SmartAgent: Extending the JADE Agent Behavior Model

Martin L Griss, Steven Fonseca[+], Dick Cowan, and Robert Kessler*

Software Technology Laboratory

HP Laboratories Palo Alto

*([+]UC Santa Cruz / HP Labs) (*University of Utah)*

January 2002

(Submitted to the AOSE Workshop, SCI, Orlando, Florida, July 2002)

ABSTRACT

A key component of effective multi-agent systems (MAS) software development is a set of models, technologies and tools that efficiently support flexible and precise specification and implementation of agent-to-agent conversations, standardized conversation protocols, and corresponding agent behaviors. This motivated a substantial extension to the JADE agent behavior model, which we call "HP SmartAgent." Previous experience with the ZEUS and JADE agent toolkits highlighted the need for more flexibility and precision in programming agent behavior and managing conversations. For example, the JADE infrastructure routes ACL messages in round-robin fashion to interested behavior "threads," making it hard to control ordering; reuse of behavior subclasses is difficult; and, support for default and exceptional behavior is inadequate. SmartAgent extends JADE behaviors with uniform message and system events, a multi-level tree of dispatchers that match and route events, and a hierarchical state machine that is based on the UML statechart model. Adherence to the UML helps bridge object-oriented to agent-oriented programming using an industry familiar modeling language and tools. Combining events, dispatcher tree and hierarchical state machines simplifies programming of default and context dependent behavior. This hypothesis was confirmed in a meeting scheduler prototype where code previously written using pure JADE was refactored using SmartAgent.

Keywords: agent, multi-agent system, conversation management, protocol, agent behavior, state machine, UML, JADE, ZEUS, state-oriented programming.

1 Introduction

Before effective agent-oriented solutions can be made widely available, systematic agent-oriented engineering and process methodologies must be matured. This lesson [Jennings, 2000] was internalized by our agent research group while prototyping a mobile shopper demonstration [Fonseca et al., 2001a; Griss and Letsinger, 2000] (using ZEUS [Nwana et al., 1999]) and a personal assistant with meeting scheduler [Griss et al., 2001] (using JADE [Bellifemine et al., 1999]). The software engineering and programming of multi-agent systems should use the same techniques used for any complex, distributed system: decomposition, abstraction, and organization [Booch et al., 1999; Jennings & Wooldridge, 1998].

The overall purpose of our agent-engineering research is to express and adapt these three techniques for the agent-oriented paradigm. Our goals are to develop a programming paradigm and implementation that facilitates agent construction through libraries of composable, reusable behavior elements, generated and assembled from input (UML) models. We and others are working in several related areas, including: UML-based modeling for agent systems (AUML [Odell 2000], [Griss 2000] and others), agent patterns [Kendall, 1999], Java-bean based agent component frameworks Gschwind [Gschwind et al., 1999], and aspect-oriented programming applied to agents [Kendall, 1999; Griss 2000a].

In this paper, we describe some of our recent work on a flexible conceptual and implementation framework that allows a number of conversation management (monitoring and control) policies and mechanisms to be uniformly described, negotiated, composed and implemented. MAS frameworks such as Zeus and JADE provide some support for constructing and coordinating sets of behaviors that support a particular conversation, but we found these mechanisms fairly rudimentary and idiosyncratic; in particular, it was hard to both control the precise order of behavior invocation and flexibly decompose behavior into coordinating parts that could be updated dynamically.

To realize flexibility, partitioning, and control when programming an agent, we chose to use an architecture supporting hierarchical state machine based programming of behavior, augmented with several flexibility enhancing mechanisms (such as events and dispatcher chains). We chose to base our conversation models and engines as accurately as possible on the UML standard hierarchical StateCharts [Booch et al., 1999]. Figure 1 shows as a UML interaction diagram ACL message exchange

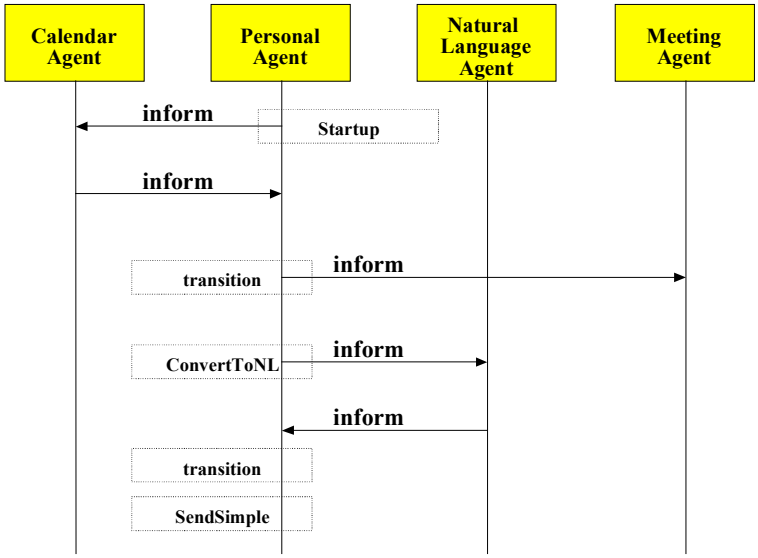
between three agents in our personal meeting arrangement assistant. These messages trigger state transitions shown in Figure 2, which models one of the hierarchical state machines that define a personal agent's behavior.

