Generating Hierarchical State Machines from Use Case Charts

Jon Whittle
Information & Software Engineering
George Mason University
Fairfax VA 22030
jwhittle@ise.gmu.edu

Praveen K. Jayaraman
Information & Software Engineering
George Mason University
Fairfax VA 22030
pjayara1@gmu.edu

Abstract

There are many examples, in the literature, of algorithms for synthesizing state machines from scenario-based models. The motivation for these is to automate the transition from scenario-based requirements to early analysis and design models. A major challenge for such algorithms, however, is that the relationships between scenarios are usually not explicitly defined. This means that synthesis algorithms have to infer the relationships and this cannot generally be done without also inferring false positives. An alternative is to require users to explicitly give scenario relationships. The challenge here is that the additional burden placed on the user must be less than the effort saved by automatic synthesis. In this paper, we address this problem by defining a synthesis algorithm for use case charts, a language for precisely describing use cases and their relationships. Use case charts are sufficiently precise to allow the automatic generation of hierarchical state machines but retain the benefits of existing scenario-based notations by being based on UML. Use case charts provide an easy way of specifying scenario relationships but also have a formal semantics that can be used both in synthesis and to execute the use case charts. This paper presents the synthesis algorithm for use case charts and illustrates it on a significant example based on students' solutions to an analysis and design problem.

1. Introduction

Since their introduction, use cases have become a method of choice for elaborating software requirements. A use case can be defined as a set of scenarios (including the "happy day" scenario and alternatives), where a scenario is an expected execution trace of a system. Use cases are a part of many major UML-based OOAD methodologies. Typically, they are used as a starting point for developing interaction diagrams which are in turn used in developing state machines for objects with state.

The transition from interaction diagrams (e.g., UML sequence diagrams) to finite state machines (FSMs) is one of the key activities in OOAD. The transition is essentially from the global view of interaction diagrams to a local, object-based view. Each interaction diagram contributes to the state-based definition of one or more objects participating in the interaction. Many authors (e.g., [1,2,3]) have tried to automate the transition from interaction diagrams to FSMs. This is important research for the following reasons. First, it automates a key activity of many OOAD processes. Secondly, it transforms scenarios (given as interaction diagrams) into an executable form (namely, FSMs). Since FSMs are executable, they can be simulated. Hence, automation of the transformation is a way of simulating scenario-based requirements. The simulation can be used in requirements validation.

Algorithms that transform scenarios into state machines are often called *synthesis algorithms*. There are two principal gaps in existing synthesis algorithms. Firstly, scenario-based specifications often do not make explicit the relationships between scenarios or between use cases. In other words, scenarios are written in isolation and their associations (e.g., overlapping, parallelism) are not specified. This is partly because early versions of UML did not support the specification of these relationships. This has changed in UML2.0 [4] which introduces interaction overview diagrams (IODs). In the absence of scenario relationships, synthesis algorithms have taken one of two approaches to elicit them. Either the algorithm infers the relationships (e.g., [3] uses inductive learning to do this) or the algorithm requires the scenario writer to explicitly give the relationships in some form (e.g., by explicitly identifying overlapping states [1]). The inference approach is problematic because it results in false positives. Specification of explicit relationships is problematic because it may rely on a non-standard methodology with which users are not familiar.

14th IEEE International Requirements Engineering Conference (RE'06)
0-7695-2555-5/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

This paper takes the approach of explicit relationship specification but does so in a way that is based on existing notations, namely UML. A notation for specifying use case scenarios, called *use case charts*, is used that is a natural extension of existing UML notations. The key contribution of this paper is a synthesis algorithm for use case charts. Use case charts, first introduced in [5], are a 3-level notation based on extended UML activity diagrams. The main application of use case charts to date has been to simulate use cases but use case charts are also precise enough for test generation and automated validation. Our synthesis algorithm goes beyond previous algorithms:

- It takes into account a rich set of relationships between scenarios such as preemption, parallelism, negation.
- It takes into account relationships between *use cases*—previous algorithms do not consider use case relationships, only scenario relationships.
- It synthesizes *hierarchical* state machines which are therefore human-readable and can easily be built upon in subsequent analysis and design steps.

The remainder of this paper is organized as follows. Section 2 describes use case charts. Section 3 is the main contribution—a synthesis algorithm for use case charts. Section 4 describes a preliminary validation on a significant example based on a set of students' OOAD models. Section 5 compares our approach to related work and is followed by conclusions.

2. Use Case Charts

Use case charts are a precisely defined, graphical language for use cases for which a formal semantics has been defined [6]. The idea behind use case charts is illustrated in Figure 1.

For the purposes of this paper, a use case is considered to be a set of scenarios, where a scenario is an expected execution trace of a system. The functionality of a system can be given as a set of use cases—that is, a set of sets of scenarios. A use case chart specifies the scenarios for a system's use cases as a 3-level description: level-1 is the use case chart, an extended UML activity diagram in which the nodes are use cases; level-2 is a set of scenario charts, or extended activity diagrams where the nodes are scenarios; level-3 is a set of UML2.0 interaction diagrams. Each level-1 use case node is defined by a level-2 scenario chart (i.e., a set of connected scenario nodes). Each level-2 scenario node is defined by a UML2.0 interaction diagram.

