INFERRING STATE MACHINES FROM TRACE DIAGRAMS

Kai Koskimies and Erkki Mäkinen

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TAMPERE

REPORT A-1993-3

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS A
A-1993-3, JULY 1993

INFERRING STATE MACHINES FROM TRACE DIAGRAMS

Kai Koskimies and Erkki Mäkinen

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

ISBN 951-44-3393-9
ISSN 0783-6910

Inferring state machines from trace diagrams

Kai Koskimies and Erkki Mäkinen

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

Abstract

The automatic synthesis of state machines describing the behavior of a class of objects in object-oriented software modeling is studied. It is shown that the synthesis can be carried out on the basis of trace diagrams giving possible sequences of events during the execution of the system. An algorithm originally developed for the automatic construction of programs on the basis of their execution traces is applied to the problem, and an experimental state machine synthesizer is implemented. It is demonstrated that such a synthesizer is a highly useful component in a practical object-oriented CASE system.

CR Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic programming; D.2.2 [**Software Engineering**]: Tools and Techniques - CASE; I.2.6 [**Artificial Intelligence**]: Learning - Induction.

Additional Key Words and Phrases: inductive inference, synthesis of state machines, event trace, object-oriented software development.

1. Introduction

Object-oriented (OO) techniques are rapidly becoming prevalent in industrial software development. This trend is partly explained by the fact that OO techniques allow a systematic development process in which the conceptual, abstract world of the application is transformed into an executable program. During the last few years, a myriad of OO design formalisms and graphical tools have been presented to support such software development; among the most popular methods are the ones proposed by Jacobson [Jac92], Rumbaugh et al. [Rum91], Coad and Yourdon [CoY90], and Shlaer and Mellor [ShM88]. Although the proposed OO software design methods differ in their emphasis and notations, they share much in common. Typically, the static view of the objects is presented as a kind of entity-relationship diagram, and the dynamic behavior of the objects is described using some variant of a finite state automaton.

The work reported in this paper is part of a project aiming at advanced support for dynamic modeling in OO software development. We will use the OMT method developed by Rumbaugh et al. [Rum91] as a guideline, but it should be emphasized that the results can be exploited in other methods as well. OMT consists of three modeling techniques: object modeling for describing the static relations and properties of objects, dynamic modeling for describing the behavior of objects, and functional modeling for describing the input-output relations of operations. Of these models, OMT emphasizes the role of the object model - this part is relevant for all applications. Dynamic modeling is needed for specifying the external behavior of active software like embedded real-time systems or interactive user interfaces. Since most modern systems have components falling into these categories, dynamic modeling is essential in many cases. Functional modeling plays only a secondary role in OMT: it is used mainly in computation-oriented applications, like spreadsheet programs, compilers, CAD systems, etc.

In the OMT method, dynamic modeling starts with the construction of so-called *scenarios*. A scenario is a sequence of events occurring during a particular execution of a system. A scenario is presented graphically as a *trace diagram* describing in which order certain events are sent from one object to another. (In [Jac92], scenarios and event trace diagrams are called use cases and interaction diagrams, respectively.) Scenarios are given first for "normal" behavior, and then for "exceptional" behavior. Based on an understanding of the mutual interaction of the objects achieved through scenarios, a *state machine* (a *state diagram* in OMT terminology) is constructed for each class of objects appearing in the scenarios. The OMT method gives no exact procedure for constructing the state machines, but only some informal hints.

In this paper we demonstrate that it is possible to carry out the construction of the state machines automatically on the basis of the trace diagrams. This result is important for the design of CASE systems supporting OO software development, because it suggests that the attention in the construction of the dynamic model can be focused on trace diagrams rather than on state machines: a state machine is a more or less automatic synthesis of trace diagrams. Hence, a CASE system should offer an environment for creating trace diagrams, and a generator synthesizing state machines from these. At least the latter component is missing from the current systems we know of. Moreover, the support offered for constructing trace diagrams is modest at the best: the emphasis has been on the direct construction of a state machine.

A synthesized state machine serves as an initial approximation of the desired dynamic model. It is likely that the designer wants to directly edit the state machine later. For this reason the system should also provide a state machine editor (which is a standard component in current systems) and guarantee that the state machines are consistent with the trace diagrams. This kind of system is being implemented in our research group.

Our work is based on an application of the ideas presented by Biermann et al in their "autoprogramming" system ([BBP75], [BiK76]). We will show how this technique can be used in the context of trace diagrams and state machines, and discuss the implementation and first experiences of an experimental state machine synthesizer. The next section reviews the relevant concepts of the OMT method. In Section 3 we discuss our problem in terms of inference theory. A brief introduction to Biermann's method is given in Section 4. In Sections 5 and 6 we build a bridge between Biermann's method and the OMT framework, and present our state machine synthesis technique in detail. Section 7 discusses the implementation and some early experiences of the proposed technique. Concluding remarks are presented in Section 8. The paper is self-contained and assumes no prior knowledge of the OMT method or inductive inference.

2. Dynamic modeling in OMT

In the sequel we will briefly review the concepts of the OMT method regarding the dynamic model. With few exceptions, we follow the terminology of [Rum91]. We will leave out some concepts of OMT that are irrelevant for the present discussion.

As an example system, consider an alarm clock. The clock has a digital display which shows the current time. The alarm time can be set by "set new alarm time" function. Otherwise the previous alarm time is valid. The alarm time set is displayed after setting. The alarm can be turned off and on. "Set new alarm time" function automatically turns the alarm on. When the selected alarm time is reached (and the alarm is on), the clock starts buzzing. Buzzing stops when the alarm is turned off (and the user hopefully gets up) or a new alarm time is set.

Scenarios

A *scenario* is a sequence of events that occurs during an execution of a system. A scenario is presented informally as a sequence of statements in natural language. For an alarm clock a scenario could be given as follows:

> user sets new alarm time
> user sleeps
> alarm time is reached, and the clock starts buzzing
> user wakes up and turns the alarm off
> user sets new alarm time
> user falls asleep again
> alarm time is reached
> user turns alarm off

Trace diagrams

A scenario is formalized as a *trace diagram*. In a trace diagram the objects appearing in the scenario are drawn as vertical lines, and an event is represented as a horizontal arrow from the object causing the event to the object receiving (or reacting upon) the event. Time flows from top to bottom. Often the objects in a trace diagram are representatives of their classes, but it is also possible that several objects of the same class appear in a trace diagram (in different roles).

In Figure 1, a trace diagram is shown for the example scenario. We have divided the alarm clock into two objects: the control unit and the buzzer unit.

