

Extending the JADE Agent Behavior Model for More Flexibility and Control

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

Software Technology Laboratory

HP Laboratories Palo Alto

(⁺UC Santa Cruz/HPL) (*University of Utah)

November 2001

ABSTRACT

A key component of effective multi-agent systems (MAS) software development is a set of models, technologies and tools that make it more effective to flexibly and precisely specify and implement agent-to-agent conversations, standardized protocols, and corresponding agent behaviors. In this technical report, we provide a background and motivation for a substantial extension/revision to the JADE agent behavior mode, which we call "HP SmartAgent." These capabilities allow for more flexible and precise control of agent activities and conversations. Previous experience with systems built using both the ZEUS and the JADE agent toolkits have highlighted the need for this capability. In JADE, each behavior is a largely independent "pseudo-thread." Behaviors are scheduled in round robin fashion; all are woken up on the arrival of a message or timeout, and several behaviors that match the incoming message have an equal chance of execution. One is selected randomly and run until control is relinquished. The message is eventually consumed and the remaining active behaviors are put back to sleep. In our new model, JADE behaviors are decomposed into distinct activities. A new set of classes enables programming activities as state machines. Dispatchers and controllers handle message routing and activity execution. The controller maintains a chain of dispatchers. Incoming events are passed to the top dispatcher of this chain. Dispatchers perform matching on the incoming event. In SmartAgent, we extend and unify the notion of events to include messages, timers and system events allowing for activity activation based on richer event patterns. If the event is handled, all of the activities under control of the dispatcher are executed with the event serving as input. Upon completion, dispatchers are responsible for deciding if the event should be forwarded along the chain. The dispatcher chain can be ordered using a hierarchical state machine (HSM) based on the UML statechart model or using a simple sequential ordering. All of these SmartAgent features allow more precise control and sequencing of behaviors while retaining and enhancing the flexibility of the JADE behavior model. Furthermore, SmartAgent provides a flexible way for defining shared default and exception handling.

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

1 Introduction

The Internet can be thought of as a large, broadly distributed, heterogeneous multi-agent system (MAS) populated by anywhere from thousands to millions of agents. Some substantial subset of those agents will be deployed for the purpose of conducting electronic commerce, providing intelligent assistance and information filtering, and supporting other tasks. It is inevitable that a variety of communication, interaction capabilities and regimes will exist in such a world of heterogeneous agents because their interactions will include negotiation, brokering, contract formation, selling, scheduling, etc.

Multi-agent system framework development requires a shift in programming paradigm from object-oriented to agent-oriented. Agent-oriented programming involves using an agent as the main unit of encapsulation. This requires that, "we program an agent with the help of high-level elements such as goals, choices, skills, beliefs, and so on, and that the types of message that agent exchanges with each other also refer to high-level communication mechanisms in defining messages about data, requests, offers, promises, refusals, acceptances and so on" [Shoham 1993].

Before effective agent-oriented solutions are available, agent-oriented engineering and process methodologies must be matured. This lesson has been articulated by others [Jennings, 2000] and was internalized by our research group after building an experimental mobile shopper agent-based e-commerce framework [Fonseca et al., 2001a; Griss and Letsinger, 2001] (using ZEUS[Nwana 1999]) and an experimental agent-based personal assistant and meeting scheduler [Griss et al., 2001] (using JADE[Bellifemine et al., 1999]). Focusing on the engineering of multi-agent systems, the programming techniques used to handle distributed system complexity is essentially the same as dealing with any type of sophisticated software; decomposition, abstraction, and organization are key [Booch, 1999; Jennings & Wooldridge, 1998]. In a very general sense, the purpose of our research is to take these three mechanisms and apply them to the agent-oriented paradigm. This attempts to answer a “fundamental question” of agent-oriented software engineering [Jennings 2000; Jennings & Wooldridge, 1998]: “What are the essential concepts and notions of agent-based computing.” Jennings [Jennings & Wooldridge, 1998] identifies agents, agent organizations, interactions, and environment as possible concepts. While focusing on abstracting and decomposing agent behavior rather than studying agent or agent-society level issues, we build on his ideas introducing alternate vocabulary.

MAS framework support for flexible conversation management is a key subsystem needed for developing agent societies where interaction is dynamically composed of agents and services. In this paper we describe the situation, problem, and opportunity for innovation for agent-to-agent communication. We are developing 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. Both 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 that certain behaviors might be invoked, yet be flexibly decomposed into coordinating parts that could be updated dynamically.

The following introduction provides some general background information on agent abstraction and decomposition, conversation management, agent modeling, and concludes with our research goals for developing a state-machine-based agent with enhanced, event-driven, dispatching of messages. In section 2, the design of several frameworks is evaluated that support agent-oriented programming, and more specifically, state machines and message dispatching. Architectural elements from these platforms influenced the design of SmartAgent whose features, architecture, and implementation are discussed in Section 3. The conclusion in Section 4 provides an analysis of SmartAgent and directions for future work. In the appendix, we provide annotated examples of SmartAgent code.

1.1 Agent Abstraction and Decomposition

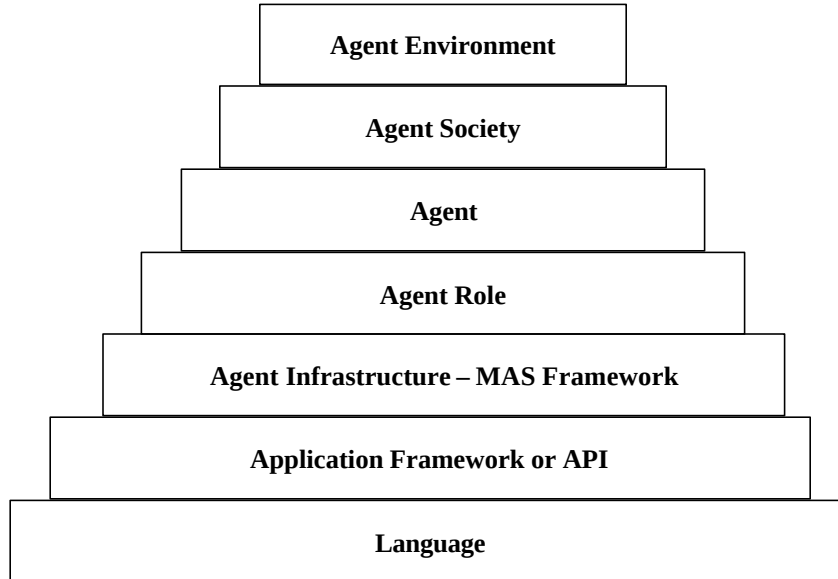
Like the transition from functional to object-oriented programming, moving from object-oriented to agent-oriented programming involves building on top of and re-encapsulating units of computation. The additional layers of abstraction introduced by this paradigm appearing in Figure 1 are named by their encapsulation component. Agent-oriented concepts are first introduced at the multi-agent system level.

Although multi-agent system frameworks can be thought of as being parallel to rather than extending from traditional object-oriented frameworks, they are placed at a higher level of abstraction because they typically utilize object-oriented API's. Well-written MAS frameworks recast object calls using agent-oriented concepts. JADE, the Java Agent Development Environment [Bellifemine et al., 1999], is an example multi-agent system framework that provides the ability to communicate with FIPA compliant society infrastructure agents.

An agent API provides the fundamental capabilities that an agent must possess to participate in the default society chosen by the MAS developers. JADE agents are given the ability to communicate with FIPA compliant society infrastructure agents. The agent API also defines interfaces to the agent subsystems. The number and types of subsystems provided by a MAS differ from implementation to implementation – domain dependent agent requirements often dictate design choices. The trend is for MAS's to minimally provide some level of conversation management, rule-based behavior, and function-based behavior subsystems.

As was stated, most MAS frameworks provide an agent behavior subsystem. These subsystems are often called “engines.” Whether rule-based or object-based, the function of a behavior engine is to provide built-in support for programming agent behavior. Regardless of the application domain, there is a set of actions or utilities that are commonly needed by any agent interacting within a society. Much like typical libraries, factoring these utilities into

Figure 1 - Agent-Oriented Programming Abstraction Layers



MAS support classes alleviates agent programmers from reinventing behavior. Message storage is one example facility provided by the ZEUS Coordination Engine [Nwana et al., 1999]. As is the case with ZEUS, a behavior engine can offer domain specific functionality (buy, sell) in addition to general-purpose agent behavior support. At a minimum, behavior engines need to offer general-purpose support because the intention is for programmers to extend, customize, or tailor agent behavior to the problem domain. Such engines typically provide a pseudo-concurrent mechanism enabling multi-tasked agent behavior execution. This offers a single-threaded multi-tasking solution.

JADE and ZEUS behavior engines adopt an architecture similar to Container-Based Component Management [Kassem et al., 2000] that is used by the J2EE environment as well as the HP Bluestone architecture [BLUESTONE, 2001]. Containers or engines provide guaranteed services to their respective components. For example, a web container provides request services to its Servlet components. Analogously, an agent engine container might provide a new dialogue convenience service to its agent role component.

An agent role is meant to represent a well-defined and encapsulated unit of high-level agent behavior. An agent role can denote the same behavior expressed in defining a protocol role. Alternatively, it can represent some other action an agent performs that may not involve communicating with the society. An agent role is composed of primitive operations from the MAS framework abstraction layer and can also encapsulate other lower level agent roles. In JADE and ZEUS, primitive operations are contained in objects implementing an interface that enables the behavior engine to manage and execute them.

Agent roles can be rule-based or object-based; JADE and ZEUS support both. An interesting encapsulation implication for rule-based behaviors is if all rules appear at the same level, a system rapidly becomes unwieldy as the number of rules increase. Instead, programmers think about “rulesets” or “rule modules,” and use a top-level set of rules to select the active rule model.

While MAS framework designers purport to offer API’s for agent-oriented programming, the current generation of MAS’s offer much more support for programming at the agent role level.

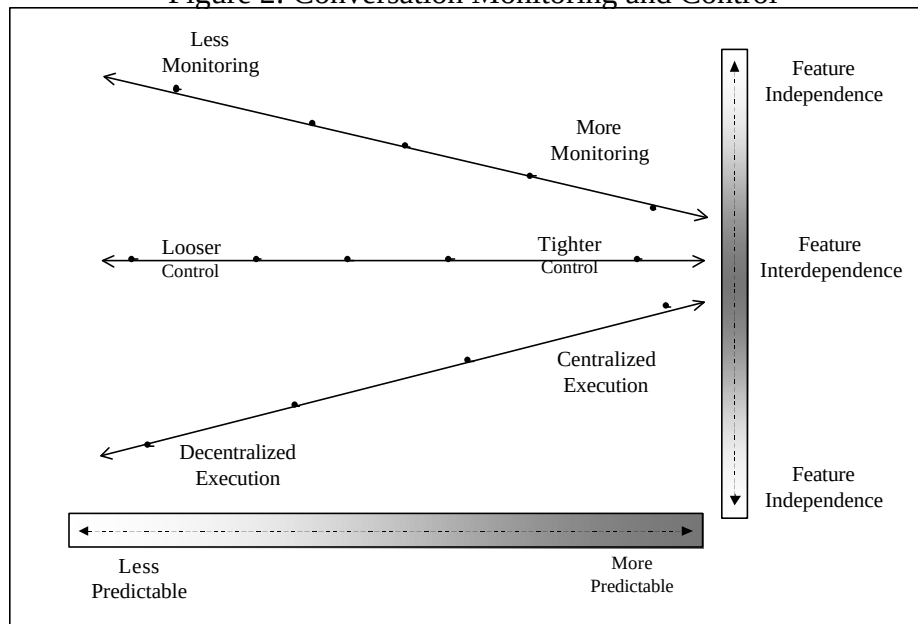
The remaining two abstraction layers are agent society and agent environment:

- An **agent society** encapsulates multiple agents that collectively provide higher-level service or functionality. The CoolAgent implementation [Griss et al, 2001], a previous project in which a multi-agent meeting scheduling society was developed, did not use a single society level interface for meeting scheduling but instead opted for having personal assistants communicate directly with members of the scheduling society. Agent society abstraction could have been achieved using the façade pattern [Gamma et al., 1995] by using a single point of contact to the scheduling facilities. This implementation would have dramatically reduced the interface requirements of the personal assistant and also made the scheduling society functionality more reusable. In general, a good strategy is to minimize the coupling between infrastructure agents and their beneficiaries while maximizing the collaboration (cohesion) within an agent society.
- Above an agent society is the **agent environment**, the highest level of abstraction and unit of encapsulation that is a collection of agent society clusters.

