IMPLEMENTING MINIMALLY ADEQUATE SYNTHESIZER

Erkki Mäkinen and Tarja Systä

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES
UNIVERSITY OF TAMPERE

REPORT A-2000-9

UNIVERSITY OF TAMPERE
DEPARTMENT OF COMPUTER AND
INFORMATION SCIENCES
SERIES OF PUBLICATIONS A
A-2000-9, JUNE 2000
REVISED AUGUST 2000

IMPLEMENTING MINIMALLY ADEQUATE SYNTHESIZER

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 951-44-4869-3
ISSN 1457-2060

Implementing Minimally Adequate Synthesizer [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`

Abstract. Minimally Adequate Synthesizer (MAS) is an algorithm that synthesizes UML statechart diagrams from sequence diagrams. It follows Angluin's framework of minimally adequate teacher to infer the desired statechart diagram with the help of membership and equivalence queries. The purpose of this paper is to discuss problems related to an practical implementation of MAS.

In order to be able to handle most membership queries without consulting the user, MAS maintains a structure of strings already known to belong to the unknown language. User's erroneous answers and mind changes require an implementation of the structure that is capable of handling backtracking. We sketch a trie based implementation to allow this.

Moreover, we discuss the interaction between the user and the algorithm as a medium of improving the algorithm and further decreasing the number of membership queries needed. Furthermore, we show how MAS can be used to synthesize sequence diagrams into an edited or manually constructed statechart diagram. Finally, we extend MAS to handle UML sequence diagrams containing also other concepts than objects and message calls.

Keywords and Phrases. Software design; Minimally adequate teacher; Synthesis algorithm; UML; Sequence diagram; Statechart diagram.

1 Introduction

The Unified Modeling Language (UML) [14, 15] has been accepted as an industrial standard for specifying, visualizing, understanding, and documenting object-oriented software systems. In object-oriented analysis and design (OOAD), dynamic modeling aims at the description of the dynamic behavior of objects using

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

a variant of a finite state machine. The UML variant of a state machine is called *a statechart diagram*. The semantics and notation used in UML follow Harel's statecharts [6]. 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*. Automated support for constructing statechart diagrams from sequence diagrams provides considerable help for the designer. It helps her to quickly shift from constructing example sequence diagrams to tuning the final specification of the behavior as a statechart diagram. Automatic generation of statechart diagrams from variations of Message Sequence Charts (MSCs) [8] (e.g., UML sequence diagrams) is implemented in several tools [7, 9, 10, 16, 17].

MAS [11] is an algorithm that synthesizes UML statecharts diagrams from sequence diagrams. It uses Angluin's [2] framework of minimally adequate teacher to infer the desired statechart diagram with the help of membership and equivalence queries. MAS is made practically applicable by interpreting a UML sequence diagram as sentences of a language. This interpretation allows the algorithm to conclude the correct answer to most of the membership queries without consulting the teacher, i.e. the designer. In order to do so, the algorithm maintains a trie structure containing all the strings that are known to belong to the language to be inferred. The implementation of the structure would be easy if the positive samples once inserted to the structure were always kept there until the algorithm halts. However, it is reasonable to believe that the designer occasionally makes errors or she may want to change her mind during the design process. Hence, deletions from the structure are inevitable. Deleting a string from the structure might have its effects to several decisions made by the algorithm. Namely, it is possible that the algorithm has concluded, based on the later deleted string, that certain other strings cannot belong to the unknown language. Allowing deletions from the structure makes its maintenance problematic. One of the purposes of this paper is to show how the data structures of MAS can be efficiently maintained under arbitrary deletions.

Interaction with the user is the main advantage of MAS over previously known synthesis algorithms. Totally automatic synthesis algorithms, e.g., the algorithm used in SCED [9], may result in a state machine that contains undesired generalizations. Because MAS consults the user during the synthesis process, the user can be confident that such generalizations do not appear in the resulting statechart diagram. Moreover, the user can help the synthesis process, for example, by marking certain (sub)paths in the statechart diagram as forbidden. This guarantees that the algorithm does not perform queries containing such a subpath more than once, and hence, decreases the number of membership queries needed. We also discuss the user's possibility to edit the resulting statechart diagram. Changing the statechart diagram means that the corresponding changes must be performed in the data structures from which the statechart diagram was concluded. Moreover, we will show how sequence diagrams can be synthesized to an existing, possibly manually constructed, statechart diagram. We also consider various ways to support the user when she is providing a counterexample after rejecting a conjecture.

This paper is organized as follows. In Chapter 2 we define the concept of state determinism required from the resulting statechart diagram. In Chapter 3, we show how state determinism is taken into account in the synthesis algorithm. Chapter 4 deals with the data structures used in MAS, especially those allowing backtracking caused by user's erroneous answers and mind changes. Chapter 5 introduces various methods for transfering additional information to the algorithm in order to decrease the number of membership queries, and study the operations needed to maintain the consistence between the statechart diagram and the observation table when the user is allowed to edit the statechart diagram. In Chapter 6 we extend MAS to handle sequence diagrams containing also other concepts than objects and message calls. Finally, in Chapter 7 we give some concluding remarks.

2 Mapping between sequence and statechart diagrams

A UML sequence diagram consists of participating *objects* and *messages* occurring between these objects. Objects are shown as vertical lines called *lifelines* 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. UML sequence diagrams may also contain various other items. These items are discussed in Chapter 6.

