Acta Cybernetica 15 (2002) 547–565.

Implemeting a Component-Based Tool for Interactive Synthesis of UML Statechart Diagrams

Johannes Koskinen[*], Erkki Mäkinen[†] and Tarja Systä[*]

Abstract

The Unified Modeling Language (UML) has an indisputable role in object-oriented software development. It provides several diagram types viewing a system from different perspectives. Currently available systems have relatively modest tool support for comparing, merging, synthesizing, and slicing UML diagrams based on their semantical relationships. Minimally Adequate Synthesizer (MAS) is a tool that synthesizes UML statechart diagrams from sequence diagrams in an interactive manner. It follows Angluin's framework of minimally adequate teacher to infer the desired statechart diagram with the help of membership and equivalence queries. MAS can also synthesize sequence diagrams into an edited or manually constructed statechart diagram. In this paper we discuss problems related to a practical implementation of MAS and its integration with two existing tools (Nokia TED and Rational Rose) supporting UML-based modeling. We also discuss information exchange techniques that could be used to allow the usage of other CASE tools supporting UML.

1 Introduction

The different diagram types provided by UML [23] have strong semantical dependencies. These dependencies allow, among other operations, slicing, synthesizing, and abstracting a UML diagram based on the information included in another diagram. A lot of tool support is available for constructing syntactically correct UML diagrams, but the present tools provide rather modest support for analyzing and using the semantical relationships of these diagrams.

In UML-based behavioral modeling, examples of object interactions are usually visualized as *sequence diagrams* or *collaboration diagrams*. The final specification of an object is modeled as a *statechart diagram*. A statechart diagram can be used as a protocol specification, showing the legal order in which the operations of an object may be invoked.

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

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

Automated support for constructing statechart diagrams from sequence diagrams provides considerable help for the designer. Automatic generation of state machines from scenario diagrams, such as UML interaction diagrams, has been studied extensively [7, 8, 12, 13, 25, 27].

MAS [10, 14, 15] is a tool that interactively synthesizes UML statecharts diagrams from sequence diagrams. Totally automatic synthesis algorithms, e.g., the one used in SCED [8], 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 "overgeneralizations" do not appear in the resulting statechart diagram. The user consultancy is organized as membership and equivalence queries posed by the algorithm. The user can help the synthesis process, for example, by marking certain (sub)paths in the statechart diagram appearing in a membership query as forbidden. This guarantees that the algorithm does not perform queries containing such a subpath more than once. We also consider various ways to support the user when she is providing a counterexample after rejecting a conjecture, i.e., after giving a negative answer to an equivalence query.

Tools like MAS, which support UML-based "model operations" [26], are desirable in all CASE tools. These techniques and tools can be integrated with CASE tools or they can be provided as separate components interoperable with CASE tools. One of our implementation platforms, the Nokia TED [28], is a multi-user software development environment that has been implemented at the Nokia Research Center. MAS interacts with TED through a COM interface. It imports the source sequence diagrams from and exports the resulting statechart diagram to the TED repository. The other implementation platform used is Rational Rose. In principle, MAS can be implemented for any tool supporting UML and providing a reasonable API for accessing the model repository. Moreover, commonly accepted exchange formats like XMI provide even more flexible integration of MAS with other CASE tools supporting UML. In addition to the diagram import and export mechanisms, the interactive nature of MAS brings additional challenges for the integration.

2 From Sequence Diagrams to Statechart Diagrams

In this section we briefly introduce the function of MAS (for further details, consult [10, 14, 15]). MAS tackles the problem of statechart diagram synthesis as a language inference task. The behavior of a selected participant described in a set of sequence diagrams is first mapped to strings belonging to the language to be inferred. MAS is then used to infer the language based on these strings. The resulting language is given as a finite state automaton. Finally, the automaton is transformed into a UML statechart diagram.

Before we discuss the actual synthesis algorithm, we briefly introduce a few aspects in UML sequence and statechart diagrams considered during the synthesis. The basic UML sequence diagrams to be considered in the rest of this paper 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. 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 e_i or e_j is missing, then add $NULL$ instead. If the explicit deletion of the object is not shown at the end of the sequence diagram, let the right hand side of the last pair be $VOID$. The traces read in the above fashion are used as strings during the synthesis process.