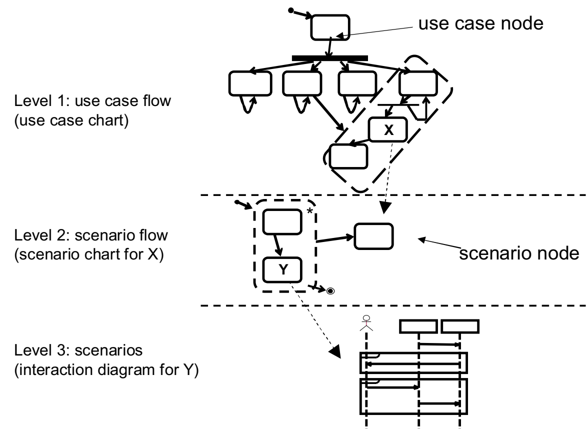

Figure 1: Use Case Charts.

A formal denotational trace-based semantics for use case charts is given in [6]. Informally, control flow of the entire use case chart starts with the initial node of the use case chart (level-1). Flow then passes between use case nodes along the arrows of the level-1 activity diagram. When flow reaches a use case chart node at level-1, the level-2 scenario chart defining this node is executed, with flow starting from the scenario chart's initial node. Flow exits a scenario node when a final node is reached. Scenario charts may have two types of final nodes—a final success node represents successful completion of the scenario chart and a final failure node represents completion but with failure. Flow only continues beyond the current use case node if a final success node is reached in the use case's defining scenario chart. The semantics of each scenario chart is similar to that for high-level message sequence charts (hMSCs) [7]. Each scenario chart node is defined by a UML2.0 interaction diagram. Hence, when flow passes into a scenario chart node, the defining interaction diagram is executed. When the interaction diagram completes, flow returns to the level-2 scenario chart, exits the scenario node at that level and continues with the next scenario node.

The intention is to reuse as much of the notation of UML2.0 as possible. This makes it easy for practitioners to learn the language. The activity diagrams used in use case charts and scenario charts are a restricted version of UML2.0 activity diagrams but with some additional relationships between nodes. They are restricted in that they do not include object flow, swimlanes, signals etc. They do include additional notations, however. The concrete syntax reuses as much of the activity diagram notation as possible. Informally, the allowed arrow types between nodes (either in use case or scenario charts) are given

14th IEEE International Requirements Engineering Conference (RE'06)
0-7695-2555-5/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

as follows, where, for each arrow, X and Y are either both scenario nodes or both use case nodes:
1. X continues from Y (i.e., the usual activity diagram arrow)
2. X and Y are alternatives (the usual alternative defined by a condition)
3. X and Y run in parallel (the usual activity diagram fork and join)
4. X preempts Y—i.e., X interrupts Y and control does not return to Y once X is complete, shown by the stereotype <<preempts>> from X to Y.
5. X suspends Y—i.e., X interrupts Y and control returns to Y once X is complete, shown by the stereotype <<suspends>> from X to Y.
6. X is negative—i.e., the scenarios defined by X should never happen. This is shown by an arrow stereotyped with <<neg>> to X and where the source of the arrow is the region over which the scope of the negation applies.
7. X may have multiple copies—i.e., X can run in parallel with itself any number of times. This is shown by an asterisk attached to node X.

In addition, use case charts and scenario charts may have regions (graphically shown by dashed boxes) that scope nodes together. Arrows of type (4) and (5) may have a region as the target of the arrow. Arrows of type (6) and (7) may have a region as the source of the arrow. All other arrows do not link regions. Arrow types (4), (5), (6) are not part of UML2.0 activity diagrams (although there is a similar notation to (4) and (5) for interruption). Activity diagrams do have a notion of region for defining an interruptible set of nodes. Regions in use case charts, however, are a general-purpose scoping mechanism not restricted to defining interrupts. In addition, there are minor extensions to interaction diagrams.

Use case charts are particularly suited for defining the scenarios in concurrent, distributed systems. Note that actors are not explicitly shown on use case charts—they appear instead as triggering participants at level 3. We do not intend use case charts to replace UML use case diagrams but rather to complement them. Therefore, we would expect actors to appear on use case diagrams as normal.

2.1 Use Case Chart Example

This section shows how to model an automated shuttle system case study [8] using use case charts. This case study is a non-trivial application based on "New Rail Technology Paderborn." The University of Paderborn made this case study available as a benchmark problem. The case study is used in Section 4 as a validation of our synthesis algorithm.

In the case study, autonomous shuttles transport passengers between stations. When a passenger requires transport, a central broker asks all active shuttles for bids on the transport order. The shuttle with the lowest bid wins. A complete set of requirements for this application is given in [8]. Figure 2 shows level 1 of a use case chart that includes use cases for initialization of the system, maintenance and retirement of shuttles, and transportation (split into multiple use cases). Figure 3 is a scenario chart (level 2) that defines the Carry Out Order use case. The Make a Bid use case consists of a single scenario and is shown as an interaction diagram (level 3) in Figure 4.

Figure 2 gives the relationships between the major use cases and, for example, shows that there are 3 use cases involved in transporting a passenger. The execution of these use cases can be preempted by the retirement use case. Figure 3 is a refinement of the Carry Out Order use case. It consists of transporting a passenger and then informing a central broker that the task is complete. In addition, it states that a shuttle cannot move to an intermediate station during this transportation process (as specified in the requirements). Finally, Figure 4 is an interaction diagram of the bidding process. The interaction fragments **all** and **exist** mean, respectively, that all shuttles must receive new order information and at least one bid must be received by the Controller. The semantics of these fragments are explained in [6].

