An Interactive Approach for Synthesizing UML Statechart Diagrams from Sequence Diagrams

Erkki Mäkinen[1] and Tarja Systä[2]
[1]Department of Computer and Information Sciences, P.O.Box 607,
FIN-33014 University of Tampere, Finland
em@cs.uta.fi
[2]Tampere University of Technology, Software Systems Laboratory,
P.O.Box 553, FIN-33101 Tampere, Finland
tsysta@cs.tut.fi

Abstract. Minimally Adequate Synthesizer (MAS) is an interactive algorithm that synthesizes UML statechart diagrams from sequence diagrams. It follows Angluin's framework of minimally adequate teacher to infer the desired statechart diagram by consulting the user. To minimize the consultations needed, MAS keeps track of the interaction with the user. Together with its general knowledge about sequence diagrams, this allows MAS to operate without user's help in most of the cases.

A synthesized statechart diagram is a generalization, which accepts additional behavior to that described in the sequence diagrams given as input. During the synthesis process MAS asks the user if certain generalizations are allowed or not. We sketch the usage of two different kinds of inaccurate answers the user can provide. We allow *Probably yes* and *Probably no* answers, i.e. weak *Yes* and *No* answers. The information obtained from these answers is considered less significant than that obtained from normal, definite answers. The user can also postpone answering by saying later.

1. Introduction

The Unified Modeling Language (UML) [8, 9] 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.

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 [2]. *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 (MSCs) [3] (including UML sequence diagrams) is implemented in several tools [4, 5, 10, 11, 12].

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 overgeneralization. To achieve this goal, the algorithm asks the designer for guidance during the synthesis process, when needed.

MAS [6, 7] is an algorithm that synthesizes UML statecharts diagrams from sequence diagrams. It models the synthesis process as a language inference problem and uses Angluin's [1] framework of minimally adequate teacher to infer the desired statechart diagram with the help of membership and equivalence queries. The algorithm can conclude the correct answer to most of the membership queries without consulting the teacher, i.e., the designer.

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 symmetric difference of the languages accepted by the output statechart diagram and the unknown statechart diagram must be given (the equivalence queries).

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 [4], 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.

We have a practical implementation of MAS, intergrated to a real-world UML modeling tool, the Nokia TED [13]. TED is a multi-user software development environment that has been implemented at Nokia Research Center and is currently used at Nokia. It supports most of the UML diagram types.

2. An overview of MAS

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.

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 lifeline 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. 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.

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_i" and an outgoing transition labelled e_j in the state machine.

In a UML statechart diagram, a transition from a state to another state can also be a so called *completion transition*. We require that if a state has an outgoing completion transition without a guard (i.e., a condition that must hold to allow the transition to fire), it cannot have any other outgoing transitions. In UML, such a transition is implicitly triggered by the completion of any internal activity in a state [9]. 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.

The UML sequence diagram notation [8, 9], 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. These UML sequence diagram concepts can be taken into account when constructing the trace for MAS (see [7] for details). For instance, a conditional branching can be interpreted as a shorthand notation for several merged sequence diagrams, each of them expressing one branch in the structure. 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.

The input of MAS is a set of traces, and MAS tries to construct a statechart diagram consistent with the input traces. However, this is usually not possible without a help from the user. MAS can pose *membership queries* and *equivalence queries*. In a membership query, MAS asks the user whether a given sequence of operations is acceptable. (In the original terminology of grammatical inference this means "whether or not a given string belongs to the target language".) By making membership queries, MAS gathers information to its data structures. Based on its application specific knowledge, MAS can conclude the correct answer to most of the membership queries. This is essential for the usability and efficiency of the use. (In its original form, Angluin's minimally adequate teacher makes far too many queries to be practically usable.)

When the data structures of MAS fulfill a certain condition, it makes an equivalence query (a conjecture). This means that MAS outputs a statechart diagram, and the user can accept or reject it. If the user accepts the conjecture, the algorithm halts. In the case of rejection, the user is expected to give a counterexample to guide the further execution of the algorithm. Consult [6,7] for the details concerning MAS.

3. An example

Assume 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:

(display time, set alarm time),
(register, NULL),
(flash alarm time, NULL),
(display time, alarm time reached),
(start ringing, turn alarm off),
(stop ringing, NULL),
(display time, VOID).