It is this decomposition of agent behavior and conversation management that SmartAgent supports. Section 2 provides a brief review of UML state machines and several agent platforms that support state-based programming of behavior and conversation management. Section 3 describes the SmartAgent architecture and design of our event-based, state-oriented extensions to JADE behaviors. Section 4 summarizes our experience using these extensions, and Section 5 concludes with a summery and suggests next steps.

2 State Machines and MAS Frameworks

State-based programming recognizes that any useful system must respond to the dynamic conditions of its environment. This fits well with the notion that software agents should autonomously react to their society. State-based programming for agent

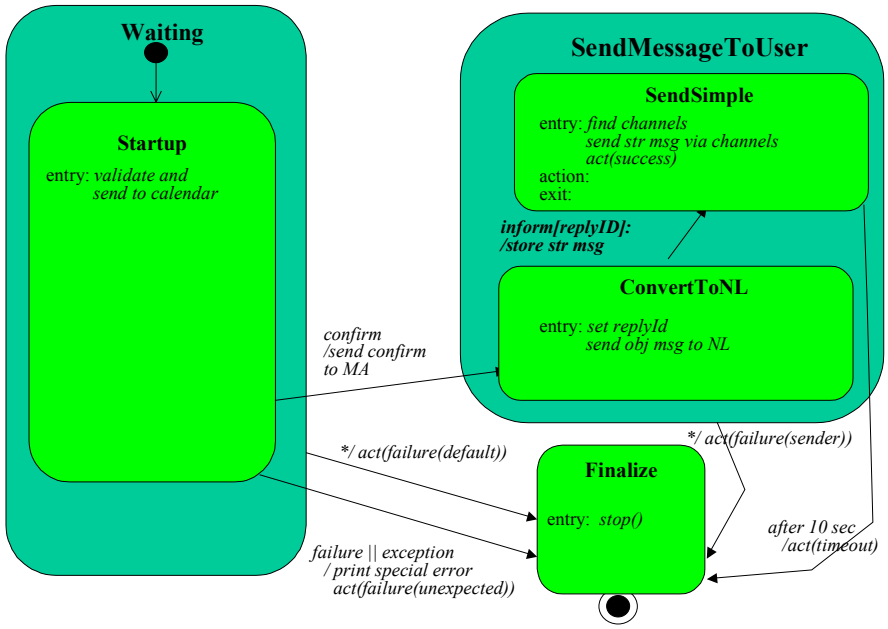
Figure 1 – Meeting arrangement conversation



behavior has been explored to model agent communication and behaviors [Odell 2000], as well more general reactive systems [Harel and Politi, 1998]. JADE offers a lightweight non-hierarchical state machine class with limited functionality while ZEUS provides a more complete non-hierarchical state machine execution subsystem, but the API is not well-developed. The JACKAL conversation engine also uses a state machine model [Cost 1998, 1999], and explored (colored) Petri nets [Cost et al., 2000].

We will discuss JADE in more detail than ZEUS, since it was our chosen platform for our state-based programming extensions. When messages arrive, a pool of currently interested behaviors is woken-up and the first ready behavior has an opportunity to process the message. A logical combination of string matching on the attributes of the incoming message is performed to determine if the action method of this behavior is executed. If the matching criteria are not satisfied, the current behavior is put back to sleep and matching with the next behavior is attempted. Once a behavior accepts a message, the pool of active behaviors are put back to sleep. Explicit reposting of a message is required when it should be handled by multiple behaviors. JADE recently provided a simple state machine, whose FSMBehaviour class maintains the relationship (transitions) between states and selects the next state behavior to execute. When a state is registered, a string name is associated with the corresponding behavior object. After naming all states, the transitions between them are defined and stored. Each transition is assigned an integer representing its event number. Once the start and finish states are registered, the

Figure 2: Sub-conversation StateChart



state machine is ready for execution. When a state behavior finishes, an *onEnd* method returns the event number that determines the one-and-only next transition to fire. If no transitions are associated with the current event number, the default transition is taken if one was specified. Note that transitions only serve to link states; they do not encapsulate agent behavior. Dealing with transition event numbers is also troublesome.