State machines

A *state* is an abstraction of attribute values of an object. In a system, objects stimulate each other, causing changes to their states. A stimulus from one object to another is an *event*. A change of state caused by an event is called a *transition*. A system consisting of states of objects, events, and transitions can be described as a *state machine* (or *state diagram*). A state machine is a directed graph in which states are represented by nodes and transitions are represented by directed edges. In addition to values of objects, states can also represent *actions*. An action is either a continuous activity ending when leaving the state (e.g., displaying a moving picture on the screen), or a finite sequence of operations ending when the operations are completed (e.g., opening a valve);

in both cases the action starts immediately after entering the state. Also, in both cases the action is interrupted in case an external event causing a state transition is received. A state machine may have a special initial state and final states. A state machine describes the behavior of a single class of objects. The state machines for various classes combine into a single dynamic model via shared events.

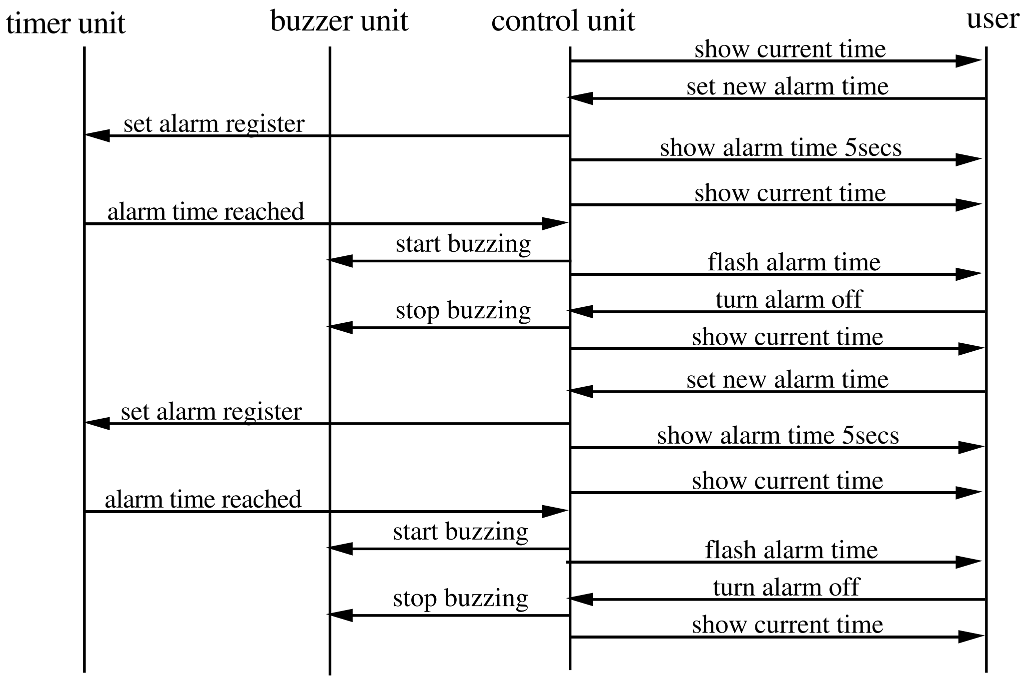

Figure 1. Event trace diagram related to the sample scenario.

An action can also be associated with a transition, provided that the action is instantaneous; i.e. its duration is insignificant as far as the behavior described by the state machine is concerned. Such an action is executed when the transition fires. A transition action is written after the event name, separated by a slash. In many cases it is possible to associate a particular (instantaneous) action either with a state or with a transition. Note that a transition action can always be removed by adding an extra state with the corresponding state action.

A *guard* is a Boolean function on the values of the attributes of objects. In contrast to an event, a guard has time duration: it holds (i.e. returns true) during a certain time interval. A guard can be associated with a transition: a guarded transition fires only if the guard holds at the time the transition event occurs. In state machines guards are shown in brackets following the corresponding event name. State machines are assumed to be deterministic: two out-going transitions of a state can have the same event only if they are both guarded and the guards never hold simultaneously.

Figure 2 shows a state machine for the control unit of an alarm clock. Note that all the events are here unguarded. If a transition has no event, it fires immediately when the action in the state is completed.

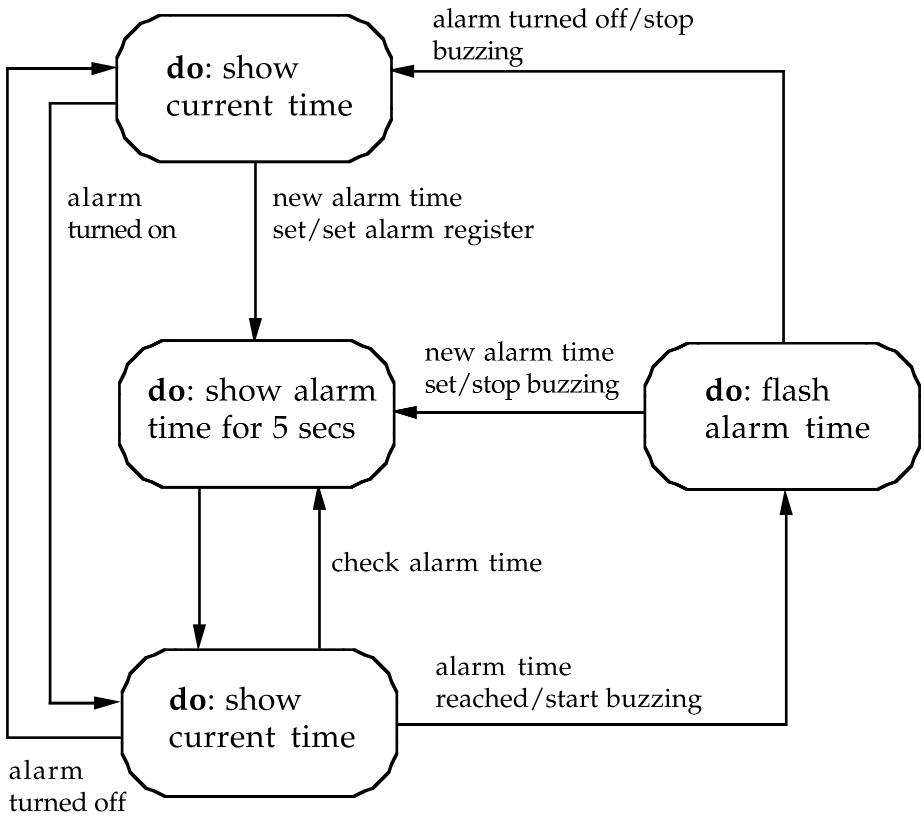

Figure 2. A state machine for the control unit of an alarm clock.

Note that there is an intimate relation between the trace diagram given previously and the above state machine. Following the vertical line of the control unit object from top to bottom, the events in the trace diagram can be found on a path in the state machine, starting from the topmost state. Each leaving event appears as an action and each arriving event appears as a transition event.

3. State machine synthesis as an inductive inference problem

Inductive inference studies algorithms for inferring general rules from their application instances. This chapter introduces some basic concepts related to inductive inference. For a more thorough treatment of the subject, see e.g. [AnS83, Ang92] and the references given there.

