MINIMALLY ADEQUATE TEACHER DESIGNS SOFTWARE

Erkki Mäkinen and Tarja Systä

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES
UNIVERSITY OF TAMPERE

REPORT A-2000-7

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER AND
INFORMATION SCIENCES
SERIES OF PUBLICATIONS A
A-2000-7, APRIL 2000

MINIMALLY ADEQUATE TEACHER DESIGNS SOFTWARE

Erkki Mäkinen and Tarja Systä

University of Tampere
Department of Computer and Information Sciences
P.O.Box 607
FIN-33014 University of Tampere, Finland

ISBN 951-44-4834-0
ISSN 1457-2060

Minimally adequate teacher designs software [1]

Erkki Mäkinen [a] **and Tarja Systä** [b]

[a] Department of Computer and Information Sciences
P.O. Box 607
FIN-33014 University of Tampere, Finland
E-mail: `em@cs.uta.fi`

[b] Software Systems Laboratory
Tampere University of Technology
P.O. Box 553
FIN-33101 Tampere, Finland
E-mail: `tsysta@cs.tut.fi`

Dedicated to Professor Kai Koskimies on the occasion of his 50th birthday.

Abstract. We consider the problem of synthesizing UML statechart diagrams from sequence diagrams as a language inference problem, and we solve it in Angluin's framework of minimally adequate teacher. The designer has the role of teacher who answers membership and equivalence queries made by the algorithm. It turns out that there are several natural restrictions concerning the form of languages accepted by state machines. These restrictions can be used to decrease the number of membership queries needed, and thus, to make the method practically applicable.

Keywords. Software design; Grammatical inference; UML; Minimally adequate teacher.

1 Introduction

The Unified Modeling Language (UML) [20, 18] has been accepted as an industrial standard for specifying, visualizing, understanding, and documenting object-oriented software systems. It provides several diagram types that can be used to view and model the software system from different perspectives and/or at different levels of abstraction. UML supports all lifecycle stages of the forward engineering process from requirements specification to implementation.

[1] Work supported by the Academy of Finland (Project 35025) and TEKES.

In object-oriented analysis and design (OOAD), dynamic modeling aims at the description of the dynamic behavior of objects using a variant of a finite state machine. A UML variant of a state machine is called *a statechart diagram.* A statechart diagram is a graph that represents a state machine. The semantics and notation used in UML follow Harel's statecharts [7]. *Sequence diagrams* and *collaboration diagrams* are also used for behavioral modeling in UML based approaches. A sequence diagram shows the interaction over time but does not show other relationships between objects than the events belonging to the interaction. A collaboration diagram, in turn, does not show time as a separate graphical dimension. While sequence diagrams and collaboration are used for showing object interactions, a statechart diagram can be used as a protocol specification, showing the legal order in which operations of an object may be invoked.

In UML based behavioral modeling, example scenarios are usually given first for "normal" cases, and then for various cases representing "exceptional" behavior. Such scenarios are visualized as sequence diagrams. When a sufficiently complete set of sequence diagrams exists, statechart diagrams are constructed for desired participating objects.

Automated support for constructing state machines from scenarios provides considerable help for the designer, allowing her to concentrate on sequence diagrams rather than on statechart machines. Automatic generation of statechart machines from variations of Message Sequence Charts (MCSs) [13] (including UML sequence diagrams) is implemented in several tools [12, 15, 16, 21, 22].

The synthesis algorithms generalize information given in MSCs, i.e., the resulting statechart machine accepts more paths through the modeled system than represented as MSCs. The generalization is usually the desired affect. However, in some cases a synthesized statechart machine might also accept unwanted or erroneous paths and thus be "overgeneralized". Applying the synthesis algorithm to an incomplete set of sequence diagrams may result in an overgeneralized state machine that does not meet user's intentions. Inaccuracies in the contents of the sequence diagrams can have the same effect. The synthesis algorithm presented in this paper avoids undesired results, due to overgeneralization. To achieve this goal, the algorithm asks the designer for guidance during the synthesis process, when needed.

As already noticed in [14], the synthesis process can be modeled as a language inference problem. In this paper we continue this line of research and consider the synthesis as a language inference problem under Angluin's [2] paradigm of minimally adequate teacher. In our model, the designer is the teacher who is able to answer the membership and equivalence queries posed by the inference algorithm which now coincides with the synthesizer.

Being a minimally adequate teacher requires that the designer can answer two kinds of simple questions:

1. she must decide whether a given sequence of events (or messages) is possible in the system she is implementing (the membership queries)

2. she must accept or reject the state machines proposed by the algorithm, and moreover, if she rejects, a counterexample from the symmetric difference of the languages accepted by the state machine proposed by the algorithm and the desired state machine must be given (the equivalence queries).

The membership queries are easy to answer, since the designer must have a general idea of the behavior of the system she is implementing. The equivalence queries are a bit more demanding, but an experienced designer with a minimal formal training should find also them easy to answer.

This paper is organized as follows. In Chapter 2 we recall the basics of grammatical inductive inference and properties of minimally adequate teacher. The readers not familiar with inductive inference are referred to [4]. We also use elementary concepts of formal language theory following the notations of [10]. In Chapter 3 we show how Angluin's algorithm can be used to synthesize state machines by solving an equivalent language inference problem. This is done by introducing a new algorithm called Minimally Adequate Synthesizer (MAS). In Chapter 4 we give example runs of our modified algorithm, and finally in Chapter 5, we discuss the use of our algorithm in OOAD.

2 Minimally adequate teacher

The classical setting in grammatical inference is that of "identification in the limit" [8]. The inference algorithm is supposed to obtain an infinite sequence of input strings, and after reading each string, the algorithm outputs a hypothesis. The inference process is successful if the hypotheses "converge" to the correct language, i.e. if the hypotheses eventually keep unchanged and they define the correct language.