The UML state machine [Booch et al., 1999] evolved from the hierarchical statechart work of Harel [Harel and Politi, 1998]. A UML state machine defines the states that a machine is allowed to enter and the transitions that connect states together. States and machines can be built within other states and machines. This provides a hierarchy of processing elements that make it easier to respond to dynamic conditions of the environment and to handle default conditions. UML state machines are event driven; all actions in the environment come in the form of events that are presented to the state machine. Transitions define the conditions under which changes in state occur and are often event driven.

3 Design of SmartAgent

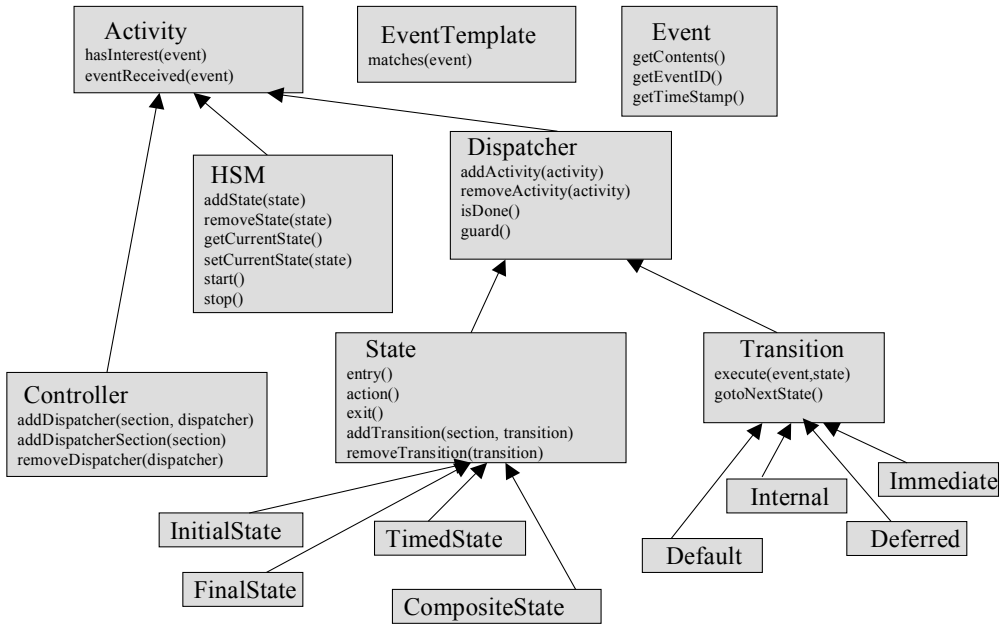
Our goal was to provide to the JADE programmer an easy to use set of classes and mechanisms (a conversation manager "kit") to enable the building of robust JADE behaviors. We wanted an explicit representation of the states and transitions, allowing future model-based generation, testing and monitoring tools. We wanted more control over the order of event dispatching, we wanted more powerful event matching patterns and rule-based dispatchers (using Jess™ and other pattern matchers). We wanted more flexible and extensible, dynamic addition of activities, transitions, states and complete behaviors, yet still provide precise handling of complex protocols. Finally, we wanted to easily express, combine and reuse core protocol elements and important default and exception handlers.

To this end, we implemented a fairly complete version of the UML hierarchical StateCharts, augmented with events and flexible dispatching extensions. UML StateCharts provide a standard graphical notation with numerous existing tools. The ability to run actions on both states and transitions provides flexibility in expression, and the hierarchy (nested states) provides a good mechanism for default handling.

3.1 SmartAgent Behavior Architecture

SmartAgent consists of three major subsystems: an event fusion and matching system that provides uniform event handling, an event dispatching system that provides event routing, grouping and ordering of agent Activities, and hierarchical state machine classes that partition agent behavior into states and transitions. Figure 3 shows a somewhat simplified class structure[1]. Figure 4 shows the event Controller/Dispatcher, and Figure 5 illustrates the state machine mechanism.