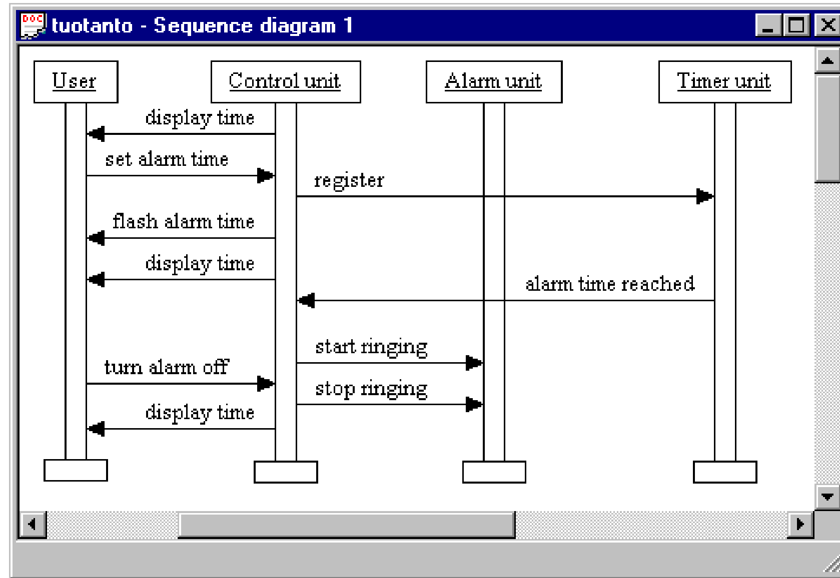

Figure 1. A sequence diagram describing an example usage of an alarm clock.

To be able to produce the first conjectured statechart diagram, MAS needs to consult the user four times. It makes a membership query concerning the following paths:

1) No user inputs are given, i.e., the alarm clock just displays time,
2) A new alarm time is given,
3) A new alarm time is given and it is changed once before that alarm clock starts ringing.
4) A new alarm time is given, the alarm clock starts ringing when the alarm time is reached, and another alarm time is set.

The user accepts all these four paths and MAS conjectures the statechart diagram depicted in Figure 2.

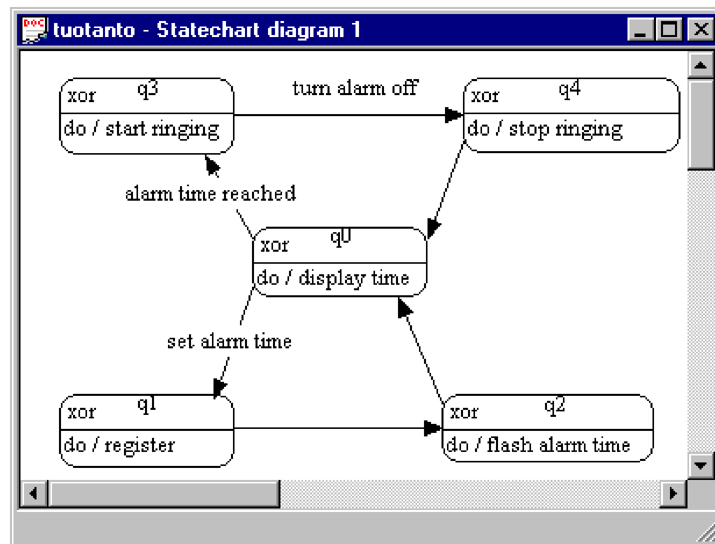

Figure 2. The first conjectured statechart diagram.

The user can then either accept or reject this conjecture. If she rejected, she should provide a counterexample. A counterexample can be positive or negative. In a case of a positive counterexample, MAS should change the conjecture so that it also accepts the given counterexample. A negative counterexample, in turn, describes a generalization in the conjecture that should be prohibited. The normal way to give a positive counterexample is to present an extra sequence diagram.

Semantically, the first conjecture in Figure 2 contains two generalizations, compared to the sequence diagram in Figure 1: (1) the alarm time can be reset several times before the clock starts ringing, and (2) if the alarm time is reached, the clock starts ringing even if the alarm is not set on. The algorithm has merged two logically different states together, thus overgeneralizing the statechart diagram. The former example of overgeneralization (1) is not harmful but the latter one (2) is (from the semantical point of view). Thus, in the case a statechart diagram in Figure 2, a negative counterexample should be given. In this case, a sufficient counterexample could be, e.g., a path from state *q0* to state *q4* via state *q3* (using transitions *alarm time reached* and *turn alarm off*).

4. Managing inaccuracy in MAS

As described in [6, 7], MAS allows only *Yes* and *No* answers to membership queries. User's mind changes cannot be considered as incomplete or inaccurate answers, since at any moment MAS handles all information as "certainly true". A much better approach is to allow the user to give genuinely inaccurate answers. This, however, makes the structure of MAS considerably much more complicated. In this section we sketch ways to equip MAS with a possibility to handle incomplete and inaccurate answers. We consider here only membership queries, since we suppose that user's answers to equivalence queries are always correct and definite.