It is currently difficult to partition agent behavior into reusable components that are organized according to cleanly defined levels of abstraction. Though levels of abstraction have been proposed, only after the agent-oriented programming paradigm has been repeatedly applied to solve real software problems will best design practices emerge. Software engineering techniques for increasing modularity and composability, such as model-driven generators, patterns, toolkits, Java Beans and aspect-oriented programming all show promise, as discussed in [Griss 2000a].

1.2 What is Conversation Management?

Figure 2: Conversation Monitoring and Control



It is common in multi-agent systems for a group of agents to coordinate their interactions to achieve some higher-level task, such as requesting, offering and accepting a contract for some services. A group of agents might be expected to have a series of stylized message exchanges to accomplish this task, perhaps by advertising and using a brokered service, bidding during an auction, responding to a call for proposals, or negotiating a price. A variety of regimes might exist in order to achieve various levels of coordination. A conversation management policy is a description of a coordination regime.

A conversation management policy embraces at least three dimensions¹:

1. Conversation Protocol

A conversation protocol is a definition of a structured interaction (i.e. a series of stylized message exchanges). For example, there is a specific structure of interaction for an English auction. In fact, most business transactions follow a well-defined interaction structure [Bach and Harnish, 1979; Finin 1997]. Different standard conversation protocols are used for different types of business transactions. Few business transactions are truly free form. Even ad hoc interactions may follow part or all of a well-defined conversation protocol. Many other activities, such as negotiating a date and location for a meeting, involve a stylized exchange of messages. Conversation participants may be rigid in their expectations that each side is following the same conversation protocol or they may exhibit some flexibility as long as the conversation more or less follows a well-understood general pattern. HP's E-Speak [ESPEAK, 2000], FIPA (Foundation for Intelligent Physical Agents)[O'Brien, 1998], RosettaNet [RosettaNet], and EDI provide some mechanisms for describing conversation protocols. See also [Sachs et al, 2000].

2. Monitoring Policy

Monitoring is the act of collecting information on conversations as they occur. The monitoring can be non-intrusive and covert (by eavesdropping) or intrusive and overt (by explicit arrangement with the conversation participants or through subscribing to a monitoring bus on which participants knowingly publish). Conversation monitoring can be done at various levels. In the simplest case, conversational "utterances" can be reported and perhaps some minimal level of information (e.g. which participant said what and when it was said) can be presented. A slightly more sophisticated monitoring policy might include some analysis and correlation to group the utterances into actual conversations (e.g. by using *conversation-id* or *reply-with* and *reply-to* fields in the ACL messages). Diagnostic monitoring is another level. A diagnostic monitoring policy might enable a monitoring agent with a set of known conversational patterns to pattern match observed conversations against the known set. This enables discovering and observing conversations to determine structure. New patterns might even be learned as conversations are observed. Once a conversation is matched to a pattern, the monitor can provide some information about the state of a conversation and the monitor can recognize deviations from the pattern (note: the monitor would only be guessing since a pattern match is not guaranteed to be 100% correct in every case). If a conversation monitor received information that documented the conversation pattern that was to be followed by one or more conversation participants, the monitor could track the progress of the conversation with absolute certainty. This would enable the possibility for the monitor to send alarms in case of deviations from the documented pattern.

The "levels" of monitoring outlined above are a sample. There are almost certainly more gradations of sophistication that are useful and interesting. See also e-service conversation management [Sahai et al, 2001].

3. Control Policy

Conversation control is concerned with exerting some level of control and enforcement on the progress of conversations. Such control might reside entirely within the conversation participants. In the case of *loose control*, each agent has a protocol (e.g. a set of rules, state machine, or colored Petri net), which it uses to guide its participation in a conversation. This is *loose control* because each agent has its own policy and no provision is necessarily made to ensure that the policies are mutually compatible. For *tighter control*, compatible agents would be expected to use a common protocol (e.g. a particular protocol described using a nested state machine or workflow representation and a specified execution engine [Chen et al, 2000; Griss, 2000]).

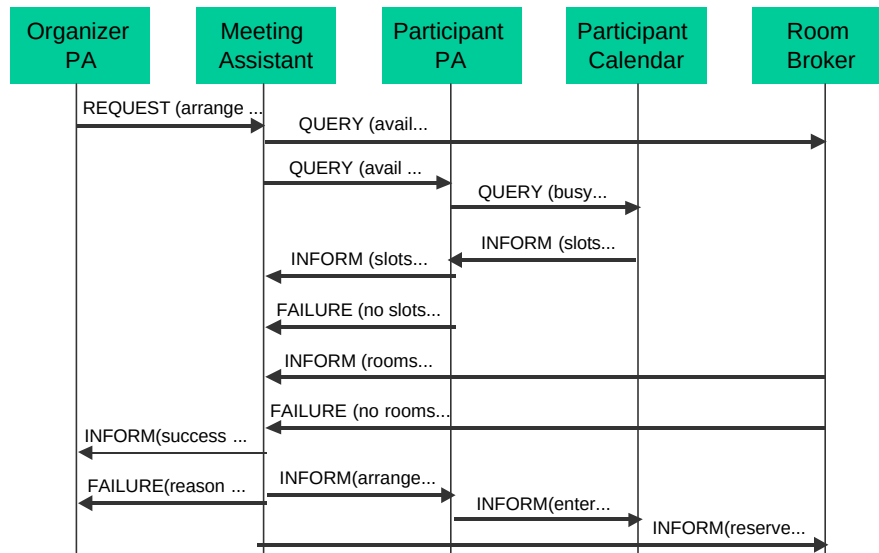
Control might be external to the conversation participants. For *workflow control*, a single collaboration workflow model is associated with a group of agents. This central controller dispatches tasks to conversation participants and mediates all communication between participants, thereby controlling the

¹ This discussion and diagram is summarized from an unpublished report by [Bell and Griss, 2000].

conversational progress and flow. There may also be gradations of external control. An external control agent might share a collaboration workflow model with the participants and closely monitor the conversation progress but only have the ability to stop and start participants. Such control might be exerted if one or more of the participants have deviated from a collaboration workflow pattern to which it should adhere.

It should be noted that centralized vs. decentralized execution is not equivalent to *tight* vs. *loose* control. Relatively tighter or looser control is determined by the degree of autonomy in selecting and following a

Figure 3 - CoolAgent MAS Meeting Protocol



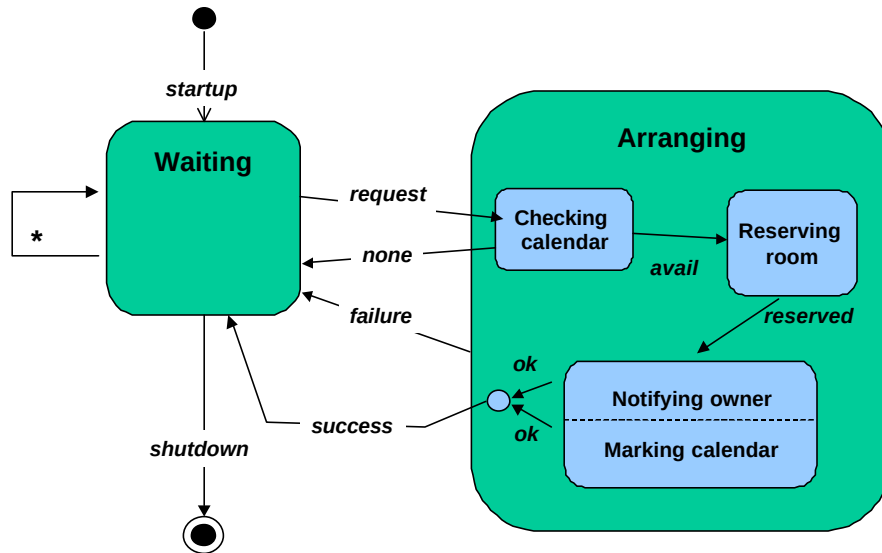
protocol.

Figure 2 is a way to visualize part of the conversation management space. The dimensions of monitoring, control, and centralization are represented as continuous. It is more likely that there are discrete points along each dimension that represent meaningful levels (as represented by the dots). Note that these dots are illustrative only and not meant to imply that there are exactly four levels of centralization, for example. In addition to there being discrete points along each dimension, there are certain combinations of these features that are meaningful and other combinations that don't make sense. In some cases, the choices along each of the dimensions may not be entirely independent of each other. For example, a tight level of control may not be possible without a fairly high level of centralization. Figure 2 shows the dimensions converging as the level of monitoring, control and centralization increase and they diverge as the levels decrease. This is meant to convey the idea that the dimensions are more interdependent as the levels increase and more independent as the levels decrease. Finally, note that as the levels of monitoring, control and centralization increase, so does the level of predictability for interactions and overall behavior. In summary, conversation management is a policy that combines conversation protocols with some monitoring and control objectives, with enough information to make it possible to: raise an alarm, log performance and deviations, attempt to take corrective action, etc. as conversations occur. See also a related report on agent-management in the context of Bluestone [Cowan et al, 2001].

1.3 Modeling Agents and Multi-Agent Systems

First efforts to decompose problems into multi-agent system solutions occur during the design phase when models are developed. Identifying useful components and their relationships establishes the programming abstraction layers that are subsequently implemented. Effectively representing a MAS across abstraction layers requires multiple types of modeling diagrams. The UML [Booch, 1999] is a good choice for modeling agents and multi-agent systems because its semantics are well defined, UML is an industry standard, and several authors are already applying it to

Figure 4 – Example HSM from Meeting Arranger



MAS designs, with minimal extensions in some cases [Griss 2000a] and AUML [Odell, 2000]. As an example drawn from the CoolAgent meeting arranger, an interaction diagram (such as Figure 3) is a good way of showing the pattern of message exchange between agents, and a statechart (such as Figure 4) is a good way of showing the handling of messages within an agent role. These modeling techniques, in addition to Dooley graphs and colored Petri nets, offer possible solutions for how to decompose agents and multi-agent systems. Minimally, they provide MAS framework developers with languages with which to explore the agent-oriented paradigm.

1.4 Research Objectives

The general purpose of our research is to raise the level of abstraction in creating an agent or set of agents that behave according to a desired plan, protocol, or collection of rules. This may collectively be called “instructing software agents.” Mechanisms for monitoring, modifying, and reusing these instructions need to be developed. Of further interest are making agent construction easier and their behavioral implementations more understandable. At the beginning and end of realizing all of these technologies are the tasks of developing techniques for expressing (modeling/defining) agent behavior and code generation tools. Our agent system is a mixture of the JADE MAS framework, custom Java classes used to extend the JADE behavior and message dispatching model, and the Jess rule evaluation system. In building our agent system we attempted to address the questions that follow.

