A Hierarchical State Machine using JADE Behaviours with Animation Visualization

Robert Kessler
School of Computing
University of Utah
Salt Lake City, UT 84112
kessler@cs.utah.edu

Martin Griss
Computer Science
University of California, Santa Cruz
1156 High Street
Santa Cruz, CA 95064
griss@soe.ucsc.edu

Brian Remick
School of Computing
University of Utah
Salt Lake City, UT 84112
remick@cs.utah.edu

Ryan Delucchi
Associate of Martin Griss
Independent Contributor
350 Bean Creek Rd.
Scotts Valley, CA 95066
scate@metaforte.com

Abstract

In previous papers [5], [6] we describe an event-based UML state machine execution engine as well as a Microsoft Visio-based visual editor that has many benefits over the built-in JADE FSM (Finite State Machine) behaviour model. These benefits are an integration of message handling, timeouts, exceptions and the use of hierarchy and common action factoring to reduce the complexity of behaviour construction, while ultimately improving system scalability. In that work we engineered a consistent, largely self-contained component, with several key properties, but relatively incomplete integration with other JADE behaviour mechanisms.

Since then, we have developed a new Hierarchical State Machine (HSM) model that shares many of the same goals of reducing the complexity of developing protocols and behaviours, but is a vastly improved companion to JADE. It leverages the newer JADE Composite Behaviour and offers substantial improvements over the JADE FSM. Furthermore, we describe a new Java based graphical tool that facilitates designing, visualizing, and reverse engineering of HSMs. Finally, we describe one of several systems that have been built using HSM.

1. Introduction

In previous work [4], [7], and [6], we built several different multi-agent systems using ZEUS [9] and JADE [2]. We found that once you began to have agent-to-agent conversations and protocols that were more than simple message exchanges, a state machine model was well-suited for managing the interaction. As have others (such as AUML [10]) we began examining state machine models and looked toward the UML State Machine. The UML state machine [3] evolved from the hierarchical statechart work of Harel [8]. A UML state machine defines states and transitions that connect states. Hierarchy appears because states and machines can be built within other states and machines. This hierarchal processing of elements makes it easier to describe complex conditions and transitions. As we built more comprehensive agents, we found that we needed ways to handle common default conditions and exceptional situations. Hierarchy was a perfect solution as it permits factoring common code, allowing more natural and concise expression. UML state machines are event driven; actions come in the form of events that are presented to the state machine, typically driving transitions.

The UML state machine is very powerful, allowing the implementation of both Mealy and Moore styles of state machine. That is, activities of the state machine can be expressed as an action within the state itself or expressed as an action on the state transition. One can also mix and match this style depending on the problem at hand, leading to the concise, understandable and maintainable machines.

Given that the behaviour models already in JADE were either too simple (the FSMBehaviour model essentially only permits flat state machines) or not easy to map into, we created our original Hierarchical State Machine model with Microsoft Visio as the drawing tool. This system gave us all of the benefits of hierarchy and common action factoring, but failed to integrate sufficiently with JADE. It forced the agent author to mix traditional JADE behaviours with straight Java HSM code which often caused thorny coordination problems.

Recognizing the FSMBehaviour benefit of allowing any behaviour to be a state, we embarked on a project to create a new version of HSM that kept the full power of

AAMAS '04,…..

hierarchy yet allowed any behaviour to be a state. In addition, we created an HSM editing tool and a visualization engine that lets you view an animation of an HSM during its execution and also permits reverse engineering of a hand-written HSM into a graphical HSM realization.

2. Behaviours as States

The poor integration of our original HSM with JADE led us to investigate the possibility of making any HSM state be a first class JADE behaviour. The benefit is enormous, as it permits existing behaviours to be "linked" together in a state machine execution framework. In addition, users would not need to learn a new mechanism when building state machines; they merely create behaviours. Incidentally, this change required that we redesign the entire HSM.

In the JADE model, an agent's functionality is implemented as a single-threaded set of cooperating behaviours, which voluntarily give up control to other behaviours. You could almost view this as a "time slicing" model, where after a behaviour's action code is executed for a short while, it gives up control allowing some other behaviour to execute. Eventually control returns and it is permitted to execute for another short cycle. A behaviour completes when its *done* method returns true. JADE provides several behaviour classes including "ComplexBehaviour" whose children include behaviour execution models such as "SequentialBehaviour", "ParallelBehaviour", and "FSMBehaviour".

We used FSMBehaviour as our starting point and created HSMBehaviour which inherits from ComplexBehaviour. HSMBehaviour manages a set of nested HSMBehaviours and other Behaviours (states) for its hierarchical state machine. It also maintains the list of transitions and manages which state is the "current" state. Finally the HSMBehaviour is responsible for handling the computation necessary for transitions firing.