During the membership query phase MAS collects information from the user to its observation table. In order to make the conjecture, MAS requires exact information indicating whether or not certain paths are possible in the system considered (or in the grammatical inference terminology: "whether or not certain strings belong to the target language"). Hence, in order to allow the algorithm to accept inaccurate answers, we have to have some kind of mapping from inaccurate answers to normal *Yes* and *No* answers. This is pictured in Figure 3.

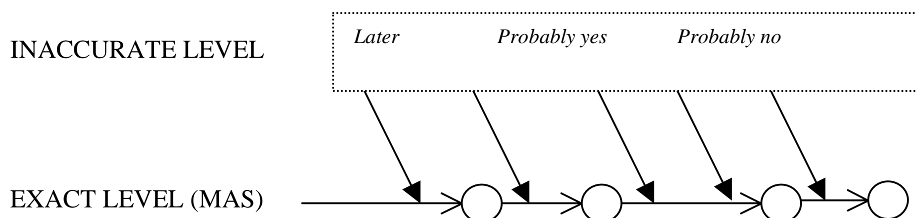

Figure 3. A mapping from the inaccurate level to the level of MAS. A line stands for a membership query phase and a circle stands for an equivalence query phase, i.e., a conjecture.

A simple way to support the user in cases of uncertaincy is to allow her to answer *Later* to a membership query. In some cases, a question that is difficult to answer might become obvious, when the synthesis process gets further and the user learns more about the overall behavior. Furthermore, a difficult membership query might even get automatically solved later during the synthesis process. This happens if another query contains information from which the answer can be concluded. Implementing *Later* answers is straightforward. The algorithm just needs to keep track of postponed questions and poses them later (if

needed) when other membership queries have been asked. Eventually, the user has to give answers to all the questions that cannot be solved by MAS, since MAS is not able to give conjectures otherwise.

It is natural to expect that although the user is not sure how to answer a membership query, she usually has some kind of assumption of the correct answer. A *Probably yes* (resp. *Probably no*) answer to a membership query is interpreted as a "*Weak yes*" (resp. "*Weak no*"). This means that the algorithm can change a *Probably yes* (resp. *Probably no*) answer to be normal *No* (resp. normal *Yes*) if the user elsewhere gives definite information from which *No* (resp. *Yes*) can be concluded.

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 by 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 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 discussed 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.

References

1. Angluin D.: Learning regular sets from queries and counterexamples, *Inf. Comput.*, 75, 1987, 87-106.
2. Harel D.: Statecharts: A visual formalism for complex systems, *Sci. Comput. Program.*, 8, 1987, 231-274.
3. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), ITU-T, Geneva, 1996.
4. Koskimies K., Systä T., Tuomi J., and Männistö T.: Automated support for modeling OO software, *IEEE Softw.,* 15, 1998, 87-94.
5. Leue S., Mehrmann L., and Rezai M.: Synthesizing software architecture descriptions from message sequence chart specification, In *Proc. of the 13th IEEE International Conference on Automated Software Engineering (ASE98)*, Honolulu, USA, 1998, 192-195.
6. Mäkinen E., Systä T.: Minimally adequate teacher designs software, Dept. of Computer and Information Sciences, University of Tampere, Report A-2000-7, April 2000. Submitted. (ftp://ftp.cs.uta.fi/pub/reports/pdf/A-2000-7.pdf)
7. Mäkinen E., Systä, T.: Implementing minimally adequate synthesizer, Dept. of Computer and Information Sciences, University of Tampere, Report A-2000-9, June 2000. (ftp://ftp.cs.uta.fi/pub/reports/pdf/A-2000-9.pdf)
8. Rational Software Corporation: *The Unified Modeling Language Notation Guide v.1.,* http://www.rational.com, 2000.
9. Rumbaugh J., Jacobson I., and Booch G.: *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.
10. Schönberger S., Keller R., and Khriss I.: Algorithmic Support for Model Transformation in Object-Oriented Software Development, In *Threory and Practice of Object Systmes (TAPOS)*, John Wiley & Sons, 2000.
11. Somé S., Dssouli R., and Vaucher J.: From scenarios to automata: building specifications from users requirements, In *Proc. of APSEC'95*, Brisbane, Australia, 1995.
12. Whittle J. and Schumann J.: Generating Statechart Designs From Scenarios, In *Proc. of ICSE'00*, Limerick, Ireland, 2000, 314-323.
13. Wikman J.: Evolution of a Distributed Repository-Based Architecture. (http://www.ide.hk-.se/~bosch/NOSA98/JohanWikman.pdf), 1998.