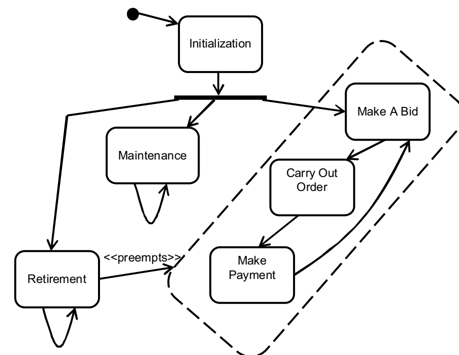

Figure 2: Use Case Chart (Level 1)

The motivation to have 3 levels is because level 1 shows use case relationships whereas level 2 shows scenario relationships.

2.2 Use Case Chart Syntax

For completeness, we give here a formal definition of the use case chart abstract syntax. The concrete syntax is based on activity diagrams and has already been described. The abstract syntax for level 2 scenario charts is as follows.

14th IEEE International Requirements Engineering Conference (RE'06)
0-7695-2555-5/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

A scenario chart $(S, R_S, E_S, s_0, S_F, S_{F^*}, L_S, f_S, m_S, L_E)$ is a graph where S is a set of scenario nodes, R_S is a set of regions, $E_S \subset \mathrm{P}(S \cup R_S) \times \mathrm{P}(S \cup R_S)$ is a set of arrows, with labels from L_E, $s_0 \in S$ is the unique initial node, $S_F \subset S$ is a set of final success nodes, $S_{F^*} \subset S$ is a set of final failure nodes, L_S is a set of scenario labels, $f_S : S \rightarrow L_S$ is a total, injective function mapping each scenario node to a label and $m_s : S \cup R_S \rightarrow \{+,-\}$ is a total function marking whether or not each scenario or region can have multiple concurrent executions. The labels in L_S are references to an interaction diagram. L_E is defined to be the set *{normal, neg, preempts, suspends}*. L_S is the set of words from some alphabet Σ.

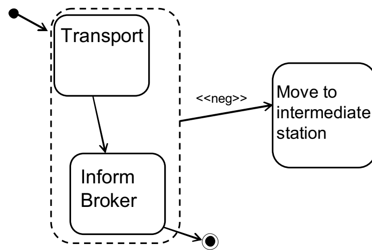

Figure 3: Scenario Chart for Carry Out Order

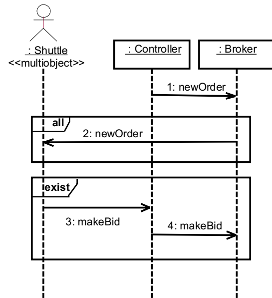

Figure 4: Interaction Diagram for Make A Bid

This definition describes a graph where edges may have multiple source nodes and multiple target nodes. This captures the notion of fork and join from activity diagrams which can be taken care of by allowing edges to have multiple source nodes and/or multiple target nodes. Multiple source nodes lead in the use case chart graphical notation to a join and multiple target nodes lead to a fork. An edge with both multiple sources and multiple targets is equivalent to a join followed by a fork. Regions are a scoping mechanism used to group scenario nodes.

As stated previously, the intuition behind final success and final failure nodes is that a final success node denotes successful completion of the scenario chart; a final failure node denotes that the scenario chart completes but unsuccessfully.

The definition omits the notion of conditions on edges, for the sake of clarity, but it is enough to say that guards could be placed on arrows leaving a node.

The abstract syntax for a use case chart is almost identical except that a use case chart has only one type of final node (for success) and each use case node maps to a scenario chart not an interaction diagram. Only one type of final node is required for use case charts because there is no notion of success or failure—either a use case chart completes or it does not.

UML2.0 sequence diagrams are defined by a metamodel in [4]. In the algorithm description in the next section, we view a sequence diagram as a sequence of events for each participant object, ordered vertically along the participant's lifeline. An event can be a message, a UML2.0 continuation[1], or a fragment. Fragments are defined recursively by a sequence of events. We assume that all messages are horizontal and that, for each sequence diagram, there is a single top-level fragment with operator **seq**. According to this definition, the diagram in Figure 4 would be represented by three sequences, one for each participant. The event sequence for Broker is *seq[newOrder, all[newOrder], exist[makeBid]]*, where *[..]* denotes a recursive definition.

3. Synthesis of State Machines

This section presents a synthesis algorithm for converting use case charts into a set of hierarchical state machines. The novelty of the algorithm is outlined in the three bullet points on page 2.

3.1 Level 3 Synthesis

This subsection details the conversion to hierarchical state machines for a level 3 UML2.0 sequence diagram. In UML2.0, sequence diagrams consist of nested interaction fragments, each of which has an interaction operator. A sequence diagram has a single top-level fragment with the operator **seq** denoting that the fragment operands are joined by weak sequential composition [4]. (Participants in different fragments joined by weak sequencing may progress independently of each other unless there are explicit messages sent between the participants.) Other operators considered here are **par**, **opt**, **neg**, **alt**, which define parallel, optional, negative and alternative sequences, respectively. The new operators **all** and

[1] A continuation is essentially a label on a participant's lifeline. Two continuations with the same name indicate that the continuations refer to the same state. Continuations have also been called state labels.