Let D be a sequence diagram describing a scenario with an instance I of class C. The *trace* originating from D with respect to I is obtained as follows. Consider the vertical line corresponding to I. Starting from the top, for two successive messages labeled e_i and e_j associated with I, where e_i is a sent message and e_j is a received message, add item (e_i, e_j) into the trace. If either of e_i or e_j is missing, then add $NULL$ instead. 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 at the end of the sequence diagram, let the right hand side of the last pair be $VOID$. Note that the semantics of the original sequence diagram is preserved.

Note, that the original sequence diagram remains unchanged. The additions needed for the statechart synthesis are made to the trace, which is an internal data structure that stores the information given in sequence diagrams.

An operation call for an object is shown with an arriving call arc in a sequence diagram. The corresponding return from the operation is shown with a dashed leaving arc, which is called a *return message* in UML. All the leaving arcs between them are internal calls of operations of other objects, and all arriving arcs are returning counterparts of these calls. Hence, the trace of the operation call consists of the internal calls shown by leaving arcs. Operation synthesis does not cause any restrictions or extensions to MAS; only the sequence diagram items to be read vary. While the synthesis of a statechart diagram for an object is based on all the sequence diagram items that involve the object, the operation synthesis is based on only those items between the operation call and the corresponding return value.

The destruction of an object in UML sequence diagram notation is visualized as a large cross at the end of a lifeline. It is placed at the message that causes the object to be destroyed or may be shown with the stereotype << *destroy* >> [15]. When forming the trace for our synthesis algorithm, the deletion of the object is specified by the last item $(e_n, DESTROYED)$ (instead of $(e_n, VOID)$). For MAS, the end of a sequence diagram is marked with the $VOID$ event, if not defined otherwise. From the point of view of the algorithm, the $DESTROYED$ messages (or other indicators of reaching a final state) are handled analogously. When MAS 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.

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 reacts 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. If the receiver of the message can be uniquely determined from the sequence diagrams, the name of the receiver is ignored for simplicity. Otherwise, it is included in the name of the message (e.g., *Action1 TO Object1*). The receivers cannot be uniquely determined, for instance, in the case of a broadcast. Analogously, the names of the senders are included in the names of transitions when the senders cannot be uniquely determined (e.g., *Message1*

FROM Object1).

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 automaton which accepts the desired language over A. This inference problem is clearly equivalent to the synthesizing problem for statechart diagrams. 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 statechart diagram. The left hand sides of the symbols correspond to the states and the right hand sides correspond to the transitions of a statechart diagram. 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 statechart diagram. 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 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 that decrease the number of membership queries needed. Consider now a trace

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

which is in the unknown language. Since e is in S, then so is its prefixes including $e = (e_1, e_2) \ldots (e_{i-2}, e_{i-1})$. The left hand side e_i of (e_i, e_{i+1}) defines the action in the state reached by the subtrace $e = (e_1, e_2) \ldots (e_{i-2}, e_{i-1})$. Hence, we do not have to make membership queries for strings $e = (e_1, e_2) \ldots (e_{i-2}, e_{i-1})w$, where $w = (e_j, e_{j+1}) \ldots (e_{m-1}, e_m)$ and $e_j \neq e_i$.

This suggests that we should maintain 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. Such a structure is discussed in Chapter 4.

In a UML statechart diagram, a transition from a state to another state can also be a so called *completion transition*. If a state has an outgoing completion transition, it cannot have any other outgoing transitions. In UML, a completion transition without a guard (i.e., a condition that must hold to allow the transition to fire) is implicitly triggered by the completion of any internal activity in a state [15] (p. 479). UML allows a completion transition without a guard and a transition with an event trigger to be attached to a same state as leaving transitions, although it is not a common practice. However, if the activity in the state was an instantaneous one (i.e., an action), then the labeled transition would never get a chance to fire. Hence, in MAS we do not allow a completion transition and a labeled transition to be leaving transition of the same state. Two leaving completion transitions, in turn, would result in a nondeterministic state.

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*.

3 An overview of MAS

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

1. she must decide whether a given behavior is possible in the system she is implementing (the membership queries)

2. she must accept or reject the output statechart diagram, and moreover, if she rejects, a counterexample from the the symmetric difference of the languages accepted by the output statechart diagram and the unknown statechart diagram must be given (the equivalence queries).

In a very early stage of the software design process, "I don't know" answers to membership queries might also be useful. However, we do not allow them here since their proper treatment would make the synthesis algorithm considerably more complicated. The reader is referred to [3] and the references given there for theoretical results concerning so called limited membership queries to which "I don't know" answers are possible.

MAS contains an *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 of the symbols appearing in the strings so far known to belong to the desired, and S is a prefix-closed set of strings in A^*. 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 $T(u) = 0$. The bit string on the row labeled x in T is denoted by $row(x)$. It is *non-null*, if at least one of its entries is 1. An observation table is said to be *closed* if for each t in $S \cdot A$ with a non-null row in the table 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, $row(s_1 \cdot a) = row(s_2 \cdot a)$.

The inference algorithm starts with $S = R = \emptyset$ and first asks membership queries for λ (the empty string) and for all symbols a in A. T is updated by the answers of these queries. 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 the 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 automaton $M(T)$ can be constructed by setting

- the state set as $Q = \{row(s) \mid s \in s\}$
- the inital 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 set of final states as $F = \{row(s) \mid s \in S \text{ and } T(s) = 1\}$
- the transition relation as $\delta(row(s), a) = row(s \cdot a)$ for each $s \in S$ and $a \in A$ such that $s \cdot a \cdot t$ is known to be in the desired language for some $t \in A^*$.

