Using UML State Machine Models for More Precise and Flexible JADE Agent Behaviors

Martin L Griss[1], Steven Fonseca[2], Dick Cowan[1], and Robert Kessler[3]

```
¹Software Technology Laboratory
    HP Laboratories MS 1137
1501 Page Mill Road, Palo Alto 94304-1126
   {martin_griss, dick_cowan}@hp.com
            650-857-8715
          ²UC Santa Cruz
       {fonseca@cse.ucsc.edu}
        ³University of Utah
       {Kessler@cs.utah.edu}
```

Abstract. In order to effectively develop multi-agent systems (MAS) software, a set of models, technologies and tools are needed to support flexible and precise specification and implementation of agent-to-agent conversations, standardized conversation protocols, and corresponding agent behaviors. Experience trying to build complex protocols with the ZEUS and JADE agent toolkits motivated a substantial extension to the JADE agent behavior model. This extension (called SmartAgent) enables more flexible, reusable and precise modeling and implementation of agent behavior. We augment JADE behaviors with uniform message, timing and system events, a multi-level tree of dispatchers that match and route events, and a hierarchical state machine (HSM.) HSM is represented using UML statechart diagrams, and implements a significant subset of UML state machine semantics. Adherence to the UML standard helps bridge object-oriented to agent-oriented programming, and allows us to use industry familiar modeling language and tools such as Rose or Visio. These extensions were tested in a meeting scheduler prototype.

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

1 Introduction

Systematic agent-oriented engineering techniques must integrate traditional software engineering techniques with multi-agent technology [Jennings, 2000; Jennings and Wooldridge, 1998] in order to develop robust, industrial strength multi-agent systems. Previous papers [Fonseca et al., 2001a; Griss and Letsinger, 2000; Griss et al., 2001] describe our experiences building several multi-agent systems using ZEUS [Nwana et al., 1999] and JADE [Bellifemine et al., 1999].

Recently we have extended the JADE behavior and communication protocol management subsystem [Griss et al., 2002] to allow richer handling of conversations and

behaviors. 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], Javabean based agent component frameworks Gschwind [Gschwind et al., 1999], and aspect-oriented programming applied to agents [Kendall, 1999; Griss 2000a].

MAS frameworks such as ZEUS and JADE provide some support for constructing and coordinating sets of behaviors for a particular conversation. However, we found these mechanisms difficult to use when conversations required complex protocols. It was hard to both control the precise order of behavior invocation and flexibly decompose behaviors into coordinating parts that could be updated dynamically.

We chose to use an architecture and implementation supporting hierarchical state machine based (HSM) programming of behavior, augmented with several flexibility enhancing mechanisms (such as events and dispatcher chains). We chose to base our models, tools and implementation as accurately as possible on the UML standard hierarchical state machines [Booch et al., 1999]. In [Griss et al., 2001], we showed a UML interaction diagram of ACL message exchange between three agents in our personal meeting arrangement assistant, and a partial state machine to handle one side of this conversation. In this paper, we show enhancements to the model and automatically generated code for a state machine handling one side of a simplified bidding conversation. Figure 1 is a Visio 2002 statechart diagram of this 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 development. Section 3 describes the SmartAgent architecture and design of our event-based, state-oriented extensions to JADE. Section 4 summarizes our experience using these extensions. Section 5 discusses tools and reusability. Section 6 concludes with a summary and 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 suits the notion that software agents should autonomously react to their society. State-based programming for agent 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].
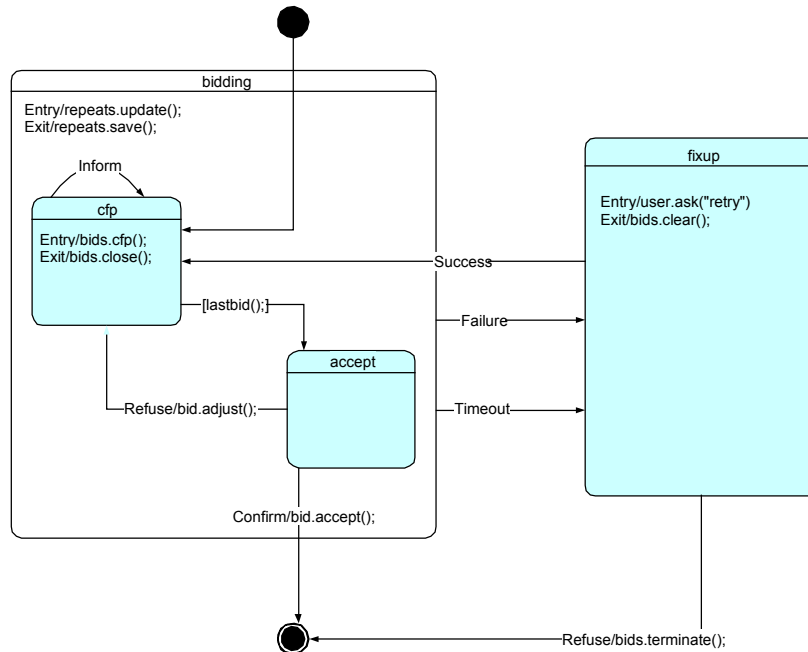
Figure 1 – An example HSM behavior specification as a statechart diagram