- **MAS Platform Agent Behavior Support**
What is a good algorithm and implementation for routing incoming messages to the correct execution unit? What amount of intelligence is useful in dispatching messages and how is this intelligence programmed? How can rule-based programming be utilized? What high-level message management policies should the MAS platform provide to an agent by default? What built-in support, if any, should be provided by a platform to support state-based agent composition? How can a MAS platform help rapidly construct agents?
- **Agent Behavior**
How can agent behavior be decomposed into high-level actions? How can an agent be constructed from reusable components? What are these components and what interface do they share? What architectures make it easier to understand and maintain agent behavior? Is state-based agent behavior a useful

programming paradigm? How are the processing steps an agent executes partitioned into states and their transitions? How can agents be programmed to respond in dynamic and unpredictable societies?

- **Agent Construction Tools**

How should agent behavior and state be monitored? What tools are needed to help understand and debug the runtime behavior of agents? How can agents and their objects be inspected? What modeling techniques help design and document an agent? Can a tool be developed that takes an agent statechart, possibly expressed with the UML, and generates state-based behavior components compatible with a chosen MAS platform.

In this first release of our SmartAgent agent building toolkit reported in this and companion reports, our work was guided by these questions and the overall vision described above; however, we have certainly not addressed all of the questions, nor in complete depth. In the following section, we review some related work on message and event dispatching in agent systems, and particular state machine approaches.

2 Review of Event Dispatching and State Machine Implementations

The JADE and ZEUS MAS platforms that we have used extensively provide support for state-based agent behavior programming. JADE offers a lightweight state machine class with limited functionality while ZEUS provides an entire state machine execution subsystem. The capabilities of each framework are assessed in this section as motivation for, and as a foundation from which, an improved state machine unit was built. Additionally, a hierarchical state machine implementation in C and C++ [Samek and Montgomery, 2000], and some other agent conversation control work is also reviewed.

2.1 ZEUS

The ZEUS MAS platform provides a coordination engine that executes state machines. These state machines are called graphs in ZEUS. A graph, which encapsulates a high-level agent behavior, is composed of processing nodes and the arcs that link nodes together. Each node is split into two methods called `exec` and `continue_exec`. The `exec` method is called the first time a node is executed while the `continue_exec` method is executed thereafter. Precondition code is defined in the `exec` method of arcs that determine what node should be executed next. Backtracking of node traversal is supported as each node defines a `reset` method that is responsible for undoing processing.

The coordination engine manages executing nodes and provides convenience conversation methods. Pseudo-parallel execution is supported by interleaving node processing using a first-in-first-out policy. Nodes typically perform some processing, are placed back on a queue while waiting for dependent processing to finish, and then re-queued for execution. There is built-in support for node timeouts. The engine reduces the tedium of explicitly coding ACL message exchange and management, and subsequent data structure translation provides conversation utilities. ZEUS also has built-in methods that execute graphs for buying and selling.

All ACL messages are routed by a message handling class. This class allows rules to be registered that match incoming messages with the correct processing object. Rules are regular expression strings for specified fields of the arriving performative. For example, it is possible for a single processing object to handle all incoming messages that have a content field starting with 'R'. Once a rule is satisfied, the registered method of the processing object is executed with the performative as its parameter. After execution, the message handler checks the next rule to see if it is satisfied by the incoming message. This continues sequentially until all active rules are given an opportunity to process the message. Rules can be registered with the message handler dynamically. There is built-in support for one-shot rules such that after processing a message for the first time, the rule is deregistered.

ZEUS provides several useful miscellaneous features. The ZEUS coordination engine can process multiple graphs in parallel. Nodes are in one of several predefined states that dictate how they are processed. A visualization tool is provided that represents state information about the nodes and executing graphs. The message handler object contains statically defined rules that ensure agents follow society conventions. This is problematic when building agent societies with different norms.

2.2 JADE

Similar to ZEUS, the JADE MAS framework provides a behavior engine for programming agent behavior using homogeneous processing elements that are executed in pseudo-concurrent fashion. JADE, however, defines a hierarchy of behavior classes implementing common execution patterns rather than tailoring the engine to process state machines. New versions of JADE do offer state machine execution support on top of their existing infrastructure. But this mechanism is not nearly as well developed as in ZEUS. The JADE engine processes behaviors as opposed to states within a given state machine behavior.

Behaviour, the root class of the behavior hierarchy, defines several core methods whose semantics are similar to methods of the UML state class. The onStart and onEnd methods are executed upon entry and exit of a behavior. An action method embodies the core code for the behavior. The JADE MAS does not have predefined states for behaviors as with ZEUS. Instead, programmers are left the task of defining application specific (as opposed to general states such as ready, waiting, finished) states and it is suggested that these states be implemented using a Java switch statement. Like ZEUS nodes, every behavior has the option of implementing the reset method to undo state changes and therefore enable backtracking.

The JADE MAS provides a round-robin dispatching mechanism for routing incoming ACL messages. The 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 typically programmed to be put back to sleep. Explicit reposting of a message is required when it should be handled by multiple behaviors.

CompositeBehaviour and SimpleBehaviour are children of the Behaviour class. JADE provides classes descending from SimpleBehaviour that fix the number of times the behavior runs. Classes descending from CompositeBehaviour implement support for the ordered execution of multiple behaviours according to a policy. The most sophisticated class from this branch of the hierarchy is FSMBehaviour. The policy for this behavior is implemented by defining a state machine of behaviors. Because each state is itself a behavior, it is possible to embed state machines.

The FSMBehaviour class has the responsibility of maintaining the relationship (transitions) between states and selecting 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 noted, the state machine is ready for execution. When a state behavior finishes, the 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.

2.3 State-Oriented Programming

Though not related specifically to programming agent behavior, research on state-oriented programming by Harel and Politi [Harel and Politi, 1998], and Samek and Montgomery [Samek and Montgomery, 2000] is applicable to developing MAS frameworks. The work of Harel and Politi and colleagues on reactive system models had a major influence on the development of the UML standard Statechart state machine model. Samek and Montgomery implemented a hierarchical state machine (HSM) in C and C++ that offers a subset of UML statechart functionality. The HSM they implemented was meant to provide a base of core functionality from which future extensions, such as adding concurrent state processing, could be developed. Therefore, in evaluating the HSM implementation, more attention is placed on the constraints introduced by the architecture than the lack of features.

Their HSM design includes three main classes, HSM, State, and Msg. The Msg class encapsulates the data transferred with a particular event along with the event type. Because Msg encapsulates data passing, events are uniformly handled. Msg is a misnomer; intuitively one would expect this class to be named Event. When an event is received by HSM, it is forwarded to the correct state class for processing. The responsibilities of the State class are minimal. By design choice, and in contrast to the JADE and ZEUS implementations, the State class is instantiated and not sub-classed. This is because State objects are used to define the state hierarchy but not to encapsulate code.

Instead, each state object maintains a function pointer to an event handling method within its HSM. These methods employ a switch statement that dispatches the incoming *Msg* by event type. Event types include at a minimum: *entry*, start (*action*), and *exit*. Additional event types can be defined that are state specific. Whereas dispatching in JADE and ZEUS meant passing an event to another object, for this HSM implementation it simply means jumping to a specific line of code within the event handling function. To be clear, a state has only one event-handling pointer. That method uses a switch statement in deciding how to process the event.

In addition to holding all event-handling functions, their HSM is responsible for determining which state should handle an incoming event. It also ensures the correct ordering of entry and exit method execution chains. The HSM constructor builds the state machine and its *onStart* method activates the state machine. Its main method sits in a loop waiting for incoming events.

The HSM architecture has some constraints to consider. First, the main body of code is not distributed across objects since HSM maintains the event-handling method for all states. The coupling between a state and an HSM is high; making state objects non-reusable components while suggesting that an HSM is the most primitive building block. The advantages of this include: 1) Easy access to HSM methods and attributes during state processing. 2) Avoiding the creation of lots of small state subclasses that contain very little code. The disadvantages include: 1) The HSM class could be quite large. 2) The programming granularity is fixed at the HSM level but encapsulating agent behavior in state objects might be more reusable.

The HSM architecture only dispatches events based on the current state and the type of event received. There is no support for dynamically adding event-handling code because static case statements are used and the code is not encapsulated in an object that can be referenced. In the agent world, events are routinely routed by inspecting the attributes of an ACL message. This means that a case must be defined for each set of attribute criteria. Using a case statement effectively prohibits adding state transitions at runtime.

Their HSM does not support triggerless transitions or the use of guards.

Since our implementation is designed for more dynamic modification as part of the extended JADE system, we do not follow the Samek and Montgomery approach in any great detail, but it did offer inspiration and several ideas on implementation.

2.4 Other notable systems

Workflow has been proposed and used to coordinate agent conversations [Chen et al, 2000; Griss, 2001]. Scott Cost developed an independent state machine based conversation engine named JACKAL [Cost et al, 1999] and discussed use of (colored) Petri nets for this purpose [Cost et al., 2000].

3 Architecture and Design of Our SmartAgent Extensions to JADE

The agent behavior units reviewed in the last section offered alternate combinations of features. Their strengths and weaknesses varied accordingly. For example, the message dispatching mechanism of ZEUS is of finer granularity than JADE. Recall that ZEUS uses regular expressions while JADE uses simple string matching with logical operators (and, or, not). ZEUS provides more sophisticated support for a form of state machines (called graphs) than does JADE. The JADE API is much easier to use than ZEUS. JADE and ZEUS both provide monitoring tools that are not easily compared because they offer different capabilities. Despite their differences, by raising the level of abstraction it is possible to identify common architectural elements. They both have a behavior engine that serves as a container for running encapsulated and primitive units of agent action, the engines interface with the messaging subsystem and provide pseudo-concurrency, they both have message matching and message dispatching mechanisms, and the behavior engines are both distinct agent subsystems.

However, we found that doing certain simple things in either MAS was difficult; for example, in JADE we found it hard to implement a single behavior that acted as a default if no other behavior would respond to an event, or a behavior that would cause an action if a specified amount of time would pass before any message was responded to, etc. These things could be done, but it was more complex than necessary. The classes and mechanisms provided did not quite match our desires to compose complex behaviors from piece parts.

Given that MAS frameworks are generally composed of the same high-level components and that it is the designers who determine their particular features, a question must be asked: What combination of features, for each high-

level component, should a general purpose MAS framework offer? The answers must consider the characteristics of the agent operating environment. Deployed solutions, as has been noted, will likely operate in a distributed, dynamic, and heterogeneous information space. While some requirements are evident, the MAS framework capabilities needed to enable rapid development of agent-oriented applications are more likely identified through iterative MAS framework refinement as programming and domain experience is gained. Currently, only educated guesses can be made and these hypotheses tested. We make assumptions on the usefulness of some attributes in defining the design requirements for adapting the JADE MAS to support richer message routing and state machine-based agent behavior programming.

In the context of the JADE MAS, our goal was to provide an easy to use set of classes and mechanisms (a “kit”) that would make it feasible to build the kind of robust behaviors we needed. We wanted more control over the order of event dispatching, we wanted to use more powerful event matching patterns and rule-based dispatches (using Jess), and we wanted dynamically extensible, state machine control of message traffic and activities.