On the contrary to Angluin's [2] original presentation we define the transition relation by using only words known to be in the desired language. Omitting the other transitions cannot affect the correctness of the algorithm.

Following [11] we can now give MAS 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 express as "extend T to $(S \cup S \cdot A) \cdot R$ using membership queries".

Algorithm MAS
Input: A set of sequence diagrams.
Output: A finite automaton.
begin
  let I be the set of traces obtained from the input sequence diagrams;
  let $S = \{\lambda\} \cup I \cup$ *pref(I)* and $R = \{\lambda\}$;
  let A be the set of symbols appearing in the input sequence diagrams (A is augmented whenever a new symbol is found in the information provided by the user);
  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;
  until the teacher replies *yes*;
  Output M;
end; { MAS }

Whenever the algorithm learns a new element to be in the desired language, it checks that it is not contradicting with the previously known elements with respect the general conditions given above.

The above algorithm outputs a finite automaton. Actually, we need a statechart diagram which is obtained by fine tuning the output automaton as described elsewhere in this paper and in [11]. The output finite automaton is called the *underlying finite automaton* of the resulting statechart diagram.

Note that the execution of the step "extend T to $(S \cup S \cdot A) \cdot R$ using membership queries;" can often be slightly speeded up by handling longer strings first. An affirmative answer to a membership query concerning a long string is likely to be more informative than an affirmative answer concerning a short string, simply because longer strings usually contain more different pairs of consecutive symbols. These pairs can be later used to check the validity of other strings.

Implementing MAS in its given basic form is straightforward. However, any implementation of MAS should take care of the consistency of the information stored. This means that each answer given by the user should be checked against the information stored in the observation table and in the other data structured to be introduced later, and any inconsistency found should be reported.

In the following chapters we improve the algorithm to better meet the requirements of practical object-oriented design process. This makes the implementation somewhat more difficult. The problems found are settled in the following chapters.

4 Data structures allowing backtracking

It is possible that the user changes her mind or accidentally gives an incorrect answer to a query. However, MAS always operates as if all the information so far obtained were correct. Hence, we do not apply such concepts as "malicious omissions and errors" or the corresponding inference algorithms introducted by Angluin et al. in [3]. This means that we have to update the observation table when incorrect information is detected in order to keep the observation table and auxiliary data structures consistent. This task is the subject of this chapter.

Trie is a data structure for maintaining a set of strings over a given alphabet of k symbols. The set is represented as a k-ary tree consisting of all the prefixes of the strings in the set (for details, see e.g. [12]).

MAS maintains a trie containing the strings known to be in the desired language. This trie is referred to as W. Initially, W contains the information related to the input set I. New information is inserting in W when the user gives a positive answer to a membership query, or when she gives an counterexample not belonging to the language accepted by the conjectured automaton.

Suppose now that MAS is looking for the correct value for an entry in the observation table T. It first checks that the string (say w) in question ends up with a symbol of the form $(e, VOID)$. If so, it accesses W and compares the existing links against w. There are three different possibilities:

1. the links can be traversed to a leaf which means that the trie contains w; the correct entry in T is, of course, 1,

2. w is of the form $w = w_1(e_i, f)w_2$, where w_1 is the longest possible common prefix of w and any string (say y) in W, and y continues with a symbol (e_j, g) where $e_i \neq e_j$; now we know that w cannot belong to the desired language and the correct entry in T is 0, and

3. w is of the form $w = w_1(e_i, f)w_2$, where w_1 is the longest possible common prefix of w and any string y in W, and y continues with a symbol (e_j, g) where $e_i = e_j$ (and hence, $f \neq g$); MAS cannot conclude the correct entry in T, and a membership query is needed.

In the case (2) above, we conclude that w cannot be in the desired language. We prepare ourselves to possible backtrack operations by maintaining lists of pointers to the concluded observation table entries in the nodes of W. If a trie node is later deleted, the observation table entries concluded can be easily found by following the pointers.

Inserting new elements to the trie is as straighforward as accessing. However, problems arise when we have to delete a string from the structure because of a found error or of a mind change of the user. The deletion itself is easy, but it is possible that we have updated the observation table based on the existence of a string, which now turns out to be erroneous. Hence, we have to check that all observation table entries are concluded from existing trie elements also after the deletion.

We use a trie structure in which we are able to efficiently backtrack, and then re-update the observation table if necessary, i.e., we use a structure that resembles so called persistent data structures.

In an ordinary data structure an update destroys the old version, leaving only the new version available for use. Such a data structure is called *ephemeral*. A data structure is *persistent*, if it supports operations not only in the newest version but also in all the previous versions of the structure. Persistent data structures are systematically developed by Overmars [13] and Driscoll et al. [5].

There are straightforward, but inefficient, solutions to make a data structure persistent: we can either store a copy of each version of the structure or store the sequence of update operations and then build up the version desired from scratch. Better methods for arbitrary linked structures are developed in [5]. We shall not discuss these general methods here. Instead, we descride a method of making tries persistent in the sense needed in MAS.

Figure 1 shows the contents of W when the strings $(a,b)(c,d)(e,f)(g,VOID)$, $(a,b)(c,d)(e,VOID)$, and $(a,b)(c,h)(i,VOID)$ (in this order) are known to be in the unknown language. Hence, the input set was $I = \{(a,b)(c,d)(e,f)(g,VOID)\}$, and after that, the user has indicated by answering membership queries or by given counterexamples that also $(a,b)(c,d)(e,VOID)$, and $(a,b)(c,h)(i,VOID)$ are in the unknown language. Related to the nodes c, e and i there is a list of pointers to the observation table entries given as observation table coordinates of the form (xi, yi).