The input strings can contain only words of the unknown language (inference from positive data) or all strings over the alphabet in question with an additional label indicating whether or not they are in the unknown language (inference from positive and negative data).

The infinite nature of identification in the limit seems to imply that it is not suitable model for a practical synthesis algorithm. Moreover, Gold's [8] famous result says that regular languages are not inferable in the limit from positive data only.

However, as shown in [15], positive data is usually sufficient if some additional condition is applied. In SCED, the synthesizer seeks the state machine with minimum number of states by using an exhaustive method of Biermann and Krishnaswamy [5]. Minimizing the number of states in the resulting automaton is known to be NP-complete even when positive *and* negative data is available [9].

In some cases, state machine synthesis algorithms that use positive data only might get slow or result in a overgeneralized statechart machine. It would thus be advantageous to have a synthesis algorithm that makes use of a stronger form of grammatical inference than inference from positive data only.

In this paper we suppose that regular languages are presented by a minimally adequate teacher which can answer membership queries and test whether the language defined by the hypothesis is equal to the unknown language and provide a counterexample if not. The second type of question is *a conjecture* consisting of a description of a regular language, in our case a state machine. The counterexample is a word w from the symmetric difference of the unknown language L and the language K defined by the hypothesis. Hence, w belongs either to L or to K, but not to both, i.e. $w \in (L \cup K) \setminus (L \cap K) = (L \setminus K) \cup (K \setminus L)$.

Angluin's [2] inference algorithm for minimally adequate teacher maintains a so called *observation table* T containing the current information about members and non-members of the unknown language. The rows of T are labelled by the elements of $(S \cup S \cdot A)$ where A is the alphabet over which the unknown language is defined, and S is a prefix-closed set of strings in A^*. (We use the concatenation operation $\cdot$ to make the notations clearer.) The columns of T are labelled by the elements of a suffix-closed set R. (A set is *prefix-closed* if every prefix of every member of the set is also in the set. A *suffix-closed* set is defined analogously.) If I is a set of words, then let *pref(I)* stand for the set obtained from I by making it prefix-closed without deleting any words.

The entry for row s, $s \in (S \cup S \cdot A)$, and column r, $r \in R$, equals $T(s \cdot r)$ which, in turn, is 1, if $u = s \cdot r$ is in the unknown language U, otherwise $T(u) = 0$. An observation table is said to be *closed* if for each t in $S \cdot A$ there is an s in S such that $row(s) = row(t)$. An observation table is *consistent* if whenever s_1 and s_2 are in S such that $row(s_1) = row(s_2)$, for all a in A, $row(s_1 \cdot a) = row(s_2 \cdot a)$.

As an example, consider the observation table T in Table 1. We have $A = \{a, b\}$, $S = \{\lambda, a\}$ and $R = \{\lambda\}$. (λ stands for the empty string.) T is neither closed nor consistent. It is not closed because, for any t any S, we do not have

$row(b) = row(t)$. It is not consistent because we have $row(\lambda) = row(a)$, but $row(\lambda \cdot b) \neq row(a \cdot b)$.

T	λ
λ	1
a	1
b	0
aa	1
bb	1

Table 1: A sample observation table T with $S = \{\lambda, a\}$ and $R = \{\lambda\}$.

The inference algorithm starts with $S = R = \emptyset$ and first asks membership queries for λ and for all symbols a in A. T is updated by the answers of these queries. Now, while T is not closed and consistent, new strings are added to S and R, and the corresponding table entries are found out by membership queries. When T is closed and consistent, the algorithm makes a conjecture, i.e. an equivalence query. The algorithm halts if teacher accepts the conjecture. Otherwise, the counterexample updates T, and the while-loop is executed again.

Given a closed, consistent observation table T, the conjectured finite state machine $M(T)$ can be constructed by setting

- the state set as $Q = \{row(s) \mid s \in s\}$
- the initial state as $q_0 = row(\lambda)$
- the set of final states as $F = \{row(s) \mid s \in S \text{ and } T(s) = 1\}$
- the transition relation as $\delta(row(s), a) = row(s \cdot a)$.

Hence, rows labelled by elements of S are candidates for states of the automaton to be conjectured. Elements of R correspond to different "experiments" which test whether or not two rows can be joined to be a state. Rows labelled by elements of $S \cdot A$ are needed for constructing the transition relation δ. The conjectured automaton correctly handles the strings so far known to be members or non-members of the desired language. Other strings may be handled erroneously, which is then corrected by user's counterexamples.

There are cases where S contains elements whose table entries are all zero and which are prefixes of no words in the unknown language U. This may happen if the teacher gives a counterexample not in U. The states corresponding to such rows can be omitted from the conjectured automaton without affecting the language accepted.

Angluin [2] has proved that $M(T)$ is the smallest finite state automaton consistent with T (where "smallest" means "has the fewest number of states").

Following [2] we next formulate the inference algorithm using minimally adequate teacher in greater detail. In order to simplify the notations in the algorithm, we consider T as a function mapping $(S \cup S \cdot A) \cdot R$ to $\{0, 1\}$. Hence, updating T to contain all the necessary values can be expressed as "extend T to $(S \cup S \cdot A) \cdot R$ using membership queries". This simply means that the algorithm inserts the necessary rows and columns and fills in the table entries by concluding the values itself or by consulting the user.

Algorithm MAT

Let $S = R = \{\lambda\}$;
Ask membership queries for λ and each a in A, and store the answers in the observation table T;
repeat
  while T is not closed or not consistent **do begin**
    if T is not consistent **then**
      find s_1 and s_2 in S, a in A, and r in R such that $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot r) \neq T(s_2 \cdot a \cdot r)$;
      add $a \cdot r$ to R;
      extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;
    if T is not closed **then**
      find s_1 in S and a in A such that $row(s_1 \cdot a)$ is different from $row(s)$ for all s in S;
      add $s_1 \cdot a$ to S;
      extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;
  end $\{while\}$
  Construct $M(T)$ from T and conjecture M;
  if teacher replies with a counterexample t **then**
    add *pref*$(\{t\})$ to S;
    extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;
until the teacher replies *yes*;
Output M;

The time and space complexity of MAT can be given as in the following theorem by Angluin [2].

