Automatic Synthesis of Agent Designs in UML

Johann Schumann[1] and Jon Whittle[2]

[1] RIACS / NASA Ames, Moffett Field, CA 94035 USA,
`schumann@ptolemy.arc.nasa.gov`
[2] Recom / NASA Ames, Moffett Field, CA 94035 USA,
`jonathw@ptolemy.arc.nasa.gov`

Summary. It is anticipated that the UML, perhaps with domain-specific extensions, will increasingly be used to model and analyse agent-based systems. Current commercial tools for UML, however, contain a number of gaps that limit this growth potential. As an example, there is little or no support for automatic translations between UML notations. We present one such translation — from sequence diagrams to statecharts — and discuss how such an algorithm could be used in agent modeling. In contrast to other approaches, our algorithm makes a justified merging of the input sequence diagrams based on simple logical specifications of messages passed between agents/objects, and detects conflicting behaviors in different sequence diagrams. In addition, we generate statecharts that make full use of hierarchy, leading to generated designs that more closely resemble those which a designer might produce. This is important in the context of iterative design, since the designer will likely want to modify the generated statecharts to refine their behavior.

1 Introduction

There has recently been interest in investigating how the Unified Modeling Language (UML) [2] can be applied to the modeling and analysis of agent-based software systems. For example, the OMG Agent Working Group [6] is attempting to unify object-oriented approaches and current methodologies for developing agent systems. The result is a proposal for augmenting UML with agent-specific extensions (called Agent UML or AUML). This is based on the observation that the development of large-scale agent-based software requires modeling methods and tools that support the entire development lifecycle. Agent-based systems are highly concurrent and distributed and hence it makes sense to employ methodologies that have already been widely accepted for distributed object-oriented systems. Indeed, agents can be viewed as "objects with attitude" [3] and can themselves be composed out of objects. On the other hand, agents have certain features not possessed by objects — such as autonomy, the ability to act without direct external intervention; and cooperation, the ability to independently coordinate with other agents to achieve a common purpose. The precise nature of the relationship between objects and agents is as yet unclear. However, we anticipate that the use of UML (perhaps with further extensions) for modeling agent-based systems will increase.

UML is essentially a collection of notations for modeling a system from different perspectives. Current commercial tools supporting UML (e.g., iLogix's Rhapsody [12] and Rational's Rose [11]) can generate C++ or Java code from statechart designs, but there is no support for translating between UML notations themselves. The focus of our own research is in bridging this gap, and to this end, we have developed an algorithm for translating UML sequence diagrams into UML statecharts. Sequence diagrams model message passing between objects, or in the case of agents, can be used to model communications between agents (i.e., agent interaction protocols). Statecharts take a state-centric view and model the behavior of a class of objects as a collection of concurrent, hierarchical finite state machines. In agent-based systems, our translation from sequence diagrams to statecharts can be used in:

- developing *agent skeletons* [14], which give abstract descriptions of agents in terms of the events that are significant for coordination with other agents. Agent skeletons are important in studying interactions in multi-agent systems. If conversation instances between agents are expressed as sequence diagrams, our translation algorithm can generate agent skeletons semi-automatically. A similar approach is followed in [15] except that conversations are described using Dooley graphs;
- developing behavioral models (i.e., statecharts) of agents that are composed of objects. In this context, our algorithm generates initial models from a collection of scenarios (sequence diagrams) of expected behavior. We view this process as being highly iterative — the generated statecharts will be refined by the user which feeds back to refined scenarios. Because of this iterative approach, the generated statecharts must be human readable — i.e., they must appropriately divide behavior into orthogonal components, and include sensible use of hierarchy.

Our techniques apply equally to modeling agents and objects that make up agents. For this reason, we will use 'agent' and 'object' interchangeably in what follows.

A number of other approaches have been developed for translating from scenarios to state machines (e.g., [7,9,8,16]), but our approach has a number of advantages, namely:

- Scenarios will in general overlap. Most other approaches cannot recognize intersections between scenarios. Our approach, however, performs a *justified merging of scenarios* based on logical descriptions of the communications between agents. The communications are specified using the Object Constraint Language (OCL) [17] and allow identical states in different scenarios to be recognized automatically. This leads to statecharts both with a vastly reduced number of states, and also corresponding more to what a human designer would produce.
- Scenarios will in general conflict with each other. Our algorithm first *detects and reports any conflicts* based on the specifications of the communications.

- The statecharts generated by our algorithm are *hierarchical and make sensible use of concurrency.* Much of this structure is detected automatically from the communications specifications. Additional structure can be deduced from user-specified abstractions. This leads to generated statecharts that are human-readable, not just huge, flat state machines.