14th IEEE International Requirements Engineering Conference (RE'06)
0-7695-2555-5/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

exist are not dealt with here due to lack of space. We do not currently consider create/destroy messages or timing constraints. Table 1 gives the synthesis algorithm for a UML2.0 sequence diagram. It follows previously published algorithms such as [9] but also includes interaction fragments. For presentational purposes, some simplifying assumptions are made. We assume, for example, that all messages are asynchronous and horizontal. Figure 5 illustrates the synthesis algorithm for the interaction fragment operators and is given as an aid to understanding Table 1. The LHS (left-hand-side) of the figure shows example sequence diagrams and the RHS (right-hand-side) shows their translation into a state machine for *B*.

Input. A sequence diagram containing object O and a list of events $e_1,\ldots,e_r$ along O's lifeline.
Output. A hierarchical finite state machine (HFSM) for O.

```
1 Main
2  let C := new HFSM();
3  C.setFirst(n0 := new State());
4  C.setLast(n1 := new State());
5  let curr_state := n0 ;
6  let negMap = new Map();
7  for i = 1,…,r do
8   processEvent(ei, C);
9  done
10 createTransition(curr_state, n1, nil, nil);
11 foreach state in negMap
12  let fsm := C.getsubFSM(negMap.get(state));
13  copyAndPointToError(state, fsm);
14 done;

15 processEvent (Event e, HFSM C)
16 if (e is a message) addMessage (e, C);
17 if (e is a label) addLabel (e, C);
18 if (e is a fragment) {
19  E := e.getEvents();
20  case e.getOperator() in
21   "seq" : foreach ei in E(1)
22               processEvent(ei, C);
23              done
24   "alt" : C.addState(endFrag := new State());
25             tmp := curr_state;
26             foreach Ei in E
27               foreach ej in Ei
28                processEvent(ej, C);
29              done
30               createTransition(curr_state, endFrag, nil, nil);
31               curr_state := tmp;
32             done
33             curr_state := endFrag;
34   "par" : C.addState(endFrag := new State());
35              tmp := curr_state;
36              C.addState(parState := new State());
37              createTransition(curr_state, parState, nil, nil);
38              foreach Ei in E
39                parState.addRegion(R:=new OrthogRegion());
40                C.addState(parInit := new State());
41                R.setInitial(parInit);
42                curr_state := parInit;
43                foreach ej in Ei
44                  processEvent(ej, R);
45                done
46                createTransition(curr_state, endFrag, nil, nil);
47                curr_state := tmp;
48              done
49              curr_state := endFrag;
50   "opt" : tmp := curr_state;
51              foreach ei in E(1)
52                processEvent(ei, C);
53              done
54              createTransition(curr_state, tmp, nil, nil);
55              curr_state := tmp;
56   "neg" : tmp := curr_state;
57              foreach ei in E(1)
58                processEvent(ei, C);
59              done
60              negMap.put(curr_state, tmp);
61              curr_state := tmp;
62  esac
63 }
64 return;

65 addMessage (Event e, HFSM C)
66 C.addState(n := new State());
67 if source(e) = O
68  createTransition(curr_state, n, nil, e.getName());
69 else if target(e) = O
70  createTransition(curr_state, n, e.getName(), nil);
71 curr_state := n;
72 return;

73 addLabel (Event e, HFSM C)
74 l := e.getName();
75 State n := lookupLabel(l, C);
76 if (n == nil) C.addState(n := new State(l));
77 createTransition(curr_state, n, nil, nil);
78 return;
```

Table 1: Sequence diagram synthesis.

The input to Table 1 is a UML2.0 sequence diagram. The algorithm is given for a single object in the sequence diagram and it is assumed that the vertical ordering of fragments and messages along the lifeline for that object is known. Along the lifeline, there may be occurrences of messages, state labels or fragments. To capture this, we say that the input is a sequence of events $e_1,\ldots,e_r$ along O's lifeline. Fragment events recursively contain events. Synthesis for all objects is done by just applying the algorithm for each object. The output of the algorithm is a hierarchical state machine for O. The function *getEvents* in line 19 returns the recursively defined set of events for a fragment. Since fragments may have multiple operands, *getEvents* returns an ordered set of

IEEE COMPUTER SOCIETY

sequences of events where the cardinality of the set is the same as the number of operands. *E(j)* (e.g., see line 21) is a projection operator that returns the *j*th element of an ordered set *E*.

The algorithm in Table 1 works as follows. The *Main* procedure creates a state machine for O with initial and final nodes and then processes the input events in sequence (lines 1-10). Negative fragments require special handling (lines 6 & 11-14). *setFirst* (line 3) marks the initial state of a HFSM. Similarly, for *setLast* (line 4). These initial and final states are used as hooks to connect HFSMs generated from different sequence diagrams but that are related. (On the RHS of Figure 5, the greyed states are initial and the black states are final.)