Our long-term goals are to develop a programming paradigm and implementation that facilitates agent construction through libraries of reusable behavior. Also of interest is developing agent code generation built from a UML modeling tool that might provide support for UML stereotyped based extensions (AUML[Odell 2000], agent patterns[Kendall 1999], Gschwind[Gschwind et al., 1999], and others[Griss 2000a]).

3.1 Some approaches to the problem

Static (design time) selection of conversation protocols, the degree of centralization, and monitoring and control policies for an e-service or agent limits the interactions in which an e-service or agent can participate. E-services and agents built and deployed by different organizations will individually and collectively have a variety of conversation management policies. The heterogeneity of the web virtually assures this. As more e-systems are built dynamically, by composing increasingly autonomous e-services and agents, more flexible ways of choreographing (describing, controlling, and monitoring) the interactions between participants will be critical.

One challenge is the development of a runtime framework or kit to support runtime plugability for the various dimensions of conversation management. Such a framework would have to support a potentially wide range of capabilities for each dimension without succumbing to the “least common denominator” set of capabilities. Runtime flexibility is necessary but not sufficient for a complete solution. Runtime flexibility needs to be augmented with mechanisms that enable description and negotiation for policies. Conversation protocols, and monitor and control policies must therefore also be electronically describable and negotiable. The selected policy and any policy attributes should also be part of any electronic contract that is formed.

In some environments, all aspects of a business process or conversation policy can be described, codified, prescribed and enforced. In other environments, there may only be partial control. In some situations the process may be sufficiently static to justify up front investment in process definition and the systems that can enforce the processes (e.g. Workflow systems). In other situations the processes are modified too frequently to justify the cost of encoding them, or time to market issues make it necessary to adapt more quickly than is possible when rules are fully codified. There also exist some environments in which a complete, well-defined, process is not known or does not exist. In some cases a community or market place (c.f. [RosettaNet]) establishes the business processes.

Another model for constructing and composing policies is the use of templates and/or “pieces” (fragments) of policies that are assembled into larger units (e.g., aspect-oriented construction of agents [Kendall, 1999], [Griss 2000a]). This is possibly recursive and there are some interesting challenges that arise. In some cases, a contained fragment may be required to adhere to certain constraints specified in the outer level policy. For example, if an outer policy is supposed to guarantee transactional atomicity, then any sub-components must also be able to conform to this constraint. In other cases the contained fragment may have more freedom. The problem of properly expressing the constraints (or lack of them) and their interactions with inner, outer, or sibling policies and then enforcing them as policies are dynamically composed represents a real challenge.

Flexible and dynamic conversation management techniques can be used to address the above-mentioned challenges. See Section 1 for more discussion.

3.2 SmartAgent Dispatcher and HSM Features

Before we go into detail about our design and implementation, we will first summarize the conceptual model and the key features. Our model is event-driven. This means that as different kinds of events (such as messages, timers, exceptions) reach the SmartAgent behavior engine, each is dispatched to an appropriate handler, which will then execute one or more computational activities. Event dispatchers use pattern or rule templates to recognize different classes of events. These event dispatchers can be registered with the SmartAgent engine.

What makes our system both interesting and somewhat complicated to describe is that different kinds of dispatchers can be added and removed dynamically, default dispatchers can be set up to handle all or some events and exceptional conditions not explicitly handled by other dispatchers, dispatchers can be grouped and ordered in various ways to build a dispatch tree, and execution of a selected dispatcher can activate or deactivate whole groups of other dispatchers. For example, during an event-driven transition from one state to another. Thus an event-driven state transition is a form of event dispatcher.

A dispatch tree, as will be described in detail later, is a structure of intelligent objects that forwards incoming events to their respective execution units. What is meant by dynamically extensible is that our implemented state machine allows states and transitions to be added at runtime. By intelligent, we mean pattern and rule based responses to events. A separate, but related, goal was to support flexible web-based inspection and monitoring of agent behavior.

As a key element, we have included a fairly complete implementation of the UML state machine model [Booch 1999; Harel and Politi, 1998]. This is a powerful hierarchical state machine (HSM), which includes nested and concurrent states and a variety of event-activated transitions. The UML HSM offers a standardized way to break software solutions, or more specifically agent behavior in this case, into a series of discrete processing steps that must be performed. It has been observed that hierarchical state machines can naturally model the internal processing of an agent [Kendall 1999, Odell 2000, Griss 2000] and agent-specific extensions to the UML have been proposed. Computational activities can occur on a transition, in a state, or on entry or exit from a state. Events may be processed in a state, cause transition to another state, be deferred for processing in a different state at a later time, or even be totally ignored while in a particular state. Common activities can be abstracted to enclosing composite states and thus “inherited” for all nested states; likewise, common or default transitions can be attached to enclosing composite states, and “inherited” for use in all nested states. Our novel extensions in SmartAgent permit the dynamic addition or removal of states and transitions. Also, the grouping and ordering of dispatchers can also be applied to state transition dispatchers.

Figure 4 is an example of a simple state machine from our meeting manager example. The state machine can be first modeled diagrammatically in UML and then implemented fairly directly in SmartAgent.

In more detail, the features and user visible architectural elements of SmartAgent include:

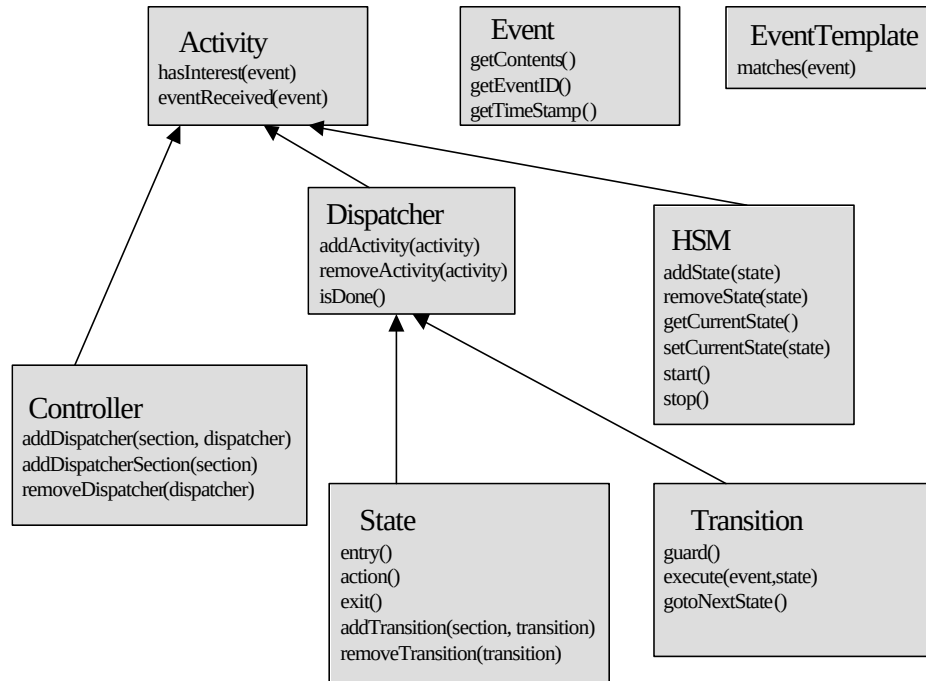
- Uniform event handling -- Most types of internal or external input is converted into an event. These include message, timer, and exception events. These are generated by the system, by incoming messages or signaled programmatically. This uniform treatment permits an agent to handle a diverse range of society and system occurrences using the same dispatch mechanism.
- Event-based dispatching - A dispatcher controls zero or more activities and is responsible for implementing an event distribution policy for these activities. The default dispatcher class (which serves as a base class for others) implements a policy of using a *hasInterest* method as a “pre-filter” and then distributing the event to every activity whose *hasInterest* method returns true. Dispatchers and activities typically use an EventTemplate in their *hasInterest* methods.
- Event Templates – The activation of an activity, dispatcher or state transition is controlled by an event template. A rich and extensible set of templates include *And*, *Or*, *Not*, *Always*, and *Never*. Event templates may also be Jess-based rules.
- Dispatcher grouping, ordering, and defaults - A number of dispatchers are managed by a controller. The controller implements a dispatcher ordering policy. The default controller manages several named sections, each with a group of dispatchers. The groups are processed in turn (priority), and the controller can stop further processing of the remaining sections if a dispatcher so indicates. This makes it easy to implement groups of dispatchers which are always run (such as loggers), then alternative key behaviors, and then default or exception

handling behaviors run only if no key behavior applies. Dispatchers can be added and removed dynamically from any section. A controller is typically run by a timer or JADE Behavior, or by another controller to produce a (dynamically changing) tree of dispatchers.

- Multiple activity firing – A single event can trigger the execution of multiple activities. This is a consequence of multiple activities being associated with dispatchers.
- UML hierarchical state machine (HSM) – An HSM is a particular kind of controller in which behavior can be explicitly partitioned into UML states and transitions. Using the UML model provides a high-level template for programming an agent, and provides many powerful ordering, grouping and default features. Using a standard UML-based modeling language makes agent construction more accessible to the programming community.
- The HSM features include (refer to UML for details):
 - States – Include initial state, final state, state, and composite state. Each state acts as a controller or dispatcher for a group of transitions, and provides standard entry, exit and “idle loop” activities. Partially implemented are a defer queue and skeleton classes for concurrent state, join state, and fork state. Not implemented are history states.
 - Transitions – with event templates, guards, actions, and several types: transition, internal transition, default transition and immediate transition.
 - A state (or complete state machine) may be nested inside other states, and sets up an “inheritance” chain of transitions and entry/exit actions. This mechanism makes it possible to reuse larger portions of code thus raising the level of abstraction. Code complexity can be hidden within a state machine; the states that interact with this machine are only required to know its input and output characteristics.
 - Default agent behavior – Nested states makes it easy to program default agent behavior. UML semantics dictate that if an event does not trigger a transition in the current state, the parent hierarchy is traversed until a transition is taken. Enclosing states can provide behavior by default that successively internal states can redefine, or use.
 - Dynamic addition and removal of states and event handlers – States and all types of transitions can be added and removed at runtime.
- Built-in logging and monitoring capability – Logging code provides agent audit trails (file) and also web-based runtime viewing of agent and system activity. The monitoring system provides several levels of debugging that set the granularity of agent and system activity that is reported. There is also support for defining programmer specified debugging levels.

In the next section we will describe the architecture and implementation of the SmartAgent dispatching model in detail. This consists of a number of interacting interfaces and classes (such as Dispatcher, Event Template, Controller, Activity, State, Transition) mentioned above, chosen to increase the composability and reusability of behavior fragments.

Figure 5 – Simplified Architecture



3.3 HP SmartAgent Behavior Architecture

Before diving into the explicit details of all classes and interfaces, the architecture and major mechanisms can be understood from Figure 5, which shows a slightly simplified set of interfaces for explanation purposes², from Figure 6 showing the Controller/Dispatcher model, and from Figure 7, the essence of the HSM. The code and later sections should be consulted for the exact details.

While JADE behaviors are like “pseudo threads”, we have chosen in this first implementation of SmartAgent to treat event dispatching and activities as a kind of “big switch,” that distributes events to activity code, or “flip the switch.” We use a JADE behavior to pump events into the dispatcher tree. The implications of this choice will be discussed later. It does have an impact on what happens while “waiting” for an event, and how one writes behaviors.

Our design consists of three major subsystems: an event fusion and matching system, an event dispatching system and the HSM code that partitions agent behavior into states and transitions.