2 Example

We will use an ongoing example to illustrate our algorithm. The example is that of an automated loading dock in which forklift agents move colored boxes from (to) a central ramp to (from) colored shelves such that boxes are placed on shelves of the same color. The example is presented as a case study in [10] of a three-layered architecture for agent-based systems, in which each agent consists of a reactive, a local planning and a coordination layer. Each layer has responsibility for certain actions: the reactive layer reacts to the environment and carries out plans sent from the planning layer; the planning layer forms plans for individual agent goals; and the coordination layer forms joint plans that require coordination between agents. We have translated part of this example into UML as a case study for our algorithm. Figure 1 gives the static structure of part of the system, represented as a UML class diagram. Each class can be annotated with attributes (typed in OCL) or associations with other classes. `coordWith` describes whether an agent is currently coordinating its actions with another agent (`0..1` is standard UML notation for multiplicity meaning 0 or 1), and `coordGoal` gives the current goal of this other agent. We assume that all agents communicate their goals truthfully to other agents when asked. Agent interaction is based on a leader election protocol which selects an agent to delegate roles in the interaction (e.g., which agent should move away). `leader` describes whether an agent is currently a leader. The filled diamonds in the class diagram represent aggregation (the 'part-of' relationship).

Figures 2, 3 and 4 are sample sequence diagrams (SDs) for interaction between two agents. SD1 is a failed coordination. Agent[i] attempts to establish a connection with Agent[j], but receives no response[1], so moves around Agent[j]. In SD2, the move is coordinated, and SD3 shows part of a protocol for Agent[j] to clear a space on a shelf for Agent[i]. Note that these are actually *extended* sequence diagrams. 'boxShelfToRamp' is a sub-sequence diagram previously defined and 'waiting' is a state explicitly given by the user. More will be said about extended SDs in Section 3.2.

3 Methodology

An increasingly popular methodology for developing object-oriented systems is that of use-case modeling [13], in which use-cases, or descriptions of the

[1] `tm` is a timeout

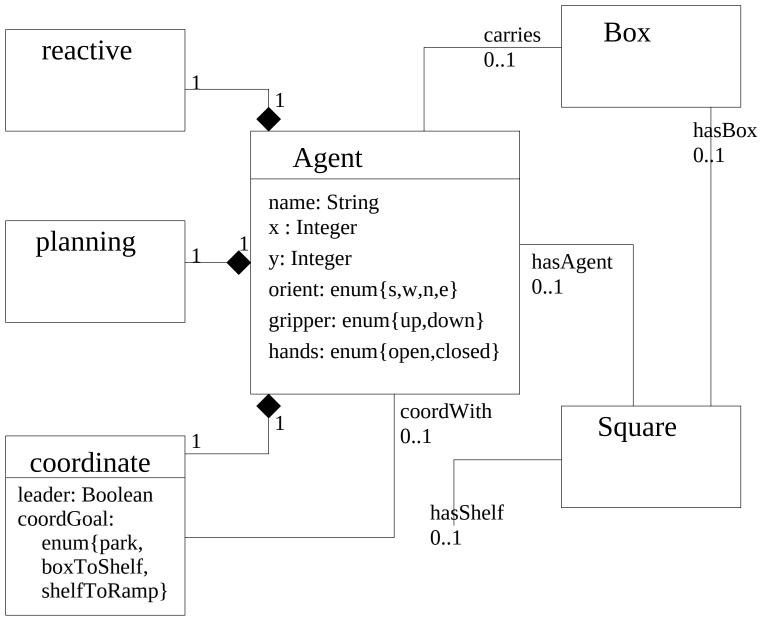
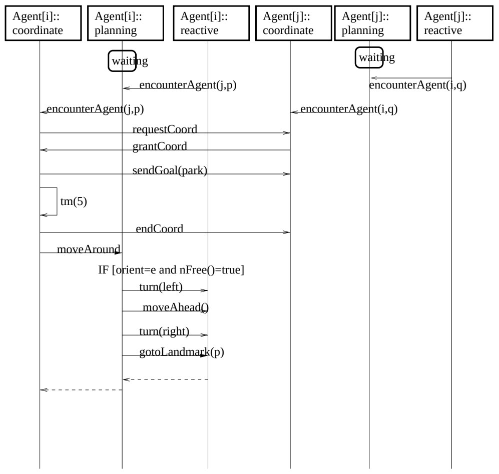


