has been taken in the current implementation. Finally, it should be noted that the user is free to edit the resulting state diagram in an arbitrary way: we expect that the synthesized state diagram is in any case only a first approximation of what the user really wants.

4. Simplifying state diagrams using OMT notation

OMT state diagrams differ considerably from general, flat state diagrams, mainly because they can be structured to permit concise descriptions of complex systems. The ways of structuring state machines are similar to the ways of structuring objects: generalization and aggregation [19]. Dynamic modeling concepts of state generalization and aggregation are based on Harel's statecharts [8]. They are included to avoid combinatiorial explosion of transitions in complicated systems. Also the dynamic modeling notation of OMT makes the structure of state diagrams clearer by attaching more information to states and transitions than in traditional state machines.

So far, **SCED** state diagrams are generated and drawn by using rather primitive, general notation including only few OMT modeling concepts. Advanced OMT notation makes state diagrams more compact and readable. In particular, the OMT notation is useful for specifying the dynamic model for complex systems. Further, well structured state diagrams show the relations between object models and dynamic models more clearly. Hence, it is desirable to add full OMT notation to **SCED** state diagrams.

Our main principle during the process of adding the OMT notation to a **SCED** state diagram is to minimize the number of transitions and states while preserving the information of the state diagram. In this paper we describe how this can be done by adopting some changes to currently used synthesis algorithm and by adding advanced OMT notation to state diagrams. We will see that some parts of the OMT notation can be added automatically after the synthesis while others require extensions to the notation of scenarios.

4.1. Modifications of the synthesis algorithm. Before concentrating on the advanced modeling concepts of OMT, we shall review some problems concerning the present synthesis method.

4.1.1. *The overgeneralization problem.* State diagrams generated by using Biermann's method have a minimum number of states [11]. This means that in a state diagram there is usually more information than in the scenarios it was synthesized from. This property also raises some problems. For example, the used algorithm may merge together states which represent logically different situations. Let's consider the following event trace concerning the use of an alarm clock:

(show current time, set new alarm time)
(show alarm time 5 secs, default)
(show current time, alarm time reached)
(buzzing, turn alarm off)
(show current time, set new alarm time)
(show alarm time 5 secs, default)
(show current time, null)

The minimal state diagram produced by using pure Biermann's method is shown in figure 4.

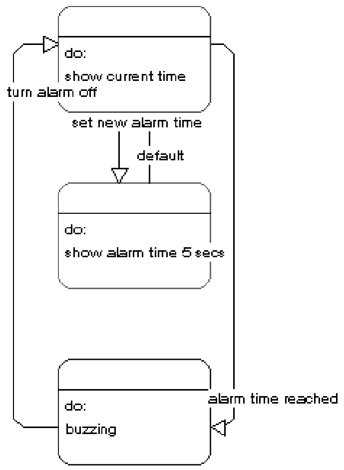

FIGURE 4. The minimal state diagram for the control unit of an alarm clock.

This is not an acceptable solution, since "show current time" phase is interrupted by "alarm time reached" event even if the alarm is not set on. The algorithm has merged two states together which represent logically different situations thus overgeneralizing the state diagram. This state diagram also allows setting a new alarm time even if the previously set alarm time is not yet reached. Such a piece of information is not included in the event trace. The latter example of overgeneralization is not harmful but the previous one is.

It is difficult to detect these cases during the automatic synthesis, because the nature of the problem is highly semantical. To avoid this problem the following heuristical "flying visit rule" was used: a trace item is not allowed to be associated with an existing state if the trace both enters and leaves this state with new transitions [11]. In such a case the new trace makes only a flying visit to an existing path, reusing a single state; this is doubtful and in any case less fruitful combining [11]. From the event trace shown above this new rule will give the state diagram in figure 5.

It has turned out that the "flying visit" rule is often too strong; it may cause the separation of states that were ment to be merged. Further, when scenarios are synthesized in a certain order, the "flying visit" rule may cause a separation of some states which would not happen if the scenarios were synthesized in a different order. For example, if event traces shown below were synthesized in order 1,2 and 3, the synthesizer would notice the possibility of overgeneralization according to the flying visit rule while trying to merge states with action "Common action". This would not happen if the event traces were synthesized in order 1,3 and 2.

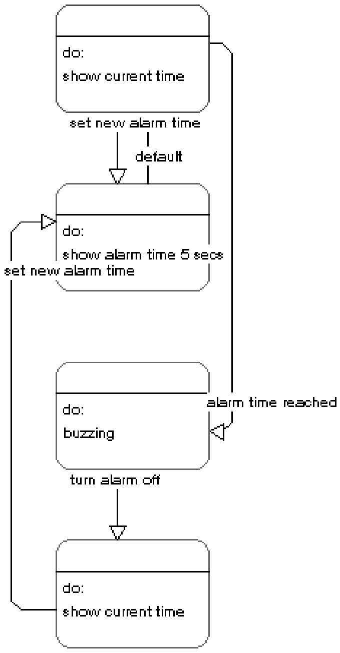

FIGURE 5. The acceptable state machine for the control unit of an alarm clock

event trace 1:

(action1,event1)
(Common action, event2)
(action2,null)

event trace 2:

(action3,event3)
(Common action, event4)
(action4,null)

event trace 3:

(action5,event1)
(Common action, event4)
(action6,null)

Since the use of the "flying visit" rule faces the problems explained above, the user was given a possibility to influence the merge during the synthesis; in every possible overgeneralization case the user decides if the states will be merged or not. Because the "flying visit" rule is rather strong, the number of these cases may be much larger than the number of harmful overgeneralization cases. This makes the use of the synthesizer unpractical. Since the state diagram editor will be built, detecting these possibly overgeneralized states during the synthesis becomes less important; spliting states should be possible afterwards as well. Hence, this checking will be leaved out and so state diagrams formed by the synthesizer will have minimum number of states.

4.1.2. *Automatic transitions vs. labeled transitions.* Allthough Biermann's notation of a program is clearly related to OMT's state machine, the relation is not trivial [11, 2]. For instance, an event in OMT seems to act in the same role as a condition in Biermann's notation. However, in Biermann's programs the absence of such a condition means that either no test was made or a test was made and the result was "false". So, if a particular instruction in a Biermann's program has several transitions leading away, an unlabeled transition and others labeled $C_1, C_2, \ldots, C_k$, an unlabeled transition is taken if none of the conditions $C_1, C_2, \ldots, C_k$ is "true" [1]. On the other hand, in OMT state diagrams an automatic transition (i.e. an unlabeled transition) without a guard fires when the activity associated with the source state is completed [19]. In other words, an automatic transition without a guard fires despite of any other events the object has received. Hence, the meaning of the absence of a condition in Biermann's notation differs from the meaning of guardless automatic transition in OMT notation. If we allowed a state to have both guardless automatic transitions and labeled transitions as leaving transitions, the labeled transitions would never fire. The problem will be the same if an automatic transition and a labeled transition have same guards. For these reasons we modified the Biermann's algorithm to satisfy the following rule: a state can not have both an automatic transition and a labeled transition as leaving transitions if these transitions have same guards or they are both guardless. The algorithm will process guards as strings. Hence, only guards which are written exactly the same way will be considered to be same.

4.1.3. *Gathering several actions into a single state.* In **SCED** state diagrams the actions of a state corresponds either to leaving arcs or to action boxes in scenarios [13]. Hence, because states are defined by their actions, there is always at most one action in a state. This is due to the Biermann's method where each state can have at most one labeled instruction [2]. In OMT notation states may have several actions. Gathering actions into a single state instead of forming a state for every action is not only semantically more reasonable but it also decreases the number of automatic transitions and the number of states in a state diagram. Hence, we modified the Biermann's method aiming to gather as many actions into a state as possible. This can be done for the longest sequence of actions which is approved by all paths in a state diagram. As an example, the state diagram in figure 6 can be changed to the state diagram in figure 7.

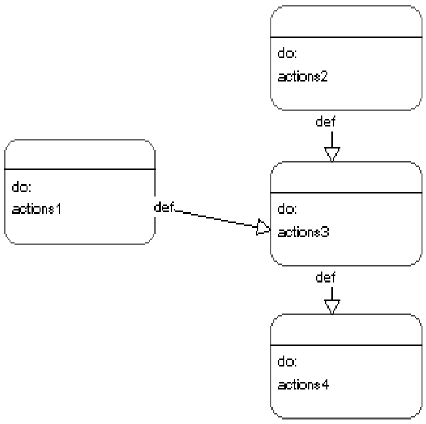

FIGURE 6. A state diagram in which each state has one action

In addition to combining, the algorithm has to be able to split states in order to separate actions into several states. For example, a state diagram in figure 7 could be constructed from two event traces:

(action1,default)
(action3, default)
(action4, null)

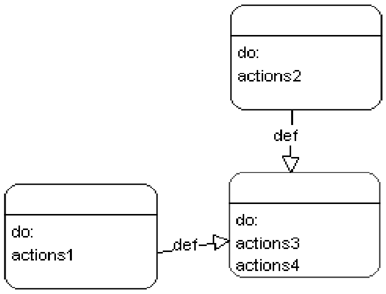

FIGURE 7. A state diagram in which each state has as many actions as possible

and

(action2,default)
(action3, default)
(action4, null)

Let's consider a situation where the second event trace is synthesized after the first one. The synthesis of the first event trace would gather actions "action1", "action3" and "action4" into a single state. While synthesizing the second event trace, the algorithm should notice that the maximal common path is:

(action3, default)
(action4, null).