Since JADE was our chosen platform for our experiments, we will discuss it in detail. When messages arrive, a pool of currently interested behaviors is woken-up and the first ready behavior has an opportunity to process the message. Pattern matching on attributes of the incoming message determines if the action method of this behavior is executed. If the match fails, the current behavior is put back to sleep and matching 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 if the message is to be handled by multiple behaviors. JADE provides a simple state machine, FSMBehaviour that maintains the transitions between states and selects the next state behavior to execute. States are registered, named and stored, and each transition between them is assigned an integer representing its event number. Once the start and finish states are registered, the 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 specified. Note that transitions only serve to link states; they do not encapsulate agent behavior. Dealing with transition event numbers is 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 states and transitions that connect states. States and machines can be built within other states and machines. This hierarchy of processing elements makes it easier to describe com-

plex conditions and transitions and to handle common default conditions and exceptions. UML state machines are event driven; environment actions come in the form of events that are presented to the state machine, typically driving transitions. Other conditions can also fire transitions and state changes.

3 Design of SmartAgent

Our goal is 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 model-based generation, analysis, testing and monitoring tools. We wanted more control over the order of event dispatching, and more powerful event and condition matching. 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 state machines, augmented with events and flexible dispatching extensions. UML state chart diagrams 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 nested state hierarchy provides a mechanism to factor and inherit common default and exceptional transitions and common behavior fragments.

3.1 SmartAgent Behavior Architecture

SmartAgent consists of three major subsystems: an event fusion and matching system for uniform event handling, an event dispatching system for event routing, grouping and ordering of agent Activities, and hierarchical state machine classes that partition agent behavior into states and transitions. Figure 2 shows a somewhat simplified class structure[1] and Figure 3 shows the event dispatchers and the state machine mechanism.

Event and EventTemplate. Every action that an agent must react to is translated into an event for uniform handling and processing. JADE behaviors and system actions convert received ACL messages, timers and exceptions into events sent into the dispatcher structure (e.g., ControllerBehavior in Figure 3.) These include message events, system events such as ExceptionEvent and TimerEvent, and internal events signaled by other behaviors and activities, such as SuccessEvent and FailureEvent.. We extended 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 Mes-

---

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

sageEvent object. EventTemplates allow matching on message arrival, timeouts or other events. Multiple matching checks can be combined using logical connectives. 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. More generally, an Observer-pattern based "software event bus" is used to combine multiple dispatchers and HSM's within an agent.
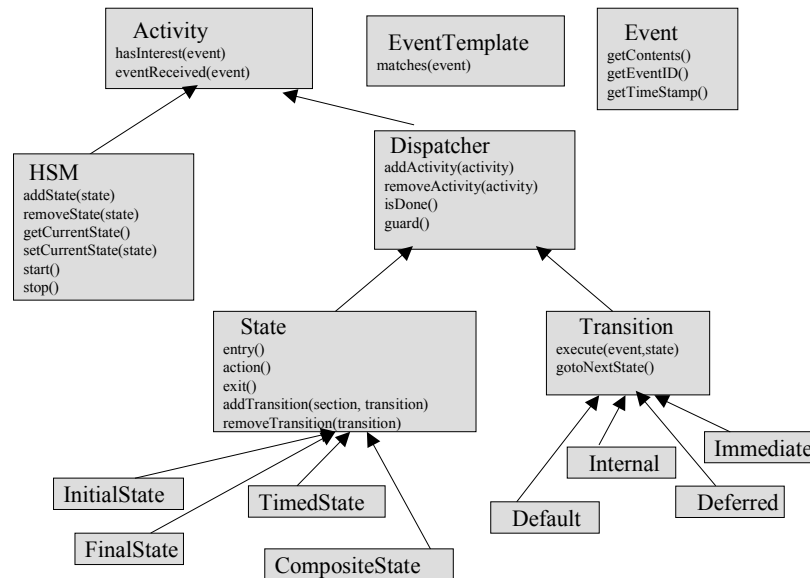