In a UML statechart diagram, a transition from a state to another state can also be a so called *completion transition.* A completion transition without a guard is implicitly triggered by the completion of any internal activity in a state [23]. Therefore, in MAS, we do not allow a state to have both labeled and unlabeled transitions (the latter corresponding to unguarded completion transitions) as outgoing transitions. In MAS we do not allow a completion transition and a labeled transition to be the leaving transitions of the same state. Two leaving completion transitions, in turn, would result in a nondeterministic state.

2.1 The Algorithm

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 related to the output statechart diagram and the desired statechart diagram must be given (the equivalence queries).

In addition to definite *Yes* and *No* answers, MAS allows the user to answer *Maybe* or *Hardly* (i.e., weak *Yes* and *No* answers). The information obtained from these answers is considered less significant than that obtained from normal, definite answers. Furthermore, the user can postpone answering by saying *Later.* These inaccurate answers are discussed in greater detail in [10].

MAS maintains an *observation table* T containing the current information about members and non-members of the desired language. The rows of T are labelled by the elements of $(S \cup S \cdot A)$ where A is the input alphabet, and S is a prefix-closed set of strings in A^*. (Notice that the alphabet A of our inference algorithm consists of pairs (e_i, e_j) and that "$\cdot$" stands for the concatenation operator.) The columns of T are labelled by the elements of a suffix-closed set R. MAS generates the columns during the synthesis process. They contain possible continuations to strings in the rows and are used to decide whether the rows represent paths that yield to the same state. In other words, the columns are used to test if two states can be joined. The entry for row s and column r, is 1, if $u = s \cdot r$ is in the desired language, otherwise the entry is 0. The bit string on the row labeled x in T is denoted by $row(x)$. 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, $row(s_1 \cdot a) = row(s_2 \cdot a)$.

The original inference algorithm [2] 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. A closed and consistent observation table defines a deterministic finite automaton in a natural way [2]. The algorithm forwards this automaton as a conjecture to the teacher. The algorithm halts if the teacher accepts the conjecture. Otherwise, the given counterexample updates T and the execution of the algorithm continues.

In our application, where the designer plays the role of the teacher, the execution of the algorithm begins so that the designer constructs a set of typical sequence diagrams describing the behavior of the system. All traces from these sequence diagrams and their prefixes are stored in S. No membership queries are needed since the traces themselves 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 have 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 $e_j \neq e_i$.

The algorithm outputs a finite automaton. Actually, we need a statechart diagram which is obtained by fine tuning the output automaton as described in [15]. The output finite automaton is called the *underlying finite automaton* of the resulting statechart diagram.

2.2 Data Structures Allowing Backtracking

MAS allows the user to give inaccurate answers to membership queries. Thus, we need data structures that are able to manage the user's mind changes or accidentally given incorrect answers.

MAS maintains a trie containing the strings known to be in the desired language. (For the definition and basic properties of tries, see e.g., [16].) This trie is referred to as W. Initially, W contains the information related to the input set. New information is inserted in W when the user gives a positive answer to a membership query, or when she gives a counterexample not belonging to the language accepted by the conjectured automaton. We need a trie structure in which we are able to efficiently backtrack, and then update the observation table if necessary, i.e., we use a structure that resembles so called persistent data structures (see [6]).

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 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 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.

The algorithm tries to determine table entry values without consulting the user. We prepare ourselves to possible backtrack operations by maintaining pointers in order to reach the table entries whose value is determined by the algorithm. If a trie node used in determining table entries is later deleted, these entries 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 the value of all observation table entries are determined from existing trie elements also after the deletion.

Consider now what happens when a string is deleted from the set of words known to be in the desired language. First, the corresponding element is deleted from W. The algorithm might have concluded an affirmative answer to a membership query based on 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. Notice, 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).

2.3 Improving the Algorithm

Depending on the case tool MAS is integrated with, the performance of the synthesis process varies. According to our studies, in the case of TED, importing and exporting diagrams to and from the TED repository is the most time consuming part with small and moderate size examples (excluding the time spend with user interactions) [9]. With large examples, in turn, also the performance of the algorithm becomes an issue.