Since we require that each state may have only either an automatic transition or labeled transition as a leaving transition but not both, two states with several actions can be joined only if both of them have same last actions (in the same order). These last common actions will be placed into a single state (and perhaps some actions will be removed from already existing states).

Gathering as many actions as possible into a single state prevents the algorithm from forming automatic self-transitions, or more generally endless loops caused by automatic transitions. That will be the case if there are two similar leaving events or action boxes in a scenario without any arriving event between them. This is illustrated in figures 8 and 9. Both of these state diagrams could be synthesized from the following two event traces.

(actions2, event1)
(action1, default)
(action1, null)

(actions2, event2)
(action3, default)
(action4, default)
(action3, null)

Note that when a state has several actions, these actions are executed in succession. In OMT notation, however, actions associated with the keyword "do" are executed simultaneously (or independently) and the actions executed in succession are separated with a semicolon.

4.1.4. *Names of transitions and actions.* The names of transitions and actions in a state diagram are directly the names of events attached to an object in a scenario [13, 11]. To simplify the presentation the specifications of receiver and sender objects have been ignored. This is due to an assumption that the event names uniquely

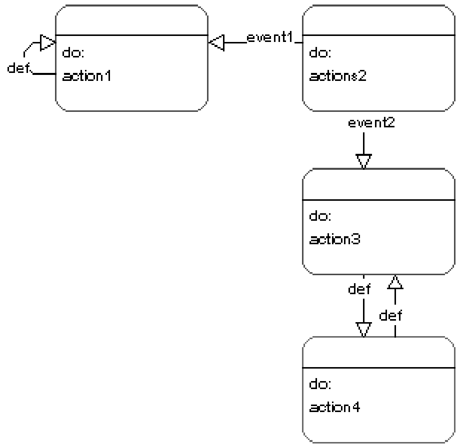

FIGURE 8. A state diagram with two endless loops

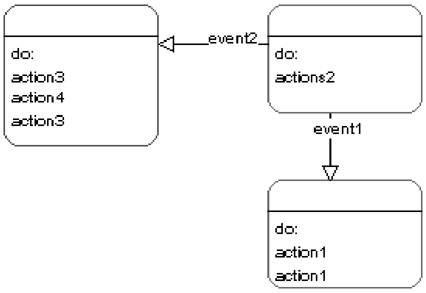

FIGURE 9. A state diagram for event traces above with no endless loop

determine receiver and sender objects. However, this may not always be the case. So, to be more specific without losing the simplicity of the presentation, the names of receiver(sender) objects will be added to the names of actions(transitions) only if receiver(sender) objects are not uniquely determined by the corresponding event names in scenarios. The notation could be the following:

action **to** object1
event **from** object2

For example, when an object is sending an event (say, action1) to several objects (say, object1 and object2) concurrently, the names of receiver objects will be added to the names of actions in a state diagram. Note that in this case actions

action1 **to** object1
action1 **to** object2

are executed without an interruption by any event and hence they are gathered into a single state. Because there is no order between such events in a scenario, all possible orders between corresponding actions are equally good. Hence, these actions can be written in any order inside states of a state diagram.

4.2. Initial and final states. OMT state diagrams can represent one-shot life cycles or continuous loops. One-shot diagrams have initial and final states, because they represent objects with finite lives [19]. Defining initial and final states should also be possible to do in **SCED**. Because state diagrams are generated incrementally from scenarios depending on the order scenarios are opened or created, we have no way to conclude which are the right initial and final states. Hence, defining initial and final states is left to the user. However, initial states of superstates can be defined automatically (see section 4.4).

4.3. Reducing the number of states and transition by combining. In this section we concentrate on internal, entry, exit, and transition actions. These concepts will be added to **SCED** state diagrams in order to make state diagrams more compact and semantically reasonable. The basic idea is to attach actions of states to other states or transitions so that relations between actions and events which cause them could be more easily seen. Combining information will also stress similar behaviour of different paths through a state diagram. Placing actions to a new location gives us the opportunity to remove some states and transitions. This will be done so that the the number of removed states and transitions will be maximized.

In some cases, there may be several, alternative ways to make state diagrams simpler by showing them in terms of these concepts. Figure 10 shows a state diagram which could be simplified for example by forming an exit action (figure 11) or a transition action (figure 12) [19]. For such cases we define priority orders between these notations.

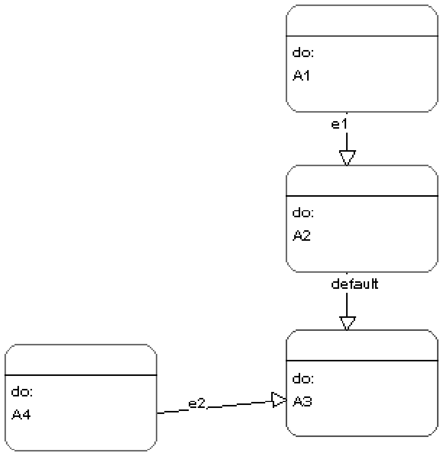

FIGURE 10. A state diagram generated by **SCED**

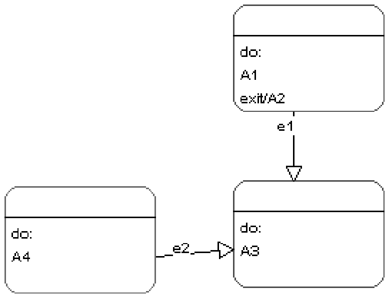

FIGURE 11. A state diagram with an exit action

Our principle is to make state diagrams as readable as possible. The information associated with a state should be easily seen. For these reasons we attach as much information to states as possible. If entry, exit, or internal actions are used, all

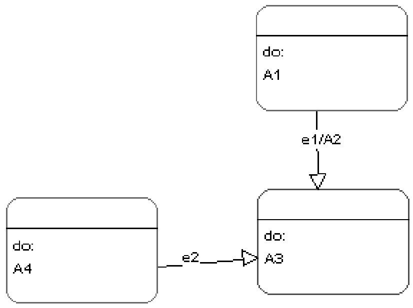

FIGURE 12. A state diagram with a transition action

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 doesn't cause a state change. This results in the conclusion that internal actions, entry actions and exit actions have higher priority than transition actions.

The behaviour of internal actions differs from the behaviour of entry or exit actions: the execution of an internal action depends on a single leaving transition while the execution of entry and exit actions depends on all entering/leaving transitions. So, an internal action can be seen to contain more specific information than entry/exit actions. It also reduces one transition more than entry or exit actions. It should be noticed that entry and exit actions can be formed for a state even if it already has internal actions. In every internal action of a state, both the event parts and the action part are different. This is true, since state diagrams are deterministic and have minimal number of states. Hence, we have decided to give the highest priority to internal actions.

We will see later (in section 4.3.2), that there can be no situation where entry and exit actions compete. So, we do not need to define any priority order between them. Hence, the final priority order is:

1) internal actions
2) entry actions and exit actions
3) transition actions

4.3.1. *Internal actions.* For cases where parts of a state diagram could be simplified in several ways by adding alternative OMT notations to it, we have given the highest priority to internal actions. Hence, internal actions will be associated to states whenever it's possible. In **SCED** we aim to detect these cases automatically during the synthesis. For creating an example of an internal action let's consider the following event trace:

(action1, event1)
(action2, default)
(action1, null)

The pure synthesis algorithm will give us the state diagram in figure 13.

For simplicity, we will use names "state 1" and "state 2" meaning states with actions "actions1" and "actions2", respectively. The information of the state diagram in figure 13 could be expressed in a following way: when the object is in state 1 and "event1" occurs, "actions2" are executed followed by the execution of "actions1". In this case after entering state 1 there is no reason to change states at all. Using an internal action the state 2 and transitions attached to it can be removed ending up to a state diagram described in figure 14.

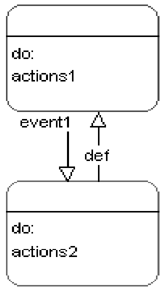

FIGURE 13. State diagram 1

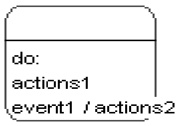

FIGURE 14. A state diagram with an internal action

Rules for attaching internal actions to states of an arbitrary state diagram. Next we examine a general case and decide when it is possible to form an internal action. First, we notice that an internal action can be formed only if in any simple state diagram there are two states (say, state 1 and state 2) and a normal labeled transition leaving state 1 and entering state 2 and an automatic transition leaving state 2 and entering the state 1. Without loss of generality, we can assume that the state diagram in figure 13 is a subgraph of such a general state diagram. Now, state 1 can not have an automatic transition as a leaving transition (see 4.1). Further, no other leaving transition of state 1 can be labeled "event1". Obviously, entering transitions have no effect on forming an internal action. This concludes that an internal action "event1/actions2" can be formed for state 1 despite of any other transitions attached to it. By the definition of an internal action we can say more generally: internal actions can be formed regardless of any entry actions, exit actions, or transitions (in addition to "event1") attached to state 1. Next we notice that there can not be any other transitions leaving state 2 except the automatic transition. Further, if there are more transitions leaving state 1 and entering state 2, they all form internal actions inside state 1. Finally, the only requirement for forming an internal action is: all transitions entering state 2 have to leave state 1.

Note that in our notation an internal action causes also performing the "do -actions" of the state. In [19] Rumbaugh does not address whether this is the right interpretation for internal actions or not.

4.3.2. *Entry and exit actions.* Next we need to find rules for generating entry and exit actions automatically. Because of the priority order, we exclude possibilities where an internal action can be formed.

Let's begin by examing a simple example of an event trace:

(actions1, event1)
(actions2, default)
(actions3, null)

The pure synthesis algorithm will give us the state diagram shown in figure 15.