Figure 2 – The simplified HSM architecture[2]

Dispatching and controller mechanism. In [Griss et al., 2001] we discuss the SmartAgent dispatching mechanism in more detail. A top-level JADE behavior, ControllerBehavior, delivers all received messages as an Event through the dispatch tree, ultimately to some set of Activity. 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 2, HSMs, states, and dispatchers all implement Activity, and so (branching) dispatch trees can be built as shown in Figure 3. Dispatchers first determine if the incoming event is for their Activities (via *hasInterest* and *guard)*. The *hasInterest* function may use an EventTemplate matcher, 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)*. The dispatch tree allows some control over which Activities will fire, but for more precise control, we use the Hierarchical State Machine.

---

[2] Figure adapted and simplified from that in [Griss et al., 2001].
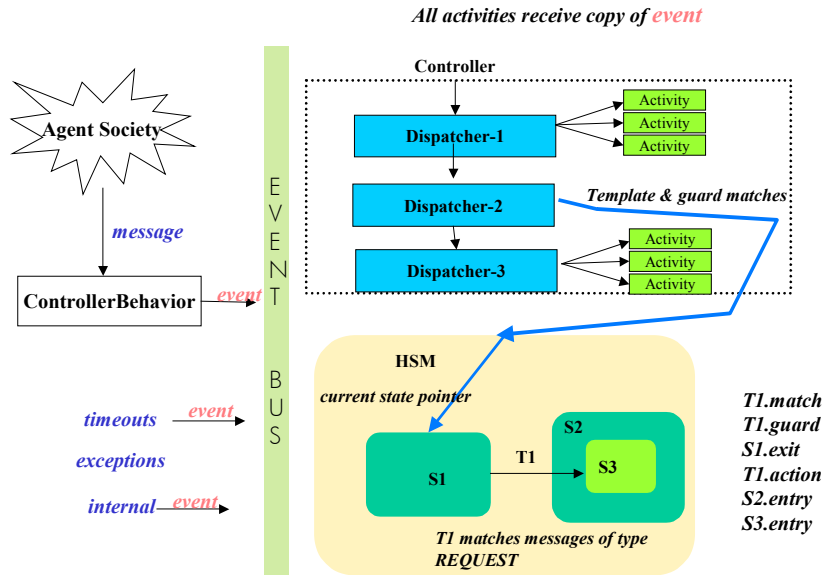
All activities receive copy of event

Figure 3 – Event flow from behavior through dispatch tree to an HSM

Hierarchical State Machine. In the HSM actions that an agent must perform are further decomposed into states. Incoming events then 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 (See Figure 3.)

All states in an HSM descend from a class called State (see Figure 2). 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.

The normal State class implements *entry*, *action*, and *exit* methods. *Entry* is called each time a new state 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 any transition leaves the state. CompositeState extends state, but does not implement *action*. InitialState and FinalState are further simplified. They restrict the kinds of transitions allowed, and do not implement the *entry*, *action*, or *exit* functions. 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 so that *entry* creates and starts a timer that generates a named timer event after a specified duration has elapsed, and stops and deletes the timer on *exit*. Transitions can trigger on a matching TimerEvent.

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 transi-

tions leading to a next state. For a transition to fire, both its event matching and guard must be satisfied. Like state objects, transitions also have an overridable *action* function, which by default runs a list of 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 (but the transition Action may run). When a deferred transition fires, the event is placed in the deferred event queue for processing in a later state[3].

Hierarchy in UML state machines provides two inheritance benefits, factoring common entry, exit, actions and tests[4]. First, if an event does not fire a transition of the current state, then the transitions for parent states are recursively checked. This makes it easy to program default agent behavior by having composite states provide default and exceptional transitions to groups of states, allowing inner states to focus only on the main cases. Further, nested states can redefine transitions provided by the parent to override shared defaults. The second benefit is that a chain of entry and exit functions will be run when transitions cross composite state boundaries, and code common to several nested states or all incoming or outgoing transitions can be factored to one place.