Figure 3 – Simplified SmartAgent architecture



Event and EventTemplate

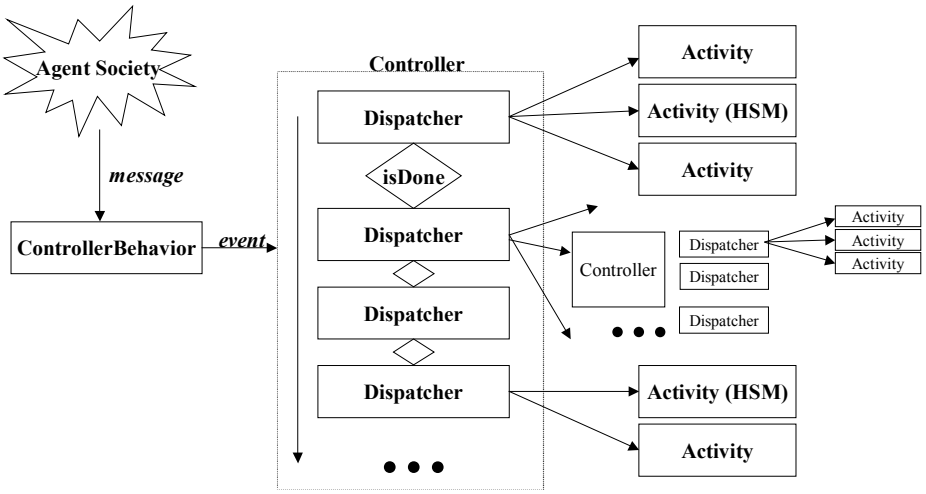
SmartAgent is explicitly event-driven. Every action that an agent is subjected to is translated into an event for uniform handling and processing. JADE behaviors convert received ACL messages and timers into events sent into the dispatcher structure (e.g., ControllerBehavior in Figure 4); other system actions and behaviors can also generate events. This includes both "external events" such as incoming ACL messages, system events such as ExceptionEvent and TimerEvent, and internal events signaled by other behaviors and activities, such as SuccessEvent and FailureEvent. ACL messages are wrapped in a MessageEvent object. Other event subclasses allow an agent and its internal systems to handle these **Events** uniformly. We extend JADE's message matching mechanism to allow more complex event matching. An event of a specified type is compared with an **EventTemplate** that defines a function called *matches* that checks to see if the matching criterion is satisfied. For example, a MessageEventTemplate object contains a JADE MessageTemplate that is compared with an incoming JADE ACL message wrapped in a MessageEvent object. EventTemplates allow matching on message arrival, timeouts or other events. Multiple matching checks can be combined using logical connectives to create a single boolean expression that can be tested for satisfaction. Loose coupling of event handlers is achieved by using the Observer pattern [Gamma et al., 1994] to allow dynamic addition and removal of multiple event listeners, each obtaining a copy of the event.

[1] We have removed several classes, interfaces and relationships to simplify explanation.

Dispatching and controller mechanism

The SmartAgent dispatching and controller mechanism routes events (including MessageEvents) to one or more execution objects (an Activity). As shown in Figure 4, ControllerBehavior continuously runs as a top-level agent behavior. It delivers all received messages as some Event to the controller. The controller contains a chain of dispatchers, grouped into sections. When a new dispatcher is added to a controller, it is appended to the end of an appropriate dispatcher section, implementing a simple priority scheme.

Figure 4: SmartAgent event-driven dispatching model



Each dispatcher manages a collection of activities. The granularity of an Activity can range from a very simple and short code sequence to another dispatcher or full multi-state state machine. As shown in Figure 3, controllers, dispatchers, and activities all implement this interface, and so dispatch trees can be built as shown in Figure 4, by attaching other controllers or dispatchers to a higher-level dispatcher.

Dispatchers first determine if the incoming event is for their Activities (via *hasInterest* and *guard)*. The *hasInterest* function may use an EventTemplate, while *guard* will check additional requirements (such as testing counters in the agent). If both tests succeed, the dispatcher distributes the event according to its distribution policy (via *eventReceived)*. After activity processing is complete, the dispatcher tests *isDone* to see if the event should be forwarded to the next dispatcher.

Figure 4 does not show the grouping of dispatchers into named sections. These allow the dynamic addition of a new dispatcher at a specified place in the chain or tree. Activities, dispatchers and sections can be added or removed at runtime. It is quite easy to establish (say) three sections, in which the first copies events to each of an extensible set of loggers and monitors, the second runs one or more activities and then the last section contains default and exception handling activities, that fire only if none of the dispatchers in the second section fire. More precise control will require more intermediate sections, or a Hierarchical State Machine.

Hierarchical State Machine

In the Hierarchical State Machine (HSM), actions that an agent must perform can also be further decomposed into states in which case incoming events trigger state transitions. We use three core class hierarchies. First, an HSM class implements the Activity interface. It is the top-level object that receives events from the dispatching subsystem. HSM maintains a current state pointer so when new events are received they can be forwarded to the current state. When transitions are taken, the current state pointer is updated.