Fig. 1. The loading dock domain



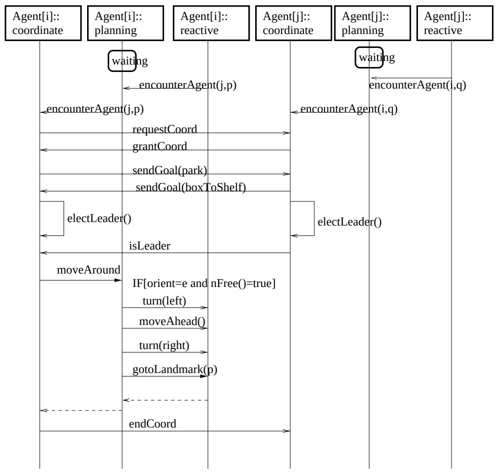


Fig. 2. Agent Interaction (SD1). **Fig. 3.** Agent Interaction (SD2).

intended use of a system, are produced initially and are used as a basis for detailed design. Each use case represents a particular piece of functionality from a user perspective, and can be described by a collection of sequence diagrams. [13] advocates developing the static model of a system (i.e., class diagram) at the same time as developing the sequence diagrams. Once this requirements

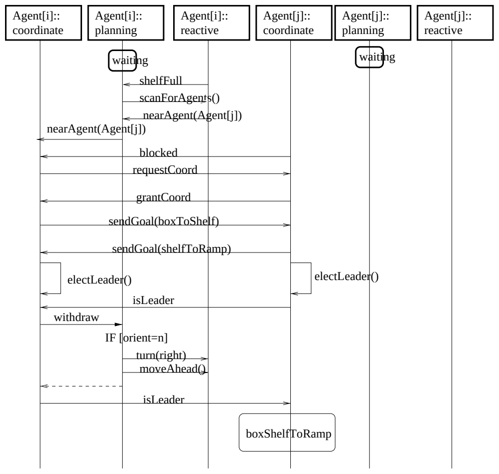


Fig. 4. Agent Interaction (SD3).

phase has been completed, more detailed design can be undertaken, e.g., by producing statecharts.

We leverage off this approach and our algorithm fits in as shown in Figure 5. From a collection of sequence diagrams, plus information from a class diagram and an OCL specification[2], a collection of statecharts is generated, one for each class. Note that the methodology is highly iterative — it is not expected that the designer gets the class diagram, sequence diagrams, or OCL specification correct first time. On the contrary, sequence diagrams will in general conflict with each other or the OCL spec, be ambiguous, or be missing information. The insertion of our algorithm enables some conflicts to be detected automatically and allows a much faster way of making modifications and seeing these modifications reflected in the statechart designs. Note that one statechart is generated for each class. This is in line with the way in which statecharts are used in tools such as Rhapsody.

3.1 OCL specification

The lack of semantic content in sequence diagrams makes them ambiguous and difficult to interpret, either automatically or between different stakeholders. In current practice, ambiguities are often resolved by examining the informal documentation but, in some cases, ambiguities may go undetected leading to costly software errors. To alleviate this problem, we encourage the user to give pre/post-condition style OCL specifications of the messages passed between objects. These specifications include the declaration of *state variables*, where a state variable represents some important aspect of the system, e.g., whether or not an agent is coordinating with another agent. This

[2] OCL is a side-effect free set-based constraint language.

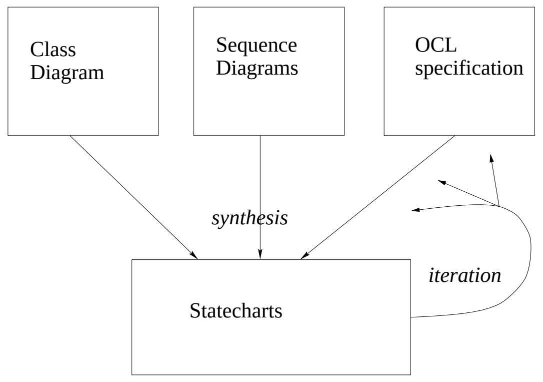


Fig. 5. Use-case modeling with Statechart Synthesis.

OCL specification allows the detection of conflicts between different scenarios and allows scenarios to be merged in a *justified* way. Note that not every message needs to be given a specification, although, clearly, the more semantic information that is supplied, the better the quality of the conflict detection. Currently, our algorithm only exploits constraints of the form $var = value$, but there may be something to be gained from reasoning about other constraints using an automated theorem prover or model checker.

Figure 6 gives specifications for selected messages in our agents example. The state variables, in the form of a *state vector*, are used to characterize states in the generated statechart. The state vector is a vector of values of the state variables. In our example, the state vector has the form:

$$\langle \texttt{coordWith\^{}}, \texttt{leader\^{}}, \texttt{coordGoal\^{}} \rangle$$

where $var\hat{} \in Dom(var) \cup \{?\}$, and ? represents an unknown value.

Our algorithm is designed to be fully automatic. The choice of the state vector, however, is a crucial design task that must be carried out by the user. The choice of state variables will affect the generated statechart, and the user should choose state variables to reflect the parts of the system functionality that is of most interest. In this way, the choice of the state vector can be seen as a powerful abstraction mechanism — indeed, the algorithm could be used in a way that allows the user to analyse the system from a number of different perspectives, each corresponding to a particular choice of state vector.