Angluin and Smith [AnS83] give a list of items which must be specified in order to define an inductive inference problem. First, one must specify the class of rules considered. Typically, inductive inference research has considered functions and languages. Second, one must specify a set of descriptions (called *hypothesis space*) for the rules considered. If one considers the inference problem related to languages, a hypothesis space could be e.g. finite automata or generative grammars. Third, for each rule, one must specify a set of examples. A sequence of valid examples constitutes an *admissible presentation* of the rule to be inferred. In the case of languages, examples are sentences belonging to the language in question. Fourth, one must specify criteria for a successful inference.

Examples can be either positive or positive and negative. In the former case we know that each example follows the rule to be inferred. In the latter case, we must specifically indicate whether a given example is positive or negative. In this paper we consider inference from positive data only.

The commonly used criterion for a successful inference, called *identification in the limit*, is due Gold [Gol67]. An infinite sequence of examples of the unknown rule is presented to inference algorithm M. After each example, M outputs a conjecture. If after some finite number of steps M guesses the correct rule and never changes its guess after this, then M is said to *identify the rule in the limit* (from positive examples).

A fundamental result by Gold [Go67] states that no family of languages containing all the finite languages and at least one infinite language can be inferred in the limit from positive samples. This result shows that even the family of regular languages is too general to be inferred from positive data only. Later, Angluin [Ang82] has shown that a natural subfamily of regular languages, the family of reversible languages, can be inferred from positive data in the limit. Recently, Yokomori [Yo91, Yo93] has presented an inference algorithm using only positive data for *very simple languages* (also known as *left Szilard languages*). The family of very simple languages is incomparable with the family of regular languages.

Following the list of Angluin and Smith, we define the problem of automatic synthesis of state diagrams as an inductive inference problem. We consider state machines as rules to be inferred. We do not define any specific descriptions for state machines, i.e. we do not make any distinction between state machines and their descriptions. In our case, traces derived from trace diagrams are used as examples.

State machines bear obvious resemblance to finite state automata. Since it is not possible to infer a finite state automaton from positive data (sentences), one might be inclined to conjecture that it is not possible to infer a state machine from positive data (traces). However, this is not the case since traces contain essentially more information than sentences. In fact, we do not consider a state machine as a language recognizer, but as an abstraction of a process. In Section 5 we show how Biermann's method [BiK76] can be applied to infer state machines from trace diagrams. The relationship between Biermann's method and standard inference problems related to languages will be discussed in greater detail in an accompanying paper [KoM93].

4. Biermann's method

In Biermann's method [BiK76], the user can synthesize a program by giving example traces of the program as sequences of instructions and conditions; the latter are Boolean expressions on the variables (or memory contents) of the program.

We first introduce some notations and definitions related to Biermann's algorithm [BiK76]. A computation consists of condition-instruction pairs $r_t = (c_t, i_t)$ being executed at discrete times t = 1, 2,... Operation i_t operates on memory contents m_{t-1} resulting a new memory contents m_t; this is denoted by $m_t = i_t(m_{t-1})$. Branching is made possible by using conditions which are (possible empty) conjunctions of atomic predicates or their negations. The value of condition c_t on memory contents m_{t-1} is denoted by $c_t(m_t)$. The value of the empty condition ϕ is true. The validity of condition c_t is checked before instruction i_t is executed.

The set of instructions available is denoted by I_1, I_2,... Moreover, we have two special instructions: I_0 is the start instruction which does nothing and I_H is the halt instruction. An instruction may appear several times during a computation. The appearances are separated by labels; for example, the appearances of I_1 are labelled by $1I_1$, $2I_1$, $3I_1$ and so on.

A *program* is a finite set a triples (called *transitions*) of the form (q_i, c_k, q_j) where q_i and q_j are labelled instructions and c_k is a condition satisfying the following restriction: if (q,c,p) and (q,c',p') are transitions and there is a memory contents m such that c(m) and c'(m) are true, then we must have c = c' and p = p'. Hence, it is not possible that two transitions were applicable at the same time. Thus, programs are always deterministic. (This is called *incomplete program* in [BiK76]. According to their terminology, an incomplete program is *program*, if it has the following two additional properties: (1) Each program has exactly one start instruction $1I_0$, and (2) if there is a transition (q, c, p) where c is a non-empty condition, then there must also be transitions from q for every possible combinations of the atomic predicates and their negations present in c.)

A *trace* of a computation is a (2n+1)-tuple T = (m_0, r_1, m_1, r_2, m_2, ..., r_n, m_n), where each r_t is a condition-instruction pair such that $m_t = i_t(m_{t-1})$ and $c_t(m_{t-1})$ is true, for each t = 1, 2,.., n. (This is called *partial trace* in [BiK76]; *trace* has then the additional properties that $r_1 = (\phi, I_0)$ and $r_n = (c_n, I_H)$.)

Programs can be illustrated as directed graphs with instructions as nodes and transitions as edges. Non-empty conditions are used as edge labels. Figure 3 illustrates the program with transitions ($1I_0$, ϕ, $1I_1$), ($1I_1$, {[a}, $1I_H$), ($1I_1$, {a}, $2I_1$), ($2I_1$, ϕ,$1I_2$), and ($1I_2$, ϕ, $1I_1$). Notice that this is a program even in then the sense of [BiK76].

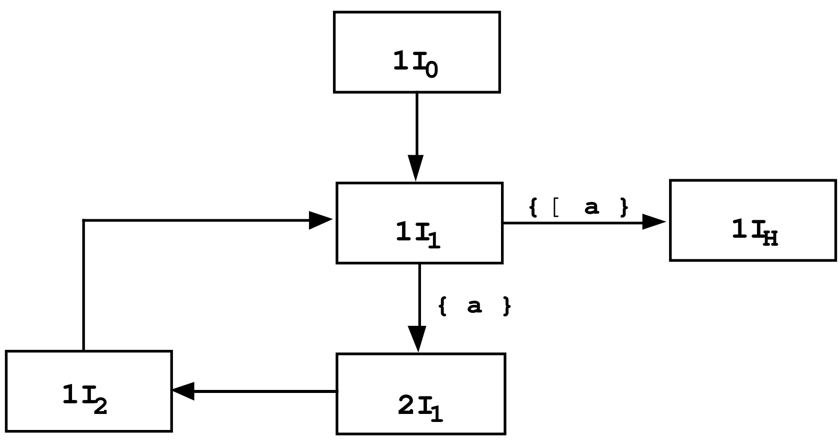

Figure 3. The illustration of program P = { ($1I_0$, ϕ, $1I_1$), ($1I_1$, {[a}, $1I_H$), ($1I_1$, {a}, $2I_1$), ($2I_1$, ϕ,$1I_2$), , ($1I_2$, ϕ,$1I_1$) }.