Event and EventTemplate

As an event-based architecture, every action that an agent is subjected to is translated into an event for uniform handling and processing. For example, incoming ACL messages from the agent society are wrapped in a `MessageEvent` object. Other implemented event objects include `ExceptionEvent`, `TimerEvent`, `SuccessEvent`, and `FailureEvent`. Because these objects descend from a common parent, **Event**, this allows an agent and its internal systems to handle them uniformly. In order for events to be processed, they must be routed to objects interested and able to handle them. Our implementation of event matching is very similar to JADE’s message matching solution. Each event encapsulates an object of a designated type. The object wrapped in an event 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. Multiple matching checks can be combined using logical connectives to create a single boolean expression that can be tested for satisfaction. The logical connectives, event types, and event templates are shown in more detail in Figure 8.

² Figure 5 illustrates the interfaces and relationships but is not the exact UML class hierarchy used in the current implementation.

Dispatching and controller mechanism

The HP dispatching and controller mechanism augments the JADE message routing system by taking responsibility

Figure 6: SmartAgent Dispatching Model

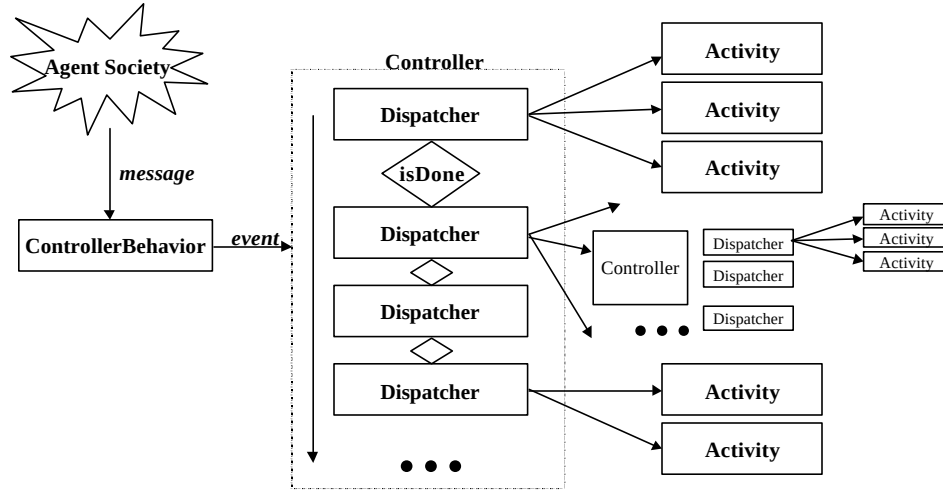
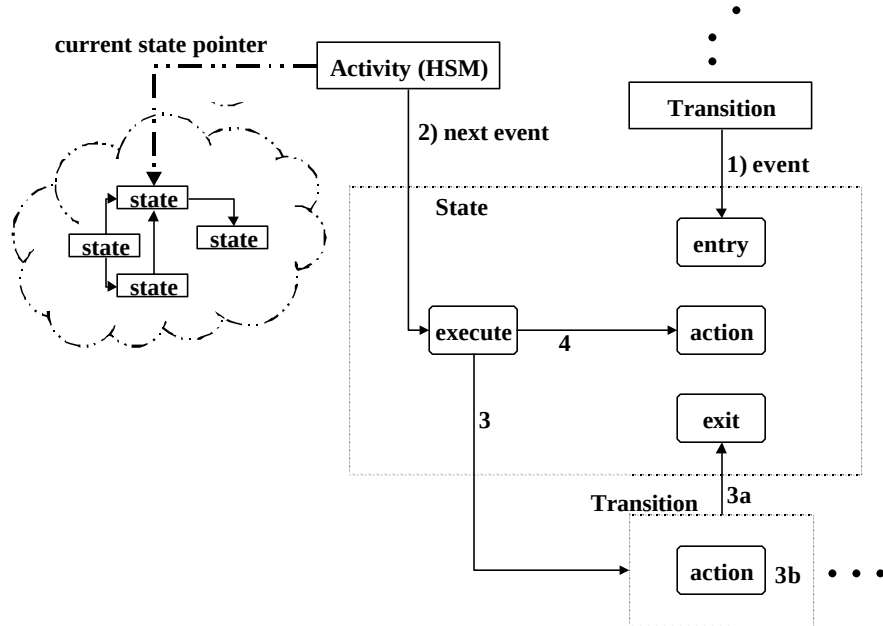


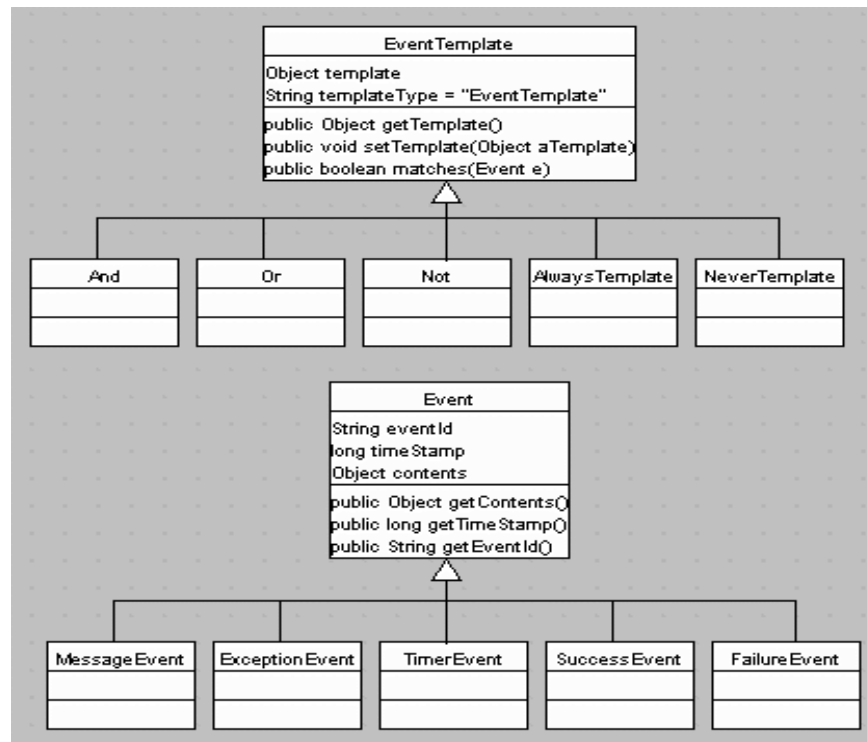
Figure 7: SmartAgent HSM Model



for channeling messages to their correct execution object (an Activity).

As shown in Figure 6, a simple and standard JADE behavior called **ControllerBehavior** is continuously run as a top-level agent behavior whose responsibility is to retrieve all incoming messages. Once retrieved, the message is

Figure 8: Event model



wrapped in an appropriate Event (usually a MessageEvent, but sometimes a TimerEvent) and forwarded to the controller for routing. The controller contains a chain of dispatchers. When a new dispatcher is added to a controller, it is appended to the end of the chain (or end of a named section, when sections are used). Dispatchers are first-level event routers that determine if the incoming event is of interest (via *hasInterest*) to their group of activities. Activities are the fundamental processing elements of an agent in the HP SmartAgent Architecture. Their behavior granularity can range from a very simple and short code sequence to another dispatcher or full multi-state state machine. The activity interface allows events to be passed using the *eventReceived* function. As shown in Figure 5, controllers, dispatchers, and activities all implement this interface, and so dispatch trees can be built as shown in Figure 6, by hanging other controllers or dispatchers off a higher-level dispatcher.

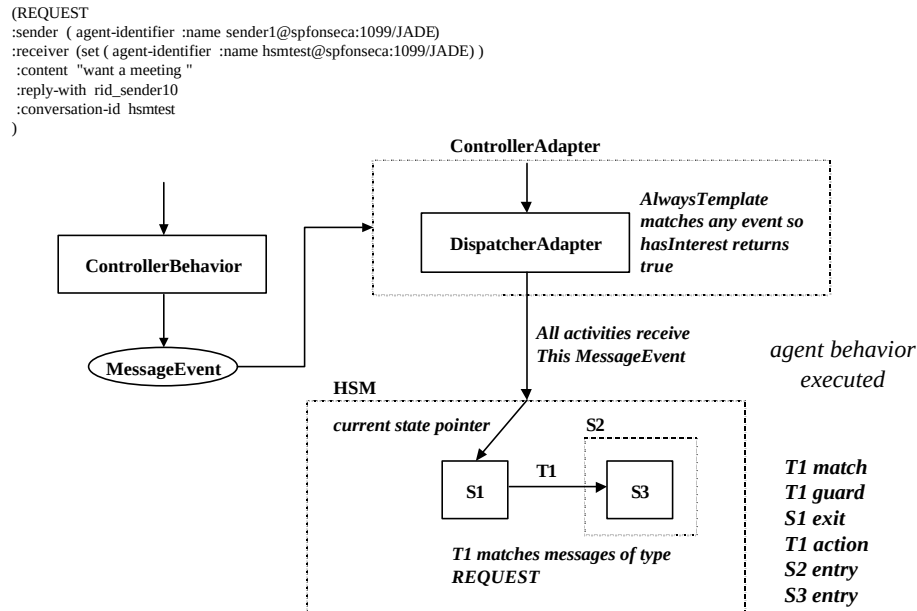
Each dispatcher manages a collection of activities (added and removed using *addActivity* or *removeActivity*). When the dispatcher receives an event, it determines if this event is of interest (via *hasInterest*) to the collection of activities it controls and if it is, distributes it according to its distribution policy (with a default of invoking all associated activities). After activity processing is complete, the dispatcher is responsible for telling the controller if the event should be forwarded to the next dispatcher in the controller chain. To allow other dispatchers the opportunity to process the event, the current dispatcher returns true when the controller calls its *isDone* function.

Figure 6 does not show the grouping of dispatchers into named sections, which are a convenience for dynamically adding a new dispatcher at a specified place in the chain or tree. Activities, dispatchers and sections can be added or removed at runtime, using the *addDispatcher(section, dispatcher)*, *removeDispatcher(dispatcher)* and *addDispatcherSection(section)* methods.

Hierarchical State Machine

Actions that an agent must perform can also be further decomposed into states and incoming events trigger state transitions. Our state machine implementation uses 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. The current state pointer is set to the initial state when the HSM is first created. When transitions are taken, the current state pointer is updated. The HSM also maintains a reference to the final state. The simplified process is shown in Figure 7.

Figure 9: SmartAgent Dispatching Model



All states in an HSM descend from a class called `BaseState`. `BaseState` provides the basic functionality that all states share and defines template functions that succeeding states must override. For example, the `eventReceived` method must be rewritten and is dependent on the type of the state object. Figure 9 shows the dispatching model in more detail, and Figure 10 the state class hierarchy. `InitialState` and `FinalState` are the simplest extensions to `BaseState`. As defined in the UML, these state objects differ from other state types because there are restrictions on the types of transitions they are allowed and they do not implement the entry, action, or exit functions. These functions are introduced into the hierarchy by the `State` class.

The `State` class, its descendents `CompositeState` and `TimedState`, all implement the entry, action, and exit functions. Entry is called each time one of these new states is entered just after the action method of a transition. The action function is invoked when an event is received but does not result in the firing of an event handler (a more general type of transition). The default behaviour for action is to run the set of attached activities (see `addActivity` and `removeActivity`). The exit function is called before a transition is taken to leave the state. A `CompositeState` provides the additional capability of encapsulating a collection of state objects similar to the HSM. Like HSM, a `CompositeState` includes initial and final states. A `TimedState` extends `State` by providing a built-in timeout mechanism that causes a timer event to fire after a specified duration has elapsed once its entry method executes.