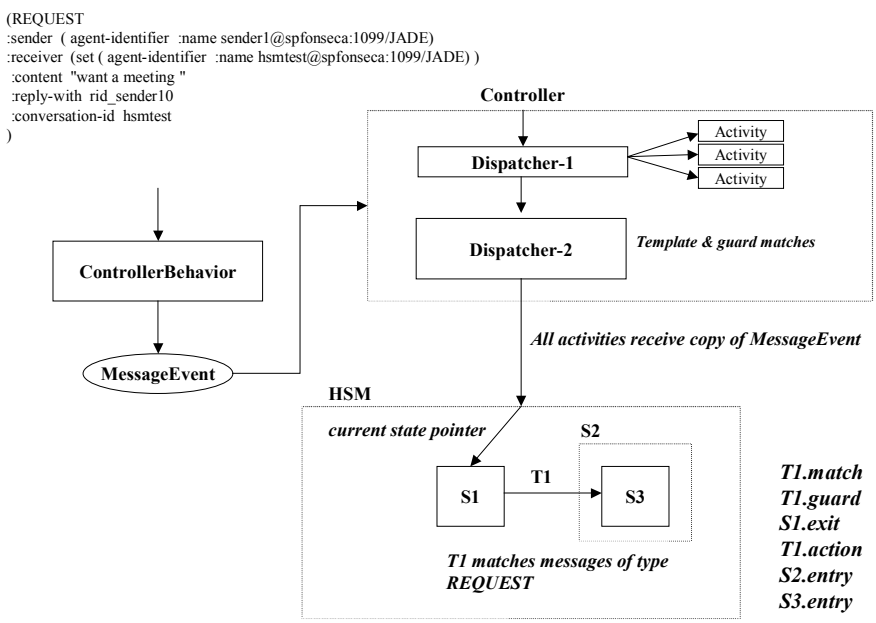
All states in an HSM descend from a class called State. State provides the basic functionality that all states share and defines default methods that subclasses might override. For example, the *eventReceived* method is often rewritten and is dependent on the type of the state object. Figure 3 shows a simplified state class hierarchy.

The normal State class and its subclasses CompositeState and TimedState, implement *entry*, *action*, and *exit* methods. InitialState and FinalState are simplified kinds of State. They restrict the transitions allowed, and do not implement the *entry*, *action*, or *exit* functions. *Entry* is called each time a new states is entered. The *action* method is invoked when an event is received but does not result in the firing of any transition. By default, *action* runs a set of attached Activities. *Exit* is called before a transition leaves the state. A CompositeState provides the additional capability of encapsulating a collection of nested states. A CompositeState can include its own initial and final states. TimedState extends State by providing a built-in timeout mechanism that on *entry* starts a timer that generates a timer event after a specified duration has elapsed.

States are connected by transitions, a subclass of Dispatcher. When an event is received by the current state, a check is performed to see if it matches any of the transitions leading to a next state. For a transition to fire, its guard and event matching criteria must be satisfied. Like state objects, transitions also have an overridable *action* function (by default running attached activities).

Internal and deferred transitions do not leave the current state. When an internal transition fires, the HSM continues to remain in the same state, and *entry* and *exit* methods are not run. When a deferred transition fires, the event is placed in the deferred

Figure 5: SmartAgent Dispatching Model



event queue for processing in a later state[2].

Hierarchy in UML state machines provides two inheritance benefits, factoring common actions and tests. First, if an event does not fire a transition of the current state, then the transitions for parent states are recursively checked. This inheritance makes it easy to program default agent behavior by having composite states provide default and exceptional transitions to states that then need explicitly handle only the main cases. Further, nested states can redefine transitions matching the same type of event as its parent to override what is already provided by default. The second is that a chain of entry and exit functions will be run when transitions cross composite state boundaries.

For example, Figure 5 shows an incoming ACL message received by the ControllerBehavior, which is sent as a MessageEvent to the Controller. This passes the event through the tree of Dispatchers. Dispatchers that match *haveInterest* and *guard* run their Activities. In this example, the Template associated with the second Dispatcher matches, and the event is received by a single state machine activity named HSM. Its currentState pointer relays the event to the correct state, in this case S1. S1 searches for a transition that matches the event just received. If transition T1 exists between S2 and S3, where S1 is a top level state, and S3 is a child of S2, the expected sequence of method invocations is as specified by UML: S1.exit, T1.action,

[2] Deferred transitions were only partially implemented.

S2.entry, and S3.entry. This ensures that exit code is always run; this is used for example in TimedState to shut off the timer. First, S1 is exited. Though not shown, the current state pointer of HSM is updated to S3.

We have introduced two new types of transitions: immediate and default. After a state's entry code is executed, all immediate transitions are evaluated without waiting for an event. When an event arrives, if no transitions fire, we recursively search the parent chain. Often we created a parent state enclosing the child state just to hold a single transition to handle any exceptional events. This idiom was so common that we introduced a default transition as a "macro" for this common case.