Theorem 2.1 *Let U be an unknown regular language presented by a minimally adequate teacher. If n is the number of states in the minimum deterministic finite state automaton accepting U, and if m is an upper bound on the length of any counterexample provided by the teacher, then the total running time of MAT is bounded by a polynomial in m and n. Moreover, the observation table T needed in MAT is of size $O(m^2n^2 + mn^3)$.*

It should be noted that the bounds in Theorem 2.1 are related to a random sampling oracle. In our application, the teacher is a software designer who is likely to give "good" examples. Thus, in practice, the situation is much better than the worst case time and space bounds suggest. However, MAT is not efficient enough for practical use. It needs much too many membership queries to be useful in OO software design. In the next chapter we consider how MAT can be streamlined to be practical.

We know that for a polynomial time algorithm it is not sufficient to use only membership queries [1] or only equivalence queries [3] but both types of queries are needed. Equivalence queries are enough, if counterexamples are always (lexicographically) "smallest" as studied by Ibarra and Jiang [11] and Birkendorf et al. [6]. However, it is not reasonable to expect the user to provide smallest counterexamples to the algorithm.

Improvements for Angluin's algorithm are suggested by Rivest and Schapire in [19]. However, we do not apply these improvements here since they involve random processes and the resulting automaton is correct only with probability less than 1, and since we are able to reduce the number of membership queries also by making use of the properties of the synthesis application as will be described in the next chapter.

3 Minimally Adequate Synthesizer (MAS)

In order to adapt the inference algorithm described in the previous chapter to handle state machine synthesis, we first discuss the form of state machines and sequence diagrams.

A sequence diagram consists of participating *objects* and *messages* occurring between these objects. Objects are shown as vertical lines and messages as horizontal arrows extending from a sender object to a receiver object. Time flows from top to bottom. Spacing is irrelevant, i.e. only the order of messages matters, not the difference between them.

Let D be a sequence diagram describing a scenario with an instance of class C. The *trace* originating from D with respect to C is obtained as follows. Consider the vertical line corresponding to C. If the explicit deletion of the object (or other kinds of UML sequence diagram concepts that indicate reaching of a final state) is not shown in the end of the sequence diagram, add a entering arc with label $VOID$ to be the last arc in the sequence diagram. This arc represents the end of one sequence diagram and is needed by the synthesis algorithm. If necessary, augment the sequence diagram with additional unlabeled entering or leaving arcs so that the column corresponding to C consists of an altering sequence of leaving and entering arcs, starting with a leaving arc. Starting from the top, for two successive messages labeled e_i and e_j associated with C, add item (e_i, e_j) into the trace.

The deletion of an object in UML sequence diagram notation is visualized as a large cross at the end of a swim lane. When forming the trace for our synthesis algorithm, the deletion of the object could be specified by the last item $(e_n, DELETE)$ (instead of $(e_n, VOID)$). In the MAS algorithm, the end of a sequence diagram is marked with the $VOID$ event. From the point of view of the algorithm, the $DELETE$ events (or other events that indicate reaching of a final state, e.g., $EXIT$) would be handled analogously. When the MAS algorithm gives the user conjectures, the artificial end point indicators ($VOID$) and final states are omitted but the explicitly specified ones are included. Ignoring the final states in the former case can be argued, since the sequence diagrams do not always show the whole lifetime of the objects. In Chapter 4 we show examples of both cases.

A trace item (e_i, e_j) (with respect to C) implies that at a certain point during the execution of the system, C sends a message e_i to some other object and then responds to message e_j sent by another object. Thus, we map each sent message with an action (for simplicity, we do not distinguish activities from actions) performed in a state and each received message with an event that causes a state change (and possibly a new action to be performed). For example, corresponding to the trace item (e_i, e_j) there is a state with action "do:e_1" and an outgoing transition labelled e_j in the state machine. To simplify the presentation we ignore the specification of the receiver in the sending of a message, and assume that the message name uniquely determines the receiver.

The language accepted by a state machine consists of words where the actions related to the states take turns with the labels of transitions. Hence, the alphabet A of our inference algorithm consists of pairs (e_i, e_j), where e_i is an action of a state and e_j is a transition. The input traces are words over this alphabet, and the task of the inference algorithm is to output a finite state automaton which accepts the desired language over A. This inference problem is clearly equivalent to the synthesizing problem for state machines. For notational convenience, in the sequel we discuss the inference of such regular languages. There is a straightforward mapping from the resulting regular language to the corresponding state machine. The left sides of the symbols correspond to the states and the right hand sides correspond to the transitions of a state machine. The left hand sides of all the symbols related to the outgoing transitions of a state in a conjecture have to be the same. Thus, they can be placed as an action in a single state in the state machine. The right hand sides are used to label the outgoing transitions.

The designer starts by constructing typical sequence diagrams describing the behavior of the system. All traces from these sequence diagrams and their prefixes are stored in S in the beginning of the inference algorithm. No membership queries are needed since the traces itself are in the unknown language but all the proper prefixes are not. Indeed, if a string ends up with a symbol (e_i, e_j) with $e_j \neq VOID$, the membership query is not necessary since we know that the string in question cannot belong to the unknown language.