The input of Biermann's algorithm is a set of traces from which the algorithm constructs (infers) a program consistent with the traces. As the input traces contain the instructions used, the problem is to assign labels to the instructions. Suppose we are given a set of traces with n instructions (which are not usually all different from each others). A trivial solution is to assign different labels to each of the instructions in the input. The resulting program is a linear one with no branching or loops. A more interesting result is obtained when an order is defined in the set of possible labelling of the instructions, and the minimum labelling defining a program is chosen to be the output.

If $U = (u_1, u_2,..., u_n)$ and $V = (v_1, v_2,..., v_n)$ are two labellings of n instructions, then we denote $U < V$ if $u_i = v_i$, for $i = 1,..., j - 1$, and $u_j < v_j$. Let k be the number of different instances of instructions in the corresponding program, i.e. k is the number of nodes in the directed graph illustrating the program. The order among labelling is defined for pairs of the form (k, U). If (k, U) and (k', U') are two such pairs, we denote $(k, U) < (k', U')$ if $k < k'$ or if $k = k'$ and $U < U'$. Hence, we start with $k = 1$ and labelling (1,1,..., 1), enumerate all the pairs, and check which one is the first (according to the given order) defining an incomplete program. This process will always halt, since the pair (n, (1, 2,..., n)) defines the linear program mentioned. Notice that this process minimizes the number of different instances of instructions in the resulting program, i.e. the number of nodes in the directed graph illustrating the resulting program.

As an example, consider trace $(m_0, (\phi, I_0), m_1, (\phi, I_1), m_2, (\{a\}, I_1), m_3, (\phi, I_2), m_4, (\phi, I_1), m_5, (\{\lceil a\}, I_H), m_6)$. Since the trace contains four different instructions (I_0, I_1, I_2, I_H) it is useless to try values less than $k = 4$. The only possible labelling with $k = 4$ is (1,1,1,1,1,1). This labelling would give transitions $(1I_1, \{a\}, 1I_1)$, $(1I_1, \phi, 1I_1)$, and $(1I_1, \{\lceil a\}, 1I_H)$. This set of transitions contradicts the requirement of determinism. Hence, we set $k = 5$. Labellings (1,1,1,1,1,1), (1,1,1,1,2,1), and (1,1,1,2,1,1) imply nondeterminism as above, but labelling (1,1,2,1,1,1) gives the program illustrated in Figure 3.

We say that a program P can *execute* trace $(m_0, (c_1, i_1), m_1, (c_2, i_2), m_2, ..., (c_n, i_n), m_n)$ if there is a labelling $(u_1, u_2,..., u_n)$ such that $(u_j i_j, c_{j+1}, u_{j+1} i_{j+1})$ is a transition in P and $c_{j+1}(m_j)$ is true, for each $j = 1, 2,..., n$. Biermann and Krishnaswamy [BiK76] has proved the following: Given a set of traces, the program inferred by the above algorithm can execute all input traces.

Moreover, if P is any program whose traces are given as input, then the above algorithm identifies P in the limit in the sense discussed in the previous chapter.

When traces are long, the enumeration of the labelling candidates may take quite a long time. Biermann et al. [BBP75] have discussed techniques for speeding up the algorithm.

Another inference algorithm for synthesizing programs from their traces is due Takada [Tak92]. This algorithm shows that by restricting the class of programs one can obtain more efficient synthesis algorithms; Takada's algorithm has polynomial time complexity. Takada considers programs (called "processes") similar to those of Biermann, consisting of instructions (called "non-branch statements") and conditions (called "test statements"). However, Takada requires a process to be *historically deterministic*. A process is historically deterministic if for any statement s in process F the following criterion holds: if s_1, s_2,..., s_n are the possible dynamic follower statements of s then the statements s_1, s_2,..., s_n are all different and there are exactly n - 1 test statements on the path between s and s_1, s_2,..., s_n. Notice that this condition quantifies over all instances of a particular statement in process flowchart.

An *execution sequence* in a process flowchart is a sequence of statements from BEGIN to END (special statements for starting and ending a process). Takada's algorithm infers historically deterministic process flowcharts from execution sequences. The inference algorithm is interactive. The algorithm predicts the next statement in step-by-step manner. Algorithm's guess is based on a conjecture constructed from previous examples. The user confirms correct guesses, and gives a counterexample in the case of mistakes or when the algorithm makes a correct but unsatisfactory choice.

Since OMT is intended to be a general software design method, it seems slightly too restrictive to assume the property of historical determinism. Moreover, the possibility to use partial traces (i.e traces not starting from an initial state or ending at a final state) favors Biermann's method: in real time systems it is often more natural to give such partial traces. In fact, the use of trace diagrams implies the existence of partial traces because it would not be sensible to require that a scenario always begins at an initial state for all the participants. In the sequel we will concentrate on Biermann's method.

5. Applying Biermann's method to state machine synthesis

Although Biermann's notion of a program is clearly related to OMT's state machine, the relation is not trivial. For instance, a guard looks like a condition in Biermann's method, but they act in somewhat different roles. On the other hand, an event in OMT acts in the same role as a condition, but these concepts are rather different. Note that there is also a fundamental difference between programs and state machines in that a program (together with its memory cells initialized with some input) is a self-contained process which needs no external stimuli, whereas the whole purpose of a state machine is to describe the behavior of an object in the presence of external stimuli. Hence for a program the flow of control is fully determined by the program itself, but for a state machine the flow of control depends on the sequence of events sent by some processes that are in principle unknown.

In OMT, a state is defined as an abstraction of a particular combination of the attribute values of the object, relevant for responding to certain events received by the object. However, this definition is somewhat inaccurate because a state is often characterized by a particular point reach in a computation rather than by attribute values. Indeed, it is possible that the object has no attributes (except for the implicit state indicator). Jacobson [Jac92] makes this more explicit: in his terminology, a *computational state* describes how far we have come in the execution of the system, while an *internal state* represents a particular collection of attribute values. In Jacobson's method, states are essentially computational states, and it seems useful to adopt this higher-level view in OMT, too. This view justifies the unification of the concepts of a state in OMT and a labelled instruction in Biermann's method.