For simplicity, we use the names "state 1", "state 2" and "state 3" meaning states with actions "actions1", "actions2" and "actions3", respectively. In section 4.1 we introduced a way to pack state diagrams by gathering actions to a single state when

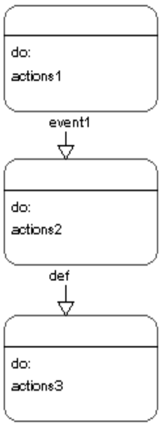

FIGURE 15. State diagram 2

possible. After packing, the state diagram in figure 15 can exist only if state 3 has at least one entering transition in addition to the shown automatic transition. We assume that this is the case.

The information of the state diagram in figure 15 is: when object is in state 1 and "event1" occurs, "actions1" and "actions2" are performed in this order and the object ends up in state 3. To simplify the state diagram we can remove state 2 and the automatic transition, change transition "event1" to enter state 3, and place an exit action "exit/action2" to state 1 (figure 16). Depending on the extra transition entering state 3, we might be able to place an entry action "entry/actions2" to state 3 (figure 17), as an alternative to the exit action "exit/actions2".

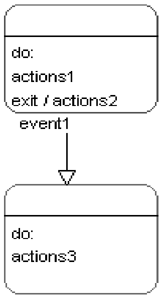

FIGURE 16. A state diagram with an exit action

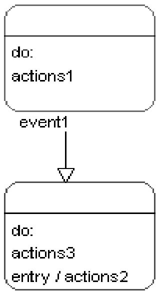

FIGURE 17. A state diagram with an entry action

Next we define the rules for forming entry and exit actions for an arbitrary state diagram. Generally, entry and exit actions can be formed only if in any state diagram there are at least two states (say, state 1 and state 2), a normal labeled transition leaving state 1 and entering state 2, and an automatic transition leaving state 2 and either entering state 1 or a third state (say, state 3). The automatic transition cannot enter state 2, since that would cause endless execution of "actions2" and that would have been prevented by gathering "actions2" to state 2 as many times as they appear in event traces. The two possible cases are shown in figures 13 and 15. From now on these cases are called *state diagram 1* and *state diagram 2*, respectively.

Without loss of generality, we can assume that either the state diagram 1 or the state diagram 2 is a subgraph of any general state diagram with possibilities to form entry or exit actions.

Rules for forming entry actions. First we examine the requirements for forming an entry action. We will do this in two parts:

<u>case 1:</u> State diagram 1 is a subgraph of a general state diagram.

All transitions entering state 2 (note that there can not be any leaving transitions other than an automatic transition attached to state 2) result in the execution of "actions1" and so entering the state 1. This case does not prevent us from forming an entry action as long as we take care that all transitions entering state 2 are changed to enter state 1. So, we need to concentrate on transitions attached to state 1.

There can not be automatic transitions entering state 1 and leaving a state with actions "actions2" other than state 2, since the state diagram is minimal (algorithm would have merged these states). All other transitions entering state 1 prevent us from forming an entry action. Obviously, transitions leaving state 1 and entering a state other than state 2 have no effect on forming an entry action. Hence, the only requirement for forming an entry action is: there cannot be any transition entering state 1 other than the automatic transition which leaves state 2. While forming an entry action we have to change all transitions entering state 2 to enter state 1.

<u>case 2:</u> State diagram 2 is a subgraph of a general state diagram.

First, we should remember that states 2 and 3 can be separated only if there is an extra transition entering state 3. On the other hand, such an entering transition would prevent the algorithm from forming an entry action, because even if it would be an automatic transition, it has to be leaving a state with actions other than "actions2" (see case 1). Hence, no entry actions can be formed in this case.

Rules for forming exit actions. Again we divide the examination into two parts:

<u>case 1:</u> State diagram 1 is a subgraph of a general state diagram.

Because of the priority order, all internal actions are already formed. So, if all transitions entering state 2 had left state 1, internal actions (at least one) would have been attached to state 1 instead of exit actions. Hence, we can assume that there has to be at least one transition leaving a state other than state 1 and entering state 2. If we wish to remove state 2 and form an exit action, we will have to form exit actions to all the states these transition are leaving from (if possible). That would not really decrease the size of the general state diagram nor would it clear the notation, since only one transition can be removed (the automatic transtion) and several (at least two) exit actions has to be formed. Hence, in case 1 exit actions won't be formed.

<u>case 2:</u> State diagram 2 is a subgraph of a general state diagram.

There can only be entering transitions attached to state 2. For the same reasons as mentioned in case 1, we decide not to form an exit action if there are such transitions and at least one of them is leaving a state other than state 1. Obviously, transitions attached to state 3 have no effect on forming an exit action and neither do the transitions entering state 1. Finally, transitions leaving state 1 and not entering state 2 prevent the algorithm from forming an exit action if at least one of those transitions enters a state with actions other than "actions2". Hence, the requirements for forming an exit action for state 1 are: all transitions leaving state 1 have to be entering a state with actions "actions2" and all transitions entering state 2 have to leave state 1.

While attaching an exit action to a state, necessary changes have to be done to all paths leading out of this state; transitions which were leaving removed states have to be changed to leave this state.

The competition between entry and exit actions. Entry and exit actions can compete only if actions of a removed state could be placed both as entry actions and as exit actions. If state diagram 1 is considered, "actions2" can only be placed as an entry action. Correspondingly, in state diagram 2 an exit action is the only possibility (see sections 4.3.2 and 4.3.2). Hence, there won't be any competition between entry and exit actions in any case. This concludes that no priority order between them is needed.

4.3.3. *Transition actions.* According to the priority order, attaching actions to a transition requires that no internal, entry, or exit action can be formed. This limits considerably the amount of transition actions in a final refined state diagram.

Rules for forming transition actions for states of an arbitrary state diagram. Transition actions can be formed only if in any state diagram there are at least two states (say, state 1 and state 2) and a normal, labeled transition which leaves state 1 and enters state 2 and an automatic transition which leaves state 2 and either enters state 1 or a third state (say, state 3). The automatic transition cannot enter state 2, since that would cause endless execution of "actions2". These two possibilities are exactly the same as used when finding general rules for forming entry and exit actions. This is natural because of the equivalency (mentioned above) between entry actions (or exit actions) and transition actions. Hence, we can use state diagrams presented in figures 13 (state diagram 1) and 15 (state diagram 2) to detect the general rules for forming transition actions. The examination is divided into two parts like before.

<u>case 1</u>. State diagram 1 is a subgraph of a general state diagram.

Obviously, transitions which are attached to state 1 but not to state 2 have no effect on forming a transition action. In addition, there can not be other leaving transitions attached to state 2 than the automatic transition (see section 4.1). If all entering transitions attached to state 2 left state 1, internal actions would be formed instead of transition actions, because of the priority order between them. Hence, the only possibility is that there are entering transitions attached to state 2 which leave a state other than state 1. If we removed state 2 and attached actions "actions2" to all the transitions which enter state 2 (there are at least two such transitions), the presentation of the resulting state diagram would not be much simpler nor more readable; only one state and one transition would be removed but "actions2" would be attached to at least two transitions. Moreover, that would be against our principle to combine information. Hence, in this a case no transition action will be formed.

<u>case 2.</u> State diagram 2 is a subgraph of a general state diagram.

First we notice that transitions attached to state 1 or state 3 have no effect on forming a transition action if they are not attached to state 2. Remembering that there can not be leaving transitions attached to state 2 other than the automatic transition shown in the figure 15, we only need to concentrate on transitions entering state 2. If there are transitions entering state 2 other than "event1", no transition action is formed because it is not convenient to attach "actions2" to all these transitions (see case 1) and hence separate the information which is already combined in state 2. If "event1" is the only entering transition attached to state 2 and no exit action can be formed for state 1 (i.e. there is at least one transition leaving state 1 and entering a state with actions other than "actions2"), the transition action will be formed and state 2 will be removed. Hence, if all exit actions are formed, the only requirement for forming a transition action is: "event1" has to be the only entering transition attached to state 2.

4.3.4. *Results.* The number of states and transitions could be reduced by combining if at least one of the state diagrams in figures 13 (state diagram 1) and 15 (state diagram 2) were found as a subdiagram in the desired state diagram. The states in state diagram 1 are called "state 1" and "state 2" in a top down order. Correspondingly, the states of state diagram 2 are called "state 1", "state 2", and "state 3". All such subdiagrams are handled separately. The desired state diagram can be modified by forming internal, entry, exit, or transition actions. The priority order between these OMT concepts is:

1) internal actions
2) entry actions and exit actions
3) transition actions

The rules for forming internal actions:

State diagram 1: All transitions entering state 2 have to leave state 1.
State diagram 2: Internal actions won't be formed.

The rules for forming entry actions:

State diagram 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.
State diagram 2: Entry actions won't be formed.

The rules for forming exit actions:

State diagram 1: Exit actions won't be formed.
State diagram 2: All transitions leaving state 1 have to be entering a state with actions "actions2" and all transitions entering state 2 have to leave state 1.

While attaching an exit action to a state, transitions which were leaving a removed state must be changed to leave this state.

The rules for forming transition actions:

State diagram 1: Transition actions won't be formed.
State diagram 2: "event1" has to be the only entering transition attached to state 2.

4.4. State generalization. In OMT states may have substates that inherit the transitions of their superstates. Further, any transition or action that applies to a state applies to all its substates, unless overridden by an equivalent transition on the substate [19].