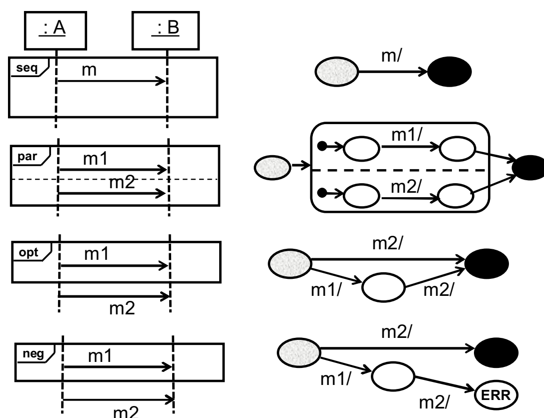

Figure 5: Example Synthesis at Level 3

The heart of the algorithm is the *processEvent* procedure (line 15) which creates states and transitions for each event. Line 16 handles messages by calling *addMessage* (lines 65-72). A message directed away from O to another object becomes an action to send that message in the HFSM for O (lines 67-68). A message directed towards O becomes a triggering event in the HFSM and results in a transition to a new state (lines 69-70). *createTransition(n,m,ev,ac)* creates a transition from state *n* to state *m* with event *ev* and action *ac*. Either *ev* or *ac* may be empty (given as *nil*).

A state label event results in a named state with that label (lines 17 and 73-78). All references of the label result in a transition to this labeled state. Line 75 checks if a state with the same label already exists. If so, a transition is created to this state. Otherwise, a new state is created.

A fragment's events are recursively processed (lines 18-63) and the fragment operator determines what kind of states are introduced into O's HFSM – **alt** leads to branching states; **par** leads to a hierarchical state containing an orthogonal region; and **opt** gives two paths (with and without the optional events). **neg** leads to a path containing the negative events with a transition to a special ERROR state. That is, negation is handled by constructing a branch in the HFSM such that if this branch is taken, the entire HFSM goes into the ERROR state. **seq** just results in a new state with a transition from the current state. The pseudo-code for these operators is self-explanatory but Figure 5 gives some simple examples.

In Table 1, *Main* contains special handling for the **neg** case (lines 11-14 & 56-61). A sequence of negative events (e.g., m1 in Figure 5) is usually followed by a sequence of positive events (e.g., m2 in Figure 5). The semantics of this is that the error only occurs if the negative events are followed by the positive events (i.e., m1, m2 in Figure 5). Hence, the positive events are replicated – they appear once for the positive case and once for the concatenation of the negative and positive events. The map in lines 6, 12 and 60 in Table 1 is used to keep track of the points where this replication must occur. Once all processing of the negative events has occurred (line 59), the last state for the negative events is stored in the map (*curr_state* at line 60). Lines 11-14 paste the positive events onto the end of the last state for the negative events. This is done by querying the map to return the sub-state machine corresponding to the positive events (*getsubFSM* at line 12) and then copying this sub-state machine to the end of the negative events (line 13).

3.2 Level 2 Synthesis

Table 2 gives the algorithm for converting a level 2 scenario chart into a HFSM. Again, the algorithm is given for a single object O but is easily extended to generate HFSMs for all participating objects.

The input is a set of scenario nodes $S_1,\ldots,S_p$. S_1 is the initial node of the scenario chart and S_p and S_{p-1} are finalSuccess and finalFailure nodes, respectively. The scenario chart also contains regions $r_1,\ldots,r_q$ where each region contains a set of scenario nodes. Arrows $a_1,\ldots,a_r$ are from sets of scenarios or regions to sets of scenarios or regions. The output is a HFSM for O.

The *Main* procedure (lines 1-33) starts by recursively applying the level 3 synthesis algorithm to each scenario node (lines 2-7). The level 3 algorithm returns, for each scenario node, a HFSM for O with special first and last states (lines 5-6) that denote the initial and final state of the interaction diagram associated with the scenario node. (As stated earlier, these are given by grey and black states, respectively, in Figures 5 and 6.) *Main* then creates a new HFSM for O and copies there the states and transitions from each of the state machines derived from the scenario nodes (lines 8-18). This new HFSM has special final success

0-7695-2555-5/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

(see line 10) and final success (see line 11) nodes that will be required during level 1 synthesis to connect the HFSMs for O generated for each use case node.

Input. A scenario chart, containing object O, and consisting of scenarios $S_1,\ldots,S_p$ (S_1, S_p and S_{p-1} being initial and finalSuccess/finalFailure nodes), regions $r_1,\ldots,r_q$, arrows $a_1,\ldots,a_r$ between $S_1,\ldots,S_p$ and/or $r_1,\ldots,r_q$
Output. HFSM for O

1 *Main.*
2 for i = 1,…, p do
3 // Apply level 3 synthesis for S_i.
4 let C_i := S_i.convertToFSM(O);
5 let $first_i$:= C_i.getFirst();
6 let $last_i$:= C_i.getLast();
7 done
8 let C := new HFSM();
9 C.setFirst(n_0 := new State());
10 C.setLastSuccess(n_1 := new State());
11 C.setLastFailure(n_2 := new State());
12 let curr_state := n_0;
13 createTransition (n_0, $first_1$, nil, nil);
14 createTransition ($last_{p-1}$, n_1, nil, nil);
15 createTransition ($last_p$, n_2, nil, nil);
16 for i = 1,…, p do
17 Copy states and transitions from C_i to C
18 done

19 for i = 1,…,q do
20 C.addState($hierarchicalState_i$:= new State());
21 foreach S_j in r_i
22 Place C_j inside $hierarchicalState_i$
23 done