We will first define a concept corresponding to Biermann's trace. Let F be the set of events in system S. We suppose that all systems have two special events called *default* and *null* in their event sets; default is an event that is considered to occur when no other event occurs and null is an event that never occurs. A *trace* in S is a sequence of pairs (e1,e2) where both e1 and e2 are elements of F. (We adopt this term from Biermann's method although the concept is not exactly the same.) Let D be a trace diagram describing a scenario in S with an isntance from class C. For simplicity we assume that there can be at most one instance from class C in a trace diagram; the following discussion can be straightforwardly generalized to the case of several instances. The trace originating from diagram D with respect to C is obtained by using the following algorithm:

Algorithm 1: Trace derivation.
Input: Trace diagram D with instance of class C.
Output: Trace T originating from D with respect to C.

```
Let T be an empty trace.
Consider the vertical line corresponding to the C object.
Starting from the top, consider each arriving or leaving arc in
turn until the bottom is reached. For each arc, do the following:
  if  the current arc is a leaving one with event e then
     if the previous arc is a leaving one with event f then
          add item (f,default) to trace T
     end;
     if the current arc is the last one then
          add item (e,null) to trace T
     end
  else
     let the event associated with the current arriving arc be e;
     if the previous arc is a leaving one with event f then
          add item (f,e) to trace T
     else
          if the previous arc is an arriving one with event f
  then
               add item (null,e) to trace T
          end
     end;
     if the current arc is the last one then
          add item (null,null) to trace T
     end
  end;
```

For example, the above algorithm produces the following trace for the control unit object of the trace diagram in Figure 1:

(show current time, set new alarm time)
(set alarm register, default)
(show alarm time 5 secs, default)
(show current time, alarm time reached)
(start buzzing, default)
(flash alarm time, turn alarm off)
(stop buzzing, default)
(show current time, set new alarm time)
(set alarm register,. default)
(show alarm time 5 secs, default)
(show current time, alarm time reached)

(start buzzing, default)
(flash alarm time, turn alarm off)
(stop buzzing, default)
(show current time, null)

We define the concatenation of traces in the obvious way: if T_1 is trace $(a_1,b_1)...(a_m,b_m)$ and T_2 is trace $(c_1,d_1)...(c_n,d_n)$, then T_1T_2 is trace $(a_1,b_1)...(a_m,b_m)(c_1,d_1)...(c_n,d_n)$. The empty trace is denoted by λ.

A trace item (e1,e2) implies that, during a particular execution of the system, the object sends event e1 to some other object and then responses to event e2 sent by another object. From the OMT point of view, trace item (e1,e2) means that the execution has reached a state with state action "do: send e1" and with a transition labelled by e2. If e1 is null, the do-action is missing, and if e2 is null, the transition is missing. Default is treated like any other event, except that it cannot be sent. To simplify the presentation we ignore the specification of the receiver in the sending of an event; we assume that the event name uniquely determines the receiver. This is of course not true in practice, but it is straightforward to extend the presentation to include the identification of the receiver objects: if the same event is sent to different receivers, the sending actions are considered to be different actions.

Note that we essentially unify the concepts of action and event: from the receiver's point of view an arc in a trace diagram denotes an event, while from the sender's point of view the same arc denotes an action. In OMT, other actions than event sendings can appear as do-actions as well, but we ignore them here. Usually such actions do not appear in trace diagrams, since an arc implies some sort of inter-object activity, although sometimes an event has a "virtual" receiver that is not a true object (like a "user" in the example scenario in Figure 1). If desired, actions with no receiver can be added to trace diagrams (although this is not a standard procedure in OMT) and these actions can be included in the traces (and in the synthesized state machines) in the same way as the sending actions: our method does not care what is the meaning of an action.

It should be emphasized that actions are by no means state labels. Thus, there is no concept of a state in trace diagrams. Different states may very well have the same action. In a practical system it might be reasonable to allow also state labels in traces (indicating that certain trace items must originate from the same state), we will discuss this in Section 6.

When viewed from the receiver's side, we regard a guard as a part of an event: a guarded event e[g] denotes an event "e such that g holds". If event e is associated with two different guards g1 and g2, we assume that g1 and g2 never hold simultaneously. Hence, in our method, e[g1] and e[g2] are simply different events from the reveiver's point of view. It is not allowed that the same event appears with and without a guard in the outgoing transitions of a particular state. On the other hand, from the sender's point of view a possible guard is ignored, i.e. a guard is not sended (in the do-action) but only the event.

Since a trace diagram does not show explicitly whether an action is associated with a state or with a transition, we ignore transition actions and consider every action as a state action (i.e. do-action). For this reason we also assume that in a state machine every transition action has been removed and replaced by an additional state with a state action. Transition actions are assumed to be introduced as a separate step not discussed in this paper.

We require that every state machine has a single initial state with a special (unique) action "start". This is a slight deviation from OMT which allows state machines without an initial state. An initial state makes it possible to relate a trace with a state machine in an unambiguous way, assuming that the trace starts with the initial state. This is formulated in the sequel as trace execution, which is an essential concept for establishing the equivalence of state diagrams in terms of traces.

We say a state machine M can execute trace T from state q if there is a path starting from q and matching with T; in that case T is called a *partial trace* of M. Formally, a path starting from state q coincides with trace T if either T is empty or T = (b,e)T', q is associated with action b, there is a transition from q with event e to state q', and there is path starting from q' which coincides with T'. M can *execute* trace T if M can execute T from the initial state; in that case T is called a *complete trace* of M.

In the synthesis algorithm given below, we assume that there is an unknown state machine M whose (partial) traces are given as input to the algorithm. One of the input traces is assumed to contain the start action. The output of the algorithm is a (deterministic) state machine M' which is able to execute all the given traces. Furthermore, under certain assumptions (we will discuss these after giving the algorithm), given some finite number of input traces, M' becomes a minimal state machine such that M can execute trace T if and only if M' can execute T. Hence, the guess provided by the algorithm becomes functionally equivalent with the original state machine after some finite number of sample traces - the algorithm identifies M in the limit. These arguments follow directly from the results in [BiK75].

For the purposes of the synthesis algorithm, we define a state machine as a 6-tuple (S, E, A, d, p, I), where S is the set of states, E is the set of events, A is the set of actions, d is the *transition function* $d{:}S{\times}E \rightarrow S$, p is the *action function* $p{:}S \rightarrow A$, and $I \in S$ is the initial state. The transition function defines the next state on the basis of the current state and the event, and the action function associates a (possibly empty) action with every state. We require that p(I) = start for all state machines.

In the algorithm below, we make use of stack G which can store 4-tuples of the form (i', S', d', p') where i' is trace item, S' is a set of states, d' is a partial function $d'{:}S{\times}E \rightarrow S$ and p' is a partial function $p'{:}S \rightarrow A$. The purpose of the stack is to store the so far computed state set, transition function and action function, to be restored when the algorithm backtracks. Function sta associates a state with each trace item. Set Used(t) gives the set of states so far considered for trace item t.