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. However, some restrictions have to be defined. In **SCED** state diagrams all actions are considered instantaneous (duration of an action is insignificant compared to the resolution of the state diagram). Moreover, several "do -actions" in a state are executed in succession (see section 4.1.3). Hence, in **SCED** superstates cannot have continuous activities which could be overriden by actions of substates. Although a superstate may not have any "do -actions" it may have an exit action: common exit actions of substates should be replaced with one exit action of the superstate.

Superstates can be formed if in a state diagram there are several (at least two) 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. Figures 18 and 19 show state diagrams for the same state machine. The transitions with label "x" in figure 18 are replaced with one leaving transition of the superstate in figure 19. The number of needed transitions decreases from five to three.

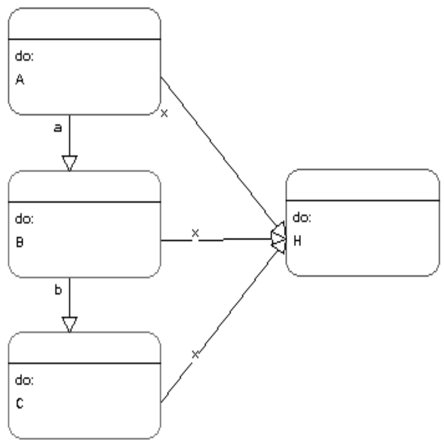

FIGURE 18. A flat state diagram

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 can not make difference between these possibilities on the basis of semantics. Hence, it is our aim to find such a combination of superstates which gives the maximal reduction in the number of transitions needed. Let's consider the example of a "state diagram matrix" in table 1.

This matrix corresponds to a state diagram in a following way: **A**, **B**, **C**, **D**, **E**, **H**, and **K** are states and **a**, **b**, **c** and **d** are transitions leaving a state in the same row and entering a state in the same column. For example, there is a transition labeled **a** which leaves state **B** and enters state **H**.

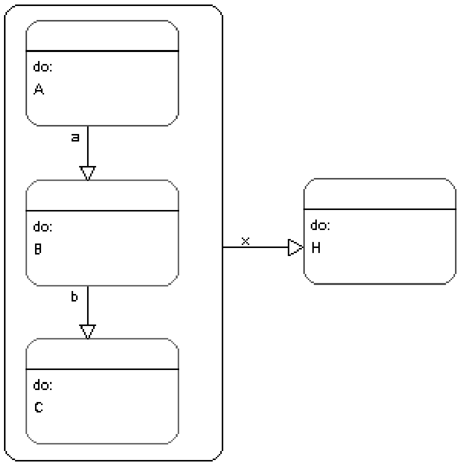

FIGURE 19. A state diagram with one superstate

	H	B	A	K
A		b		d
B	a			
C	a	b	c	
D	a		c	d
E			c	d

TABLE 1.

Generally, in state diagram matrices we show only those states and transitions which are interesting while forming superstates. Since each superstate has to include at least two substates, every column in a state diagram matrix contains at least two transitions.

If we demand that all states, which have similarly labeled leaving transitions which enter the same state, have to be associated as substates of the same superstate, there will be only four different possibilities to form a superstate:

1) states **B**, **C** and **D** (because of the transitions with label **a**)
2) states **A** and **C** (because of the transitions with label **b**)
3) states **C**, **D** and **E** (because of the transitions with label **c**)
4) states **A**, **D** and **E** (because of the transitions with label **d**)

All these possibilities are mutually contradictional. If we choose one of the possibilities 1, 2 and 4, the number of transitions needed will be reduced by two. The second possibility reduces the amount of transitions needed by one. So, the possibilities 1, 2 and 4 are equally good. Yet, these are not the optimal solutions. The maximal reduction can be obtained e.g. by forming first superstate for states **B** and **C**, second superstate for states **A**, **D** and **E** and third superstate for states **D** and **E**. In this case the reduction in the number of transitions is (2-1)+(3-1)+(2-1)=4.

For generating the total number of superstate candidates in an arbitrary state diagram we concentrate on states and transitions shown in the state diagram matrix. Let m be the number of different labels on transitions. Further, i is used as an index for these labels. Each superstate candidate must contain at least two and at most n_i substates, where n_i is the number of transitions with ith label. It is easy to notice that there can be

$$\binom{n_i}{2} + \cdots + \binom{n_i}{n_i} = \sum_{k=2}^{n_i} \binom{n_i}{k}$$

different superstate candidates with label index i. Hence, the total amount of superstate candidates in a state diagram is

$$\sum_{i=1}^{l} \sum_{k=2}^{n_i} \binom{n_i}{k},$$

where $l \leq m$. Note that there can be several possibilities to form a superstate with exactly the same substates. In fact, in that case only one superstate will be formed but it will have several leaving transitions. For computational reasons we regard them as different cases; each superstate has exactly one leaving transition. That can be done, since in both cases the total number of transitions reduced will be the same.

In a state diagram there can be looping transitions which leave the same state they enter. For example in the state diagram matrix in table 2, one superstate candidate could contain states **A**, **B**, and **C** as substates and **e** as a transition.

	A
A	e
B	e
C	e

TABLE 2.

In this case the leaving transition **e** of the superstate enters one of the substates. When this transition is drawn from the superstate contour to the state A, it entirely lies inside the superstate (see figure 20).

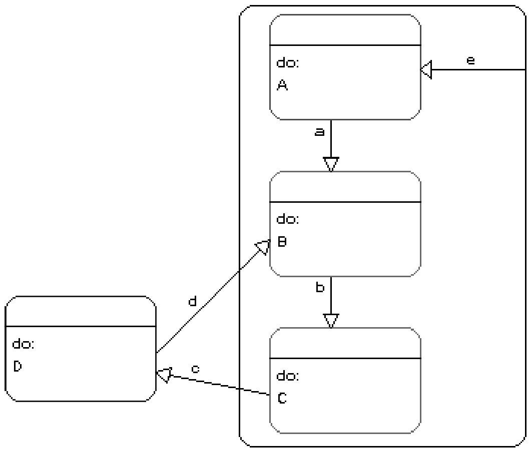

FIGURE 20. A state diagram with one superstate

The initial state inside a superstate will be the one with most entering transitions leaving states which lie outside the superstate. If there are several possibilities for an initial state, any one of them can be chosen. Superstates cannot include a final state, since the definition of a superstate demands that there is always a transition leading out from any of its substates.

4.4.1. *NP-completeness of finding an optimal combination of superstates.* In what follows we assume that the reader is familiar with the rudiments of the theory of NP-completeness as given e.g. in [6, 3]. Consider now the following problem called SUPERSTATES:

<u>SUPERSTATES</u>:
INSTANCE: An arbitrary state diagram D, positive integer M.
QUESTION: Is there a combination of superstates (on the basis of similarly labeled transitions) in D so that no two superstates are mutually contradictional and the number of transitions reduced is at least M?

Next we introduce some terms which will be used later. The number of rows in the state diagram matrix is denoted by V and the number of transitions, i.e. non-empty items in the state diagram matrix, is denoted by T. Finally, the number of different labels on transitions is denoted m.

The real problem, however, is to find the best possible combination of superstates, i.e. the combination that gives the maximal reduction of transitions. We call this problem MAXIMAL SUPERSTATES. If we could solve SUPERSTATES efficiently we could also solve MAXIMAL SUPERSTATES efficiently. Also, if SUPERSTATES is shown to be hard to solve then MAXIMAL SUPERSTATES must be hard to solve [6, 3].

First we show that SUPERSTATES $\in$ NP. This can be done by showing that a nondeterministic algorithm need only guess a combination of superstates and check in polynomial time that no two candidates are contradictional. Each superstate candidate i has a single transition label and a set of substates S_i so that $\forall S_i : |S_i| < V$. The nondeterministic algorithm checks if the first substates of both candidates are the same. If so, the algorithm needs at most $(|S_i| - 1)(|S_j| - 1)$ comparisons to check if one set of substates is a subset of the other and one comparison to check if the transition labels differ. If the first substates of both candidates differ, the algorithm needs at most $(|S_i| - 1)(|S_j| - 1)$ comparisons to check if the sets of substates are mutually disjoint. Each guess have at most m $(\leq V)$ candidates for superstates, since each label may appear at most once; condition 2 demands that labels are mutually different and if condition 1 holds and labels of two superstates are the same, we can always replace them with one larger superstate. Hence, there is a nondeterministic algorithm which is able to check in polynomial time that no two of the guessed superstate candidates are contradictional.

The following SET PACKING problem is shown to be NP-complete [6]:

<u>SET PACKING</u>:
INSTANCE: Collection C of finite sets, positive integer $K \leq |C|$.
QUESTION: Does C contain at least K mutually disjoint sets?

An instance of SUPERSTATES consists of a state diagram and an integer M, where M gives the lower bound to the number of transitions reduced and can be formulated in a following way:

$$M = \sum_{i=1}^{k}(|S_i| - 1),$$

where $k \leq m$ and each S_i is a set of substates in a superstate candidate. If we allowed only mutually disjoint sets of substates (condition 1) in SUPERSTATES, we would get a similar problem to SET PACKING; integer K would equal to k giving the number of superstate candidates. So, SUPERSTATES can be restricted to SET PACKING by allowing only mutually disjoint sets of substates (condition 1). Hence, SUPERSTATES is NP-complete and so also the larger problem MAXIMAL SUPERSTATES is NP-complete.