In what follows, the set containing the strings known to be in the desired language is denoted as U. We index the elements in U according to the order they are inserted in U. If there are n, $n > 1$, elements in the input set I, they are arbitrarily indexed by the numbers $1, \ldots, n$.

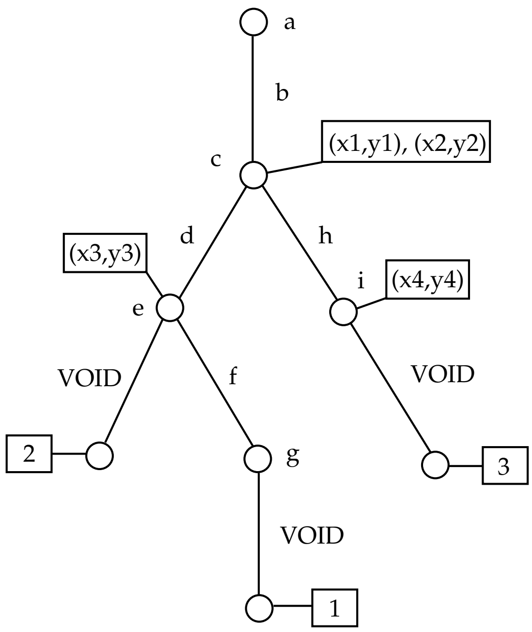

Figure 1: A sample content of W.

Consider now what happens when a string is deleted from U. First, the corresponding element is deleted from W. The algorithm might have concluded an affirmative answer to a membership query because of the (now ceased) existence of the string in question. Now, this entry in the observation table must be updated to be 0. The possible need for reconsidering the value of a table entry can be concluded by checking the lists of coordinates along the path presenting the element to be deleted from the trie structure. Note, however, that a change in the value of an observation table entry is not necessarily needed. The string corresponding to the observation table entry in question may contain other substrings, from which a negative answer can be concluded (or the user can confirm by answering a membership query that the entry should be kept unchanged). It is clear that a closed and consistent observation table can turn out to be not closed and/or not consistent as a result of a change in U. This, in turn, can imply further membership queries.

When a string w of U is deleted from the observation table, we also have to delete any such prefix of w that is not a prefix of any string left in the observation table.

Notice that when a table entry is changed from 0 to 1, no further changes are needed. The algorithm only checks that the new piece of information is consistent with the rest of the information stored in the data structures.

Deleting a string from U causes changes also in the alphabet, if the only appearance of a symbol (say (a, b)) was in the deleted string. This implies deletions in the observation table: each row and column labeled with a string containing (a, b) should also be deleted.

5 Interaction between the user and MAS

The purpose of this chapter is to introduce various methods for transfering additional information to the synthesis algorithm in order to further decrease the number of membership queries.

First, we describe how the user can mark "forbidden sequences" in the strings appearing in membership queries. The algorithm maintains the set of all marked substrings and is so prevented to make a new query with the same forbidden substring. Second, we deal with the problem of keeping the statechart diagram and the observation table consistent when the user edits the statechart diagram. Third, we consider the problems related to the user's task to provide counterexamples if she does not accept the automaton conjectured by the algorithm.

5.1 Forbidden substrings

It is obvious to the user of MAS that certain sequences of messages cannot take place, or equivalently, that certain substrings are not possible in the words belonging to the desired language. It is, however, quite 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. Such a possibility increases 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.

We need another trie (referred to as F), which contains the forbidden substrings. It is accessed if the correct answer cannot be concluded based on the information stored in W. In the nodes of F, there are lists of pointers to the observation table entries whose values are concluded from the trie element in question. Since deletions should be possible also from F, it is maintained analogously to W, i.e., a deletion may cause changes in the observation table entry values.

Checking whether a given string contains any of F's strings as its substring, is an instance of a string matching problem where several patterns are searched from a single text. This problem can be efficiently solved by the Aho-Corasick algorithm [1].

Note that F can contain strings of any length (> 1), and the set presented by F is prefix-free. Note also that the content of F must be consistent with the information stored in W and in the observation table, i.e., F should not contain a string appearing in W as a substring.

5.2 Editing the statechart diagram

Answering equivalence queries and providing a sufficient set of sequence diagrams as counterexamples can sometimes be quite tedious when defining the correct statechart diagram. The user should have a more direct method to change the conjecture. A typical object-oriented design tool allows the user to edit the statechart diagram by adding new states and transitions, by deleting existing ones, and by splitting and merging states.

So far, we have considered how to construct an automaton from a given consistent and closed observation table. When the user is allowed to edit the statechart diagram, we have to have a method for traversing also to the opposite direction from the statechart diagram (or from the corresponding finite automaton) to an observation table that defines the original finite automaton / statechart diagram.

Even a small editing operation in the statechart diagram may cause a major change in the observation table. Furthermore, the whole statechart diagram might have been constructed manually. Hence, we obey the policy to always build up the observation table from scratch.

In order to formulate an algorithm for building up the observation table for a given statechart diagram, we have to define some auxiliary concepts. We say that a *simple path* in a finite automaton is a path that does not visit any state more than once. Hence, a simple path does not contain loops. A *simple loop* has the same node in the beginning and in the end, and all other nodes are different from each other and from the one in the beginning and in the end. A *self-loop* is a special kind of simple loop with only one transition. We have earlier defined $\mathit{pref}(I)$ to stand for the set obtained from I by making it prefix-closed without deleting any words. Now we define $\mathit{suff}(I)$ to be the analogous suffix-closed set. If B is a finite automaton, then $L(B)$ stands for the language accepted by B.