There are also other application specific features in the synthesis process which decrease the number of membership queries needed. Consider now a trace

$$e = (e_1, e_2)(e_3, e_4) \dots (e_{i-2}, e_{i-1})(e_i, e_{i+1}) \dots (e_{n-1}, e_n)$$

which is in the unknown language. Since e is in S, then so is its prefixes including $e = (e_1, e_2) \dots (e_{i-2}, e_{i-1})$. The left hand side e_i of (e_i, e_{i+1}) defines the action in the state reached by the subtrace $e = (e_1, e_2) \dots (e_{i-2}, e_{i-1})$. Hence, we do not have to make membership queries for strings $e = (e_1, e_2) \dots (e_{i-2}, e_{i-1})w$, where $w = (e_j, e_{j+1}) \dots (e_{m-1}, e_m)$ and $e_j \neq e_i$. Actually, we require even a stronger form of determinism: if $(e, f)(e', f')$ and $(e, f)(e'', f'')$ are any pair of consecutive symbols in strings belonging to the desired language, we insist that $e' = e''$.

This suggests that we should keep track of the actions related to the words known to be in the unknown language. This is easily done by maintaining a structure containing the pairs of consecutive symbols appearing in the words known to belong to the desired language. When the original algorithm MAT makes a membership query, we can check whether all the pairs of consecutive symbols in the string in question are among the valid ones. If not, we know that the answer to the query must be negative.

In a state machine, a transition from a state to another can also be a so called *completion transition* or a NULL transition. If a state has an outgoing NULL transition, it cannot have any other outgoing transitions. In our regular language to be inferred this means that if, for some e_i, there is a symbol $(e_i, NULL)$, then (e_i, e_j) must always imply $e_j = NULL$.

In what follows, we say that a regular language fulfilling the three conditions given (the existence of end marker, the determinism concerning the left hand sides of the terminal symbols, and the condition related to NULL) is *state deterministic*. A similar class of languages is defined in [17] in the connection with the synthesis algorithm used in SCED.

The algorithm MAS differs from MAT in various points:

1. the designer provides positive samples which are stored in T in the beginning of the algorithm,
2. "extend T to $(S \cup S \cdot A) \cdot R$ using membership queries" should now be interpreted as "extend T and fill in the entries by using membership queries for strings whose membership status cannot be concluded by the conditions for state determinism or by the contents of T",
3. the set of words known to be in the unknown language cannot violate the conditions for state determinism, and
4. the whole alphabet is not known beforehand.

It is clear that these differences have no effect to the correctness of the algorithm.

MAS can be given as follows:

Algorithm MAS
Input: A set I samples belonging to the unknown language.
Output: A finite state automaton accepting the unknown state deterministic language.
let $S = \{\lambda\} \cup I \cup$ *pref(I)* and $R = \{\lambda\}$;
check that I fulfills the conditions for state determinism;
let A be the set of symbols in appearing in the word of S;
extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;
repeat
  while T is not closed or not consistent **do begin**
    if T is not consistent **then**
      find s_1 and s_2 in S, a in A, and r in R such that $row(s_1) = row(s_2)$ and $T(s_1 \cdot a \cdot r) \neq T(s_2 \cdot a \cdot r)$;
      add $a \cdot r$ to R;
      extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;
    if T is not closed **then**
      find s_1 in S and a in A such that $row(s_1 \cdot a)$ is different from $row(s)$ for all s in S;
      add $s_1 \cdot a$ to S;
      extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;

end $\{while\}$
Construct $M(T)$ from T and conjecture M;
if teacher replies with a counterexample t **then**
add *pref*($\{t\}$) to S;
add new symbols in t to A;
extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;
add t to I and check that I fulfills the conditions for state determinism;
until the teacher replies *yes*;
Output M;

It is actually an advantage that we usually do not know the whole alphabet in the beginning. Namely, not knowing the whole alphabet prevents us from making membership queries before we have a positive sample showing the context in which a symbol can appear. This decreases the number of membership queries needed.

Notice that "add *pref*($\{t\}$) to S;" is followed by "extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;" in MAS. Hence, the possible change in A is indeed took into consideration.

To be able to answer the equivalence queries means that the user has an idea of the correct unknown language. Note, however, that a formal equivalence algorithm for finite automata is not applicable here. Namely, if the user had an exact definition for the automaton, she would not need to apply MAS at all. Instead, she uses some kind of heuristic algorithm to check whether or not the conjecture is correct.

4 Example runs of MAS

Next we examine couple of examples of using the MAS algorithm. Assume first that the user designs the behavior of an alarm clock. An example sequence diagram is shown in Figure 1. A trace corresponding to the sequence diagram in Figure 1 from the point of view of the *control unit* is the following:

(show current time, set new alarm time),
(show alarm time 5 secs, NULL),
(show current time, alarm time reached),
(buzzing,turn alarm off),
(show current time, set new alarm time),
(show alarm time 5 secs, NULL),
(show current time, VOID).

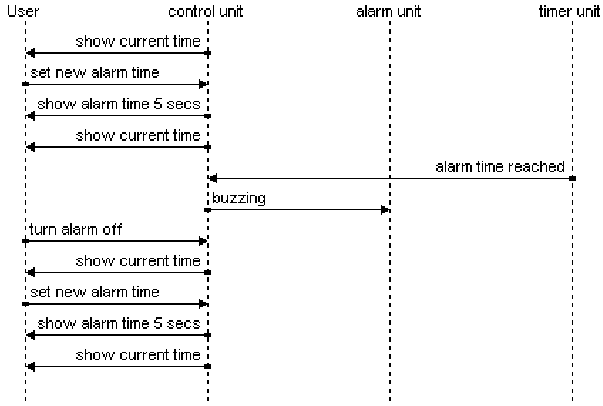

Figure 1: A sequence diagram describing an example usage of an alarm clock.

For simplicity we denote:

1. message *show current time* as s_ct,
2. message *set new alarm time* as set,
3. message *show alarm time 5 secs* as s_at,
4. message *alarm time reached* as reached,
5. message *buzzing* as buzz, and
6. message *turn alarm off* as off.