4.4.2. *Principles of forming superstates in* **SCED**. Advanced OMT notation will be added to the state diagrams after the synthesis. It is desirable that it could be done quite quickly, specially when the synthesis algorithm has exponential behaviour itself. Internal actions, entry actions, exit action, and transition actions can be formed relatively quickly, because each state and transitions attached to it need to be checked at most once. However, because of the NP-completeness, the problem of finding the optimal superstate combination has much higher time requirements; the number of superstate candidates increases quickly when when the number of states with similarly labeled leaving transitions increases. That encouraged us to give up the goal of finding the best possible superstate combinations. Instead, our aim is to find guidelines for getting relatively good combination of superstates which is also semantically reasonable; the number of transitions which enter/leave directly the substates inside a superstate, should be small with respect to the number of transitions reduced. Some heuristical assumptions has to be made in order to follow these guidelines and to avoid the use of computationally complex algorithms and large search spaces.

Let's take a closer look at the state diagram matrix by considering the next example shown in table 3.

	H	I	J	K	L	M
A		a				z
B		a				z
C	x	a	b	c	y	
D		a	b	c		z
E	x				y	

TABLE 3.

Transitions which enter/leave directly the substates inside a superstate will be called *critical transitions* from now on. Further, transitions which are removed by forming a superstate, are said to be *replaced by a superstate transition*.

If we formed two superstates, one for states **A**, **B**, and **D**, and the other one for states **C** and **E**, the number of transitions needed would be decreased from 15 to 9 and the number of critical transitions would be five (transitions **b** and **c** which leave state **D**, and transitions **a**, **b**, and **c** which leave state **C**). The critical leaving transitions in a state diagram matrix are those which won't be replaced by any superstate transition but which lie at the same row as some of the replaced transitions. The number of critical, leaving transitions is relatively small, if all the rows have either no replaced transitions or many replaced transitions but few other transitions (these other transitions would be critical ones). In other words, rows which have as much common as possible potentially form superstates with few critical, leaving transitions. Correspondingly, the number of critical, entering transitions is relatively small, if all the columns have either no replaced transitions or many replaced transitions but few other transitions. However, the situation is more complicated if there are many states which label both a column and a row. Further, states and transition which do not appear at the state diagram matrix at all, complicate the situation as well.

In our example, rows labeled by states **C** and **D** have three transitions in common: **a**, **b**, and **c**. Correspondingly, states **A** and **B** have two transition in common: **a** and **z**. If we formed these two superstates, the number of transitions needed would be 10 but the number of critical transitions would be only three.

While forming superstates it shoud be aspired to find a combination with a small number of critical transitions.

4.5. Concurrency. Information on concurrency may be attached to the dynamic model in OMT. This may occur in two different ways: concurrency between set of objects or within the state of a single object. In the former case (aggregation concurrency) objects are inherently concurrent, each with its own state and state diagram, being able to change states independently. In the latter case the state of the object comprises one state from each subdiagram while the subdiagrams need not be independent; the same event can cause a transition in more than one subdiagram [19]. **SCED** state diagrams are generated on the basis of sequential information of scenarios. This results in a state diagram in which the state of an object is defined either by a single action or by several actions which are executed in succession. Specially, the object is always in a single state. Hence, generating concurrent subdiagrams automatically on the basis of scenarios requires a way to compound events in scenarios. This implies that extensions to scenario notation for showing compound information are needed. The aggregation concurrency, for instance, requires compounding scenarios. The problem of applying concurrency notation to **SCED** state diagrams has not been solved yet.

5. Automatic layout of state diagrams

State diagrams in **SCED** will be visualized using the Harel [8] notation for statecharts. States are displayed as rectangular boxes and the labeled transition arcs are displayed as orthogonal polyline segments.

The automatic state diagram layout facility of **SCED** will be designed and implemented so that the user can choose the layout method from several different algorithms that are implemented in **SCED**.

The first implementation for a state diagram layout algorithm for **SCED** will be an adaptation of an algorithm developed by Nummenmaa [15] for automatic graph layout. The implementation of this algorithm — as applied to the layout of ER-diagrams — has been described in [16].

The state diagram is internally represented as a directed — in some parts of the algorithm as an undirected — graph where states are the vertices of the graph and transitions are (labeled) edges of the graph.

The layout algorithm is linear-time in the number of vertices for planar graphs. What makes the algorithm especially useful for state diagram layout is its capability to handle arbitrary sized vertices. While the states in the diagram usually are of small size, there are no fixed limits to the width or height of the box representing the state, as the state diagram synthesis phase may collapse several states into one, add internal actions to the state, and other similar actions. These kinds of combined states may have several lines of text in them. A state may also be a superstate of arbitrary size, consisting of several states and actions. Superstate/substate hierarchy is potentially recursive to arbitrary depth.

5.1. Constructing the initial layout. We will shortly describe the phases of the layout algorithm here:

(1) Planarity testing and finding a topological embedding.
The Hopcroft-Tarjan algorithm [9] will be used to check for the planarity of the input graph and a topological embedding specifying the circular order of edges around each vertex is constructed. A crossing in a non-planar graph is handled by adding a dummy vertex to the crossing of two edges and splitting both edges in two.

(2) Maximalizing (triangulating) the graph.
New edges will be added to the graph so that the boundary of every face of the graph is a triangle and adding one more edge would make the graph to be a non-planar one. A similar method is used as the one described by Read [18].
(3) Numbering the vertices.
A *canonical numbering* [5, 15] will be constructed — which is also an *st*-numbering [4] — for the vertices of the graph. Canonical numbering starts with a cycle u, v, w in a maximal planar graph G. These vertices are numbered as $u = v_1, v = v_2, w = v_n$ and the following holds true for each vertex v_k for $3 \leq k < n$:
   (a) The subgraph $G_k \subset G$ induced by $v_1, v_2, \ldots, v_k$ is 2-connected, and the boundary of its exterior face is a cycle C_k containing the edge (v_1, v_2), and
   (b) v_k is in the exterior face of G_k and it has at least two neighbours in G_{k-1}, and these neighbours are consecutive on the path $C_{k-1} - (v_1, v_2)$.
(4) Calculating space requirements for vertices.
The width and height of the rectangular area needed for the visual representation of each vertex is calculated. For state diagrams, we use a certain minimum area, which is large enough for common cases, but if the state contains many lines of text, we need to adjust the area requirements accordingly.
(5) Creating a *visibility representation*.
A visibility representation [17] maps each vertex to a horizontal line segment and each edge to a vertical line segment. These constraints must hold true for a visibility representation:
   (a) no segments of two vertices intersect,
   (b) the segments of edges only intersect at their endpoints, and
   (c) the segment of each edge (u, v) intersects the segments of u and v, but no segments of other vertices.
A visibility representation is built for the graph starting with vertex v_1 at the bottom and assigning successive y-coordinates to successive vertices (as defined by the canonical numbering). Each vertex will be placed so that the edges that are connecting it to the lower numbered vertices can be drawn as vertical lines.
(6) Transforming the visibility representation to an ER-diagram.
The mapping to an ER-diagram (or a state diagram) from the visibility representation is straightforward. The y-coordinate for the box representing a vertex is the line segment's y-coordinate, the x-coordinate is assigned to the middle of the line segment. The x-coordinate for an edge is straight from the visibility representation with possible extra line segments added to connect the edge to the vertex box.

Figure 21 contains an example diagram produced by the layout algorithm. The diagram is drawn in a mock state diagram style; the transition names are not drawn, and state names are somewhat meaningless. However, it illustrates the general outlook of the resulting state diagram. Vertex boxes also contain the canonical number of the vertex.

5.2. Improving the initial layout. The basic algorithm does very little in the way of optimizing the result layout wrt alignment, symmetry, balance or bend minimization or other aesthetic criteria. Some packing in the vertical direction is already done for the vertices when the visibility representation clearly allows moving the vertex downward without overlapping the line segment of the vertex below.

We will add a few processing steps which will manipulate the internal visibility rep-

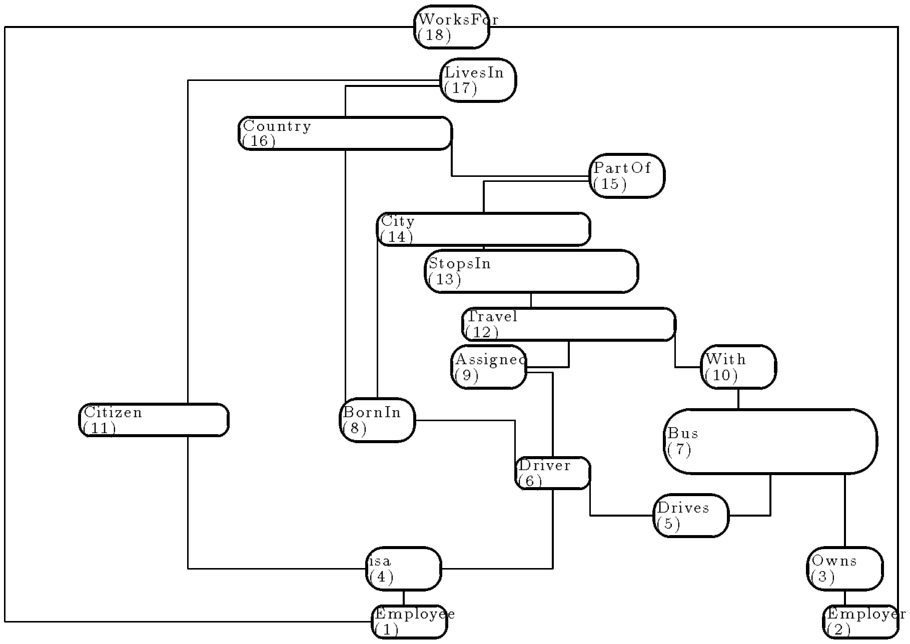

FIGURE 21. A state diagram example

resentation the diagram to improve the layout in various ways. Most of these improvements can also be directly applied to the layout of ER-diagrams, even though the possible attributes attached to the entity- and relationship-vertices will add further complications to the placement of connecting edges.