24 for i = 1,…,r do
25 case type(a_i) in
26 "sequential": seqTransition (source(a_i), target(a_i));
27 "par": parTransition(source(a_i), target(a_i));
28 "preempt": preemptTransition(source(a_i), target(a_i));
29 "suspends": suspendTransition(source(a_i), target(a_i));
30 "neg": negTransition(a_i);
31 esac
32 done
33 return;

34 *seqTransition (Scenario S_j, Scenario S_k)*
35 createTransition ($last_j$, $first_k$, nil, nil);
36 return;

37 *preemptTransition (Scenario S_k, Region r_j)*
38 createTransition ($hierarchicalState_j$, $first_k$,
S_k.getFirstMessage(), nil);
39 return;

40 *suspendTransition (Scenario S_k, Region r_j)*
41 createTransition ($hierarchicalState_j$, $first_k$,
S_k.getFirstMessage(), nil);
42 $hierarchicalState_j$.addHistoryMarker();
43 createTransition ($last_k$, $hierarchicalState_j$, nil, nil);
44 return;

45 *parTransition (Scenario Set Z_j, Scenario Set Z_k)*
46 OrthogonalState srcST = new State();
47 OrthogonalState destST = new State();
48 srcST.createRegions(Z_j);
49 destST.createRegions(Z_k);
50 createTransition(srcST, destST, nil, nil);
51 return;

52 *negTransition (Region r_j, Scenario S_k)*
53 OrthogonalState negST = new State();
54 negST.createRegions({$hierarchicalState_j$, S_k});
55 let ERR_STATE := new state();
56 createTransition(lastk, ERR_STATE, nil, nil);
57 return;

Table 2: Scenario chart synthesis.

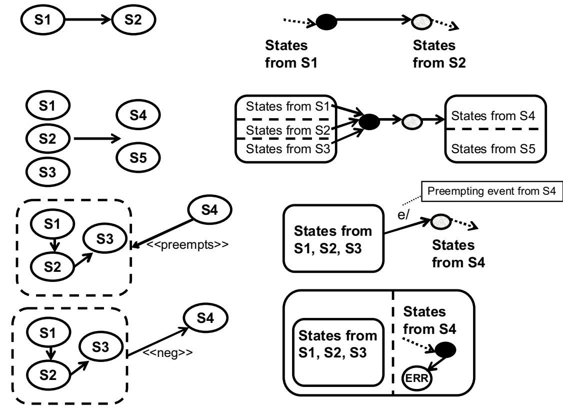

Figure 6: Level 2 Synthesis Examples

Regions are handled by creating a new hierarchical state (lines 19-23). This hierarchical state contains all states generated for scenario nodes in that region.

Arrows are dealt with according to what type of arrow they are (lines 24-33). Recall that the possible types (see abstract syntax definition is Section 2.2) are normal arrows (sequential transitions), parallel arrows, preempting arrows, suspending arrows and negative arrows. Figure 6 gives some simple examples. (Suspension is not shown because it is similar to preemption.) Sequential transitions merely connect HFSMs generated for two scenario nodes (lines 34-36). *preemptTransition* (lines 37-39) denotes the fact that a scenario node, S_k, preempts a region, r_j. r_j is transformed into a hierarchical state. A transition leaves this hierarchical state with event the first message of S_k. This captures the fact that any state in the region can be preempted by this message. *suspendTransition* works similarly (lines 40-44). The only difference between preemption and suspension is

IEEE COMPUTER SOCIETY

that once the suspending scenario is finished, control returns to the originating region. This is captured in the HFSM by introducing a history marker in the hierarchical state, so that the state caches its current state on exit, and a transition that returns to the hierarchical state once the suspending events are done.

Parallel arrows (lines 45-51) are converted into orthogonal regions in the HFSM. Parallel arrows have multiple source and target scenario nodes (hence, the sets Z_j, Z_k in line 45). A state with orthogonal regions is created to hold the source scenario nodes (with one orthogonal region for each source node) and, similarly, each target scenario node appears in the output in an orthogonal region. A transition is then introduced in the HFSM to handle the parallel arrow from the source orthogonal state to the target orthogonal state.

Finally, negative arrows also result in orthogonal regions in the output (lines 52-57). Negation is handled by monitoring for the negative events and transitioning to a special ERROR state if they occur. This is similar to level 3 synthesis. At level 2, the negative events under monitor are placed into an orthogonal region so that if the sequence of negative events ever occurs (even with other events interleaved), then the ERROR state will be entered. Note that the interpretation of negation is slightly different at level 2 than at level 3. At level 2, a negative sequence may have other messages interleaved with it. At level 3, however, the negative sequence is the flattening of the **neg** fragments. See [6] for details.

3.3 Level 1 Synthesis

Due to space restrictions, we do not give the algorithm for level 1 synthesis. The algorithm is almost identical to that in Table 2 except that level 1 use case charts have only one type of final node. The other difference is that HFSMs generated at level 1 are connected taking into account final failure and final success nodes generated at level 2.

3.4 Synthesis Example

This section gives an example of synthesis for the use case chart of the shuttle system described in Section 2. Figure 2 is the level 1 chart. Figures 3 and 4 are refinements. The shuttle system example was specified completely in our SCASP tool. SCASP implements the synthesis algorithm described in this paper and allows the user to specify a use case chart and to generate and view HFSMs graphically.