State objects are connected by transitions, a subclass of `EventHandler`, a kind of `Dispatcher`. When an event is received by the current state, a check is performed to see if it triggers any of the transitions leading to a next state. For a transition to fire, its guard and event matching criteria must be satisfied. Event matching was previously described for dispatchers. The boolean *guard* function encapsulates additional requirements that are not related to the received event (we are thinking of adding guard to `Dispatcher`, or combining as an option in `EventTemplate`). For example, a guard might check relevant internal values of an agent in determining if the transition should be taken. Like state objects, transitions also have an action function that is executed before entry of the next state. A transition's default action code is to execute any attached activities. Naturally the default behavior may be overridden to do any possible action.

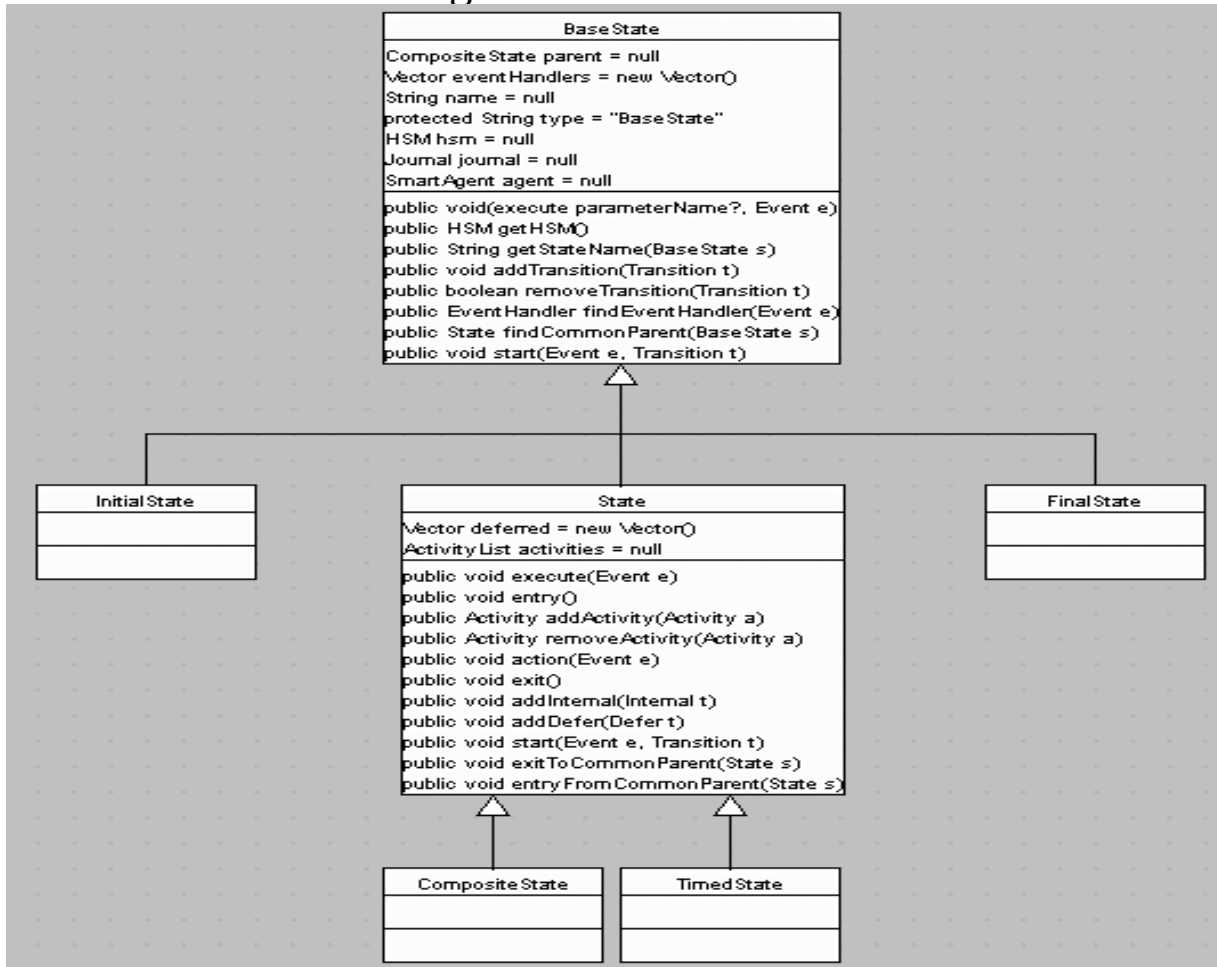
Along with these external transitions, internal and deferred transitions are also checked. An internal transition is the same as an external transition except when it fires, the HSM continues to remain in the same state, and *entry* and *exit* methods are not run. The firing of a deferred transition places the event in the deferred event queue for processing in a later state. Full support for internal transitions was completed while deferred transitions were only partially implemented.

Because UML state machines are hierarchical, there are two implications to consider. First, if an incoming event does not fire one of the event handlers (aka transitions) of the current state, then the event handlers for parent states are recursively checked. This mechanism makes it easy to program default agent behavior by having composite states provide transitions to states that handle default cases. Further, internal states can redefine transitions matching the same type of event as its parent to override what is already provided. The second implication is that the sequence of entry and exit functions must be understood when transitions cross composite state boundaries. For example, as shown in Figure 9, if transition T1 exists between S2 and S4, where S2 is a child of S1, and S4 is a child of S3, the expected sequence of method invocations is: S2 exit, S1 exit, T1 action, S3 entry, and S4 entry.

There are two more types of transitions that we must mention: immediate and default. After a state's entry code is executed, all immediate transitions are evaluated. An immediate transition goes to a new state without an event causing that transition. The immediate transition is a first-class transition and also includes a guard function that is evaluated to determine if the transition should be fired. If it returns true, then the transition is fired. If false, then processing checks to see if any deferred events are available and then the state waits for the next event.

Our last kind of transition is default. During our implementation of numerous state machines we had many situations where we had a set of transitions that would trigger on a set of events. If none of these transitions would fire, we would move up the parent hierarchy to find a default handler. Often we wanted a default handler immediately above the state to handle any exceptional events. This idiom was so common that we created a new transition called default. Default is in essence a macro for the common case of having a composite state that matches all events, contained within it is a state that matches the desired set of events. Processing of default transitions are handled by first seeing if any transition can handle the event and if not, then the default transition's guards are evaluated. If true, then the default transition is fired.

Figure 10: State Model



To summarize, after a transition's action is processed, the following occurs in this order:

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

Agent Monitoring

Without going into great detail, an important element in the goal of making agent development easier was to make the internal structure of an agent to be more regular, and to expose key status information in a uniform way. This

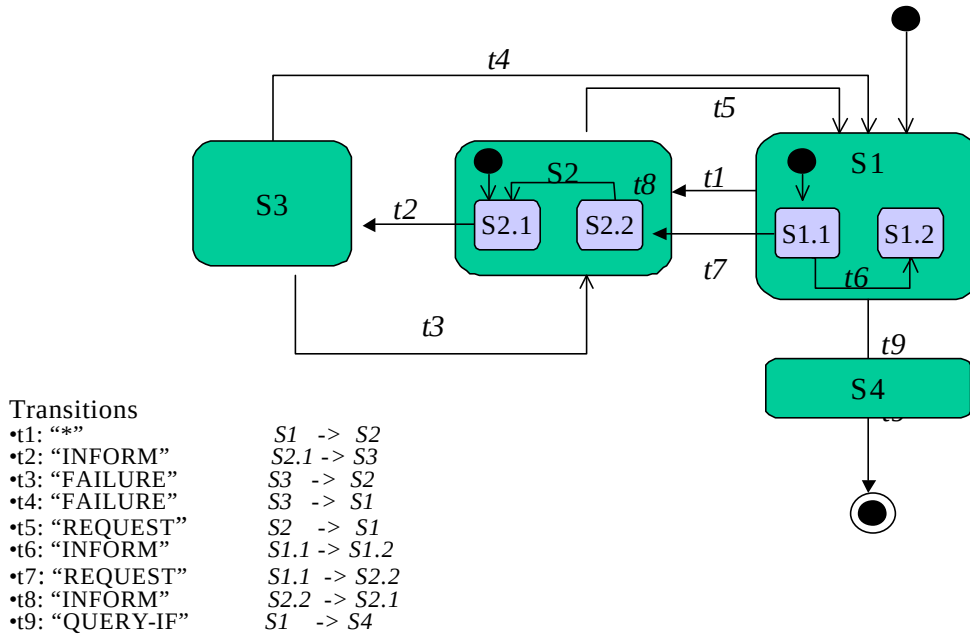
would allow us to see what an agent is doing, and permit more effective agent testing, and a variety of dynamic management functions (monitoring and control) of executing agents. JADE provides a Sniffer (to display ACL message traffic) between agents, and an Introspector (to see some status information inside an agent). We made extensions to the original Sniffer (included in the new JADE release) so that messages that are part of the same conversation are coded in the same color, again to enable better tracking of complete conversations.

To this we have added the use of Inspection[Cowan & Griss 2001], which instantiates a small web server inside an agent, that uses reflection to expose selected public *get* and *set* methods on dynamically created web pages. This allows monitoring and some limited control. By defining a standard set of key public access methods for the Controller, Dispatcher, Activities, HSM, States and Transitions, we can observe the progress of agent conversations, and adjust selected parameters.

3.4 HSM Examples

The example shown in Figure 11 traces the processing of a message received by an agent with an active state machine. Loading the agent with the dispatching mechanism, construction of the state machine, and adding the state machine as an activity of a dispatcher were performed when the agent was constructed. Figure 9 shows an incoming message that is relayed to the ControllerBehavior. The ControllerBehavior encapsulates the ACL message in a MessageEvent object and forwards this to the ControllerAdapter. Though not shown in the diagram, recall that the ControllerAdapter contains a chain of dispatchers that help route incoming events. The ControllerAdapter passes the MessageEvent to the first, and only, DispatcherAdapter whom calls its hasInterest method to determine if the