The state variables can be chosen from information present in the class diagram. For instance, in our example, the state variables are either attributes of a particular class or based on associations. The choice is still a user activity, however, as not all attributes/associations are relevant.

```
coordWith : enum {0,1}
leader : Boolean
coordGoal : enum {park, boxToShelf, shelfToRamp}

context grantCoord
   pre:  coordWith = 0
   post: coordWith = 1

context sendGoalBoxToShelf
   post: coordGoal = boxToShelf

context sendGoalShelfToRamp
   post: coordGoal = shelfToRamp

context electLeader
   pre:  leader = false

context isLeader
   post: leader = true

context endCoord
   pre:  coordWith = 1
   post: coordWith = 0 and leader = false
```

Fig. 6. Domain Knowledge for the Loading Dock Example.

3.2 Extended Sequence Diagrams

Other authors ([5,4]) have already noted that the utility of sequence diagrams to describe system behavior could be vastly increased by extending the notation. A basic SD supports the description of *exemplary* behavior — one concrete interaction — but when used in requirements engineering, a *generative* style is more appropriate, in which each SD represents a collection of interactions. Extensions that have been suggested include the ability to allow `case` statements, loops and sub-SDs. We go further than this and advocate the use of language constructs that allow behavior to be generalized. Example constructs we have devised so far include:

- $any\_order(m_1, \ldots, m_n)$: specify that a group of messages may occur in any order;
- $or(m_1, \ldots, m_n)$: a message may be any one of a group of messages;
- $generalize(m, SubSD)$: a message gives the same behavior when sent/received at any point in the sub-sequence diagram;
- $allInstances(m, I)$: send a message to all instances in I;

These extensions significantly augment the expressiveness of sequence diagrams and their utility in describing system behaviors.

4 Generating Statecharts

Synthesis of statecharts is performed in four steps: first, each sequence diagram is annotated with state vectors and conflicts with respect to the domain knowledge are detected. As the second step, the annotated SD is converted into a flat statechart. In the next step, statecharts for each class are merged into a single statechart per class. Finally, hierarchy is introduced in order to enhance readability of the synthesized statecharts.

4.1 Step I: Annotating Sequence Diagrams with State Vectors

The process to convert an individual SD into a statechart starts by detecting conflicts between the SD and the domain knowledge (and hence, other SDs). There are two kinds of constraints imposed on a SD: constraints on the state vector given by the OCL specification, and constraints on the ordering of messages given by the SD itself. These constraints must be solved and conflicts be reported to the user. Conflicts mean that either the scenario does not follow the user's intended semantics or the domain knowledge is incorrect.

More formally, the process of conflict detection can be written as follows. An annotated sequence diagram is a sequence of messages $m_1, \ldots, m_n$, with

$$s_0^{\mathsf{pre}} \xrightarrow{m_1} s_0^{\mathsf{post}}, s_1^{\mathsf{pre}} \xrightarrow{m_2} \ldots \xrightarrow{m_{r-1}} s_{r-1}^{\mathsf{post}}, s_r^{\mathsf{pre}} \xrightarrow{m_r} s_r^{\mathsf{post}} \tag{1}$$

where the s_i^{pre}, s_i^{post} are the state vectors immediately before and after message m_i is executed. S_i will be used to denote either s_i^{pre} or s_i^{post}; $s_i^{\mathsf{pre}}[j]$ denotes the element at position j in s_i^{pre} (similarly for s_i^{post}).

In the first step of the synthesis process, we assign values to the variables in the state vectors as shown in Figure 7. The variable instantiations of the initial state vectors are obtained directly from the message specifications (lines 1,2): if message m_i assigns a value y to a variable of the state vector in its pre- or post-condition, then this variable assignment is used. Otherwise, the variable in the state vector is set to an undetermined value ?. Since each message is specified independently, the initial state vectors will contain a lot of unknown values. Most (but not all) of these can be given a value in one of two ways: two state vectors, S_i and S_j $(i \neq j)$, are considered the same if they are unifiable (line 6). This means that there exists a variable assignment ϕ such that $\phi(S_i) = \phi(S_j)$. This situation indicates a potential loop within a SD. The second means for assigning values to variables is the application of the frame axiom (lines 8,9), i.e., we can assign unknown variables of a precondition with the value from the preceeding post-condition, and vice versa. This assumes that there are no hidden side-effects between messages.