Algorithm 2: State machine synthesis.
Input: Traces $T_1, T_2, \ldots, T_k$ of (unknown) state machine M.
Output: State machine M' = (S,E,A,d,p,I).

```
let T = T1T2...Tk = t1...tm (m>0);
let E = { e | (a,e) ∈ T}, A = { a | (a,e) ∈ T};
SetEmpty(G);
MAX := card(A) - 1;
while true do begin
    if IsEmpty(G) then
        i:= 1;
        let p be undefined for all states;
        let d be undefined for all states and events;
        S := ∅;
        MAX:= MAX + 1
    else
```

```
            (i,S,d,p) := Pop(G);
        end;
        let Used(tr) = ∅ for i < r ≤ m;
        undefine sta(tr) for i ≤ r ≤ m;
        while i ≤ m do begin
            if i > 0 and d(sta(ti-1),ei-1) = s then
                if p(s) = ai then  -- this is already covered
                    define sta(ti) = s;
                    i:= i + 1;
                else
                    exit   -- backtrack
                end
            else
                Cand := { s ∈ S \ Used(ti) | p(s) = ai };
                if i > 0 and Cand + ∅ then  -- add a new transition
                    pick up a member s of Cand;
                    Used(ti) := Used(ti) ∪ { s };
                    define sta(ti) = s;
                    define d(sta(ti-1),ei-1) = s;
                    Push(G,(i,S,p,d));
                    i:= i + 1;
                else
                    if card(S) < MAX then  -- add a new state
                        create new state s;
                        S := S ∪ { s };
                        define p(s) = ai;
                        define d(sta(ti-1),ei-1) = s;
                        define sta(ti) = s;
                        i := i + 1;
                    else
                        exit    -- backtrack
                    end;
                end;
            end;
        end;   -- while i
        if i > m then exit end;    -- success!
    end;  -- while
    let I = s such that p(s) = start.
```

Following the basic idea of the suggestions in [BBP75], Algorithm 2 implements the synthesis as a sequence of repeated attempts, each increasing the maximal state number: if there is no way to associate the trace items with states using the allowed number of states MAX, the process is repeated for MAX+1. In contrast to [BBP75], we use the number of different actions as the initial approximation for the number of states: clearly this is a lower bound. Moreover, it seems that in those applications we have in mind the number of states is usually close to the number of actions (provided that every state is associated with a non-null action). In our case the actions are conceptually high-level since they are part of a relatively abstract characterization of an object's behavior, whereas in Biermann's framework actions are simple statements of a program (like variable assignments). High-level actions tend to be more bound to the "state" of an object than the fairly arbitrary low-level statements. Note that the algorithm avoids unnecessary attempts if the initial approximated state number is close to the real one.

Algorithm 2 can be easily modified to work in an incremental mode; i.e., the traces can be given to the algorithm one by one, and synthesized into the existing state machine. Assume that we have state machine M and would like to extend it so that it can execute trace T as well. The updated state machine is constructed as follows.

Algorithm 3: Incremental state machine synthesis.
Input: State machine M = (S,E,A,d,p,I), trace T = $t_1 \ldots t_m$.
Output: Updated state machine M'.

```
let E' = E ∪ { e | (a,e) ∈ T };
let A' = A ∪ { a | (a,e) ∈ T };
MAX := card(S) + card(A') - card(A) - 1;
SetEmpty(G);
while true do begin
    if IsEmpty(G) then
        (i,S',d',p') := (1,S,d,p);
        MAX:= MAX + 1
    else
        (i,S',d',p') := Pop(G);
    end;
    let Used(t_r) = ∅ for i < r ≤ m;
    undefine sta(t_r) for i ≤ r ≤ m;
    while i ≤ m do begin
        if i > 0 and d'(sta(t_i-1),e_i-1) = s then
            if p'(s) = a_i then  -- this is already covered
                define sta(t_i) = s;
                i:= i + 1;
            else
                exit   -- backtrack
            end
        else
            Cand := { s ∈ S' \ Used(t_i) | p'(s) = a_i } ;
            if i > 0 and Cand + ∅ then  -- add a new transition
```

```
                pick up a member s of Cand;
                Used(t_i) := Used(t_i) ∪ { s };
                define sta(t_i) = s;
                define d'(sta(t_i-1),e_i-1) = s;
                Push(G,(i,S',p',d'));
                i:= i + 1;
            else
                if card(S) < MAX then  -- add a new state
                    create new state s;
                    S' := S' ∪ { s };
                    define p'(s) = a_i;
                    define d'(sta(t_i-1),e_i-1) = s;
                    define sta(t_i) = s;
                    i := i + 1;
                else
                    exit    -- backtrack
                end;
            end;
        end;
    end;   -- while i
    if i > m then exit end;    -- success!
end;  -- while
let M' = (S',E',A',d',p',I)
```

The incremental version builds on the existing state machine as if it had been created by the synthesis algorithm. Note that the initial value of MAX is set to be the sum of the number of existing states and the number of new actions introduced by the new trace: this sum is a feasible lower bound since a new trace can never eliminate existing states, and a new state must be created at least for each new action.

Algorithm 2 is sensitive to the order of traces in its input. Indeed, there are some special cases where the state machine outputted may vary depending on the order of the input traces. On the other hand, the following holds: if Algorithm 2 is run with input T_1,.., T_k (in this order), and then the resulting state machine is given as input with a trace T_{k+1} to Algorithm 3, the final result equals the state machine obtained by running Algorithm 2 with input T_1,.., T_k, T_{k+1} (in this order).

6. On the accuracy of the guesses

If we want that the output of Algorithm 2 is excatly equivalent with the unknown state machine from which the sample traces are taken, then the concept of "completeness" is essential. As mentioned above, a program was defined to be complete in [BiK76] if in all cases there is a valid continuation of the program (except after halt statement). This guarantees that the traces will eventually give sufficient information about the program for inference. If this condition is satisfied, the original unknown program (state machine) and the one inferred are indeed equivalent [BiK76, Corollary of Theorem 2]. The problem is that this condition is hardly ever satisfied in practical state machines.

The problem is illustrated in Figure 4. If the completeness requirement need not be satisfied, the source state machine could be the one shown on the left-hand side. This state machine can execute traces (a1,e1)(a2,e1)(a4,e1)(a5,e1)(a7,null) and (a1,e2)(a3,e2)(a4,e2)(a6,e2)(a7,null) and since these cover the whole state machine no other traces are necessary. The state machine inferred by Algorithm 2 on the basis of these traces is given on the right-hand side. As can be seen, the inferred state machine is more general, executing traces that cannot be executed by the original state machine. In a complete state machine there would be e1- and e2-transitions from both a4 states, this would be shown in the traces, and the a4 states could not be merged (unless, of course, the transitions were identical).

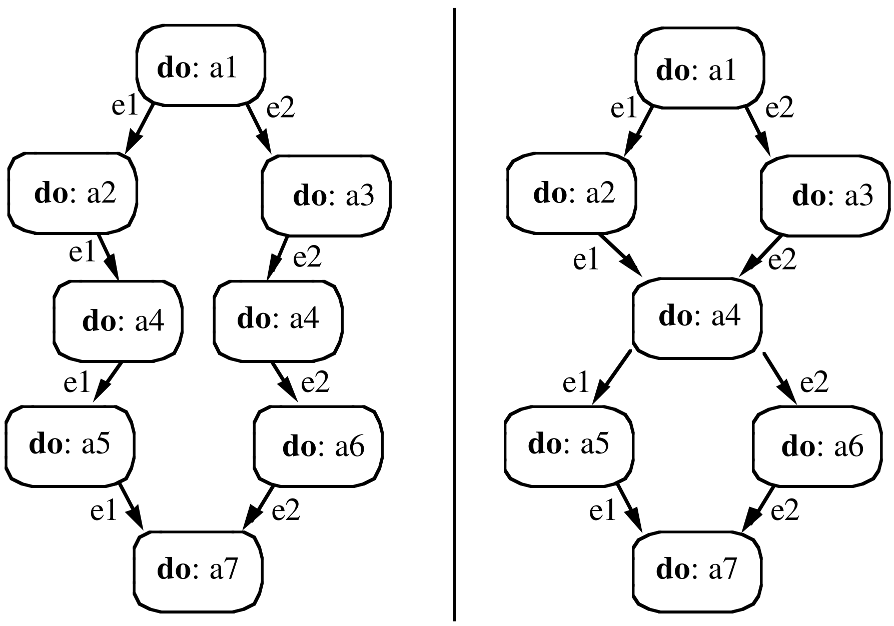

Figure 4. The source state machine (on the left) and the inferred state machine (on the right).