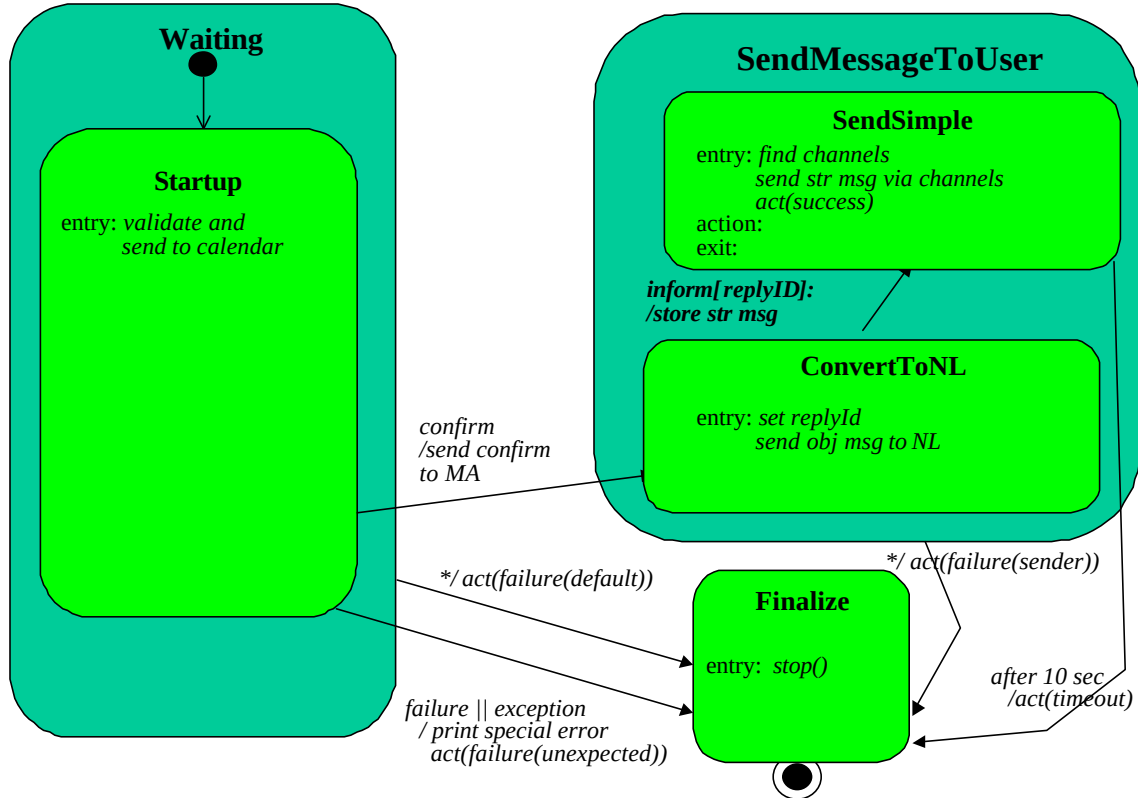
Figure 11: A sample nested HSM



event should be processed by its associated activities. DispatcherAdapter, being a default implementation of the Dispatcher interface, returns true because this event is matched with the AlwaysTemplate.

Having established an interest in handling the event, the DispatcherAdapter sequentially forwards it to each activity. In the example case, the event is received by a single activity named HSM that is a state machine. Its `currentState` pointer is used for relaying the event to the correct state object. State S1 searches for a transition that matches the event just received. Because the event criteria of transition T1 matches the `MessageEvent` and the guard function

Figure 12: Detailed CoolAgent Example, without Concurrency



also returns true, T1 is fired. This results in a sequence of method executions as defined by the UML state machine specification. First, S1 is exited. Then T1 action is called. Finally, the chain of entry calls, as dictated by the path to the new state, is performed beginning with the outermost state object S2. Though not shown, the current state pointer of HSM is updated to S3.

Figure 11 shows a set of nested states and a description of the transitions, while Figure 12 is a complete example taken from the CoolAgent meeting arranger protocol which expands detail on that shown in Figure 4; since concurrency has not yet been implemented, the concurrent states have been replaced with an almost equivalent sequential pair.

3.5 UML Compliance

Because the UML state machine semantics are detailed and intricate, it is beyond the scope of this paper to identify all of the discrepancies between the specification and our implementation. Our goal was not *a priori* to comply fully with the UML; it was to build a state machine infrastructure for the structured programming of agent behavior that could benefit as much as possible from the specification, understanding and toolsets that came with UML state charts. As has been described, we implemented a major subset of UML state charts, but also made some extensions or specialized interpretations.

One significant difference is that UML state charts seem to be static. They are designed and defined to act on a set of events in a well-structured and static manner. For example, when an event arrives that is not handled by a state, the analysis of the state machine knows that the event is to be processed by the grandparent of the state. In our case, we have to dynamically scan the state hierarchy to see if any state is interested in handling the event. This in fact allows us to add new states dynamically, and indeed to include nested clusters of “pre-wired” states.

Our implemented immediate transitions correspond closely with triggerless transitions defined in the UML state chart specification. Deferred event handling was partially implemented. The storage mechanism used to hold descriptions of what events to defer is available but are not matched with incoming events, nor are the deferred events released to an active queue when the state with the deferred event is exited.

As described above, we implemented the concept of default transitions. These are really not new as they are semantically equivalent to adding an intervening composite state to which these default transitions are attached. However, they result in much less verbose code.

The most visible features missing from our state hierarchy are a concurrent state, fork state, join state, history state, and deep history state. The first three of these objects allow states to execute in parallel. The history states are used to maintain a reference to the substate of a composite state that is interrupted during processing. Though having these capabilities was not a concern when the first iteration of the HP SmartAgent architecture was developed, we speculate that concurrent processing of states is desirable when programming agent behavior.

In principle, implementing concurrent states with our model is not difficult: at a concurrent fork state, multiple state pointers are cloned from the current state pointer, and a copy of the event distributed to each child state; at the concurrent join state, events are discarded, or deferred until all the concurrent child states have their pointers at the join state.

Finally, we are left with one open issue related to our choice of the state machine as a form of “passive” dispatcher (distributor of events or “switch”), rather than an active free running machine. (See also the discussion below about metaphor). This is the problem of when to run the Action clause. In the “switch” model, Entry and Exit code is always run on entering and leaving a state; but then the state “just sits there” waiting for an event. When an event arrives, if it does not match any transition in this State or inherited from enclosing composite state, then the action activities are run. This is related also to the problem of triggerless transitions. One idea is to have a system “clock” event that is periodically pumped into the state machine, which will ensure that the action is run (multiple times) if no transition is taken instead. The UML state chart specifications tell us that the action should be executed as soon as the entry code is processed. That action processing is then “interrupted” by the arrival of an event. Since we are single-threaded, this implementation is really not possible.

3.6 Reconnecting with the JADE Behavior Model

Up to this point, we have described and treated our Controller, Dispatcher and HSM model as distinct from the JADE behavior model, except for JADE behaviors such as ControllerBehavior and RunTimedBehavior which pump events into the dispatch tree. As a partial solution to the “switch vs. thread” conundrum, and as a step to a tighter integration, it seems quite feasible to embed an HSM into the body of a JADE behavior. The following steps are just a sketch, and will be developed further :

- We subclass Behavior, and implement Activity
- In the Behavior setup, we instantiate the desired HSM, and set it to the initial state
- We implement the eventReceived methods, and coordinate those with the Action
- We have the JADE Action (of the Behaviour) coordinate with the eventReceived, else runs the action of the current state. By coordinate, we mean first fish out events that cause transitions, otherwise, deliver to the action.
- When the outermost state exits, it sets *isDone* to true
- Since Behavior (like HSM) are activities, they can be related sub Behaviors or nested HSM's

This plan is not without some concerns:

- Other JADE Behaviors with overlapping templates could still get and consume our events and messages

- We would still have to outlaw blockingReceives

3.7 Reconnecting/Simplifying the Controller/Dispatcher/Activity and HSM models

As indicated in Figure 5, we can view the HSM, State and Transitions as specialized Controllers and Dispatchers; however, this is not how the system is currently implemented, but indicates a direction we are exploring. Both the Controller/Dispatcher/Activity model, and the HSM model are different but related ways of grouping event/message invoked activities to better control the order of Activity execution. In the Controller/Dispatcher model, groups of dispatchers (called sections) allow us to order a set of dispatchers, each with its own template to fire a set of associated activities (which themselves may be dispatchers or controllers with nested activities).

This feels like a state, with a rather weakly defined transition, or like a cascade of nested states. So one approach is to express the Controller/Dispatcher/Activity bundle as essentially the same as an Internal Transition, with Guard=True.

On the other hand, an HSM switches into different states, each of which represents a different set of active transitions; the addition of a default transition essentially adds two "dispatcher" sections to each state; so what seems more promising (as telegraphed in Figure 5), is the idea of integrating the Controller/Dispatcher into the State/Transition model. One can then think of the immediate, normal or default transitions being Dispatcher sections. These extensions to the HSM and unification with the Controller/Dispatcher are being investigated.

4 Summary, Conclusions and Future Work

Our original goal was to develop an enhanced JADE agent behavior model in order to have more precise control over robust agent behavioral issues such as timeouts, default and exception handling, and order of behavior execution. In addition, we wanted to increase the flexibility and adaptability of our system. In our previous experiments with ZEUS and JADE [Fonseca et al, 2001; Fonseca et al, 2001a; Griss et al, 2001], we found writing and validating a complex set of related behaviors too difficult. Thus we developed our model that integrates events, activities, event dispatching and hierarchical state machines.

Keys to this integration:

- 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).

Though not described in detail in this report, we have also integrated the Jess forward chaining rule system to allow more flexible recognition and dispatching of events (rule-based pattern templates), to provide flexible selection between specific state machine behaviors, and programming of behaviors that are completely rule-based.

A companion report [Letsinger 2001] describes a model in which state machines are divided into two classes. Conversation managers are state machines designed to guide an agent through a conversational protocol. They keep track of the flow of messages between the agent and other agents. The events they respond to are message events, whose encoding depends on the choice of a particular content language and ontology. The other class of state machines is used to structure the agent's problem-solving activity. These problem-solving state machines respond to events encoded in an internal format that is independent of the choice of communication language. Incoming messages have their content converted from the designated external ontology and language to the more convenient internal form. In [Katz 2001], some discussion of rule-based dispatching is provided.

As we developed and evaluated the integrated model, we used two experimental vehicles. One was an extensive set of sample state machines, driven by a table driven event sender. The other was the continuous re-implementation of portions of our CoolAgent Personal Assistant and Meeting Assistant system [Griss et al., 2001], with the addition of a table driven meeting request sender. In the former case, we were looking for exhaustive tests and corner cases; in the latter case, we wanted to add new features, improve the robustness of the meeting system and assess the benefits of state-based programming, as well as develop some higher level tools for creating and monitoring the behaviors.

Our method for inserting HSM into the system was to first carefully study the original code. In our case, the code was written using a state-like model, so adapting it for the HSM was relatively straightforward. After understanding the conversation model, we drew pictures of the state machines, decided on transitions, templates, guards, and actions. We then mapped from the diagram into code that created the various states and transitions. Interestingly, we never used a single state “action.” We ended up putting all of the action code either inside of the entry code of a state or in the action part of the transition. The reason is that you can’t really tell when the action code will be executed. As described above, it is only executed when an event arrives that the state cannot handle. If only valid events arrive, then the action code will never be executed. Clearly, this needs to be re-investigated. It is almost like the action code should only be used to indicate those exceptional conditions when no event template can be found. This is clearly not what was intended by the UML state chart designers. In addition, we have not used “exit,” it just hasn’t been needed, except turn off the timer in a timed state. Not using action or exit may be due to our lack of familiarity, but this may also be related to the “metaphor” question. Since our current approach is to treat the state machine as a “switch,” action is just not reliably triggered.

Another important aspect of the new HSM model is that it is very easy to add default handlers. One can use timed states to handle those situations where a target agent fails to respond, you can use the hierarchy to handle the exceptional times when an event arrives that was unexpected. These ultimately add to more robust conversations. What is important is that this new model allows the designer to concentrate on the “correct” flow of the conversation and to cleanly separate out exception handling.

The model that we use for interaction between dispatchers and the state processing is initiated when the agent receives a message that will begin a new “sub-conversation” (for example, the agent receives a request to do something, the agent starts a sub-conversation with another agent to deal with the request). Our model is to create a new unique conversation ID and add a new dispatcher to the main controller that looks for only those conversations. Then, the HSM starts, sends out its initial message and waits for a response (using the conversation ID). When the response arrives, it is dispatched to the HSM and the state firing continues. Once the HSM is finished, we remove the dispatcher and the agent goes back to waiting for new requests. This model is nice as it allows us to manage multiple conversations and sub-conversations simultaneously, each with its own instance of an HSM and conversation ID. Overall, the dispatcher/HSM model works very well and is an important improvement over the straight behaviour-oriented system.