To summarize, after a transition's *hasInterest* and *guard* return true, the following occurs in this order:

1. The current state's exit method is run.
2. The transition's action is run.
3. The target state's entry code is evaluated.
4. The guards on any immediate transitions are checked and fired if appropriate.
5. If there are deferred events, they are processed like any other external event, firing transitions if appropriate.
6. The target state then waits for a new event to arrive. When that happens, the regular, deferred and internal transition triggers and guards are checked and then fired as appropriate.
7. If no transitions fire in Step 6, any default transitions are processed.
8. The parent hierarchy is scanned to see if any parent wishes to process the event.
9. If no one will process the event, it is discarded, the state's action is evaluated, and processing continues back at Step 5.

4 Experience using SmartAgent

As we developed and evaluated the integrated model, we used two experimental vehicles. One was an extensive set of sample state machines. The other was a re-implementation of our CoolAgent Personal Assistant and Meeting Assistant system [Griss et al., 2001] (similar in spirit to [Maes, 1994]). We used table-driven event and meeting request senders for debugging, exhaustive testing of corner cases, exploring new features, improving the robustness of the meeting system and assessing the benefits of explicit state-based programming.

The meeting system was originally written using a state-like model (with case statements over integer states), so adapting it for the HSM was relatively straightforward. The original conversation models were informally sketched as state machines (such as in Figure 2), so formalizing states, transitions, templates, guards, and actions was fairly direct. Mapping from the UML diagrams into code was immediate. We translated code that looked like that in Figure 6A into code in Figure 6B. What you should notice are many switch statements (lines 4 and 23), both for state processing and for message handling. State changes occur by changing the state variable to the value of an integer constant. Line 21 shows a call to receive a message. By forcing message retrieval through one function, it allows you to add a message logging capability. However, if you wish to be selective and only log certain messages, the code would require additional arguments or global state to decide which messages to log. Under the dispatcher model, one simply adds a new message dispatcher that decides which messages to log and inserts that at the top of the dispatch loop. Line 30 shows the addition of another behavior that is responsible for sending the message to the user (essentially the right side of the state chart in Figure 2). Outer code not shown coordinates the blocking of this behavior until the child completes. Figure 6B shows a piece of the state machine code for Figure 2. Lines 2 through 11 create the states. Some states, such as finalize on lines 8 through 11, are simple enough that you merely instantiate a state object and define the various methods inline. Others, such as startup, sendSimple, and convertToNL on lines 4 through 7 are more involved and so we chose to create a subclass that defines the various entry, action, or exit code. Lines 28 through 41 show how easy it is to create transitions. Lines 28-39 shows the definition of a transition between convertToNL and sendSimple that has both guard and action code. Lines 40-41 show a simpler transition that goes from startup to finalize when any FAILURE message is received. As you might imagine, modifying the state machine to handle other situations is as simple as adding a new transition.

```
01 public void actionBody() {
..
04  switch (state) {
05    case STARTING: {
..
20    case WAITING_FOR_UPDATE_CONFIRMATION: {
21      ACLMessage msg = agent.receiveACLMessage(MessageTemplate.MatchInReplyTo( reply_id ));
22      if ( msg != null ) {
23         switch ( msg.getPerformative() ) {
24           case ACLMessage.CONFIRM: {
..
29             // Start behavior for informing the participant of the meeting details
30             addChildBehavior(new PA_SendMsgToUser_Behavior(agent, this, announcement ));
31             state = FINISHED;
32             break;
33           } //~ case ACLMessage.CONFIRM
34           case ACLMessage.FAILURE: {
35             state = FINISHED;
36             break;
37           } //~ case ACLMessage.FAILURE
..
48         } //~ switch (msg.getPerformative())
49      } // END if ( msg != null )
50      break;
51    } // END case WAITING_FOR_UPDATE_CONFIRMATION:
52  } // END  switch (state)
53 } // END actionBody()
```

Figure 6A – Partial code for Figure 2 before change into a state machine.

```
01 // *** Create the States.
02 CompositeState def = new CompositeState("Default", this);
03 CompositeState sendMessageToUser = new CompositeState("sendMessageToUser", this);
04 State startup = new StartupState("Startup", def);
05 State sendSimple = new SendSimpleState("SendSimple", sendMessageToUser,
06                    agent.getTimeout("waitForSendMessage", 5000), this, timerEventKey);
07 State convertToNL = new ConvertToNLState("ConvertToNL", sendMessageToUser);
08 State finalize = new State("Final", this) {
09    public void entry() {
10        stop();
11    }
12 };
13 // *** Create the transitions.
..
28 Transition gotNL = new Transition("gotNL",
29         new MessageEventTemplate(MessageTemplate.MatchPerformative(ACLMessage.INFORM)),
30     convertToNL, sendSimple) {
31          public boolean guard(Event e) {
32              return (e instanceof MessageEvent) &&
33                   (((ACLMessage)e.getContents()).getInReplyTo().equals(replyId));
34          }
35          public void action(Event e, State s) {
36              convertedMessage = ((ACLMessage)e.getContents()).getContent();
37          }
38     };
39 convertToNL.addTransition(gotNL);
40 Transition.add(startup, finalize, "failure",
41       new MessageEventTemplate(MessageTemplate.MatchPerformative(ACLMessage.FAILURE)));
```