Algorithm BuildUp
Input: A statechart diagram D obtainted from a finite automaton B.
Output: A consistent and closed observation table T with the sets S and R such that T defines B.
begin

1. Let P be the set of strings related to the simple paths from the initial state to the final state in B;
2. Let Q be the set of strings related to the simple loops in B;
3. **for** each $y \in Q$ **do** take some $xyz \in L(B)$ to P;
4. Let $S = \mathit{pref}(P)$ and $R = \mathit{suff}(P)$;
5. Extend T to $(S \cup S \cdot A) \cdot R$ by adding necessary rows for the strings in $S \cdot A$ and by filling up the entries by consulting B;

end { BuildUp }

It is clear that the observation table can be filled up without consulting the user. Moreover, the observation table obtained is closed and consistent, and it defines B. Since all the simple paths from the initial state to the final state and their prefixes are inserted to S, and the same strings with their suffixes to R, the size of T can increase considerably. However, this should not cause any problems in practice, since queries to the user are not needed when filling up the table.

As an example, consider the statechart diagram shown in Figure 2. It is related to an alarm clock example discussed in [11]. There are following messages:

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

The alphabet consists of the following action/event pairs: (s_ct,set), (s_at,NULL), (s_ct,reached), (buzz,off), and (s_ct,VOID).

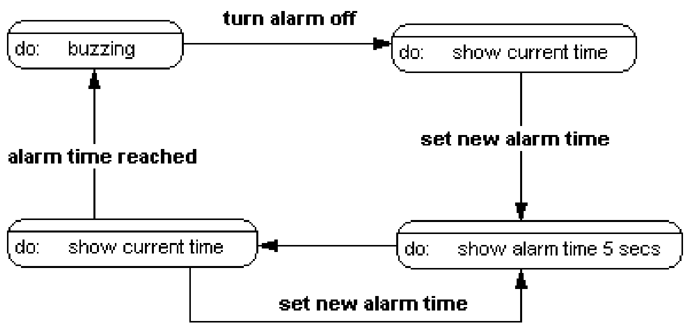

Figure 2: A sample statechart diagram.

The underlying finite automaton is shown in Figure 3, where the initial state is denoted by q_0 and the final state by q_+. The algorithm first sets

$$P = \{(s\_ct,\, VOID), (s\_ct, set)(s\_at, NULL)(s\_ct,\, VOID)\}$$

and

$$Q = \{(s\_ct, set)(s\_at, NULL)(s\_ct, reached)(buzz, off), (s\_at, NULL)(s\_ct, set)\}.$$

The strings added to P are

$$(s\_ct, set)(s\_at, NULL)(s\_ct, reached)(buzz, off)(s\_ct,\, VOID)$$

and

$$(s\_ct, set)(s\_at, NULL)(s\_ct, set)(s\_at, NULL)(s\_ct,\, VOID).$$

The observation table output by the algorithm BuildUp is shown in Table 1. It is easy to verify that it indeed defines the finite automaton of Figure 3. Note

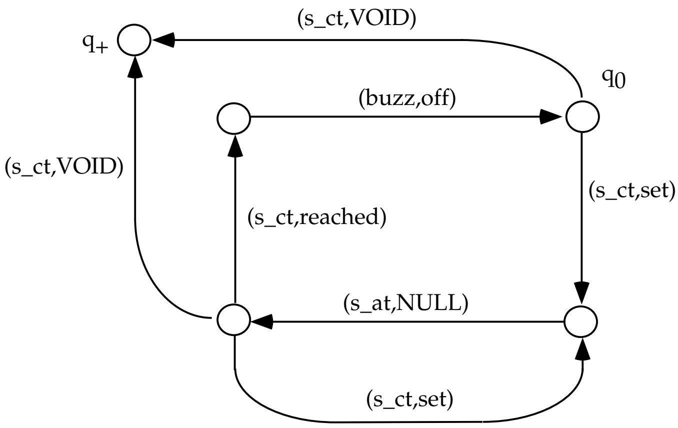

Figure 3: The underlying finite automaton of the statechart diagram shown in Figure 2.

that we have omitted the rows corresponding to the strings not in S from Table 1. However, these rows are needed when further iterations of MAS are executed. The fact that the table in question is closed and consistent follows from the construction.

T	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
λ	0	1	0	1	0	0	0	1	0	1
(s_ct,VOID)	1	0	0	0	0	0	0	0	0	0
(s_ct,set)	0	0	1	0	0	0	1	0	1	0
(s_ct,set)(s_at,NULL)	0	1	0	1	0	1	0	1	0	1
(s_ct,set)(s_at,NULL)(s_ct,VOID)	1	0	0	0	0	0	0	0	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)	0	0	0	0	1	0	0	0	0	0
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)	0	1	0	1	0	0	0	1	0	1
(s_ct,set)(s_at,NULL)(s_ct,reached)(buzz,off)(s_ct,VOID)	1	0	0	0	0	0	0	0	0	0
(s_ct,set)(s_at,NULL)(s_ct,set)	0	0	1	0	0	0	1	0	1	0
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)	0	1	0	1	0	1	0	1	0	1
(s_ct,set)(s_at,NULL)(s_ct,set)(s_at,NULL)(s_ct,VOID)	1	0	0	0	0	0	0	0	0	0