The alphabet of the alarm clock example consists of the following action/event pairs: (s_ct,set),(s_at,NULL),(s_ct,reached),(buzz,off), and (s_ct,VOID). The left column in Table 2 shows the strings for which MAS initially asks the membership queries. The elements of S are shown in the upper compartment. In the beginning, λ is the only member in R. For some of the rows that do not belong to S the membership queries always give 0 as a result. For example, let $s = (s\_ct, set)(s\_ct, set)$. Then $u = s \cdot r$ can not be in the unknown language U with any $r \in R$, since a pair with *set* in the action part always needs to follow the pair (s_ct,set). Thus, membership queries never need to be asked for the row (s_ct,set)(s_ct,set). Such rows in the observation table are marked with "-".

When making the membership queries in the first step, the values in the column λ result. The algorithm can conclude the answers, except for two rows $(s\_ct, VOID)$ and $(s\_ct, set)(s\_at, NULL)(s\_ct, VOID)$. Hence, it needs to consult the user. The user accepts both of the rows, hence allowing additional

T	r_1	r_2	r_3	r_4
λ	0	1	0	0
(s_ct,set)	0	0	0	0
(s_ct,set)(s_at,NULL)	0	1	0	1
(s_ct,set)(s_at,NULL)(s_ct,reached)	0	0	1	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)	0	1	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,VOID)	1	0	0	0
(s_at,NULL)	-			
(s_ct,reached)	0	0	0	0
(buzz,off)	-			
(s_ct,VOID)	1	0	0	0
(s_ct,set)(s_ct,set)	-			
(s_ct,set)(s_ct,reached)	-			
(s_ct,set)(buzz,off)	-			
(s_ct,set)(s_ct,VOID)	-			
(s_ct,set)(s_at,NULL)(s_ct,set)	0	0	0	0
(s_ct,set)(s_at,NULL)(s_at,NULL)	-			
(s_ct,set)(s_at,NULL)(buzz,off)	-			
(s_ct,set)(s_at,NULL)(s_ct,VOID)	1	0	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,set)	-			
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_at,NULL)	-			
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,reached)	-			
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,VOID)	-			
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,set)	0	0	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_at,NULL)	-			
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,reached)	0	0	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(buzz,off)	-			

Table 2: Observation table with columns $r_1 = \lambda$, $r_2 = (s\_ct, VOID)$, $r_3 = (buff, off)(s\_ct, VOID)$, $r_4 = (s\_ct, reached)(buzz, off)(s\_ct, VOID)$.

behavior for the alarm clock.

The observation table after the first step is closed. The algorithm then checks whether it is consistent. For $s_1 = (s\_ct, set)(s\_at, NULL)(s\_ct, reached)$, $s_2 = (s\_ct, set)(s\_at, NULL)$ $(s\_ct, reached)(buzz, off)$, and $a = (s\_ct, VOID)$, $row(s_1) = row(s_2)$ but $row(s_1 \cdot a) \neq row(s_2 \cdot a)$. Hence, the observation table is not consistent. The algorithm then adds $a = (s\_ct, VOID)$ to R and makes membership queries. This time, the algorithm can conclude all the answers. The values of the queries are shown in the column labeled $(s\_ct, VOID)$.

The observation table after the second step is again closed but not consistent, since for $s_1 = (s\_ct, set)$, $s_2 = (s\_ct, set)(s\_at, NULL)(s\_ct, reached)$, and $a = (buzz, off)$, $row(s_1) = row(s_2)$ but $row(s_1 \cdot a) \neq row(s_2 \cdot a)$. The algorithm adds $a = (buzz, off)(s\_ct, VOID)$ to R and makes membership queries again. The user needs to be consulted once: determining whether the row $(s\_ct, reached)$ should be accepted or not. This case indicates a situation where the alarm clock starts buzzing even when the alarm is set off (assuming that the alarm is off in the beginning). The user thus does not accept this row. The values of the third step are shown in the column labeled $(buzz, off)(s\_ct, VOID)$.

The observation table is closed but not consistent also after the third step. Rows $s_1 = (s\_ct, set)(s\_at, NULL)$ and $s_2 = (s\_ct, set)(s\_at, NULL)(s\_ct, reached)(buzz, off)$ with $a = (s\_ct, reached)$ prove the inconsistency. Hence, MAS adds a string $(s\_ct, reached)(buzz, off)(s\_ct, VOID)$ to R and makes membership queries. This time, the algorithm is able to conclude all the answers. The values of the queries are shown in the last column.

After the fourth step, the observation table is closed and consistent. Rows $s_1 = \lambda$ and $s_2 = (s\_ct, set)(s\_at, NULL)(s\_ct, reached)(buzz, off)$ are the only similar rows in S, and for all $a \in A$, $row(s_1 \cdot a) = row(s_2 \cdot a)$.

After getting a closed and consistent observation table, the MAS algorithm makes a conjecture of the acceptor M with an initial state $(0, 1, 0, 0)$ and one final state $(1, 0, 0, 0)$. The conjecture is depicted in Figure 2 and the state machine corresponding to the conjecture is shown in Figure 3. The user accepts the state machine and the execution of the algorithm halts. Note how the final state is deleted from the state machine. Note also that the user was consulted only three times because of the membership queries.

Next we will run the MAS algorithm again on the alarm clock example, but now assuming different consultancy from the user. Whenever the algorithm cannot conclude the answer to a membership query, it asks the user. Suppose now that the user accepts the rows in all cases. Table 3 shows the correspond-

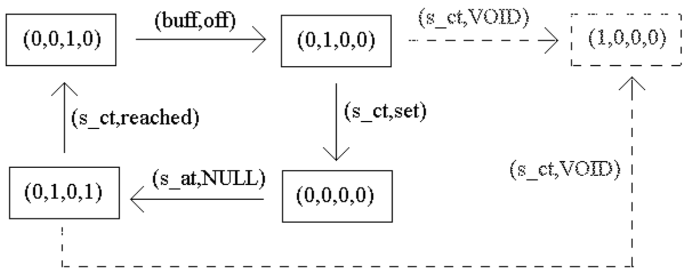

Figure 2: A conjecture.