Figure 6B – Partial state machine code for state diagram in Figure 2.

One interesting observation is what we have NOT used in the system. So far we have not needed any state *action*; instead, we put all action code either inside the state *entry* or in the transition *action*. In part, this is because one cannot ensure that the state *action* code will ever be executed. It is only executed when an event arrives that the state transitions cannot handle. If only expected valid events arrive, then the action code will never be executed. In the "event switch" model, the action code is used to indicate those exceptional conditions when no event template can be found anywhere in the hierarchy. In the UML state chart design, action is the "idle loop" action or "default action." We could easily be enabled this in our model by sending in (several frequencies of) "clock tick" event. Another rarely used feature is the state *exit* capability. So far, our only use was to turn off the timer in a TimedState. We suspect that as more complex conversations are developed, we will certainly make use of these features. We must note that nested states are used extensively to factor our exceptional and default case transitions, and caused us to introduce the DefaultTransition for the most common case.

The way we handle multiple conversations is simple: a message begins a new "sub-conversation" allocates a new unique conversation ID (CID) and then instantiates an appropriate HSM, which is attached to a new dispatchers that matches only messages with that CID. When a message with that CID arrives it is directly routed to the specific HSM. Once the HSM is finished, we remove the dispatcher. Logging, monitoring and default handlers will still apply as appropriate to all messages.

We have built about a dozen state machines to handle various parts of the personal assistant system. We have used dispatchers and HSM's to implement a loosely coupled internal agent architecture, in which a rich set of internal events is exchanged between HSM's [Letsinger 2001]. Anecdotally, it is clear to us that using this model is far superior to using the model that is provided by the base JADE system. We can construct more robust, more complicated state-based interactions in less time and with less code.

5 Summary, Conclusions and Future Work

We have developed an enhanced JADE agent behavior model allowing more precise control over robust agent behavioral issues such as timeouts, default and exception handling, and order of behavior execution. In addition, we have increased the flexibility and adaptability of our system. Our model integrates events, activities, event dispatching and hierarchical state machines to overcome the difficulties we encountered in our previous experiments with ZEUS and JADE [Fonseca at al, 2001; Fonseca et al, 2001a; Griss et al, 2001]. Key to this integration was:

- Event fusion, which converts all messages, exceptions, timeouts and other systems events into a common event hierarchy that can be matched and dispatched uniformly by rules, dispatchers and state machines.
- Event invoked activities, with a simple Activity interface that can be implemented by most other elements (simple expressions, rule modules, JADE behaviors and state machines).

It is very easy to add default handlers. Timed states handle those situations where a target agent fails to respond, and state/transition hierarchy cleanly separates the "main" flow of the conversation and its exception handling.

While we tried to follow the UML state chart semantics very closely, we made some modifications, interpretations and extensions to better suit a natural, precise yet compatible extension to JADE. Close compatibility with UML yields a rich, yet understandable system, and enables the use of tools such as Visio 2000, ArgoUML and Rational Rose to design state charts and interaction diagrams. As mentioned above, the significant changes and extensions include:

- Dynamic addition of dispatchers, states, transitions and activities
- Immediate and default transitions
- Dynamic inheritance of transitions in the state hierarchy
- Embedded HSM, remote invocation of HSM

To better support the factoring of more complex state machines, we expect to complete the implementation of deferred transitions, and of concurrent states, with fork and join states.

Acknowledgements

Special thanks to David Bell for earlier work that helped motivate and define the scope of this work, and to Reed Letsinger for critique and feedback as we developed these extensions.

References

[Bellifemine et al., 1999] F. Bellifemine, A. Poggi & G. Rimassi, "JADE: A FIPA-Compliant agent framework", Proc. Practical Applications of Intelligent Agents and Multi-Agents, April 1999, pg 97-108 (*See http://sharon.cselt.it/projects/jade for latest information*)

[Booch et al., 1999] G. Booch, J. Rumbaug & I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999

[Cost et al., 1998] RS. Cost, T. Finin, Y. Labrou, X. Luan, Y. Peng, I. Soboroff, J. Mayfield & A. Boughannam, "Jackal: A Java-Based Tool for Agent Development." *Working Notes of the Workshop on Tools for Developing Agents (AAAI '98) (AAAI Technical Report)*.