An obvious direction for improving the performance of MAS is to further decrease the amount of user consultancy. There are two different lines to follow. First, we can equip the user with methods to transfer her knowledge about the system to the algorithm. These methods are introduced in Section 3. Second, we can try to make use of the general improvements suggested to Angluin's original algorithm in the literature.

In what follows, we shortly discuss the suggestions by Rivest and Schapire [24] (see also a survey by Balcázar *et al.* [3]). The idea of Rivest and Schapire is to use a "characteristic member" of each class of strings in S with an equal row. This means that the observation table is always consistent. Clearly, the principle also decreases the size of the observation table, and as a consequence, the number of membership queries is decreased too, at least in the worst case. The crux of the improvement is the handling of counterexamples. Instead of inserting the counterexample and its prefixes to S, the method of Rivest and Schapire finds out a new member for R. This string is chosen so that it makes the observation table non-closed, and in order to retain closeness, a prefix of the counterexample is inserted in S. Although membership queries are needed to find the correct prefix of the counterexample, it can be shown that this method indeed decreases the number of membership queries in the worst case. However, it is still open whether this method actually decreases the number of membership queries in our application. Namely, it is essential how many membership queries the algorithm can answer without consulting the user. The queries induced by the method of Rivest and Schapire may be difficult for the algorithm.

If applied in its basic form, the method of Rivest and Schapire has the drawback that the new conjecture does not necessarily classify the previous counterexample correctly [3]. This feature is not acceptable since it would confuse the user by making the user interface illogical. However, this problem can be settled by not showing the new conjecture to the user and using the same counterexample as long as the counterexample is not correctly classified. This would also decrease the number of equivalence queries.

It is even known that membership queries are not necessary at all for a polynomial time inference algorithm for regular languages, provided that the teacher always gives (lexicographically) smallest counterexamples (see e.g., Birkendorf *et al.* [4]). However, this result does not help us, since it is unreasonable to expect the user to provide smallest counterexamples to the algorithm. Still the choice of the counterexamples does have its effect to the efficiency of MAS: short (positive) counterexamples are, of course, desirable.

There are also various ways to streamline the data structures. For example, the observation table is very sparse, i.e., a great majority of its entries are zeroes. This fact can be utilized by storing only the entries containing ones.

3 Interaction Between the User and MAS

In this section we discuss the information exchange and visualization techniques between MAS and the user. We introduce various methods for transfering additional information to the synthesis algorithm in order to further decrease the number of membership queries. For the membership queries to be informative and interesting enough, they need to vary from each other. Moreover, the amount of queries should be considerable small not to cause the user to lose her interest. Ideally, MAS should draw the user's attention to crucial and ill-defined parts of the current design but not bother her with trivial questions. In what follows we sketch techniques that allow the user to give "general guidelines" to MAS, thus decreasing the amount of queries. Especially, the proposed techniques aim at decreasing the amount of similar or closely related questions, answers to which depend on the same key question.

3.1 Visualizing the Membership Queries

Choosing an appropriate information visualization technique is important in interactive systems. A membership query needs to be shown to the user in a way that is easy to understand and answer. An intuitive way to visualize a membership query would be highlighting the corresponding path in a statechart diagram. However, since some of the membership queries are posed before a conjecture for a statechart diagram can be represented, this is not possible for all membership queries.

Currently, MAS poses the membership queries in a form of a simple sequence diagram with two participants: the object of interest and a participant (called *System*) that represents all other participants (inside or outside the system border) the object interacts with. A membership query often consists of subpaths that have already been accepted by the user. In such a case, the membership query can be translated to a question: "Can these subpaths occur in the presented order?". To make it easier for the user to recognize such components (subpaths) in the membership query, MAS uses a different color for each one of them. Figure 1 shows a sample membership query. It consists of two subpaths already accepted by the user. The equivalence queries, in turn, are visualized as statechart diagrams by the CASE tool (currently TED or Rose) itself.

In order to give the user a flexible way to express her mind changes concerning the status of a piece of inaccurate information, MAS provides a window in which all the inaccurate information is presented. The user can browse the questions and modify the answers simply by clicking the mouse button. If the user now gives an accurate answer, the question disappears from the window. This window is opened when the user gives an inaccurate answer (*Maybe* or *Hardly*) for the first time.