Table 1: Observation table with columns
$r_1 = \lambda$,
$r_2 = (s\_ct,\ VOID)$,
$r_3 = (s\_at,\ NULL)(s\_ct,\ VOID)$,
$r_4 = (s\_ct,\ set)(s\_at,\ NULL)(s\_ct,\ VOID)$,
$r_5 = (buzz,\ off)(s\_ct,\ VOID)$,
$r_6 = (s\_ct,\ reached)(buzz,\ off)(s\_ct,\ VOID)$,
$r_7 = (s\_at,\ NULL)(s\_ct,\ reached)(buzz,\ off)(s\_ct,\ VOID)$,
$r_8 = (s\_ct,\ set)(s\_at,\ NULL)(s\_ct,\ reached)(buzz,\ off)(s\_ct,\ VOID)$,
$r_9 = (s\_at,\ NULL)(s\_ct,\ set)(s\_at,\ NULL)(s\_ct,\ VOID)$, and
$r_{10} = (s\_ct,\ set)(s\_at,\ NULL)(s\_ct,\ set)(s\_at,\ NULL)(s\_ct,\ VOID)$.

5.3 Providing counterexamples

The task of providing counterexamples is the most difficult part of using MAS. Hence, the user interface should support the user to find proper counterexamples and to check their consistency with the other information available.

Suppose that MAS has output a statechart diagram with B as the undelying finite automaton and that the user does not accept the conjecture. The user is now expected to provide a counterexample. If she gives a positive counterexample w, i.e., a string not in $L(B)$, MAS should change the conjecture so that w is contained in $L(B)$. Otherwise, the user gives a negative counterexample (a string w in $L(B)$) and MAS should omit w from $L(B)$.

The normal way to give a positive counterexample is to present an extra sequence diagram, which is then transformed to a trace as explained earlier. When the user gives her counterexample, the interface should confirm whether or not it is in $L(B)$, so that she can be sure that the counterexample is of the desired type. An instructive way of telling this is to animate the function of the conjectured statechart diagram with the input w. This ensures that the counterexample has the desired effect to the statechart diagram.

The task of giving a negative counterexample is often natural to replace by editing the statechart diagram. For example, deleting a transition from the statechart diagram is equivalent with giving a set of negative counterexamples, which are now longer accepted by the statechart diagram when the transition is missing.

An easy method to define a very general type of negative counterexamples is to allow the user to select paths from the conjectured statechart diagram by clicking its states on the screen. Suppose the user clicks a pair of states s_1 and s_2 one after another. This can be interpreted so that all paths from state s_1 to state s_2 are forbidden. In other words, all the substrings of the form $(a, x)y(b, z)$, where a and b are the actions related to the states s_1 and s_2, respectively, x and z are any messages, and y is any sequence of pairs, are forbidden. Hence, by clicking states we can define even more general classes of strings as forbidden than by marking substrings in membership queries.

It is known that membership queries are not necessary for a polynomial time inference algorithm for regular languages if the teacher always provides (lexicographically) smallest counterexamples (see e.g. Birkendorf et al. [4]). This result does not help us, since it is unreasonable to expect the user to provide smallest counterexamples to the algorithm. However, the chosen counterexamples also have their effect to the efficiency of MAS: short (positive) counterexamples are, of course, desirable.

6 The UML sequence diagram notation

So far, we have studied MAS for synthesizing statechart diagrams from simple UML sequence diagrams that consist only of objects (vertical lines) and message calls (horizontal arcs). The UML sequence diagram notation [14, 15], however, is much richer. It contains, for instance, the following notation concepts: *conditional branching*, *iteration*, *recursion*, *explicit return messages*, and *destruction of objects*. In addition, states of objects can be attached to a sequence diagram. Next we will study how these concepts can be taken into account when synthesizing a UML statechart diagram from a set of sequence diagrams using MAS.

The statechart diagram synthesis process consists of two distinct phases: forming the trace and applying MAS. The sequence diagram notation concepts are interpreted and managed in the first phase, i.e., they are explained and added to the trace when the sequence diagram is read. Thus, these concepts do not cause any major changes to MAS itself.

In Section 2 we showed how the trace, consisting of pairs (e_i, e_j), where e_i is a sent message that maps to an action of a state and e_j is a received message that maps to a transition, is formed. A received message causes the object to react and possibly to perform an action. In some cases, the reaction of the object may also depend on a primitive condition. Such conditions appear, for instance, in conditional and iteration expression as *guard conditions*. Thus, in addition to a received message, the right hand side of a pair might also represent a condition that is mapped to a guard condition of a transition in the resulting statechart diagram.

MAS defines the states by their actions, not by their names. However, the UML sequence diagram notation allows state symbols (with state names) to be attached to a lifeline of the object to show a change of states [15]. This encourages us to extend the structure of the trace items from a tuple to a triple: in a trace item (e_i, e_j, e_k), e_i is a sent message, e_j is a received message, and e_k is a name of the state in which e_i is an action. The proposed change does not require major changes in MAS. The algorithm can determine that a certain action can be executed after a particular received message (a state merge) only if both the action and the state name allows that. Thus, a state name is used by MAS when the action part (sent message) of the same trace item is used, giving stronger requirements for a state merge. The state name is also needed for interpreting iteration expressions. However, in most cases the state name is NULL. For simplicity, we use tuples in such cases.

6.1 Conditional branching

UML sequence and collaboration diagrams may contain messages whose execution depends on the truth of a *condition-clause.* The condition-clause is attached to the message label, before the actual message name. UML does not define the format of the condition-clause. It is typically expressed in pseudocode in square brackets. A conditional branching is shown by multiple arrows leaving a singe point, each labeled by a guard condition. Depending on whether the guard conditions are mutually exclusive, the construct may represent conditionality or concurrency [15].