A conflict (line 11) is detected and reported if the state vector immediately following a message and the state vector immediately preceding the next message differ.

```
*Input.* An annotated SD
*Output.* A SD with extended annotations

 1 for each message $m_i$ do
 2     if $m_i$ has a precondition $v_j = y$ then $s_i^{\mathsf{pre}}[j] := y$ else $s_i^{\mathsf{pre}}[j] := ?$ fi
 3     if $m_i$ has a postcondition $v_j = y$ then $s_i^{\mathsf{post}}[j] := y$ else $s_i^{\mathsf{post}}[j] := ?$ fi
 4 for each state vector $S$ do
 5     if there is some $S' \neq S$ and some unifier $\phi$ with $\phi(S) = \phi(S')$ then
 6         unify $S_i$ and $S_j$;
 7         propagate instantiations with frame axiom:
 8         for each $j$ and $i > 0$ : if $s_i^{\mathsf{pre}}[j] = ?$ then $s_i^{\mathsf{pre}}[j] := s_{i-1}^{\mathsf{post}}[j]$ fi
 9                                   if $s_i^{\mathsf{post}}[j] = ?$ then $s_i^{\mathsf{post}}[j] := s_i^{\mathsf{pre}}[j]$ fi
10     if there is some $k, l$ with $s_k^{\mathsf{post}}[l] \neq s_{k+1}^{\mathsf{pre}}[l]$ then
11         Report Conflict;
12         break;
```

Fig. 7. Extending the state vector annotations.

Example. Figure 8 shows how SD2 from Figure 3 is annotated, and how the values of the state vectors are propagated (Figure 9). In our case, there are no conflicts with the domain knowledge, and a loop (marked by a dashed line) is detected due to successful unification of different state vectors.

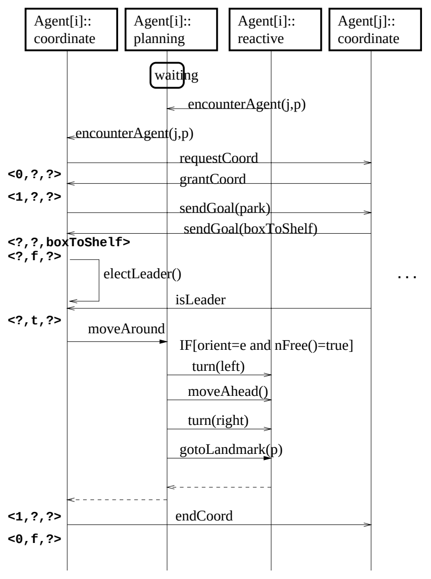


Fig. 8. SD2 (parts) with state vectors $\langle$ `coordWith^`, `leader^`, `coordGoal^` $\rangle$.

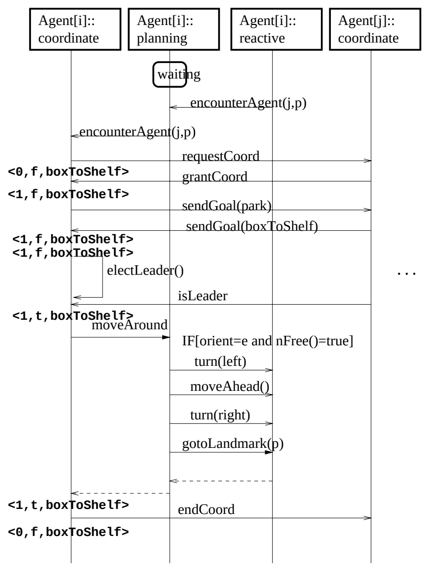


Fig. 9. SD2 after extension of state vector annotations.

4.2 Step II: Translation into a Finite State Machine

Once the variables in the state vectors have been instantiated as far as possible, a flat statechart (in fact, a finite state machine (FSM)) is generated for each individual SD, one for each involved agent or object in the SD. The finite state machine for agent A is denoted by Φ_A; its set of nodes by N_A; its transitions by $\langle n_1, \langle type, label\rangle, n_2\rangle$ for nodes n_1, n_2 where *type* is either *event* or *action*[3]; and μ_A is a function mapping a node to its state vector. $\mathcal{C}_{\mathcal{A}}$ denotes the currently processed node during the run of the algorithm. Messages directed towards a particular agent, A (i.e., $m_i^{to} = A$) are considered events in the FSM for A. Messages directed away from A (i.e., $m_i^{from} = A$) are considered actions.

The algorithm for this synthesis is depicted in Figure 10. Given a SD, the algorithm constructs one FSM for each agent (or for each object, in case we consider agents consisting of objects) mentioned in the sequence diagram. We start by generating a single starting node $n^0_{A_i}$ for each FSM (line 2). Then we successively add outgoing and incoming messages to the FSMs, creating new nodes as we proceed (lines 7-9).