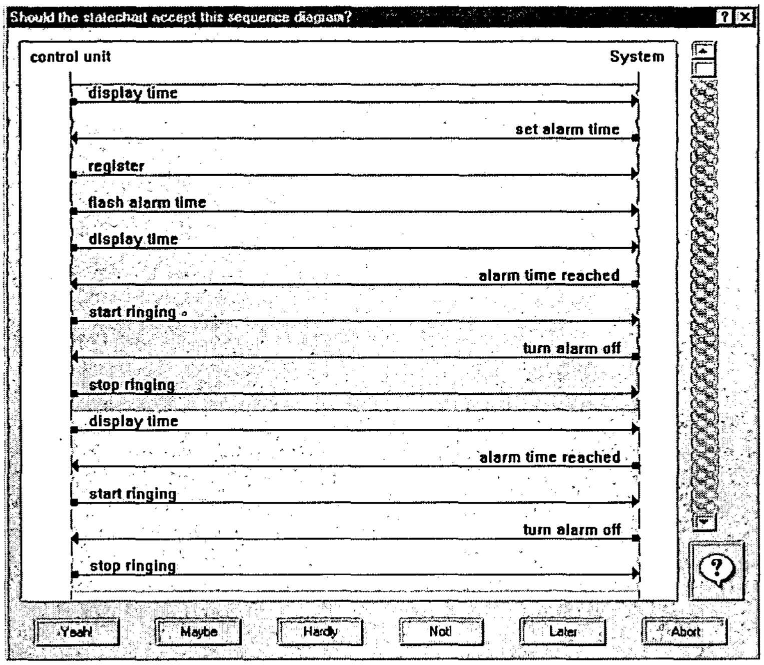

Figure 1: A sample membership query

3.2 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 subpaths beforehand. A user-friendly way to transfer this information to the algorithm is to give the user a possibility to mark any subpath of a membership query as invalid. This guarantees that the algorithm does not make membership queries with the same invalid subpath 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. For example, from the membership query in Figure 1 the user might want to select a block from the sixth message (alarm time reached) to the 12th message (start ringing) as a forbidden subpath, indicating that an alarm clock should not start ringing if the alarm is not set on (even though the alarm time is reached).

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.

3.3 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 the construction of 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. This is possible by using the algorithm *BuildUp* introduced in [14]. 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 the edited statechart diagram.

3.4 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 underlying 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 the language $L(B)$ accepted by 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. 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. In our approach, the conjectured statechart diagram is visualized by the CASE tool. This means that the API of the CASE tool in question should allow its extension with such animation property. Considering our current implementation environments, the Rational Rose Extensibility Interface (REI) allows this while the TED API does not.

The task of giving a negative counterexample is often more 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. Again, the API of the CASE tool should allow activation of the states.

4 Integrating MAS with CASE Tools

A variety of CASE tools supporting UML is available. These tools provide syntactic support for UML-based software development. Moreover, code generation and/or basic round-trip engineering facilities (typically limited to relations between a class diagram and source code) are supported by many of these tools. A more interesting and more challenging problem is to provide semantical support for applying operations among different UML diagrams. Selonen *et al.* [26] divide *model operations* into two groups: (1) *basic operations* that apply set theoretical operations (union, difference or intersection) for two diagrams of the same type and (2) *transformation operations* that take a UML diagram as an operand and produce a diagram of another type as its result. Both basic and transformation operations involve the semantics of the diagrams and need a case tool for providing the information content of the diagrams and for visualizing the resulting UML diagram. Thus, model operations could be implemented as separate components that provide import and export services for the tools (supporting UML) they are interoperating with.

For managing interoperability, the information exchange format should be agreed on. In what follows we discuss the advantages and disadvantages of different integration techniques from the point of view of MAS.

4.1 Integrating MAS with TED

The TED version of MAS is implemented as a stand-alone program, which connects to the TED's repository using a special TED COM server. This server is located in a local computer and it establishes a connection to a remote TED server. The user needs to specify where the server and the repository are located and how the traces to be synthesized can be found.