We can solve the problem by *completing* a state machine with additional "error" transitions: if there is no transition with event e from state s, the state machine is augmented with such a transition whose target is a new state; this state has a special "error-and-halt" action and no continuation. In this way the state machine explicitly indicates which transitions are not allowed, and this information is passed via traces to the inference process. In the final state machine the error transitions and states can be again removed, presenting thus the state machine in the desired form. Since the original state machine is merely an imaginary concept, in practice this solution means that the user should be able to express in the traces "negative" cases as well, that is, cases which end with a forbidden move. Hence, we can use negative information to replace the missing completeness property.

For instance, in the case of Figure 4 the two given traces should be accompanied by two other, say (a2,e1)(a4,e2)(err,null) and (a3,e2),(a4,e1),(err,null), where err is the "error-and-halt" action. Recall that the traces can be partial. This is sufficient information for the synthesis algorithm, which will produce now the exact copy of the original state machine (assuming that the err-state and its transitions are finally removed). Note that in this case we need not give similar "negative" traces for the states a2, a3, a5, a6, and a7 since they are not threatened by inadvertent merge. However, it may be hard for the user to understand which "negative" traces are really necessary.

On the other hand, in practice this is not necessarily a problem: we may regard state machine synthesis as a software construction process rather than as a learning process. After all, there is no secret state machine which we try to guess, but our purpose is to construct a state machine describing a new system component on the basis of primitive behavior descriptions (traces). If we use non-complete state machines, the method produces the smallest (with respect to number of states) deterministic state machine capable of executing the given traces. The fact that the resulting state machine will allow various combinations of the given traces as well is usually exactly the desired effect. However, in some cases the algorithm may combine the traces in an undesirable way.

The limitations of the synthesis method, when applied to non-complete state machines, are illustrated by the following practical example. Consider a traffic light controller for which we have got the following two traces:

(NS straight, time-out[no cars in NS left lanes])
(EW straight, time-out[no cars in EW left lanes])
(NS straight, null)

(NS straight, time-out[cars in NS left lanes])
(NS may turn left, time-out)
(EW straight, time-out[cars in EW left lanes])
(EW may turn left, time-out)
(NS straight, null)

Here NS denotes north-south and EW denotes east-west; "NS straight" means a setting that allows the cars to proceed in north-south (or south-north) direction, and "NS may turn left" switches the lights so that those cars turning left in north-south direction can proceed. These traces will yield the state machine of Figure 5 (without the dashed transitions). This is a fairly successful example of automated state machine synthesis: the result is exactly the state machine given in [Rum91, p.92] as an example of dynamic modelling.

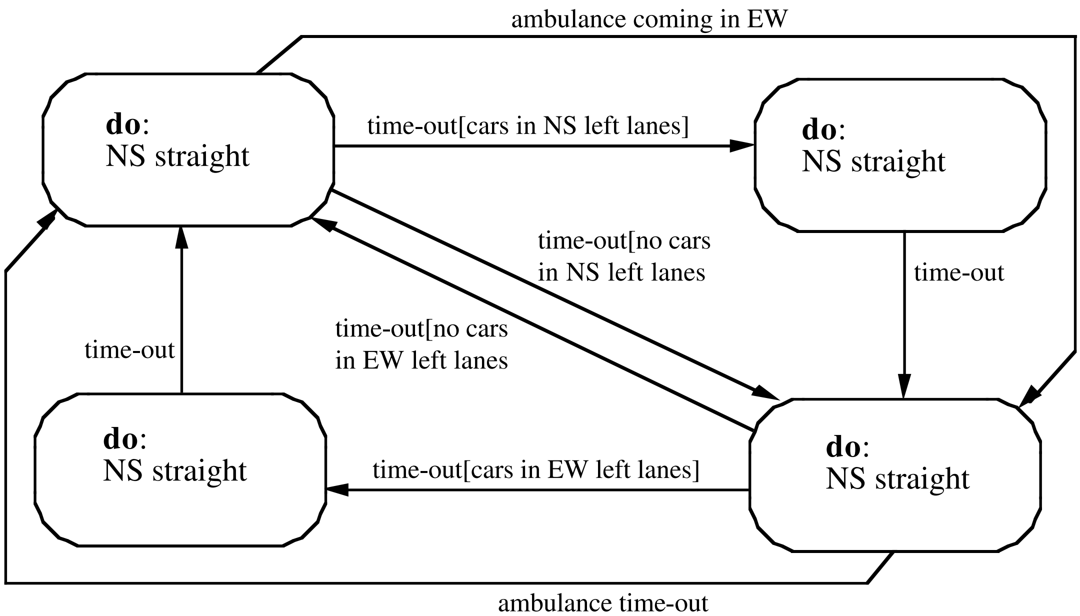

Figure 5. State machine for traffic light controller.

Suppose now that the user would like to add some behavior concerning the controlling of lights in case an ambulance arrives. For this purpose the following trace might be added:

(NS straight, ambulance coming in EW)
(EW straight, ambulance time-out)
(NS straight, null)

When this trace is synthesized into the state machine of Figure 5, the additional traces denoted with dashed lines appear. However, this is not an acceptable solution, because the "EW straight" phase is interrupted by "ambulance time-out" also in the normal case, i.e. when there has not been an ambulance. If ambulance time-out is shorter (as it presumably is) than the normal time-out, the other transitions are never taken. The algorithm has merged two states together which represent logically different situations. The reason for this is that the traces do not give sufficiently information: the algorithm has no way of knowing that the "ambulance time-out" event is allowed only after an ambulance has been observed.

Since the problem concerns the actual semantics of the actions and events, it is very hard to reliably distinguish this kind of undesired merge from the (far more common) desired one, without some new information included in the traces. A possible heuristics could be the following "rule of flying visit": 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. In such a case the new trace makes only a flying visit to an existing path, reusing a single state; this is suspect and in any case less fruitful combining. Note that this rule is sufficient to guide the synthesizer in the problem of Figure 5.