For example, Figure 3 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 *hasInterest* and *guard* run their Activities. In this example, the EventTemplate 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*, *S2.entry*, and *S3.entry*. This ensures that *entry* and *exit* code is always run; for example in TimedState this is used to start and stop the timer. First, S1 is exited. Though not shown, the current state pointer of HSM is finally updated to S3. As another example, in Figure 1, *bidding.entry* ensures that that *repeats* is *updated* or *saved* for any of the five transitions taken; transitions *Failure* and *Timeout of* composite state *bidding* are available to both states *cfp* and *accept;* and*, bids.cfp* or *bids.close* could run on one of eight direct, self or inherited transitions.

We have introduced two new types of transitions: immediate and default. Immediate transitions are essentially "triggerless." 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. Default transitions were created when we observed that we often 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.

---

[3] Deferred transitions were only partially implemented.

[4] In principle, at most only a single transition should be viable, including inheritance; in our implementation, this can be optionally checked, but we use nesting to disambiguate.

To summarize, after a transition's *hasInterest* and *guard* return true, the following occurs in this order: 1) The current state's *exit* is run; 2) The transition's *action* is run; 3) The target state's *entry* is run; 4) Guards on any immediate transitions are tested and the transition fired if appropriate; 5) Any deferred events are now processed, firing transitions if appropriate; 6) The target state then waits for a new event to arrive; regular, deferred and internal transition triggers and guards are checked and then fired if appropriate; 7) If no transitions fire in the previous step, any default transitions are processed; 8) The parent hierarchy is scanned for any matching transition; and 9)If no transition fires, the event is discarded, the state's action is evaluated, and processing continues back at Step 5.

3   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 personal assistant and meeting assistant system. 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. [Griss et al., 2001].

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 1), so formalizing states, transitions, templates, guards, and actions was fairly direct. Mapping from the UML statechart diagrams into code was immediate. In [Griss et al., 2001] we show how meeting handler code written the old way was transformed.

The original JADE code has many switch statements both for state processing and for message handling. State changes occurred by changing the state variable to the value of an integer constant. In the new code states are defined directly by instantiating class State or a subclass, such as TimedState, or CompositeState. Likewise, Transitions are also explicitly instantiated classes, and are explicitly attached to source and target States at time of creation. The code is easier to read, and tools are able to directly generate and analyze the network structure of the state machine. Listing 1 shows the one-to-one correspondence between elements in Figure 1 and lines of Java code.

We have built about a dozen state machines to handle various parts of the personal assistant system, and a loosely coupled internal agent architecture, in which a rich set of internal events is exchanged between HSM's [Letsinger 2001]. 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 hand-written code.

4 Drawing and Generation Tools

The UML statechart diagram (Figure 1) was drawn using Visio 2002. When we first started using an early HSM, we hand drew the state machines on paper or a white board. We then hand coded from these diagrams. As the protocols became more complex, we wanted a tool to draw the HSM and to enhance as we developed shorthand notations and extensions. We experimented with several tools, including Rational Rose, ArgoUML and the UML stencils in Visio. After some irritation at the restrictions imposed by the other tools, we quickly developed our own stencils in Visio (we had done this before for other diagrammatic languages), with custom properties used for the attributes. This allows us to use color to indicate important states and transitions, and easy cut-and-paste of diagrams into slides and documents.

After building several state machines from Visio diagrams, we realized how valuable the diagram was. We ended up constantly referencing back to the diagram during implementation and debugging to help "understand" what was happening in the state machine. However, we realized that it was a tedious and error-prone process to make sure that the initial implementation exactly reflected the states and transitions described in a diagram. With that motivation, we then implemented a simple code generation facility, written as a Visual Basic addin in Visio. It currently generates Java and can handle some notational extensions, such as converting trigger expressions, such as *Inform* into calls on support routines *M.Inform()*. This creates an EventTemplate to match an *Inform* performative. (See Listing 1.) It will be easy to also generate XML and RDF, needed for other projects in our group. In the future, we plan to investigate round-trip engineering and the possibility of using the visualization of the diagrams to help in debugging the state machine execution.