Additionally, the user needs two different tools, one for constructing a sequence diagram and another for synthesizing the statechart diagram, and to switch between them during the synthesis process. This is not practical with interactive synthesizers like MAS. When MAS asks for a counterexample, the user needs to make a sequence diagram or to modify the existing statechart diagram. When she is ready, MAS has to know the exact location of the diagram (ID or path to it) before it can continue. There is no way the user can point an object in the editor and tell MAS to continue synthesizing using that trace as a counterexample.

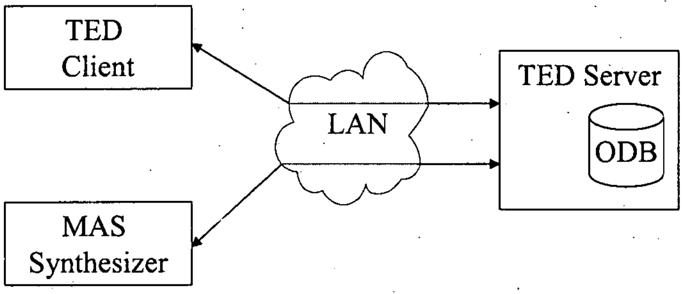

Figure 2: The TED implementation of MAS

4.2 Implementing a Software Component

Programming languages offer mechanisms to distribute and reuse software libraries, but these techiques have been vendor and language specific and communication between different libraries has been difficult. The object component model is a language and vendor independent mechanism to reuse existing software.

In Microsoft Windows environment we can choose between two object component technologies: The Common Object Request Broker Architecture (CORBA) [22] and The Component Object Model (COM) [17]. Information about the differences between these two technologies can be found in [5]. COM is designed for Windows platform. Almost all CASE tools offer some kind of COM interface for automating design. In addition, with COM Automation [18] we could use our synthesizer tool from scripts and macros.

COM is a platform independent, distributed, object-oriented system for creating binary software components that can interact. These components (objects) can be within a single process, in different processes, or even on remote machines [19]. Every component has an unique identifier (called Globally Unique Identifier, GUID) and information about available components is stored in the system registry.

The main idea of the software components is that only interfaces are provided for their users, their implementation is hidden. The COM components can be used like normal C++ classes, but the code itself may be executed on a remote machine – the client application does not have to worry about the components' location. There are some guidelines to specify a COM component [20]:

- **A COM interface is not the same as a C++ class.** The pure virtual definition carries no implementation. Unlike C++ classes (interfaces), the COM interfaces cannot have any implementation.
- **A COM interface is not an object.** It is simply a related group of functions. It is also the binary standard through which clients and objects communicate.
- **COM interfaces are strongly typed.** Every interface has its own interface identifier (a GUID), which eliminates the possibility of duplication that could occur with any other naming scheme.
- **COM interfaces are immutable.** You cannot define a new version of an old interface and give it the same identifier. Thus, each interface is a separate contract, and systemwide objects need not know whether the version of the interface they are calling is the one they expect. The interface ID (IID) defines the interface contract explicitly and uniquely.

The COM components can communicate with the client software using events.

4.3 Integration Considerations

The most trivial and flexible way to manage interoperability is to change information through files written in a predefined format. For an optimal interoperability, a file format supported by several CASE tools should be chosen. A downside of this approach is inefficiency: additional reading/writing information from/to files is time consuming compared to the direct use of the information. Since the tool that provides the information need not to be the same as that visualizing the results, this approach allows, for instance, the source sequence diagrams to be constructed with multiple tools.

XML-based Metadata Interchange (XMI) [21] is an interchange format for UML supported by most of the case tools supporting UML. Since the Document Type Definition (DTD) grammar that defines the XMI language is based on UML metamodel (or more precisely, on MOF (Meta-Object Facility) specification [21]), it can only express what is in the UML metamodel, thus lacking support for defining presentation information (e.g., layouts). However, using the extension mechanism of XMI, the tool vendors can define how to add that information to the XMI files. Since there currently does not exist a global agreement on how this should be done, the UML CASE tools can only exchange model information in practise. From the point of view of MAS, this is not a crucial problem, since the imported statechart diagrams are created by MAS and thus, they do not contain any history information on the diagram layouts to be restored.