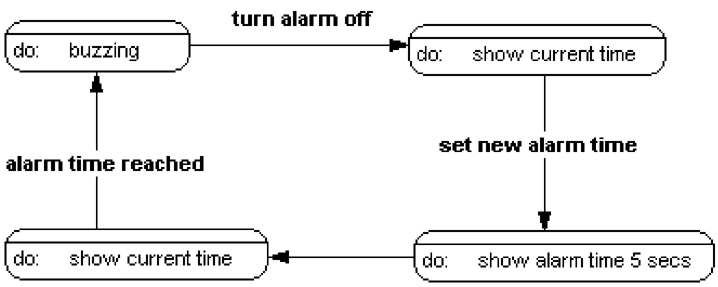

Figure 3: The state machine corresponding to the conjecture in Figure 2.

ing observation table. After the first three steps (columns λ, $(s\_ct, VOID)$, and $(s\_at, NULL)(s\_ct, VOID)$) the observation table is closed and consistent. Hence, the algorithm makes a conjecture of the acceptor M_1, which has an initial state $(0,1,0)$ and a final state $(1,0,0)$. The conjectured state machine M_1 is shown in Figure 4.

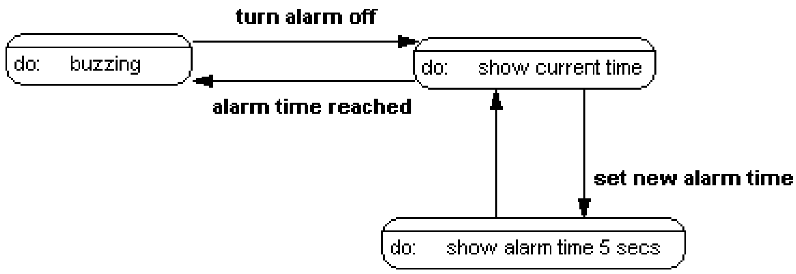

Figure 4: The conjectured state machine M_1.

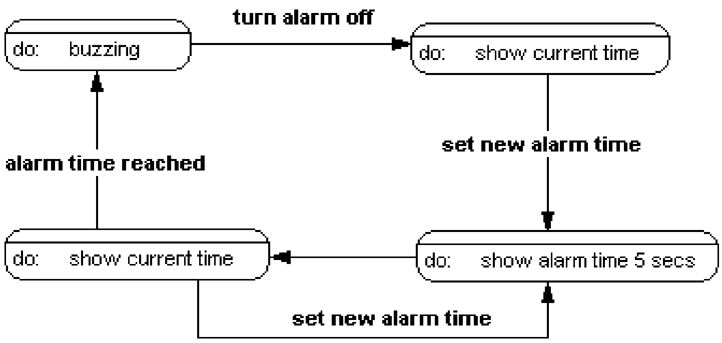

Figure 5: The conjectured state machine M_2.

The user is not satisfied with the state machine since it allows buzzing even if the alarm was not set on. Thus, the user gives a counterexample $(s\_ct, reached)(buzz, off)(s\_ct, VOID)$, which belongs to a language accepted by M_1 but should not be allowed by the correct state machine. Then the algorithm adds the rows $s_1 = (s\_ct, reached)(buzz, off)(s\_ct, VOID)$, $s_2 = (s\_ct, reached)(buzz, off)$, and $s_3 = (s\_ct, reached)$ to S. These rows are shown in the third compartment of the observation table in Table 3. Also rows $s_i \cdot a, i = 1 \ldots 3$, for all $a \in A$ need to be added to the observation table (the bottom compartment). The values of the membership queries for added rows never need to be asked since they always have to be 0. After the fourth and fifth steps the observation table is closed and

T	r_1	r_2	r_3	r_4	r_5
λ	0	1	0	0	0
(s_ct,set)	0	0	1	0	0
(s_ct,set)(s_at,NULL)	0	1	0	0	1
(s_ct,set)(s_at,NULL)(s_ct,reached)	0	0	0	1	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)	0	1	0	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,VOID)	1	0	0	0	0
(s_at,NULL)	-				
(s_ct,reached)	0	0	0	0	0
(buzz,off)	-				
(s_ct,VOID)	1	0	0	0	0
(s_ct,set)(s_ct,set)	-				
(s_ct,set)(s_ct,reached)	-				
(s_ct,set)(buzz,off)	-				
(s_ct,set)(s_ct,VOID)	-				
(s_ct,set)(s_at,NULL)(s_ct,set)	0	0	1	0	0
(s_ct,set)(s_at,NULL)(s_at,NULL)	-				
(s_ct,set)(s_at,NULL)(buzz,off)	-				
(s_ct,set)(s_at,NULL)(s_ct,VOID)	1	0	0	0	
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,set)	-				
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_at,NULL)	-				
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,reached)	-				
(s_ct,set)(s_at,NULL)(s_ct,reached)(s_ct,VOID)	-				
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,set)	0	0	1	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_at,NULL)	-				
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,reached)	0	0	0	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(buzz,off)	-				
(s_ct,reached)(buzz,off)(s_ct,VOID)			-		
(s_ct,reached)(buzz,off)			-		
(s_ct,reached)			-		
(s_ct,reached)(buzz,off)(s_ct,VOID)(s_ct,set)			-		
(s_ct,reached)(buzz,off)(s_ct,VOID)(s_at,NULL)			-		
...					

Table 3: Observation table with columns $r_1 = \lambda$, $r_2 = (s\_ct, VOID)$, $r_3 = (s\_at, NULL)(s\_ct, VOID)$, $r_4 = (buzz, off)(s\_ct, VOID)$, and $r_5 = (s\_ct, reached)(buzz, off)(s\_ct, VOID)$.

consistent. Thus, the algorithm makes a second conjecture of the acceptor M_2. The conjecture M_2 has an initial state $(0, 1, 0, 0, 0)$ and a final state $(1, 0, 0, 0, 0)$. Figure 5 shows the conjectured state machine M_2. The user accepts this state machine and the algorithm halts.