5 Reuse and Extensibility

The SmartAgent implemention of HSM and Dispatcher has taken great care to use the standard extensibility mechanisms of Java, namely Interfaces, Abstract classes, Adapter classes, and class inheritance. For example, the HSM uses ordinary public Java classes to represent states and transitions of various kinds. This means that individual states or transitions, and even groups of states and transitions, can be reused or extended, either in-line using the *new State() {...};* construction (such as lines 10-13 in Listing 1), or by defining a new class that extends or implements some other class or interface.

```
 1 public class Example extends HSM {
 2  public Example(SmartAgentAdapter anAgent,String name) {
 3   super(anAgent, name);
 4   Composite bidding = new Composite("bidding",this) {
 5      public void entry () {
 6        super.entry();
 7        repeats.update();
 8      } };
 9   State accept = new State("accept",bidding);
10   State cfp = new State("cfp",bidding) {
11      public void entry () { bids.cfp();  }
12      public void exit ()   { bids.close();  }
13   };
14   State fixup = new State("fixup",this) {
15      public void entry () { user.ask("retry"); }
16      public void exit ()   { bids.clear(); }
17   };
18   InitialState initial=new InitialState("initial",this);
19   FinalState finish=new FinalState("finish",this);
20   Transition Tran0 =new Transition("Tran0",null,initial,cfp);
21   Transition Tran1 =new Transition("Tran1",null,cfp,accept){
22      public boolean guard (Event e) { return lastbid(); }
23   };
24   Transition Tran2 =new Transition("Tran2",M.Failure(),bidding,fixup);
25   Transition Tran3 =new Transition("Tran3",M.Confirm(),accept,finish){
26      public void action (Event e, State s) {  bid.accept();  }
27   };
28   Transition Tran4 =new Transition("Tran4",M.Timeout(),bidding,fixup);
29   Transition Tran5 =new Transition("Tran5",M.Refuse(),fixup,finish){
30      public void action (Event e, State s) { bids.terminate(); }
31   };
32   Transition Tran6 =new Transition("Tran6",M.Success(),fixup,cfp);
33   Transition Tran7 =new Transition("Tran7",M.Inform(),cfp,cfp);
34   Transition Tran8 =new Transition("Tran8",M.Refuse(),accept,cfp){
35      public void action (Event e, State s) { bid.adjust();  }
36   } };
37 }
```
Listing 1 – A segment of the generated HSM code for state machine in Figure 1

We defined a new Subgraph class that extends CompositeState to support constructing reusable clusters of States and Transitions that can be instantiated in several places in a larger state machine[5]. We added a new Subgraph icon to the Visio stencil to show the use of a subgraph (shown in Figure 4); the generator code also needed small extensions. This subgraph can export selected internal states (using *public State getA()* etc.), shown as the A and B "ports" on the subgraph icon in Figure 4, so that transitions from outside the subgraph can be easily "wired in" to sources or target states in the subgraph, using code such as shown here:
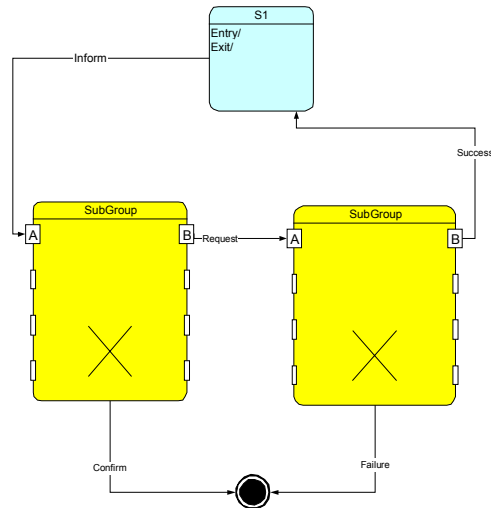
```
Subgraph sg1=new Subgraph("sg1", this);
Subgraph sg2=new Subgraph("sg2",this);
Transition t1=new Transition("t1",M.Request(), sg1.getB(),sg2.getA());
```
Listing 2 – Using and wiring subgraph

---

[5] This is different from referencing or embedding a complete HSM, since transitions can occur directly to and from States in the subgraph.

Figure 4 – Repeated use of a pre-wired subgraph

6 Summary, Conclusions and Future Work

Our enhanced state machine based JADE agent behavior model allows 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. Visio 2002 tools make it easy to quickly draw new behaviors, and generate compatible JADE code. 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. 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, HSM subgraph reuse

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

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*)

[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

[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 & 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 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, *Tech 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 et al, 2001] M. Griss, S. Fonseca, D. Cowan and R. Kessler, SmartAgent: Extending the JADE Agent Behavior Model, Proceedings of SCI SEMAS Workshop, Orlando, Florida, July 2002 (*to appear).*

[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.

[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.