Integrating MAS more tightly with different CASE tools allows us to extend the possibilities to communicate with the synthesizer. The user can start the synthesizer from the menu and all interactions can be managed using one tool (although in separate windows). While editing statecharts is a relatively easy way to express the counterexamples, the editor can support this task by previewing the conjecture and allowing the user to modify it before she continues the synthesis process.

Using repository via a server has a performance penalty as well. For example, generating a statechart to the repository takes a long time (about one second per state in our case). When the synthesizer and the editor both use the same internal data format, the repository becomes obsolete as temporary storage for synthesized conjectures. Only the final conjecture should be placed in the repository for future use.

Our TED implementation of MAS allows semi-automatic synthesis using startup parameters. The initial sequence diagrams can be given parameters without any interaction, but membership queries and counterexamples will still need user consultation. The startup parameters can also be given graphically using the visual scripting mechanism in TED [11].

To integrate MAS with CASE tools we need to build MAS as a software component. This component implements an interface offered by the CASE tool and either uses a specific interchange format (e.g., XMI) or a special interface for exchanging UML models between the component and the tool. Using an internal data format is a simpler and faster solution, but it limits us to use the single specific CASE tool. XMI is a universal format, but exporting and importing XMI files (even memory mapped files) might slow down the performance drastically with large data structures, especially when chaining the components. In some cases, using pre-saved XMI files allows us to speed up the synthesizing, but we have not made a speed comparison between these two techniques.

A MAS COM component should provide a high-level synthesis interface to start and control the synthesizer from CASE tools. The interface should have at least the following methods:

- `Synthesize(IDataInterface* in, IDataInterface* out)`
  A very high-level synthesis method to start a synthesizer. The input and output interfaces (*IDataInterface*) are used to get a sequence diagram and to put a conjectured statechart diagram back to the editor. This interface is similar to startup parameters with the exception of events. Using this interface method, the synthesizer can notify the editor in different synthesis phases.

- `InsertCounterExamples(DWORD count, IExampleInterface** examples)`
  Inserts a number of counterexamples (sequence diagrams or edited statechart diagrams) for the synthesizer. This method is used after the synthesizer has notified the editor to give a counterexample with an event.

- `InsertForbiddenStrings(DWORD count, BSTR* strs)`
  Tells MAS not to accept traces (strings) by default. This is useful when handling the forbidden strings considered in Section 3.2.

All input and output data is exchanged using interfaces. These interfaces provide required methods to access internal data structures. The data itself could be in XMI files, internal data format, or in server's repository. In addition to the methods, we need several events to tell the editor (or more generally, the client) when the synthesizer will need user action. This allows the client to set *callbacks* and to modify the synthesizer's behavior and user interface.

4.4 First Rays of the New MAS

We have already made the first version of MAS for Rational Rose (later *roseMAS*) using the techiques described in Section 4.2. The synthesizer is a COM component which is connected to the modeling software using *Rational Rose Extensibility Interface* (REI). REI is a COM automation interface for various plugins. It contains methods to manipulate Rose models (e.g., creating new elements) and to extend the user interface (like additional menus). Although roseMAS has been designed to be a Rose plugin, it can also be used with other tools because of its automation interface. On the other hand, roseMAS component cannot be run without additional command line utility.

When roseMAS has been installed using the setup program, it registers itself to the Windows registry, so that CASE tools can use it. The registry contains information on events that roseMAS is interested in, such as selecting a menu item from Rose. When an event occurs, Rose calls a method in roseMAS interface *EventHandler* (see Figure 3). The roseMAS installation package also includes a menu file, allowing MAS options to be changed directly from the CASE tool.