An important step during FSM creation is the identification of loops: a loop is detected if the state vector immediately after the current message has been executed is the same as an existing state vector *and* if this message is state-changing, i.e., $s_i^{\mathsf{pre}} \neq s_i^{\mathsf{post}}$. Note that some messages may not have a specification, hence they will not affect the state vector. To identify states based solely on the state vector would result in incorrect loop detection.

4.3 Step III: Merging multiple Sequence Diagrams

The previous steps concerned the translation of a single sequence diagram. For merging multiple sequence diagrams, our approach is first to convert each SD individually into a flat statechart, and then merge these statecharts into a single one. In order to reduce the size of the resulting statechart, it is important to identify as many common nodes and branches as possible.

Merging statecharts derived from different SDs is based upon identifying *similar* states in these statecharts. Two nodes of a statechart are *similar* if they have the same state vector and they have at least one incoming transition with the same label. The first condition alone would produce an excessive number of similar nodes since some messages do not change the state vector. The additional required existence of a common incoming transition means that in both cases, an event has occurred which leaves the state variables in an identical assignment. Hence, our definition of similarity takes into account the ordering of the messages and the current state. Figure 11 shows how two

[3] In statecharts, a transition is labeled by e/a which means that this transition can be active only if event e occurs. Then, the state changes and action a is carried out. We use a similar notion in our definition of FSMs

```
*Input.* A SD, $S$, with agents $A_1, \ldots, A_k$ and messages $m_1, \ldots, m_r$
*Output.* A FSM $\Phi_{A_i}$ for each agent, $1 \leq i \leq k$.

for $i = 1, \ldots, k$ do
    Create a FSM, $\Phi_{A_i}$, with a initial node, $n^0_{A_i}$; $\mathcal{C}_{A_i} := n^0_{A_i}$;
for $i = 1, \ldots, r$ do
    ADD($m_i, action, m_i^{from}$);
    ADD($m_i, event, m_i^{to}$);
where ADD($mess\ m$, $type\ t$, $agent\ A$)
    if there is a node $n \in N_A$, a transition $\langle \mathcal{C}_A, \langle t, m \rangle, n \rangle$
      and $s_i^{\mathsf{post}} = \mu_A(n)$ then
        $\mathcal{C}_A := n$;
    else if there is $n \in N_A$ with $s_i^{\mathsf{post}} = \mu_A(n)$
      and $m_i$ is state-changing then
        add new transition $\langle \mathcal{C}_A, \langle t, m \rangle, n \rangle$;
        $\mathcal{C}_A := n$;
    else
        add a new node $n$ and let $\mu_A(n) := s_i^{\mathsf{post}}$;
        add transition $\langle \mathcal{C}_A, \langle t, m \rangle, n \rangle$;
        $\mathcal{C}_A := n$;
```

Fig. 10. Translating a sequence diagram into FSMs.

nodes with identical state vector S and incoming transitions labeled with l can be merged together.

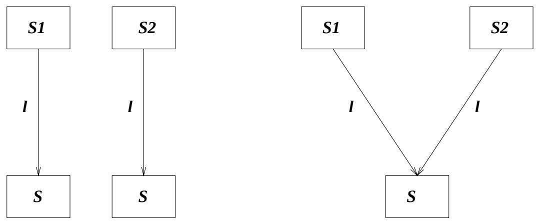


Fig. 11. Merging of similar states (before and after the merge). .

The process of merging multiple statecharts proceeds as follows: we generate a new statechart and connect its initial node by empty ϵ-transitions with the initial nodes of the individual statecharts derived from each SD. Furthermore, all pairs of nodes which are *similar* to each other are connected by ϵ-transitions. Then we remove ϵ-transitions, and resolve many non-deterministic branches. For this purpose, we use an algorithm which is a variant of the standard algorithm for transforming a non-deterministic finite automaton (NFA) into a deterministic finite automaton (DFA) [1][4].

[4] Note that the output of the algorithm is only deterministic in that there are no ϵ-transitions remaining. There may still be two transitions leaving a state labelled

5 Introducing Hierarchy

So far, we have shown how to generate FSMs without any hierarchy. In practice, however, statechart designs tend to get very large and so the judicious use of hierarchy and orthogonality is crucial to the readability and maintainability of the designs. There are several issues which comprise a "well-designed" statechart (see, for example, [?]). They include the consolidation of related behavior, the separation of unrelated behavior, and the introduction of meaningful abstractions. Our approach allows several ways for introducing hierarchy into the generated FSMs: using implicit information present in the state vectors, introducing generalizations, and using information explicitly given by the user (e.g., in a UML class diagram). These techniques will be discussed in the following.