As in the traffic light example, an undesirable merge takes place also in the alarm clock example: the algorithm will merge the two states with the "show current time" action. This would be wrong because then the state machine would not preserve the information about the position of the alarm on/off button: the clock would start buzzing even though the alarm is off. Again, the "rule of flying visit" would prohibit the algorithm from making the undesirable merge.

Another way to solve this problem would be to require that the user adds certain information to the traces (or actually to the trace diagrams; we assume that the information can be transferred to the traces). In the above example, it would suffice to give a state name in the trace, and modify the algorithm so that only trace items with identical (or none) state names can share the same state. Another piece of information could be a symbol denoting "this is the only way to proceed" in a trace item; for the above example this could be given e.g.

(EW straight, !ambulance time-out).

Since the "ambulance time-out" is marked as the only outgoing transition ("!"), this item cannot be associated with the existing state that has the same action ("EW straight"), and the correct state machine results.

7. Experiences

The state diagram synthesizer is assumed to be part of an OO software design environment in which a central tool for dynamic modeling is a trace diagram editor. The synthesizer provides automatically an initial version of the state diagram which can then be further edited through a state diagram editor. This kind of environment for OO dynamic modeling is currently being developed in our group [MT93]. As a part of this environment, a trace diagram editor (SCED, SCenario EDitor) has been implemented and integrated with a state diagram synthesizer. The system is written in Borland C++ and is running under Windows 3.1 in PCs.

The interface of SCED is shown in Figure 6. On the left, two trace diagram windows are displayed for a traffic light control system. On the right, the state diagram synthesized from these scenarios is presented. The resulting state diagram is equivalent with the one given in [Rum91, p. 92] as an example of dynamic modeling.

Although the synthesis algorithm makes use of backtracking, it is fast for practical state diagrams of reasonable size. One reason for this is that there often is a strong connection between a state and an action: very often a state has a unique action. Notice that if this holds for all the states, the state diagram synthesis can be carried out deterministically. Thus, in practice backtracking is usually not in heavy use. It should also be noted that practical state diagrams are seldom very large, because such state diagrams would become hard to understand. In this context state diagrams are above all specification tools to be read by humans. In all the examples we have tried, the updating of a state diagram on the basis of a new trace is carried out without observable delay - the time is usually a fraction of a second.

[Figure 6 is missing from the electronically distributed version of the report.]

Figure 6. The user interface of SCED.

The experiments were run in a 20MHz 386 PC. Some of the examples were taken from [Rum91], namely the phone line (PL) with 11 states, 11 actions, 24 transitions and the automatic teller machine (ATM) with 16 states, 15 actions, 27 transitions. Both for ATM and for PL exactly the state diagram given in the book [Rum91] was indeed synthesized without problems and very fast. Since [Rum91] provides only a single example trace diagram for each system, we had to invent the missing trace diagrams ourselves, trying to imitate a software designer.

When the algorithm has to backtrack, the order of input traces has its effect to the time needed. It is possible to find state diagram in which several states share the same action; this makes the task essentially harder for the algorithm. Although the number of states, actions, transitions, and trace items are even smaller than in ATM and PL, the time needed can be considerably longer (up to 5 seconds in our experiments), especially when the order of input traces is unfavourable.

It seems that the time needed does not depend directly on the size of the resulting state diagram, but rather on the state/action ratio: the synthesis algorithm is very fast when this ratio is close to 1 (ATM), but it becomes significantly slower for larger values. Although it is difficult to draw conclusions on the basis of these experiment, the time appears to grow linearly with respect to state/action ratio.

8. Concluding remarks

We have shown that automatic state machine synthesis is a realistic possibility that should be taken into account in the design of future OO software design environments. The application of this technique implies that the emphasis in the construction of the dynamic model is laid on the scenarios rather than on state machines. The latter can be considered, at least partially, as a mechanical summary of the former.

In [Jac92], Jacobson introduces so-called *use-case driven* design. This is a software design process based on the idea that the architecture of a system is controlled by what the users want to do with the system. The users' requirements are modelled as use cases, which roughly correspond to OMT's scenarios. Our method supports this view, providing automatic means to derive the dynamic model automatically from a set of use cases. In Jacobson's methodology this is only one aspect of exploiting use cases since they are the basis of the whole design, but the idea of emphasizing the role of scenarios is characteristic to both his and our approach.

So far we have studied the automatic generation of state machines in terms of rather primitive, general notions. Since we have employed only few OMT notations, our method is in principle applicable in many contexts making use of state machines or similar devices. On the other hand, if the aim is to build a true OMT model, it would be highly desirable to make use of the full set of OMT concepts in the produced state machines. Some of those concepts can be added to the state machine afterwards as an automatic or semi-automatic post-processing step; such features are e.g. transition actions and state hierarchies. Some concepts may require that the notation of trace diagrams is extended, allowing the user to give new kind of information (like names of states). Our future work will deal with these issues.

Acknowledgements
This work is in part financed by TEKES (Finnish Technology Development Centre) and Academy of Finland. The experiments reported in Section 5 were carried out by Tarja Systä. SCED has been implemented by Tatu Männistö and Jyrki Tuomi.

References

[Ang82] D. Angluin, Inference of reversible languages, *J. ACM* **29** (1982), 741-765.
[Ang92] D. Angluin, Computational learning theory: survey and selected bibliography, *Proc. 24th Ann. ACM Symp. on the Theory of Computing* (1992), 351-369.
[AnS83] D. Angluin and C.H.Smith, Inductive inference: theory and methods, *ACM Comput. Surv.* **15** (1983), 237-269.
[BiK76] A.W. Biermann and R.Krishnaswamy, Constructing programs from example computations, *IEEE Trans. Softw. Eng.* **SE-2** (1976), 141-153.
[BBP75] A.W. Biermann, R.I. Baum and F.E. Petry, Speeding up the synthesis of programs from traces, *IEEE Trans. Comput.* **C-21** (1975), 122-136.
[Gol67] E.M.Gold, Language identification in the limit, *Inf. Conrol* **10** (1967), 447-474.
[CoY90] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yordon Press, 1990.

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

[KoM93] K. Koskimies and E. Mäkinen, A reformulation of Biermann's algorithm, Manuscript in preparation, 1993,

[MäT93] T. Männistö and J. Tuomi, The design of the scenario editor SCED. Manuscript, June 1993.

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

[ShM88] S. Shlaer and S.J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1988.

[Tak92] Y. Takada, Interactive synthesis of process flowcharts, International Institute for Advanced Study of Social Information Science (IIAS-SIS), Fujitsu Labs. Ltd, Report IIAS-RR-92-4E, March 1992.

[Yok91] T. Yokomori, Polynomial time learning of very simple grammars from positive data, *Proc. 4th Workshop on Computational Learning* (1991), 213-227.

[Yok93] T. Yokomori, Polynomial time identification of very simple grammars from positive data, University of Electro-Communications, Dept. of Computer Science and Information Mathematics, Report CSIM 90-15, December 1990. Revised March 1993.