The improvements will be done in following phases:

(1) Identifying the true visibility of vertices.
Length of the horizontal segment along which a vertex can be moved without obstructing those edge segments which are not connected to the vertex will be identified. As the initial layout algorithm also processes the dummy edges added during maximalization, the "visible" area for the vertices is often smaller than it could be in reality, even though space is not reserved for dummy edges in the layout.
Widening the visibility area of vertices will add more freedom to later vertex position adjustments and will give new opportunities for further improvements to the layout.
(2) Horizontal centering of vertices.
The vertices will be placed (approximately) in the midpoint of the edges that are connected to it. If the number of edges is odd, the vertex will be placed on the same x-coordinate as the middle edge.
(3) Vertical alignment of vertices and edges.
The general goal is to align connected vertices which are on successive vertical levels so that the vertices and their connecting edges are on the same x-coordinate, i.e. they are aligned on the same vertical line.
The horizontal movement of the vertices is restricted by other vertices on the same y-level and we will avoid creating new bends in the connecting edges.
(4) Vertical packing of vertices.
In this step we will move the vertices downward as much as possible while taking into account the constraint of not introducing new bends in the edges. This process will naturally align connected vertices horizontally.

(5) Final bend elimination.
Placement of vertices will be adjusted so that the number of bends will be decreased. For each bend of an edge — note that each edge has at most two bends after the initial layout — we will check if moving either of the connected vertices to the place of the bend will not introduce new bends in other edges. This will reduce the total number of bends in all edges.

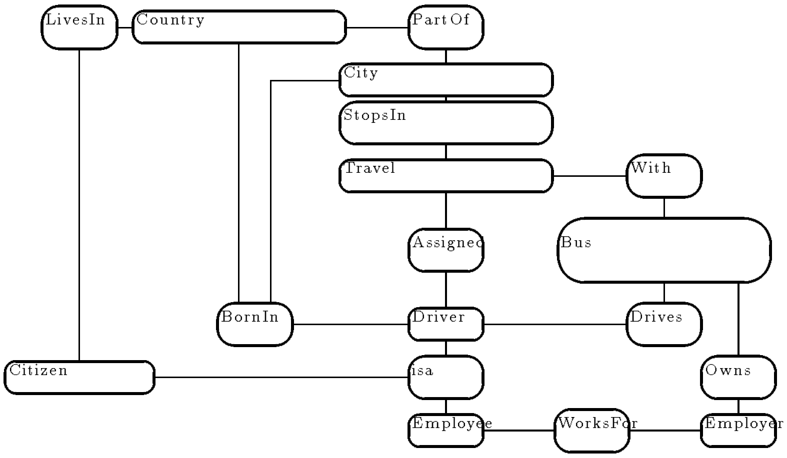

FIGURE 22. State diagram example after layout improvements (see also figure 21)

Figure 22 is our example state diagram after layout improvements. The result is a more compact diagram with considerably clearer structure than the diagram in figure 21, e.g. the total number of bends in the edges has been reduced from 15 to 1. Even the last bend could be eliminated algorithmically from this particular diagram, but we are currently planning to do only local optimizations to improve the diagram's layout.

Also, there are still other fine-tuning opportunities, e.g. adjusting vertex positions so that edge lengths will be more uniform. These will be evaluated more closely after we have the basic optimizations implemented and actual diagrams produced by the algorithm can be seen.

5.3. Placement of identifying texts. The placement of textual information inside the vertices is simple, as we have originally reserved the area of each vertex large enough to contain the necessary text.

Attaching labels (transition names) to edges is more complicated because of the potential overlap of the labels with other edges and vertices. This problem has not yet been analyzed in detail; some overlap is acceptable with edges but not with other labels and vertices, as long as it is easy to distinguish which edge a label is associated with. Probably the simplest method is to generate the layout without special considerations for edge label placement, and then assign coordinates for the edge labels, expanding the layout to accommodate the labels, if necessary.

5.4. Layout of superstates. The layout of superstates adds some interesting twists to placement of the edges (transitions) in the layout. Besides having transitions which enter (or leave) the enclosing superstate, i.e. they connect to the superstate box border in the actual layout, transitions can also enter/leave directly the substates inside the superstate.

In the internal graph representation, the superstates are in the same vertex space as are the substates. While constructing the layout for the diagram, the substate vertices are not actually "seen", i.e. the edges connecting to the substates are conceptually connecting to the superstate itself.

It appears better do defer the construction of the layout of the superstate contents until the enclosing diagram is fully known. Then the directions which the edges to the substates are coming from are known, and the substate placement can be done so that excessive edge crossings and bends are not introduced. Simple rotation/mirroring transformations of the superstate layout could also be used to reduce the number of edge crossings and bends.

The layout of the superstate contents is best done by a considerably simpler layout algorithm when the number of superstates is small. Producing truly good layouts for diagrams containing superstates and comparable hierarchical structures is clearly a non-trivial problem, to say the least. Similar layout problems are discussed in e.g. [14, 20].

5.5. Layout of user-drawn areas. With the state diagram editor, the user can modify at will the state diagram produced by the layout algorithm. It is desirable that after modifying the layout, e.g. to better display the inherent relationships in the state diagram, the user can fix certain areas and ask a new layout to be automatically generated for the rest.

The areas fixed by the user will be handled in a similar manner as superstate vertices, i.e. a fixed area will be seen as one large vertex while constructing the layout.

5.6. Incremental state diagram layout. SCED allows the user to synthesize new scenarios into an already synthesized state diagram. When the number of additional states and transitions generated by the newly synthesized scenario is relatively small when compared to those of the base state diagram, it is preferable that the overall layout of the diagram would not change very much. This can be accomplished by adding the new states and transitions so that the relative placement of the original states is not altered extensively. It is not yet clear what is the best way to do this, but the internal data structures for the state diagram contain enough information to find out where new states and transitions can be inserted.

When the number of added states is on the order of the number of original states, we might as well generate a new layout for the whole state diagram, instead of attempting to preserve the existing layout.

5.7. Handling non-planar graphs. During planarity testing dummy vertices will be added for each detected edge crossing of a non-planar graph. While allocating space for the vertices in the layout, the size of a vertex representing a crossing of edges will be zero. The layout algorithm must be careful to get decent looking crossing, because there are no absolute guarantees that the four edges — originally two before introducing the dummy vertex and splitting the edges — are perpendicular to their neighbouring edges.

5.8. Self-loops and multiple edges. The initial layout algorithm processes only simple graphs, i.e. graphs that do not contain multiple edges between any two vertices and self-loops. As these are perfectly valid constructs in a state diagram, and indeed quite common, they must be accepted and processed. Self-loops will be handled as additional information attached to a vertex which necessitates reserving additional space for the vertex in proportion to the number of self-loops. They will be drawn as circular, labelled arcs attached to the outer edge of the vertex.

Multiple edges between two nodes also means that some additional space must be reserved in the diagram. Only a single edge is present in the graph representation and the multiple edges will always be displayed parallel to each other in the state diagram.

5.9. Layout of cyclic state diagrams. The initial layout is greatly affected by the assignment of canonical numbers for the vertices and is upward growing by nature. As such it does not support directly the layout of cyclic structures. However, while applying the improvements outlined in section 5.2, the different choices of vertex movements can be prioritized so that we will progress towards better visibility of the main cycle of the state diagram. Currently we are investigating various strategies for better visualization of the main cycle.

6. Consistency between scenarios and state diagrams

When the user is able to edit both the scenarios and state diagrams the consistency between these two views becomes an important issue. If the user changes a scenario the system should either check that the state diagram still implements the new scenario or modify the state diagram to implement the new scenario. As we will see later both these goals can be met. Going the other way around, i.e. to automatically change scenarios as a result of changes in a state diagram, is a bit more complicated issue. At first we shall look into some of the problems.

The fundamental source of problems is the fact that state diagrams generalize scenarios. If they would not do that then the whole concept of generating state diagrams from scenarios would be a trivial problem but also quite useless for the user. Synthesizing algorithm needs to join states, and each time two states are joined some information is lost. Consider, for example, the scenario in figure 23 and the state diagram synthesized from it in figure 24. In the scenario there is a sequence `a1c2a3b4a` and we can see that the state diagram accepts this sequence. However, the state diagram accepts lots of other sequences as well and there is no way to tell what sequence is the "real" one specified in the scenario.

The expressive power of a state diagram is much higher than that of a scenario. A single transition in a state diagram may be the result of many events in a set of scenarios and if we consider a nested state diagram structure then one transition from a superstate could not be expressed with just one scenario. Instead, one should find the set of scenarios that form the substates, and then for each scenario in this set generate as many new scenarios as there are actions in the scenario. Even though this process could be automated, the number of resulting scenarios would be too high and the information content of those scenarios would be questionable.

Due to these two problems we do not even try to keep both scenarios and the state diagram complete, meaning that the other could be generated from the other. Instead, scenarios are consireded to be a set of assertions that must hold for the state diagram. This means that the user can always be sure that his scenarios can be run by the state diagram.

The algorithm used to synthesize scenarios into a state diagram is incremental, meaning that a new scenario can be generated into an existing state diagram. This makes it easy to add a new scenario into a state diagram, but the question is how to deal with a scenario that has already been included into the state diagram and then changed afterwards. If the user changes a scenario then we can either start from scratch and resynthesize all scenarios or we can first desynthesize the original scenario out of the state diagram and then resynthesize the changed scenario into it. The desynthesizing process works as follows: first execute each scenario and count how many times each state and transition is visited in the state diagram.