Next we study an example of withdrawing money using an ATM. This example is from [20], p. 177. The following four traces describe different cases of using an ATM from the point of view of the bank:

1.
(NULL, verify card with bank),
(verify card number, valid bank account),
(verify password, valid password),
(update account, transaction success),
(transaction OK, EXIT)

2.
(NULL, verify card with bank),
(verify card number, valid bank account),
(verify password, valid password),
(update account, transaction failure),
(transaction error, EXIT)

3.
(NULL, verify card with bank),
(verify card number, valid bank account),
(verify password, bad password),
(password error, EXIT)

4.
(NULL, verify card with bank),
(verify card number, bad bank account),
(account error, EXIT).

Assume then that the user first gives only traces 1 and 2 to the MAS algorithm. The algorithm does not need to ask any guidance from the user and produces a conjecture depicted in Figure 6.

The user does not accept the conjecture and gives traces 3 and 4 as counterexamples. The MAS algorithm is again able to produce a conjecture without user's guidance. The conjectured state machine is shown in Figure 7. The user accepts the automaton.

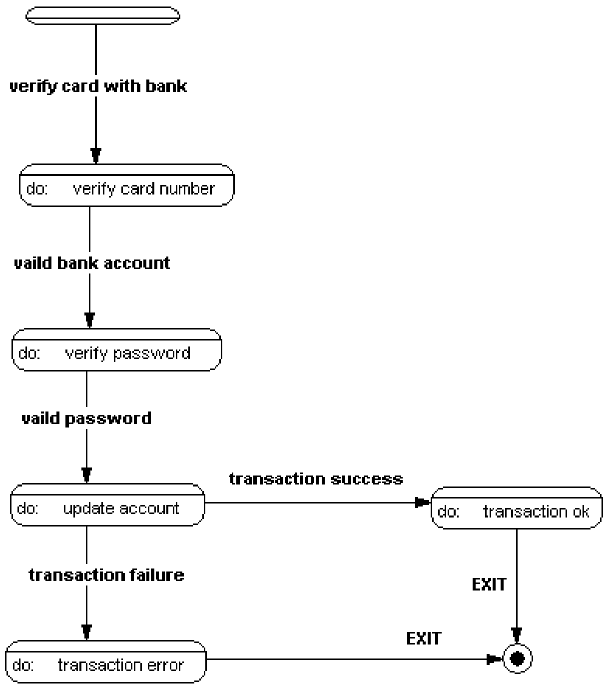

Figure 6: The first conjectured automaton for the bank. The user rejects it.

The example runs discussed in this chapter illustrate the obvious advantages of MAS over MAT. The assumption of state determinism decreases both the number of membership queries needed and the space complexity of the algorithm. Although we are not able to give exact bounds for the time and space requirements, it is clear that MAS outperforms MAT and provides a practical method for synthesizing state machines.

5 Discussion

UML provides a large set of diagrams that can be used to model different aspects of a software system. Since these diagrams contain overlapping information, automated support for constructing the diagrams can be provided. In behavioral modeling, such support can be offered for constructing statechart diagrams from sequence diagrams.

UML sequence diagrams are used for showing examples of interaction between several objects, while statechart diagrams are used for specifying the full dynamic

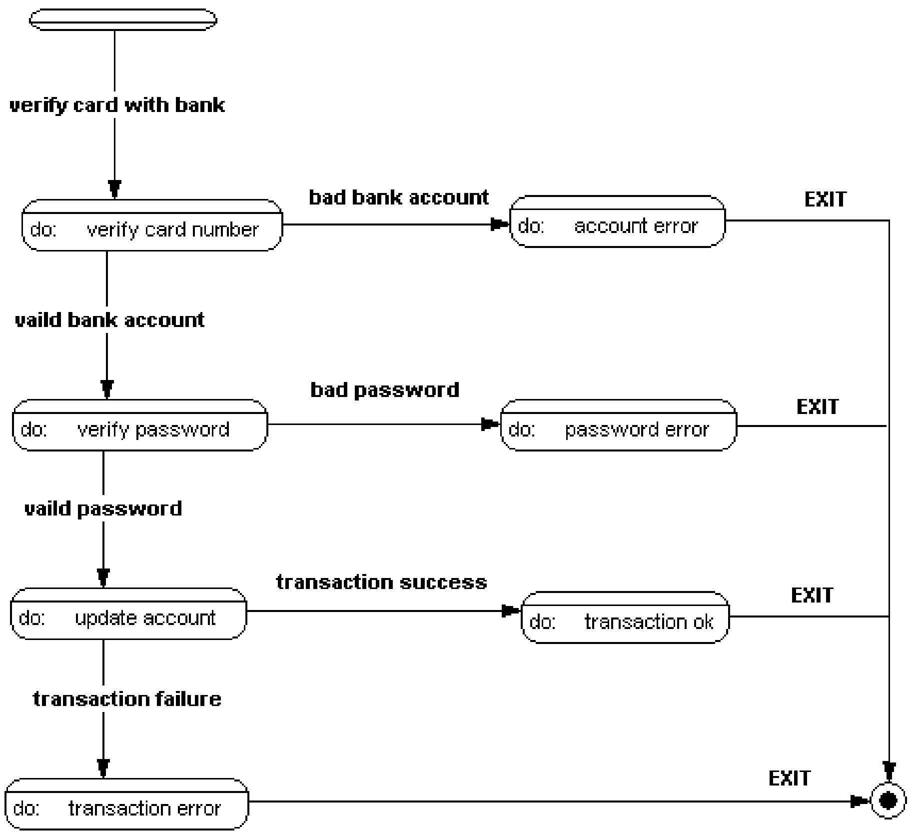

Figure 7: The second conjectured state machine for the bank. The user accepts it.

behavior of a single class of objects. Hence, the constructed set of sequence diagrams might be incomplete and there might be inaccuracies in the contents of the sequence diagrams. Generating a statechart diagram automatically from such a set of sequence diagrams might lead to undesired results. For example, a generated statechart machine might accept more traces than expressed as the set of sequence diagrams. In some cases, such generalizations might be harmful, i.e. the state machine is "overgeneralized". Even if tool support is provided, correcting the statechart diagram manually afterwards is undesirable.