5.1 Using the State Vector

State variables usually encode that the system is in a specific mode or state (e.g., holding a box or not). Thus it is natural to partition the statechart into subcharts containing all nodes belonging to a specific mode of the system. More specifically, we recursively partition the set of nodes according to the different values of the variables in the state vectors. In general, however, there are many different ways of partitioning a statechart, not all of them suited for good readability.

Thus, we introduce additional heuristic constraints (controlled by the user) on the layout of the statechart:

1. the maximum depth of hierarchy. Too many nested levels of compound states limit readability of the generated statechart. On the other hand, a statechart which is too flat contains very large compound nodes, making reading and maintaining the statechart virtually impossible.
2. the maximum number of states on a single level. This constraint is orthogonal to the first one and also aims at generating "handy" statecharts.
3. the maximum percentage of inter-level transitions. Transitions between different levels of the hierarchy usually limit modularity, but occasionally they can be useful. Thus their relative number should be limited (usually around 5-10%).
4. a partial ordering over the state variables. This ordering describes the sequence in which partitions should be attempted. It provides a means to indicate that some state variables may be more "important" than others and thus should be given priority. The ordering encapsulates important design decisions about how the statechart should be split up.

with the same events but different actions. Hence, our algorithm may produce non-deterministic statecharts, which allows a designer to refine the design later.

In general, not all of the above constraints can be fulfilled at the same time. Therefore our algorithm has to have the capability to do a search for an optimal solution. This search is done using backtracking (currently over different partial orders) and is strictly limited to avoid excessive run-times.

Example. Figure 12 gives a partitioned statechart for agent communication generated from SD1, SD2 and SD3. The flat statechart was first split over `coordWith`, followed by `leader` and finally `coordGoal`.

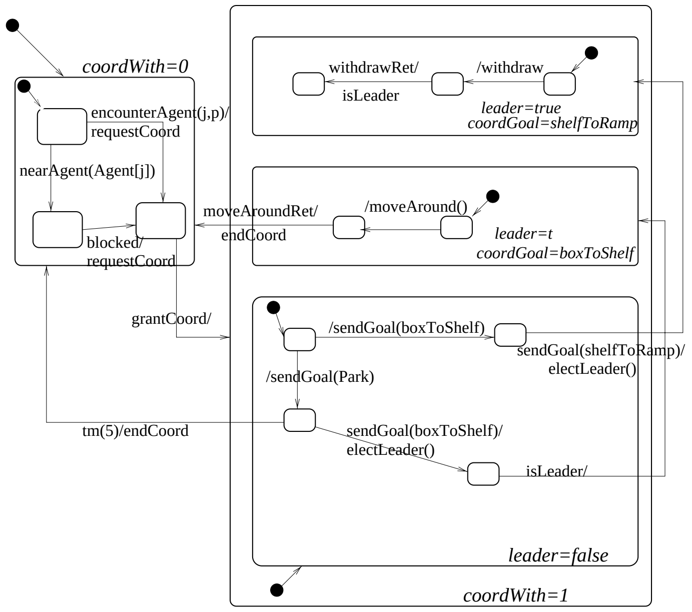


Fig. 12. Hierarchical Statechart for Agent[i]::coordinate.

5.2 Hierarchy by Language Constructs

Section 3.2 introduced extended sequence diagrams. The actual constructs used in these sequence diagrams can also be used to introduce hierarchy into the generated statechart. As an example, $any\_order(m_1, \ldots, m_n)$ can be implemented by n concurrent statecharts (see Figure 13), connected by the UML synchronization operator (the black bar) which waits until all its source states are entered before its transition is taken. This is particularly useful if $m_1, \ldots, m_n$ are not individual messages, but sub-sequence diagrams. Figure 13 shows how the other constructs mentioned in Section 3.2 can be implemented as statecharts. *allInstances* is implemented by a local variable that iterates through each instance, and sends the message to that instance.

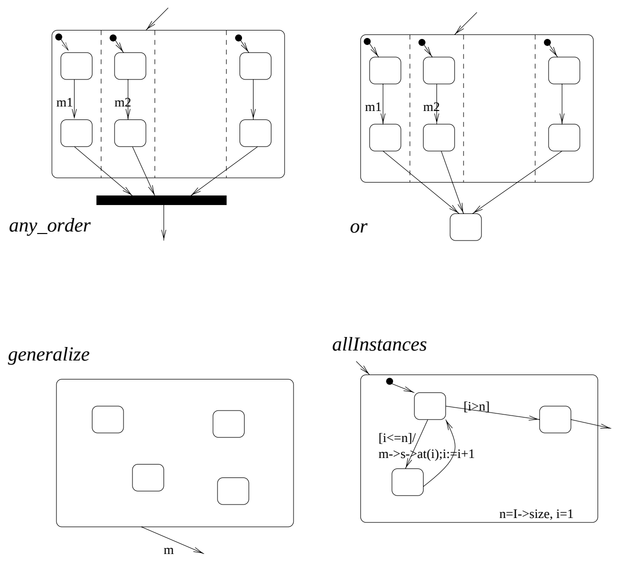


Fig. 13. Hierarchy by Macro Commands.