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

- Dynamic nature of our HSM vs. the static nature of UML StateCharts
- Immediate transitions
- Default transitions – (relates to ordering, acts as default enclosing state)
- Ordering of transitions
- Embedded HSM and remote invocation of HSM

While we believe concurrency will be important to make it easier to implement complex StateMachines, we have not yet implemented and tested concurrent substates, with supporting fork and join nodes. In part, we did not have an immediate need, and in part this is exacerbated by the metaphor question.

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 that he used and built agents upon.

References

[Bach and Harnish, 1979] K. Bach and R.M. Harnish. *Linguistic Communication and Speech Acts*, MIT Press,

- 1979.
- [Bell and Griss, 2001] David I Bell and Martin Griss, "Flexible Conversation Management," HPL Working Paper, 2001, (unpublished).
- [Bell et al., 2000] David I. Bell, Harumi Kuno, Jon Gustafson, Anna Durante, and Louis Goldstein, A Business Communication Specification for E-Services HPL-2000-80, June, 2000.
- [Bellifemine et al., 1999] Fabio Bellifemine, Agostino Poggi and Giovanni 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)
- [BLUESTONE, 2001] HP Bluestone Application Server (HP-AS), part of HP Netaction middleware suite, <http://www.bluestone.com>
- [Booch, 1999] Grady Booch, Unified Modeling Language User Guide, Addison-Wesley, 1999
- [Chen et al., 2000] Q. Chen, M. Hsu, U. Dayal, and M. Griss, "Multi-Agent Cooperation, Dynamic Workflow and XML for E-Commerce Automation," in *Proc. Autonomous Agents 2000*, June, Barcelona.
- [Cost et al., 1998] R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield and Akram 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] R. Scott Cost, Tim Finin, Yannis Labrou, Xiaocheng Luan, Yun Peng, Ian Soboroff, James Mayfield, and Akram Boughannam, An Agent-based Infrastructure for Enterprise Integration, ASA/MA, October, 1999, p. 219-33
- [Cost et al., 2000] R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng, Using Colored Petri nets for Conversation Modeling, Issues in Agent Communication, AI Vol. 1916, May 2000, p. 178-92.
- [Cowan and Griss, 2001] Dick Cowan and Martin Griss, "Web Based Object Inspector," HPL Tech Report HPL-2001-297, Nov 2001. (*In preparation*).
- [Cowan et al, 2001] Dick Cowan, Martin Griss and Bernard Burg, "BlueJade – A service for managing software agents," HPL Tech Report HPL-2001-296, Nov 2001.
- [Espeak 2000] Hewlett-Packard - E-Speak Operation. *E-Speak Service Framework Specification, Version 1.1*, July, 2000 (unpublished, see also See <http://www.e-speak.hp.com/> and <http://www.e-speak.net>)
- [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] Steven P. Fonseca, Martin L. Griss, Reed 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] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design patterns," Addison-Wesley, 1995.
- [Greaves et al., 2000] Mark Greaves, Heather Holmback, and Jeffrey Bradshaw, "What Is a Conversation Policy?" Issues in Agent Communication, AI Vol. 1916, May 2000, p.118-31
- [Griss 2000] Martin .L. 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] Martin L. Griss, Implementing Product-Line Features By Composing Component Aspects, Proc. First International Software Product-Line Conference, Denver, CO., Aug 2000

- [Griss et al., 2001] Martin L. Griss, et al., CoolAgent: Intelligent Digital Assistants for Mobile Professionals – Phase 1 Retrospective, HPL Technical Report, Nov 2001 (*In preparation*)
- [Griss, 2001a] Martin L Griss, MatchMaker: A Structure Matcher for JADE Agent Objects, HPL Tech Report, Dec 2001. (*In preparation*)
- [Gschwind et al., 1999] Thomas Gschwind, Metin Ferudin, and Stefan Pleisch, ADK - Building Mobile Agents for Network and Systems Management from Reusable Components, ASA/MA. Los Alamitos, IEEE Computer Society Press, 1999. p. 13-21
- [Harel and Politi, 1998] David Harel and Michal Politi, Modeling Reactive Systems with Statecharts, McGraw Hill, 1998
- [Huhns 1998] M.N. Huhns and M.P. Singh, “Readings in Agents”, Morgan-Kaufman, 1998.
- [Jennings and Wooldridge, 1998] N. R. Jennings, and M. R. Wooldridge, *Agent Technology*, Springer.
- [Jennings, 2000] Nicholas R. Jennings, On Agent-Based Software Engineering, Artificial Intelligence, March, 2000, vol. 117, no. 2, p. 277-96
- [Kassem et al., 2000] Nicholas Kassem et al., Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition, Java Series, Addison-Wesley, 2000
- [Katz, 2001] Ed Katz, "A Multiple Rule Engine-Based Agent Control Architecture," HPL Technical Report HPL-2001-283, October 2001 (*Submitted to AAMAS 2002*).
- [Kendall, 1999] E.A. 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 and E. Katz, A Flexible Architecture for Personalizable Agents, HPL Tech Report, Nov 2001. (*In preparation*).
- [Maes, 1994] Pattie Maes. "[Agents that Reduce Work and Information Overload.](#)" [Communications of the ACM](#), Vol. 37, No.7, pp. 31-40, 146, ACM Press, July 1994.
- [Maes 1999] P. Maes, R.H. Guttman, and A.G. Moukas. "Agents that buy and sell." Communications of the ACM, Vol.42, No.3, March 1999, pp.81-91.
- [Moore 1998] S.A. Moore, "KQML and FLBC: Contrasting Agent Communication Languages," *Proceedings. 32nd Hawaii international conference on system sciences*, 1998.
- [Moore, 2001] Scott A. Moore, A Foundation for Flexible Automated Electronic Communication, Information Systems Research, March 2001, vol. 12, No. 1, p. 34-62
- [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/>).
- [O'Brien 1998] P.D. O'Brien and R. Nicol, FIPA: Towards a standard for intelligent agents. BT Technical Journal, 16(3), 1998. Also see the Foundation for Intelligent Physical Agents (FIPA) website, <http://www.fipa.org>
- [Odell, 2000] James Odell, H. Van Dyke Parunak and Bernhard Bauer, Extending UML for Agents, AOIS Workshop at AAAI 2000, www.auml.org. Also see <http://www.jamesodell.com/publications.html>
- [RosettaNet] RosettaNet, <http://www.rosettanel.org>
- [Sachs et al., 2000] Martin Sachs, Asit Dan, Thao Nguyen, Robert Kearney, Hidayatullah Shikh, and Daniel Dias, Executable Trading-Partner Agreements, Electronic Commerce, January, 2000
- [Sahai et al, 2001] Akhil Sahai, Vijay Machiraju, Klaus Wurster, “Monitoring and Controlling Internet based E-Services,” Second IEEE Workshop on Internet Applications'01 (WIAPP'01), July 2001.
- [Samek and Montgomery, 2000] Miro Samek and Paul Montgomery, State-Oriented Programming, Embedded Systems Engineering, August, 2000, p. 21-43.
- [Shoham, 1993] Yoav Shoham, “Agent Oriented Programming,” Journal of Artificial Intelligence , 60(1), pp. 51-92, 1993.

Appendix – Sample Code

A.1 Controller and Dispatcher

First, create the SmartAgent to Listen for messages.

```

/** A simple listener agent . */
public class ListenerAgent extends SmartAgentAdapter {

    public void setup() {
        super.setup();

        // Use default toplevel Controller object , and Provide "first" named dispatch section

        getController().addDispatcherSection("first");

        // Create a message template to match INFORM
        MessageEventTemplate mtInform = new MessageEventTemplate( MessageTemplate.MatchPerformative(ACLMessage.INFORM));

        // Create a Dispatcher listening for messages/events for this template
        Dispatcher dispatcher1 = new DispatcherAdapter(this, mtInform)
        { // Local override of isDone method to "eat message" if return true
            public boolean isDone() {
                return true;
            }
        };

        // Add a single ListenerActivity to this dispatcher
        dispatcher1.addActivity("Listener", new ListenerActivity(this, "INFORM received by 'informer'"));

        // Plant this dispatcher into "first" section
        getController().addDispatcher("first", dispatcher1);

        // Create a default "catch all" section for events not seen by previous dispatchers
        getController().addDispatcherSection("last");

        // Create a Dispatcher and activity that will catch and print all messages/events
        Dispatcher defaultDispatcher = new DispatcherAdapter(this);
        defaultDispatcher.addActivity("Default", new JournalActivity(this, "Default catch-all"));

        // Add this dispatcher to "last" section
        getController().addDispatcher("last", defaultDispatcher);

        // Finally, set up a JADE behaviour (like a thread) to run the dispatcher
        // by default, prints infor on each message/timeout/event

        ControllerBehaviour db = new ControllerBehaviour(this, getController(),
            "Main Dispatch", // title for inspector
            500,           // blockingTime to "sleep" between checking
            10000);        // generate periodic10 sec timeout
    }
}

```

Then, define the ListenerActivity

```
public class ListenerActivity extends ActivityAdapter {

    protected int messageCounter=0;
    protected int timeoutCounter=0;
    protected int exceptionCounter=0;
    SmartAgentAdapter agent;

    /* Constructor for Listener activity. */

    public ListenerActivity(SmartAgentAdapter anAgent, String label) {
        super(anAgent,label);
        agent=anAgent;
    }

    /** Called by dispatcher when a message received to print received message. */

    public void messageReceived(MessageEvent anEvent) {
        ACLMessage request = (ACLMessage)anEvent.getContents();
        String perfName = request.getPerformative(request.getPerformative());
        agent.printlnInfo("ListenerActivity got message: " + messageCounter++ + " performative=" + perfName);
    }

    /** Called by dispatcher when a timeout is processed. */
    public void timerReceived(TimerEvent anEvent) {
        // ignore timeouts
    }

    /** Called by dispatcher when an exception is encountered. */
    public void exceptionReceived(ExceptionEvent anEvent) {
        //Ignore
    }
}
```

A. 2 Simple HSM example

This example is drawn from a series of HSM test machines in the HSM package. This corresponds to the very simple machine in Figure 13.

```
HSM simpleStateMachine(SmartAgent anAgent) {

    HSM anHSM = new HSM(anAgent, "simple nesting, accept all");

    // Build 2 composite states, s1 and s2
    CompositeState s1 = new CompositeState("S1", anHSM);
    CompositeState s2 = new CompositeState("S2", anHSM);

    // Build 2 states, nesting inside s1
    State s11 = new State("s11", s1);
    State s12 = new State("s12", s1);

    // Build 2 states nesting inside s2
    State s21 = new State("s21", s2);
    State s22 = new State("s22", s2);

    // Create 4 transitions, matching anything

    Transition.add(s1, s21, "t121", new StringEventTemplate("inform"));
    Transition.add(s2, s1, "t21", new StringEventTemplate("request"));
    Transition.add(s21, s12, "t2112", new StringEventTemplate("*"));
    Transition.add(s12, s21, "t1221", new StringEventTemplate("*"));

    // Announce the initial states to the HSM and to nested states
    s1.setInitialState(s12);
    s2.setInitialState(s22);

    anHSM.setInitialState(s1);

    // This HSM can now be run as an Activity from a
    // Dispatcher, or RunTimedBehaviour
    return anHSM;
}
```