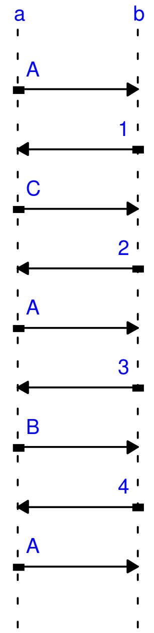

FIGURE 23. An example scenario

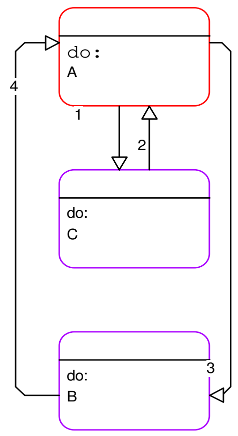

FIGURE 24. An example state diagram

Then re-execute the original scenario and decrement the count of each state and transition visited. If any of the counts went to zero then the state or edge is removed from the state diagram. One should note that executing a scenario is much faster than synthesizing it. In the current system the de-/resynthesizing process is not run automatically, instead the user must issue an explicit command to do it.

The user is free to edit the state diagram in any way he wants to. The only way an inconsistency could be created between a state diagram and the corresponding scenarios is by changing or deleting a state or a transition. Inserting a new state or a transition just makes the state diagram more general; it still can accept all scenarios. The system should have a set of support tools to deal with possible inconsistent situations. Perhaps the most useful tool is a multilevel undo operation, so that the user can back up his modifications. By executing the scenarios the system can tell which scenarios are no longer valid executions of the state diagram and at what point they differ from the state diagram. For example the contradicting scenarios could be resynthesized into the state diagram using a different color for new transitions and states or by highlighting one scenario at time up to the point where the scenario differs from the state diagram. The user can then correct the situation manually or accept the resynthesized scenarios again.

The synthesizing algorithm is eager to join states which may result to states that while being correct do not meet the user's intentions. The user should be able to split a joined state into two or more states in such a way that the result is consistent with the scenarios. For example, one scenario might enter a state s with an event e_0 and leave it with an event e_1 and some other scenario might enter the state s with e_2 and leave with e_3. Now if the user wants to split the state s then the system should prevent the situation where one of the splitted states is entered with e_0 but left with e_3 and the other entered with e_2 and left with e_1. When the user wants to split a state the system should execute each scenario and record the entry/exit event pairs for that state. The state can then be split into as many states as there are those pairs.

An equally important feature is to allow the user to join states with the same action in them into one state. This is much simpler than splitting states since the only thing that may go wrong is that the resulting state diagram would be undeterministic. The system should ensure that the state diagrams stay deterministic.

A major problem with editable state diagrams is how to preserve the user's changes. If, for example, the user has separated one state into two then the synthesizing algorithm should not rejoin those states again. It is not clear what is the best way to deal with these situations. Further, it is not clear what should be consireded as a change, the user might, for example, delete a state and then insert a new state that is identical to the deleted one. Is the synthesizing algorithm allowed to delete this state when it is not needed any more, or should it keep it because the user has manually inserted it?

7. Interfaces

The state diagram editor has an interface to almost every package in the SCED system. The editor should implement an interface to following packages or components of the system:

(1) Layout package,
(2) synthesizing package,
(3) project manager,
(4) wxWindow windowing package and
(5) to the rest of the SCED system.

We will take a more detailed look at each of these components.

7.1. Layout package. The layout package is responsible for assigning proper graphical information for each component in the state diagram. For states in the state diagram the graphical information means the x- and y-cooridinates, width and height of the state. For edges in the state diagram the graphical information means a list of straight line segments.

An automatic layout is performed either when the user explicitly requests it or when a newly synthesized state diagram is prepared for display.

7.2. Synthesizing package. The synthesizing package is used as a user support tool, giving for the user semantic help on the state diagram. Such help is to

(1) show how to split a state into separate states,
(2) show what scenarios are affected by some state or edge,
(3) show a route of a scenario through the state diagram and
(4) perform consistency check between scenarios and the state diagram.

The synthesizing package is used as a server of the editor package meaning that no call is made by it back to the editor package.

7.3. Project manager. The project manager package is responsible for loading and saving state diagrams (as well as scenarios) and keeping track of scenarios and state diagrams that belong to a project. The project manager provides means to open (show it in a scenario editor window) a scenario from the editor.

The interface functions to the project manager package can be found from file `project.h`.

7.4. wxWindows windowing package. The central point of control is the editor window. There is one instance of editor window for each state diagram. This window implement the user interface and responds to events and requests sent by the wxWindow windowing system.

The class structure of this part will be made similar to the structure found in the scenario editor. The interface functions and their prototypes are listed in the wxWindows reference manual.

7.5. The rest of the SCED system. Then there is some random stuff that needs to be taken care of. These are the OLE-support and dealing with menus, toolbar buttons and clipboard functions.

8. Editor classes

In this section we take a look at the classes that needs to be implemented.

8.1. Class SdeSubFrame. This class is used to implement the wxWindow specific functions. Its implementation is very similar to the existing class `ScSubFrame` found in scenario editor. This class is required because of the way windows are implemented in wxWindows package. All the common functionality with the class `ScSubFrame` is in class `SubFrame`.

Each instance of class `SdeSubFrame` holds one editor canvas instance of class `SdeCanvas`.

The class is inherited from class `SubFrame` which is inherted from class `wxFrame`.

8.2. Class SdeCanvas. This class implements the most of the state diagram editor. It responds to the events caused by the user's actions.

A great deal of functionality can be copied from the scenario editor class `ScCanvas`. These parts basically cover things that have something to do with user iterface. Semantic reasoning what is a legal or an illegal editing action should be implemented from scratch. Class `SdeCanvas` should be able to

(1) show states and edges (in a normal and in selected mode),
(2) recognize if the mouse cursor is over a state or an edge,
(3) move a state or an edge,
(4) insert or delete a state or an edge,
(5) change the text field or a state or an edge and
(6) implement interfaces described earlier to other packages.

`SdeCanvas` maintains a list of all visible objects (of type `SdePrimitive`) and a set of objects that have been selected by the user. Most of the operations operate on these two lists.

When the user changes or deletes a state or an edge from the state diagram the system checks if some scenarios would not be valid any more. If such scenarios are found then a warning message is displayed and the user may either proceed or take some actions to correct either the state diagram or invalid scenarios. When a scenario becomes invalid it is removed from the list of scenarios that have been used to synthesize this state diagram.

This class is inherited from class `Canvas` which is inherited from class `wxCanvas`.

8.3. Class SdePrimitive. This is an abstract base class for classes `State` and `Edge`. It is used to glue these two classes together, in order to simplify the handling of these classes, since as editable objects they act in the same way. Most of its methods are virtual the actual method being invoked either in class `State` or `Edge`.

There should be methods to

(1) paint the object into a canvas,
(2) check if a given coordinate pair is inside the boundary of the object.
(3) drag the object (composed from three separate methods),
(4) move the object and
(5) edit the text fields in the object.

This class is very similar to class `ScPrimitive`.

8.4. Modifications to class Edge. This class is inherited from class `SdePrimitive`. It basically provides the implementation to the methods in class `SdePrimitive`.

8.5. Modifications to class State. This class is inherited from class `SdePrimitive`. It basically provides the implementation to the methods in class `SdePrimitive`.

9. Prototypes for member functions

The following member functions are defined in the state diagram class `StateDiagram`.

9.1. Layout package.

`void layout(void)`: Performs automatic layout of the state diagram.
`void makeRoom(RECT rect)`: Makes empty space to the state diagram.

9.2. Synthesizing package.

`set<struct State *enter, State *leave> splitState(State *state)`: Returns how a state `state` could be splitted into separate states. Each item in the return set gives one pair of entering and leaving edges from the state.
`set<string> findScenarios(SdePrimitive *item)`: Returns a set of scenarios names that go through the `item`.
`void showTrace(ScScenario *scenario)`: Shows a trace of a scenario through the state diagram.
`int checkScenario(ScScenario *scenario)`: Checks if the scenario is consistent with the state diagram. Returns a negative value if it is ok, otherwise returns a number n meaning that nth item (from the top) in the scenario differs from the state diagram.

10. The data structure of state diagram

The state diagram class should be able to model states, substates, edges and connections between states. LEDA's graph class would directly give these properties, except substates, since the underlying graph is flat. Substates can be modeled using parametrisized graph class with a suitable state class as a parameter.

This data structure is shared between three major packages (and developers). Each of these packages has their own needs for persistent and scratch data. Instead of deciding a fixed class layout at this stage of development each package declares its own data (persistent and scratch) in a separate include file. Since the development is being done on separate computers this arrangement avoids the need to change one common file. So, for example, state diagram editor related members are found in files `ed_stdia.h`, `ed_state.h` and `ed_edge.h`.

Each class has a method to read and print itself from/to a stream. These functions are called by LEDA.