6 Conclusions

In this paper, we have presented an algorithm for automatically generating UML statecharts from a set of sequence diagrams. For the development of large-scale agent-based systems, sequence diagrams can be a valuable means to describe inter-agent communication. We extend sequence diagrams with additional language constructs to enable generalizations and augment them with communication pre- and post-conditions in OCL. This enables us to automatically detect and report conflicts between different sequence diagrams and inconsistencies with respect to the domain theory. These annotations are furthermore used in our algorithm to correctly identify similar states and to merge a number of sequence diagrams into a single statechart. In order to make the algorithm practical, we introduce hierarchy into the statechart.

A prototype of this algorithm has been implemented in Java and so far used for several smaller case-studies in the area of agents, classical object-oriented design [18], and human-computer interaction. In order to be practical for applications on a larger scale, our future work includes a tight integration of our algorithm into a state-of-the-art UML-based design tool.

The algorithm described in this paper only describes the forward or synthesis part of the design cycle: given a set of SDs, we generate a statechart. For full support of our methodology, research and development in two directions are of major importance: conflicts detected by our algorithm must not only be reported in an appropriate way to the designer but also should provide explanation on what went wrong and what could be done to avoid this conflict.

We will use techniques of model-generation, abduction, and deduction-based explanation generation to provide this kind of feed-back.

The other major strand for providing feed-back is required when the user, after synthesizing the statechart, refines it or makes changes to the statechart. In that case, it must be checked if the current statechart still reflects the requirements (i.e., the sequence diagrams), and in case it does, must update the sequence diagrams (e.g., by adding new message arrows).

The question if UML (or AUML), is an appropriate methodology for the design of large-scale agent-based systems must still be answered. A part of the answer lies in the availability of powerful tools which facilitate the development of agents during all phases of the iterative life-cycle. We are confident that our approach to close the gap between requirements modeling using sequence diagrams and design with statecharts will increase acceptability of UML methods and tools for the design of agent-based systems.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
3. J. Bradshaw. *Software Agents*. American Association for Artificial Intelligence / MIT Press, 1997.
4. R. Breu, R. Grosu, C. Hofmann, F. Huber, I. Krüger, B. Rumpe, M. Schmidt, and W. Schwerin. Exemplary and complete object interaction descriptions. In *Computer Standards and Interfaces*, volume 19, pages 335–345, 1998.
5. T. Gehrke and T. Firley. Generative sequence diagrams with textual annotations. In Spies and Schätz, editors,*Formal Description Techniques for Distributed Systems (FBT99)*, pages 65–72, München, 1999.
6. OMG Agent Working Group. Agent technology Green Paper. Technical Report ec/2000-03-01, Object Management Group, March 2000.
7. I. Khriss, M. Elkoutbi, and R.K. Keller. Automating the synthesis of UML statechart diagrams from multiple collaboration diagrams. In J. Bezivin and P.A. Muller, editors, *UML98: Beyond the Notation*, pages 132–147. Springer-Verlag, 1999. LNCS 1618.
8. S. Leue, L. Mehrmann, and M. Rezai. Synthesizing software architecture descriptions from Message Sequence Chart specifications. In *Automated Software Engineering*, pages 192–195, Honolulu, Hawaii, 1998.
9. T. Männistö, T. Systä, and J. Tuomi. SCED report and user manual. Report A-1994-5, Dept of Computer Science, University of Tampere, 1994.
10. J. Müller. *The Design of Intelligent Agents*. Springer, 1996. LNAI 1177.
11. *Rational Rose*. Rational Software Corporation, Cupertino, CA, 1999.
12. *Rhapsody*. I-Logix Inc., Andover, MA, 1999.
13. D. Rosenberg and K. Scott. *Use Case Driven Object Modeling with UML*. Addison Wesley, 1999.
14. M. Singh. A customizable coordination service for autonomous agents. In *Intelligent Agents IV: 4th International Workshop on Agent Theories, Architectures, and Languages*, 1998.

15. M. Singh. Developing formal specifications to coordinate heterogeneous autonomous agents. In *International Conference on Multi Agent Systems*, pages 261–268, 1998.
16. S. Somé and R. Dssouli. From scenarios to timed automata: building specifications from users requirements. In *Asia Pacific Software Engineering Conference*, pages 48–57, 1995.
17. J.B. Warmer and A.G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
18. J. Whittle and J. Schumann. Generating Statechart Designs From Scenarios. International Conference on Software Engineering (ICSE), 2000.