In this paper we discuss a method for synthesizing a state machine from UML sequence diagrams in an interactive manner. Whenever needed, the used algorithm asks the designer for guidance during the synthesis process. This avoids overgeneralization.

As a general rule, the user can be advised to give several initial traces as sequence diagrams, and several counterexamples, if possible. The more there are positive samples, the easier it is for the algorithm to fill in the observation table without making explicit membership queries.

One way to further decrease the number of membership queries is to give additional information to the algorithm. As an example, consider the alarm clock. It is obvious to the user that the message "alarm time reached" cannot occur after "turn alarm off". Hence, if a word contains a subword $(buzz, off)(s\_ct, reached)$, it cannot belong to the unknown language. It might, however, be unreasonable to expect that the user can list such invalid subwords beforehand. A user-friendly way to transfer this information to the algorithm is to give to the user a possibility to mark any subword of a membership query as invalid. This guarantees that the algorithm does not make membership queries with the same invalid subword more than once. We can say that such a possibility would increase the generality of the answers: instead of neglecting a single word from the unknown language, we can neglect a whole sublanguage of words containing the invalid pattern.

The concept of state determinism links the present work to the one done in the SCED project [14, 15, 23]. There is a close connection with the membership queries of MAS and joining states in the exhaustive inference algorithm used in SCED. By making conjectures, MAS allows a natural user interface for controlling the inference process. A somewhat similar interface can be used in SCED if the algorithm asks the user to confirm joining of states.

Angluin's minimally adequate teacher is a theoretical construction, an oracle, which always works correctly. In our model, minimally adequate teacher is a human being who can make errors and who might also change her mind during the design process. The system should examine the correctness of the answer provided by the user: they must obey the conditions of state determinism and they cannot contradict each others. A more demanding task is to allow the user to change her earlier answers. The details of efficient backtracking methods are left open here. They are a subject of our further studies.

References

[1] D. Angluin: Inference of reversible languages. J. ACM. **29**, 741–765 (1982)

[2] D. Angluin: Learning regular sets from queries and counterexamples. Inf. Comput. **75**, 87–106 (1987)

[3] D. Angluin: Negative results for equivalence queries. Mach. Learn. **5**, 121–150 (1990)

[4] D. Angluin, C.H. Smith: Inductive inference: theory and methods. ACM Comput. Surv. **15**, 237–269 (1983)

[5] A.W. Biermann, R. Krishnaswamy: Constructing programs from example computations. IEEE Trans. Comput. **C-21**, 122–136 (1975)

[6] A. Birkendorf, A. Böker, H.U. Simon: Learning deterministic finite automata from smallest counterexamples. In Proc. 9th ACM/SIAM Symp. Discr. Alg. (SODA), Baltimore, USA, January 1999, pp. 599–608

[7] D. Harel: Statecharts: A visual formalism for complex systems: Sci. Comput. Program. **8**, 231–274 (1987)

[8] E.M. Gold: Language identification in the limit. Inform. Contr. **10**, 447–474 (1967)

[9] E.M. Gold: Complexity of automaton identification from given data. Inform. Contr. **37**, 302–320 (1978)

[10] M.A. Harrison: Introduction to Formal Language Theory. Addison-Wesley 1978

[11] O.H. Ibarra, T. Jiang: Learning regular languages from counterexamples. J. Comput. Syst. Sci. **43**, 299–316 (1991)

[12] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, C. Chen: Formal approach to scenario analysis. IEEE Softw. **11**, 33–41 (1994)

[13] Z.120 ITU-T Recommendation Z.120: Message Sequence Chart (MSC). ITU-T, Geneva, 1996.

[14] K. Koskimies, E. Mäkinen: Automatic synthesis of state machines from trace diagrams. Softw. Pract. Exper. **24**, 643–658 (1994)

[15] K. Koskimies, T. Systä, J. Tuomi, T. Männistö: Automated support for modeling OO software. IEEE Softw. **15**, 87–94 (1998)

[16] S. Leue, L. Mehrmann, M. Rezai: Synthesizing software architecture descriptions from message sequence chart specification. In Proc. of the 13th IEEE International Conference on Automated Software Engineering (ASE98), Honolulu, USA, October 1998, pp. 192–195

[17] E. Mäkinen: On the relationship between diagram synthesis and grammatical inference. Intern. J. Computer Math. **52**, 129–137 (1994)

[18] Rational Software Corporation: The Unified Modeling Language Notation Guide v.1.3. [http://www.rational.com], January 1999

[19] R.L. Rivest, R.E. Schapire: Inference of finite automata using homing sequences. Inf. Comput. **103**, 299–347 (1993)

[20] J. Rumbaugh, J. Jacobson, G. Booch: The Unified Modeling Reference Manual. Addison-Wesley, 1999

[21] S. Schönberger, R. Keller, I. Khriss: Algorithmic support for transformations in object-oriented software development. Technical Report GELO-83, University of Montreal, 1998

[22] S. Somé, R. Dssouli, J. Vaucher: ¿From scenarios to automata: building specifications from users requirements. APSEC'95, Brisbane, Australia, 1995

[23] T. Systä: Static and Dynamic Reverse Engineering Techniques for Java Software Systems. Ph. D. Dissertation, Dept. of Computer and Information Sciences, University of Tampere, May 2000