Figure 7 gives the HFSM generated for the Shuttle class based on the use case chart defined in Figures 2-4. Figure 7 is a simplified version of the full example and was generated by giving very simple sequence diagrams at level 3 for each scenario node. The full example is reported on in Section 4 but the complete HFSM cannot be given due to its size. (Note that both preemption/suspension assume a priority of outermost transition selection in the state machine semantics.) However, even a simplified form of this example shows the power of the synthesis algorithm because a rich hierarchy of states is automatically generated.

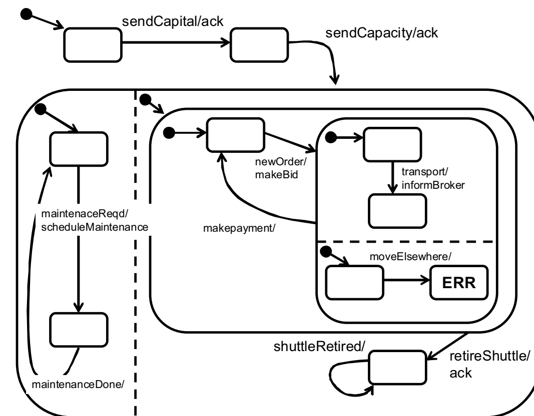

Figure 7: HFSM Generated for Shuttle

There are a number of hierarchical states generated in Figure 7. At the top level, there are two orthogonal regions—one for maintenance and one for transportation. This mirrors the parallel fork in Figure 2. The transportation region is further divided. The use case chart region in Figure 2 (denoted by the dotted rectangle) becomes a hierarchical state in Figure 7. This entire hierarchical state is exited if the retirement event occurs. This corresponds to the preempting arrow in Figure 2. Within the hierarchical state for transportation, there is a further division into orthogonal regions. This is done to capture the negative behavior from Figure 3. Recall that Figure 3 says that a shuttle cannot move to an intermediate station while it is transporting a passenger. If it does, the HFSM goes into the special ERR state. This is captured in the generated HFSM by introducing an orthogonal region for the negative behavior that essentially monitors for that behavior—if, at any point during the transport, the event *moveElsewhere* arrives, then ERR is entered.

4. Results

As a preliminary validation of use case chart synthesis, we implemented a prototype tool, SCASP, that takes any use case chart as input and generates a set of HFSMs, one for each object with state-dependent behavior. SCASP is implemented as an Eclipse plugin and the use case charts can be created using IBM

14th IEEE International Requirements Engineering Conference (RE'06)
0-7695-2555-5/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY

Rational Software Modeler (RSM) where each level is represented using standard UML 2.0 notations.

So far, we have used SCASP to generate HFSMs for a number of industrial or industrial-strength examples. We applied it to a weather update subsystem of an air traffic control system developed at NASA [10]. We have also applied SCASP to a complete version of the shuttle system case study already introduced in this paper. SCASP can be applied at multiple phases of the software lifecycle. It can be applied during requirements development when the use cases are black-box, with no internal details of the system. Alternatively, it can be applied during requirements analysis, when internal objects are identified. For the shuttle system, we applied SCASP at the requirements analysis phase. We used the shuttle system as a term project in a software architecture and design graduate course at George Mason University. We then took student solutions to the requirements analysis phase and reengineered them into a use case chart specification. This involved taking a collection of UML interaction diagrams and adding structure using the 3-level layering in use case charts. We did not have students write the use case charts directly because of the training time that would be needed. This is regarded as future work. Even so, this exercise showed that it is possible to represent complex analysis models with use case charts and, moreover, synthesis of correct HFSMs can be automatically generated using SCASP. Table 3 gives an indication of the complexity of the use case chart for the shuttle system at the analysis level and of the generated HFSMs.

The data in Table 3 provides evidence of the scalability of the use case chart notation and the synthesis algorithm. The example was industrial-strength in size but also used many of the more complex arrow types such as preemption and negation. The 3-level structure of use case charts allow a wide range of behavior to be specified in a controllable way. If UML2.0 sequence diagrams alone had been used to capture the same information, the number of modeling elements needed would be at least an order of magnitude larger. Indeed, it would be extremely difficult if not impossible to capture some of the more complex behavior, such as preemption.

The table also gives an indication of the size and complexity of the HFSMs generated. These data do not, of course, provide any evidence of how easy it is to develop the use case chart.

Table 3: Size/complexity of shuttle system study

Use Case Chart Data

Data	Value
Node-related data	
# level 1 nodes	7
# level 1 regions	1
# level 2 nodes	10
# level 2 regions	1
# participating objects at level 3 (average per sequence diagram)	7.2
# interaction fragments at level 3	23
# messages at level 3	144
Arrow-related data	
# arrows at level 1 of type:	
sequential	6
parallel	3
preempting	1
negating	0
# arrows at level 2 of type:	
sequential	2
parallel	0
preempting	0
negating	1

Generated HFSM Data

Data	Value
# state dependent objects	7
# generated states	117
# generated hierarchical states	4
# generated orthogonal regions	9
# generated transitions	184

5. Related Work

UML2.0 [4] introduces interaction overview diagrams, a notation based on activity diagrams, for specifying relationships between interaction diagrams (e.g., sequence diagrams). Interaction overview diagrams (IODs) can be used to more precisely describe use cases by a set of interaction diagrams connected by activity diagram relationships, e.g., concurrency. Whilst IODs provide much needed expressiveness for relating interaction scenarios, their semantics is still somewhat unclear since neither activity nor interaction diagrams have a formal semantics. In addition, IODs model only a single use case at a time and do not specify relationships between use cases. In essence, use case charts add a third layer of relationships to IODs at the use case level (level 1). They also introduce relationships that do not currently exist in UML, as noted in Section 2. Some of these relationships have been suggested by other authors – e.g., Krüger [1] introduces preemption. There is no existing synthesis algorithm for IODs or these additional relationships.