[Cost et al., 1999] RS. Cost, T. Finin, Y. Labrou, X. Luan, Y. Peng, I. Soboroff, J. Mayfield & A. Boughannam, An Agent-based Infrastructure for Enterprise Integration, ASA/MA, October, 1999, p. 219-33

[Cost et al., 2000] RS.Cost, Y. Chen, T. Finin, Y. Labrou, & Y. Peng, Using Colored Petri nets for Conversation Modeling, Issues in Agent Communication, AI Vol. 1916, May 2000, p. 178-92.

[Finin 1997] T. Finin, Y. Labrou and J. Mayfield, “KQML as an Agent Communication Language,” in Software Agents, J.M.Bradshaw (ed), MIT Press, Cambridge, Mass., p291-316, 1997.

[Fonseca et al, 2001] SP Fonseca, ML. Griss, & R. Letsinger, Evaluation of the ZEUS MAS Framework, Hewlett-Packard Laboratories, Technical Report, HPL-2001-154

[Fonseca et al, 2001a] Steven P. Fonseca, Martin L. Griss, Reed Letsinger, An Agent Mediated E-Commerce Environment for the Mobile Shopper, Hewlett-Packard Laboratories, Technical Report, HPL-2001-157

[Gamma et al., 1995] E. Gamma, R Helm, R Johnson & J Vlissides, “Design patterns,” Addison-Wesley, 1995.

[Griss 2000] ML. Griss, "My Agent Will Call Your Agent," *Software Development Magazine*, February. (See also "My Agent Will Call Your Agent … But Will It Respond?" Hewlett-Packard Laboratories, *Technical Report, TR-HPL- 1999-159*).

[Griss 2000a] ML. Griss, Implementing Product-Line Features By Composing Component Aspects, Proc. First International Software Product-Line Conference, Denver, CO., Aug 2000

[Griss & Letsinger, 2000] ML Griss & R Letsinger, Games at Work - Agent-Mediated E-Commerce Simulation**,** Workshop, Autonomous Agents 2000, Barcelona, Spain, June 2000. *(Also HP Laboratories Technical Report, HPL-2000-52.)*

[Griss et al., 2001] ML. Griss, et al., CoolAgent: Intelligent Digital Assistants for Mobile Professionals – Phase 1 Retrospective, HPL Technical Report, Nov 2001 *(In preparation)*

[Gschwind et al., 1999] T Gschwind, M Ferudin, & S Pleisch, ADK - Building Mobile Agents for Network and Systems Management from Reusable Components, ASA/MA. Los Alamitos, IEEE.

[Harel and Politi, 1998] D Harel and M Politi, Modeling Reactive Systems with Statecharts, McGraw Hill, 1998

[Jennings and Wooldridge, 1998] NR. Jennings, and MR. Wooldridge, *Agent Technology,* Springer.

[Jennings, 2000] NR. Jennings, On Agent-Based Software Engineering, Artificial Intelligence, March, 2000, vol. 117, no. 2, p. 277-96

[Kendall, 1999] EA. Kendall, Role Model Designs and Implementations with Aspect-oriented Programming, Proc. Of OOPSLA 99, Oct, Denver, ACM SIGPLAN, p. 353-369

[Letsinger, 2001] R Letsinger,. Three Architectural Principles for the Design of Personalisable Agents, *HP Labs Tech Report, HPL-2001-300, December 2001.*

[Maes, 1994] P Maes. "Agents that Reduce Work and Information Overload." Communications of the ACM, Vol. 37, No.7,pp. 31-40, 146, ACM Press, July 1994.

[Nwana et al., 1999] H. Nwana, D. Nduma, L. Lee, J. Collis, "ZEUS: a toolkit for building distributed multi-agent systems", in *Artificial Intelligence Journal*, Vol. 13, No. 1, 1999, pp. 129-186. (See http://www.labs.bt.com/projects/agents/zeus/).

[Odell, 2000] J Odell, H. VD Parunak & B Bauer, Extending UML for Agents, AOIS Workshop at AAAI 2000, www.auml.org. Also see http://www.jamesodell.com/publications.html

[Samek and Montgomery, 2000] M Samek & P Montgomery, State-Oriented Programming, Embedded Systems Engineering, August, 2000, p. 21-43.

[Shoham, 1993] Y Shoham, “Agent Oriented Programming,” Journal of Artificial Intelligence, 60(1), pp. 51-92, 1993.