10.1. Class StateDiagram. The state diagram is represented as a class `StateDiagram` which inherits LEDA's parametrisized class `GRAPH` with classes `State` and `Edge` parameters for state and egde, respectively. There is an extra node in the state diagram, called the `root`-node and the actual state diagram appers as its substates.

```
#ifndef STATEDIAGRAM_H
#define STATEDIAGRAM_H

#include "state.h"
#include "edge.h"
#include "LEDA/graph.h"

#undef SCRATCH

class StateDiagram : public GRAPH<State *, Edge *> {
public:
  StateDiagram();
  StateDiagram(string in);
  ~StateDiagram();

  int read(string fname);
  void print(string fname);

  void setParticipant(string participant);
  string getParticipant(void);
  set<string> getScenarios(void);
  void setScenarios(set<string> scenarios);
  void setRoot(State *root);
  State *getRoot(void);

protected:

private:
  string participant;
  set<string> scenarios;
  State *root;

#include "ed_stdia.h"
#include "lo_stdia.h"
#include "st_stdia.h"
```

```
#define SCRATCH

private:

  union {
#include "ed_stdia.h"
#include "lo_stdia.h"
#include "st_stdia.h"
  } s;
};

#endif
```

Public methods just provide access methods to the private data members and their meanings are:

participant: The name of the participant this state diagram is generated for.
scenarios: The set of scenario names this state diagram is generated from.
root: Pointer to the root state in the state diagram. The actual state diagram states are the substates of this state.

All of the graph related methods are inherited from the class `GRAPH`. See LEDA manual for details.

10.2. Class State. Class `State` is used to implement the nested state structure. It inherits from class `SdePrimitive`. A state has a set of substates and a parent state. If the set of substates is non-empty (meaning that this state is a super state) then the default state should point to the default state.

```
//
// File state.h
//

#ifndef STATE_H
#define STATE_H

#include "sdeprim.h"

#undef SCRATCH

#define DECL
#include "ed_state.h"
#include "lo_state.h"
#include "st_state.h"

#undef DECL
#undef SCRATCH

class State : public SdePrimitive {
public:
  State();
  ~State();

  friend void Print(const State&, ostream&);
  friend void Read(State&, istream&);

  void setXY(int x, int y);
  void getXY(int& x, int& y);
  void setWH(int w, int h);
```

```
  void getWH(int&w, int& h);
  void setSuperState(State *superState);
  State *getSuperState(void);
  void setDefaultState(State *state);
  State *getDefaultState(void);
  void setName(string name);
  string getName(void);
  Bool hasSubstate(State *substate);
  void addSubstate(State *substate);
  void delSubstate(State *substate);
  set<State *> getSubstates(void);
  dictionary<string, list<string> > getActions(void);
  void setActions(dictionary<string, list<string> > actions);

protected:

private:
  string name;
  set<State *> substates;
  dictionary<string, list<string> > actions;
  State *superState;
  State *defaultState;

#include "ed_state.h"
#include "lo_state.h"
#include "st_state.h"

#define SCRATCH
#undef DECL

private:

  union {
#include "ed_state.h"
#include "lo_state.h"
#include "st_state.h"
  } s;
};

#endif
```

The public methods provide access to the private data members whose meaning is:

name: The name of this state.

substates: A set of substates of this state. A non empty set indicates that this is a super state.

superState: A pointer to the super state of this state. Null if no super state exists.

defaultState: If this is a super state, this should point to the initial state of its substates.

actions: Actions to be performed in this state. Each action type (like do, exit, entry) is one entry in the dictionary (the first string argument) and its value is a list of actions.

10.3. Class Edge. An edge between two states is represented by class `Edge`. It inherits from class `SdePrimitive`. An edge is drawn as straight line segments through points in list `coordinates`. The name of an event is stored in `name` and an optional action in `action`.

```
//
// File edge.h
//

#ifndef EDGE_H
#define EDGE_H

#include "sdeprim.h"

#undef SCRATCH
#define DECL

#include "ed_edge.h"
#include "lo_edge.h"
#include "st_edge.h"

#undef DECL
#undef SCRATCH

class Point {
public:
  friend void Print(const Point&, ostream&);
  friend void Read(Point&, istream&);
  xy_coord x, y;
};

class Edge : public SdePrimitive {
public:
  Edge();
  ~Edge();

  friend void Print(const Edge&, ostream&);
  friend void Read(Edge&, istream&);

  void setCoordinates(list<Point> coords);
  list<Point> getCoordinates(void);
  void setName(string name);
  string getName(void);
  void setAction(string action);
  string getAction(void);

protected:

private:
  list<Point> coordinates;
  string name;
  string action;

#include "ed_edge.h"
#include "lo_edge.h"
#include "st_edge.h"

#define SCRATCH
#undef DECL

private:

  union {
```

```
#include "ed_edge.h"
#include "lo_edge.h"
#include "st_edge.h"
  } s;
};

#endif
```

The public methods provide access to the private data members whose meaning is:

name: the name of this edge.
action: the name of an action associated to this edge.
coordinates: a list of points describing the graphical layout of this edge.

10.4. Class SdePrimitive. This is an abstract base class for classes `State` and `Edge`. It is only used by the state diagram editor to hold and implement the common (mostly) graphical properties of those classes. Its definition is not yet complete.

```
//
// File edprim.h
//

#ifndef SDEPRIM_H
#define SDEPRIM_H

#include "prim.h"
class SdeCanvas;

enum SdeType {Sde_Edge, Sde_State};

class SdePrimitive : public Primitive {
public:
  SdePrimitive();
  SdePrimitive(SdeCanvas *Canvas, SdeType type);
  virtual ~SdePrimitive();

  void OnChar(wxKeyEvent& event);
  void OnEvent(wxMouseEvent& event);
  void OnMenuCommand(int id);
  Canvas *getCanvas(void);
  void beginTextEdit(xy_coord mx, xy_coord my,
    const char *defaultText = (char *)0, int *defaultCnt = (int *)0);
  void endTextEdit(wxMouseEvent& event);
protected:
  SdeCanvas *canvas;
  SdeType type;
};

#endif
```

10.5. An example of an include file.

```
//
// File ed_edge.h
//

#if defined(DECL)

#ifndef ED_EDGE_DH
```

```
#define ED_EDGE_DH

// declarations outside of class scope
class SdeCanvas;

#endif

// *******************************************************************

#elif !defined(SCRATCH)

#ifndef ED_EDGE_PH
#define ED_EDGE_PH

// common methods and persistent members
public:
  Edge(SdeCanvas *owner);
  void draw(wxDC *dc);
  void drawFast(wxDC *dc, int dx = 0, int dy = 0);
  void invalidate(void);
protected:

private:

#endif

// ***********************************************************************

#else

#ifndef ED_EDGE_SH
#define ED_EDGE_SH

  struct {
// scratch variables
  } ed;

#endif

#endif
```

11. Concluding remarks

An underlying idea of the SCED project is that scenarios can be exploited in object-oriented software construction much more than what is usually the case in current OOA/D methods and tools. This is essentially a design-by-example approach for OOA/D. Similar direction is taken also by Jacobson, et al [10], 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. Scenarios can clearly give all (or nearly all) information needed for state diagrams, but it seems possible to produce large parts of the static model and the user interface model on the basis of scenarios as well. To make the maximal use of scenarios, the scenario notation has to be extended to allow the user to express things like aggregation and inheritance. These issues will be the topics of our future research.

References

[1] Biermann A.W., Baum R.I. and Petry F.E.: Speeding up the synthesis of programs from traces, *IEEE Trans. Comput.* **C-21**, 1975, pp.122–136.

[2] Biermann A.W. and Krishnaswamy R.: Constructing programs from example computations, *IEEE Trans. Software Engeneering*, SE-2, 9, 1976, pp. 141–153.

[3] S. Even: *Graph Algorithms*, Pitman, 1979.

[4] S. Even and R. Tarjan: Computing an *st*-numbering, *Theoretical Computer Science*, 2, 1976, pp. 339–344.

[5] H. De Fraysseix, J. Pach and R. Pollack: How to draw a planar graph on a grid, *Combinatorica*, 10, 1, 1990, pp. 41–51.

[6] Garey M.R. and Johnson D.S.: *Computers and Intractability*, Freeman, 1979.

[7] Gold E.M.: Complexity of Automaton Identification from Given Data, *Inf. Control* **37**, 1978, pp. 30–320.

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

[9] Hopcroft, J. and Tarjan, R.: Efficient planarity testing, *Journal of the ACM*, 21, 4, 1974, pp. 549–568.

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

[11] Koskimies K. and Mäkinen E.: *Inferring state machines from trace diagrams*, University of Tampere, Report A-1993-3.

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

[13] T. Männistö, T. Systä and J. Tuomi: **SCED** *Report and User Manual*, University of Tampere, Report A-1994-5.

[14] S.C. North: Drawing ranked digraphs with recursive clusters, *presented at Graph Drawing '93: ALCOM International Workshop on Graph Drawing*, author's e-mail address: north@research.att.com.

[15] J. Nummenmaa: Constructing compact rectilinear planar layouts using canonical representation of planar graphs, *Theoretical Computer Science*, 99, 1992, pp. 213–230.

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

[17] R.H.J.M Otten and J.G. van Wijk: Graph representations in interactive layout design, *Proc. IEEE Internation Symposium on Circuits and Systems*, 1978, pp. 914–918.

[18] R.C. Read: A new method for drawing a planar graph given the cyclic order of the edges at each vertex, *Congressus Numerantium*, 56, 1987, pp. 31–44.

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

[20] K. Sugiyama and K. Misue: Visualization of structural information: Automatic drawing of compound digraphs, *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-21, 4, 1991, pp. 876–892.

Tatu Männistö, Tampere University of Technology, Software Systems Laboratory, P.O.Box 553, FIN-33101 Tampere, Finland

E-mail address: `tm@cs.tut.fi`

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

E-mail address: `cstasy@cs.uta.fi`

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

E-mail address: `jjt@cs.uta.fi`