If the conditions are mutually exclusive (conditional branching) and the guards do not cover all possible case (e.g., $[x > 0]$ and $[x < 0]$, when x = 0 is not included in either of the cases), either of the branches is taken and the next arrow is considered. If the arrows do not leave a single point, then they are considered separately, not depending on each other.

The notation used to express conditional structures in UML sequence diagrams is rather clumsy, especially for expressing nested conditional structures. When such structures are nested to more than two levels, the sequence diagrams become difficult to read and write.

When constructing a trace for MAS, a conditional branching is interpreted as a shorthand notation for several merged sequence diagrams, each of them expressing one branch in the structure. Moreover, from the sender's point of view, the name of the message is parsed so that the guard condition itself is considered as a received message (and thus will be mapped to a transition in the resulting statechart diagram) and the message name as a normal sent message. From the receiver's point of view, only the actual message name (without the guard condition) is considered.

As an example, consider the sequence diagram in Figure 4 from the point of view of the *a:A* participant. The conditional branching causes the following two sequences of pairs to be added to the trace:

1. $(NULL, op())(NULL, [x > 0])(f(), NULL)(h(),\ldots$
2. $(NULL, op())(NULL, [x < 0])(g(), NULL)(h(),\ldots$

6.2 Iteration

An *iteration expression* can be used to mark an iteration of a connected set of messages. The iteration expression consists of an iteration marker "*", an

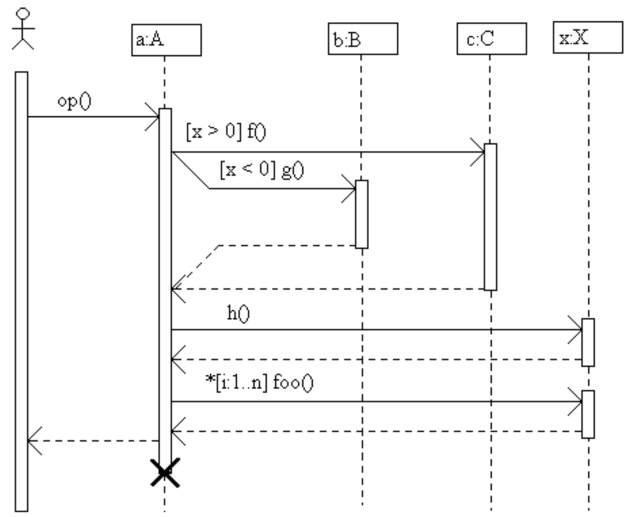

Figure 4: A UML sequence diagram with the conditional brancing, iteration, and destruction of an object.

iteration-clause, and the message to be iterated. In addition to that message, all the other messages between the iteration expression and the corresponding return message are iterated as well. The iteration marker indicates that the messages are iterated zero or more times. The iteration-clause shows the details of the iteration variable and condition, but it may be omitted (in which case the iteration conditions are unspecified) [15]. As in the case of a condition-clause, the UML does not define the format of the condition in the iteration clause. The iteration-clause is typically expressed in pseudocode in square brackets. Note that a branch is notated the same as an iteration without the iteration marker.

As a default, the connected messages in an iteration expression are executed sequentially. For expressing concurrently executed messages, a star followed by a double vertical line ($*||$) is used, for example, $*[i := 1..n]||q[i].calculateScore()$ [15].

The iteration expression can only be used to express iteration of connected messages. In some cases, it would be desirable to model repetition of a set of sequent, disconnected messages. Such a notation could be used, e.g., to express repetition of behavioral patterns [18]. This is not supported in UML sequence nor collaboration diagrams.

Consider an iteration expression **[x] foo()* when constructing a trace for MAS. From the receiver's point of view, only the message name in an iteration expression is considered. From the sender's point of view, the iteration expression is handled in the following way:

1. A trace item $(NULL, [x], Iteration\ n)$, where n is a consecutive number of the iteration expression, is added to the trace.

2. The message *foo()* is interpreted as a normal sent message.

3. After the retun message of the iteration expression has been read, an additional trace item $(NULL, NOT[x], Iteration\ n)$ is added to the trace. The new conditional expression negates the iteration clause.

As an example, consider the sequence diagram in Figure 4. The complete traces from the point of view of the *a:A* participant are the following:

1. $(NULL, op())(NULL, [x > 0])(f(), NULL)(h(), NULL)(NULL, [i : 1..n], Iteration\ 1)(foo(), NULL)(NULL, NOT[i : 1..n], Iteration\ 1)(NULL, DESTROYED)$

2. $(NULL, op())(NULL, [x < 0])(g(), NULL)(h(), NULL)(NULL, [i : 1..n], Iteration\ 1)(foo(), NULL)(NULL, NOT[i : 1..n], Iteration\ 1)(NULL, DESTROYED)$.

A condition-clause and its negation are attached as labels of the outgoing transition of a state that names the iteration in the resulting statechart diagram. Condition clauses of different iteration expressions should not be mapped with outgoing transitions of the same state. Hence, individual state names are given for determining the states and thus avoiding unnecessary membership queries.

6.3 Recursion

An activation of an object is shown as a double solid lifeline in a sequence diagram. When an operation of an object is called, the object is activated. In a recursive call, the control reenters the object (a call of the same or another operation) while the first call is still active. Recursion is shown in a sequence diagram by stacking the activation lines: the second activation region is drawn slightly to the right of the first one. Stacked calls may be nested to an arbitrary depth [15].