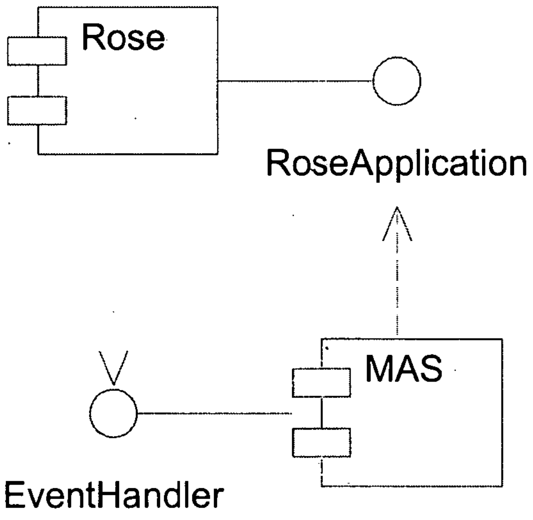

Figure 3: The component diagram describing the implementation of MAS for Rose

When the user selects the *SED→SCD* command from the pop-up menu, roseMAS looks up all selected and active items. One of them has to be an object. The other selected items are sequence diagrams, which contain the same object (or at least object with the same name). RoseMAS gets all the messages related to the object of interest from diagrams and adds them to the trace list. After that, the original MAS algorithm starts with these input traces.

All communication between roseMAS and Rose is managed via *RoseApplication* interface. Since the interface supports automation, we can use a C++ wrapper class to hide COM specific code and use the RoseApplication like a class library.

Comparing this Rose integration to the one with TED, the differences between these two are remarkable. Instead of giving startup parameters and typing location information to the dialogs, the user can select the sequence diagrams she would like to include in the synthesizing. Starting roseMAS is easy because of the direct menu support (see Figure 4). Furthermore, the conjecture is automatically created under the base class of the traced object. The user can give a counterexample, like starting the synthesis. This is accomplished by selecting sequence or statechart diagrams and pushing the *continue* button.

In addition to all this, the performance of the new roseMAS is much better than that of the old client-server system. On the other hand, Rose lacks a multi-user collaboration and database system. This means that we are back with the "one model per file" environment. Moreover, unlike TED, Rose tries to keep our model and diagrams consistent. Normally, this is what the user wants, but when the user synthesizes new diagrams, consistency checking makes some things a bit unconfortable. For example, the states of the generated statechart diagrams can no longer use the same name between diagrams, because they share the same state machine.

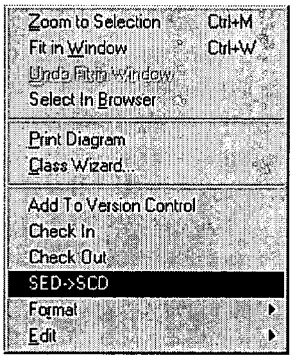

Figure 4: The MAS can be started from the menu

5 Future Work

Currently, we have integrated MAS with two different CASE tools. It would be useful to separate a CASE tool and the synthesizer with a tool-independent platform, so that we would need only one implementation of MAS for all the CASE tools supported by the platform. Such a platform (xUMLi, executable UML interface) has been introduced in [1]. When integrating MAS with xUMLi some problems might appear because of the interactive nature of MAS.

The main problems with MAS are in the user interface. We should equip the user with more efficient methods to transfer her knowledge to the algorithm, and the algorithm should have better ways to support the user in making various decisions. In addition to the topics discussed in the previous sections, at least the following things call for our attention.

We introduced uncertain answers, which allow the user to change her mind later. After giving an inaccurate answer the user might be interested in what part of the current membership query the uncertain portion is (i.e. if the user has given an uncertain answer to query A and/or query B, the dialog should show the uncertain part in query ABC). This could be indicated by using appropiate colors in the queries. In addition, the conjecture generated by MAS could distinguish uncertain and certain paths same way as in the situations mentioned above.

When synthesizing complex systems (e.g., a dialog with buttons and other control elements), it would be helpful, if all components were synthesized at the same time. The resulting conjecture would have multiple statecharts with hyperlinks between different synthesized components allowing the user to switch between generic (the statechart from the dialog) and more specific (the statechart from the button) view. Some kind of 3D-model could also be used to visualize the conjecture.

In real world applications, giving only definite answers to the membership questions could sometimes be too limiting. Since MAS allows us to use exact data only, we need to convert the user's indefinite answers to definite ones for the MAS algorithm to be able to use them. This conversion can be done completely transparently, but the user interface (especially a membership query dialog) needs to be modified to support multiple paths. The result of the synthesis would be a single nondeterministic statechart or multiple separate deterministic statecharts (depending on user's needs).