The use of scenarios in requirements engineering, has ranged from informal sketching to precise

IEEE COMPUTER SOCIETY

specification [11]. The Inquiry Cycle [12] combines goal-oriented requirements analysis with scenario scripts and is extended in [13] by exception and dependency analysis. CREWS-SAVRE [14] included a technique for discovering scenarios using a library of domain-specific alternative paths. [15] uses a notation known as scenario networks to structure a scenario walkthrough process. Use case charts are different in that they remain very close to UML and so do not require a significant learning curve.

There have been many papers on synthesizing state machines from scenario-based notations such as UML sequence diagrams [e.g., 9], message sequence charts [e.g., 1, 16] and live sequence charts [2]. These synthesis algorithms have to deal with the fact that current notations do not allow easy specifications of use case and/or scenario relationships. One approach has been to try to infer these relationships automatically using machine learning techniques ([3]). Another has been to require the user to provide additional information, in a non-standard way, that can be used to infer the relationships (e.g., [9,1]). Use case charts externalize these relationships. We believe that there is no synthesis algorithm in the literature as sophisticated as our algorithm for use case charts. Most existing algorithms cannot generate hierarchy in the state machines and cannot handle the rich set of relationships that use case charts provide.

6. Conclusion and Further Work

This paper presented a new algorithm for converting use-case based requirements into hierarchical finite state machines (HFSMs). This algorithm has been implemented in the SCASP tool and supports automatic simulation and validation of use case scenarios via conversion to executable HFSMs. The algorithm is novel in that it works for a very expressive use case language, use case charts, suitable in particular for distributed, concurrent systems. It is also novel in that the FSMs generated contain rich hierarchy which makes the generated FSMs easier for human consumption during later stages of analysis and design. The algorithm was validated on significant examples, including the use of solutions to a large term project. In the future, empirical studies will be undertaken to further evaluate use case charts and the use of the generated HFSMs in requirements validation.

7. References

[1] I. Krüger, “Distributed System Design with Message Sequence Charts”, *PhD Thesis*, Technical University of Munich, 2000.

[2] D. Harel, H. Kugler and A. Pnueli, "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements", *Formal Methods in Software and System Modeling* (H.-J. Kreowski et al, eds.), LNCS, Vol. 3393, Springer-Verlag, 2005, 309-324.

[3] E. Mäkinen and T. Systä, “Minimally adequate teacher synthesizes statechart diagrams,” *Acta Informatica 38*, 2002.

[4] Unified modeling language 2.0 specification, 2005. http://www.omg.org.

[5] J. Whittle, “Specifying Precise Use Cases with Use Case Charts,” *Satellite Events at the MODELS 2005 Conference*, LNCS, Vol. 3844, Springer-Verlag, 2005, 290-301.

[6] J. Whittle, “A Formal Semantics of Use Case Charts,” Technical Report ISE Dept, George Mason University, ISE-TR-06-02. http://www.ise.gmu.edu/techrep

[7] Message Sequence Chart (MSC). Technical Report, 1996. ITU-T Recommendation Z.120 (previously CCITT Recommendation), Formal Description Techniques.

[8] University of Paderborn Software Engineering Group. Shuttle system case study. http://wwwcs.uni-paderborn.de/cs/ag-schaefer/CaseStudies/ShuttleSystem/.

[9] J. Whittle and J. Schumann. “Generating statechart designs from scenarios.” *In ICSE ’00: Proceedings of the 22nd international conference on Software engineering*, pages 314–323, New York, NY, USA, 2000. ACM Press.

[10] D. Denery, H. Erzberger, T. Davis, S.Green & B. McNally, Challenges of Air Traffic Management Research: Analysis, Simulation and Field Test. In *AIAA Guidance, Navigation and Control Conference, 1997*.

[11] I. Alexander and N. Maiden (eds.) “Scenarios, Stories, Use Cases: Through the System Development Lifecycle,” John Wiley and Sons, 2004.

[12] C. Potts, K. Takahashi and A. Anton, “Inquiry-based requirements analysis,” *IEEE Software*, 21-32, 1994.

[13] A. Sutcliffe, N. Maiden, S. Minocha and D. Manuel. “Supporting scenario-based requirements engineering,” *IEEE Transactions on Software Engineering*, 24(12), 1998.

[14] N. Maiden, “Crews-savre: Scenarios for acquiring and validating requirements.” *Journal of Automated Software Engineering*, 1997.

[15] T. Alspaugh and A. Anton, “Scenario networks for software specification and scenario management.” *IEEE Transactions on Software Engineering*, 2001.

[16] S. Uchitel, J. Kramer and J. Magee. “Synthesis of Behavorial Models from Scenarios.” *IEEE Transactions on Software Engineering*. 29(2), February 2003.

0-7695-2555-5/06 $20.00 © 2006 IEEE

IEEE COMPUTER SOCIETY