Note that the recursive call may or may not be an activation of the same operation. Thus, the recursion indicates object recursion rather than operation recursion. If the recursive call is labelled differently than the currently active call, then it represents a normal operation call (of the same object). If the names are the same, it represents operation recursion.

MAS considers recursive calls as normal messages.

6.4 An object calling its own operation

In UML sequence diagrams, operation calls are drawn as arcs from the lifeline of the sender object to the lifeline of the receiver object. An object may also call its own operations. Such a self message is drawn as an arc that leaves and curves back to the lifeline of that object.

MAS considers a self message as a sent message, since it can be interpreted as an internal action of an object.

6.5 States in sequence diagrams

In a UML sequence diagram, a state symbol can be attached to a certain point of a lifeline of an object to show a state change and to name the activated state. An arrow can be drawn to enter the state symbol to indicate the message that caused the state change [15].

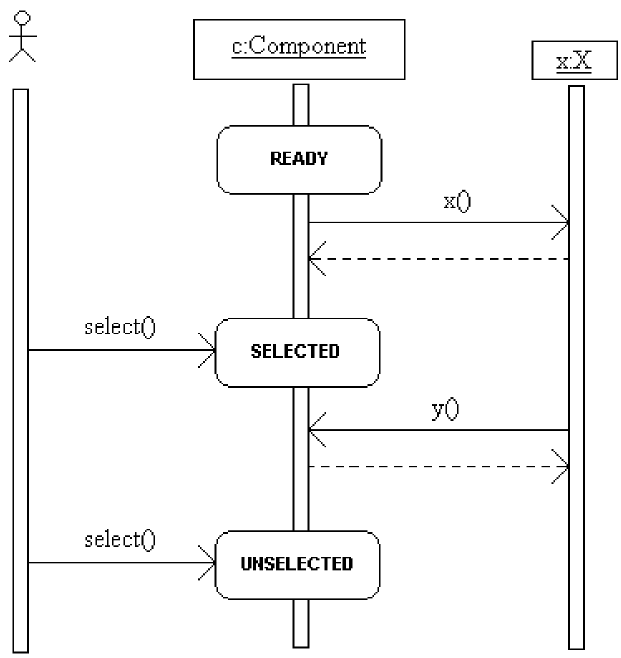

Figure 5: A UML sequence diagram with three state symbols.

Note that the message drawn to the state actually corresponds to a transition that yields to that state, i.e., the state drawn in the sequence diagram represents the new state after the object has received the message. When constructing a trace for MAS, the message that caused the state change is interpreted as a normal received message. The name of the state is added as a third component in an appropriate trace item. If there is a sent message following the state symbol, the name of the state is added to the same trace item as this message. If a received message follows the state symbol, the name of the state and the received message are placed in consecutive trace items.

Figure 5 shows a sequence diagram with three state symbols. A trace constructed for the object *c:Component* is the following:

$(x(), \mathit{select}(), READY)(NULL, y(), SELECTED)(NULL, \mathit{select}())(NULL, VOID, UNSELECTED)$

7 Conclusions

We have earlier [11] described the basic properties of MAS, a synthesis algorithm based on Angluin's framework of minimally adequate teacher. In this paper we sharpened our ideas by considering various features to be fixed when implementing the algorithm.

It has turned out that the key concept in using MAS is the interaction between the user and the algorithm. The user can improve the performance of MAS by transfering additional information to it in forms of edit operations, marking forbidden substrings, and providing desirable counterexamples. On the other hand, MAS can support the user, for example, by telling the status of a counterexample, or by constructing a new observation table from the result of user's edit operations.

An important factor in the use of MAS is the number of membership queries needed. The improvements introduced in Chapter 5 all aim at decreasing that number. Our next goal is to run MAS on various design tasks in order to estimate the average need of membership queries in practical situations, and to find out possible problematic situations where membership queries are needed more than usual.

The statechart diagram synthesis using MAS consists of two phases: constructing a trace based on the information given in sequence diagrams and applying MAS. The UML sequence diagram notation is an extension of a basic MSC notation, containing, for instance, algorithmic constructs. As shown in Chapter 6, these extensions can be interpreted and taken into account when constructing the trace. No significant changes are needed for MAS itself. Thus, using MAS for the statechart diagram synthesis does not require any restrictions nor extensions to the UML sequence diagram notation.

Acknowledgements. The authors wish to thank Prof. Kai Koskimies for his valuable comments.

References

[1] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search. *Comm. ACM* **18** (1975), 333–340.

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

[3] D. Angluin, M. Kriķis, R.H. Sloan, and G. Turán, Malicious omissions and errors in answers to membership queries. *Mach. Learn.* **28** (1997), 211–255.

[4] A. Birkendorf, A. Böker, and 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.

[5] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan, Making data structures persistent. *J. Comput. Syst. Sci.* **38** (1989), 86–124.

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

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

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

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

[10] S. Leue, L. Mehrmann, and 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.

[11] E. Mäkinen and T. Systä, Minimally adequate teacher designs software. Dept. of Computer and Information Sciences, University of Tampere, Report A-2000-7, April 2000. Submitted.

[12] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Seaching.* Springer, 1984.

[13] M.H. Overmars, *The Design of Dynamic Data Structures. Lecture Notes in Computer Science* **156**, Springer, 1983.

[14] Rational Software Corporation, *The Unified Modeling Language Notation Guide v.1.3.* [http://www.rational.com], 2000.

[15] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

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

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

[18] T. Systä, Static and Dynamic Reverse Engineering Techniques for Java Software Systems, Dept. of Computer and Information Sciences, University of Tampere, Report A-2000-4, Ph.D. Dissertation, 2000.