Currently, we have only limited experiences in using MAS. In fact, it has not been applied in any large real world application. For correctly directing the future development of MAS such experiences are essential. Therefore, case studies and gathering experiences form an important part of our future work.

Acknowledgements. The authors wish to thank Prof. Kai Koskimies for numerous useful discussions. This work was supported by TEKES, Nokia, Metso Automation, Plenware, Sensor Software Consulting, Ebsolut, and the Academy of Finland (Project 35025).

References

[1] Airaksinen J., Koskimies K., Koskinen J., Peltonen J., Selonen P., Siikarla M., Systä T.: xUMLi: Towards a tool-independent UML processing platform. *The Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER)*, 2002, to appear.

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

[3] Balcázar, J.L., Díaz, J., Gavaldà, R., Watanabe, O.: Algorithms for learning finite automata from queries: a unified view. In D.-Z. Du, K.-I. Ko (eds.), *Advances in Algorithms, Languages, and Complexity*, Kluwer Academic Publishers, 1997, pp. 73-91.

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

[5] Chung, P., Huang, Y., Yajnik, S.: *DCOM and CORBA Side by Side, Step by Step, and Layer by Layer.* [www-csag.ucsd.edu/individual/achien/cs491-f97/papers/dcom_corba.html]

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

[7] Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. *International Journal of Foundations of Computer Science* **13** (2002), 5–51.

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

[9] Koskinen, J.: Implementing MAS on Windows NT (In Finnish). M.Sc. Thesis, Software Systems Laboratory, Tampere University of Technology, 2000.

[10] Koskinen, J., Mäkinen, E., Systä, T.: Minimally adequate synthesizer tolerates inaccurate information during behavioral modeling. In *Proc. of SCASE'01*, February 2001.

[11] Koskinen, J., Peltonen, J., Selonen, P., Systä, T., Koskimies, K.: Towards tool assisted UML development environments. In *SPLST'01*, Szeged, June 2001.

[12] Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to statecharts. In F.J. Ramming (ed.), *Distributed and Parallel Embedded Systems*, Kluwer Academic Publishers, 1999, pp. 61–71.

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

[14] Mäkinen, E., Systä, T.: MAS – an interactive synthesizer to support behavioral modeling in UML. In *Proc. of the 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, 2001, pp. 15–24.

[15] Mäkinen, E., Systä, T.: Minimally adequate teacher synthesizes statechart diagrams. *Acta Inform.* **38** (2002), 235–259.

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

[17] Microsoft Corporation: *COM (Component Object Model).* [http://msdn.microsoft.com/library/psdk/com/comportal_3qn9.htm], 2000.

[18] Microsoft Corporation: *Automation Start Page.* [http://msdn.microsoft.com/library/psdk/automat/autoportal_7145.htm], 2000.

[19] Microsoft Corporation: *COM Clients and Servers.* [http://msdn.microsoft.com/ library/psdk/com/comext_8p2r.htm], 2000.

[20] Microsoft Corporation: *Interfaces and Pointers.* [http://msdn.microsoft.com/ library/psdk/com/com_37w3.htm], 2000.

[21] OMG Corporation: OMG XML Metadata Interchange (XMI) Specification. [http://www.omg.org], 2000.

[22] The Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.3.1.* [http://www.omg.org/ technology/documents/formal/corba_2.htm], October 1999.

[23] Rational Software Corporation. *OMG Unified Modeling Language Specification v.1.3.* [http://www.rational.com], 2000.

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

[25] Schönberger, S., Keller, R., Khriss, I.: Algorithmic support for transformations in object-oriented software development. To appear in *Theory and Practice of Object Systems (TAPOS).*

[26] Selonen P., Koskimies K., Sakkinen M.: How to make apples from oranges in UML. In *Proc. of HICSS-34*, 2001.

[27] Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In *Proc. of the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000, pp. 314–323.

[28] Wikman, J.: Evolution of a distributed repository-based architecture. In *Proc. of the First Nordic Workshop on Software Architecture*, Research Report 14/98, Dept. of Computer Science and Business Administration, University of Karlskrona/Ronneby, Sweden, 1998. [http://www.hk-r.se/fou/forskinfo.nsf/]