Each JADE behaviour has the concept of *onStart* and *onEnd*. These are methods that are called before and after the main action code in the behaviour is invoked. We mapped the UML concepts of "entry" and "exit" into these methods. Entry and exit codes are guaranteed to be executed in an appropriately hierarchical fashion no matter what kind of transition is taken. We will discuss this more later.

One difficulty in designing the HSMBehaviour was to follow the state chart tenet that "as soon as a transition is able to fire, it does." Conceptually this means that a state is in the middle of executing some action code and a message arrives that triggers a transition, the execution is terminated and the transition is taken. In a single-threaded environment, it is not possible to fully implement this model. Our solution was to utilize the "time-slicing" nature of behaviours and integrate transition checking and firing around the call to action. Basically, we check work through the following steps:

1) See if any transition can fire and if it can, exit the behaviour and move to the target behaviour.
2) If no such transition is fired, invoke the *action* method.
3) Again, see if any transition can fire and if it can, exit the behaviour and move to the target.
4) Otherwise, leave and schedule the behaviour for execution in the next time slice.

This model works well given the limits of a single-threaded system and allows the HSM to fire the transition "nearly" as soon is it is ready. In addition, we can fire "immediate" transitions as soon as you enter a state. An immediate transition is one that can fire without an event; it is simply tested to see if it should fire. Thus, one can make several state changes even before any event arrives.

3. HSM Events

As mentioned above, UML state machines utilize events to drive the transitions. Another design decision was to determine what kinds of events were needed and how to handle the queuing of these events. The original HSM created and managed its own queue. *Note - our original HSM could run stand-alone and did not need JADE at all.* We realized that most of the events that the state machine handles are messages and that JADE already provided a very nice queuing facility in its agent message queue. So, we decided to adopt the standard message queue as our HSM event queue. That means that when the HSMBehaviour is checking to see if any transitions can fire, the JADE *receive* method is called to retrieve the top message in the queue. This message is then passed to the transitions to see if any can fire.

The decision to use the JADE message queue for HSM events left two issues to be solved. The first was how to create other kinds of events, such as SUCCESS, FAILURE, or TIMEOUT. Our solution was to create a new class, HSMEvent, which inherits from ACLMessage. An HSMEvent uses "unknown" for its performative and then sets the language to "HSMEvent" and the ontology to be the event type (e.g. TIMEOUT). When the user must distinguish between different subtypes within a particular event type (such as a five-second and a fifteen-second timeout), an additional identification is stored in the conversation ID field. This is somewhat of a "hack" solution, but works reasonably well. Unfortunately, this requires the state machine author to be cognizant that the event argument can either be a regular ACLMessage or it may be an HSMEvent object when writing trigger and action code in a transition.

The second problem with using the traditional message queue has to do with the integration of behaviours as states. Since the regular *receive* method is used to get the next event from the queue, the state machine author is free to call *receive* inside of their behaviour action code. So, the author can effectively manipulate the queue while inside of the HSM without the HSM having the ability to fire appro-

priate transitions. Suppose that you are in behaviour s0 and you are waiting to transition to s1 when an inform message arrives. In the action code you call *receive* and then process the inform message directly. Unless you receive another inform message, the transition from s0 to s1 will never take place. So, the downside is that you could end up with your state machine not executing as you expected. However, the upside is that you have the ability to do any kind of manipulation inside of your action code that you would like and thus you have the power to effect very interesting state operations. For example, the action code could receive a failure message, process it, and post an inform message that would effectively force the transition to s1 when the action exited. So, as with many things, this kind of power must be used with caution, but it can allow the user to create very flexible state machine models.

4. HSM Transitions and Hierarchy

We introduced a new class, HSMTransition, which defines a transition. In the pure UML model, transitions include a source and destination state, a trigger, guard, and action. The trigger is an expression that is evaluated to indicate whether or not a transition should fire based on the current "event." The guard is another function that is evaluated to provide a secondary guard to decide again if the transition should fire (the guard does not utilize the event). Triggers and guards are often very declarative. In our solution, we chose a procedural model and collapsed both functions into a single *trigger* function. Action is code that is invoked as the transition is being taken and in the HSM case is a method of HSMTransition.

The execution model works as follows: the *trigger* method is initially invoked with a null event. This effectively asks whether the transition can fire without any event (i.e. an immediate transition). When an event arrives, the HSMBehaviour invokes *trigger* with the event. The *trigger* method then decides if the transition should fire. If the *trigger* returns true, then *action* is called and the transition to the target state is then performed.

Hierarchy is created in an HSM by simply adding an HSMBehaviour state to an existing HSMBehaviour. This establishes the appropriate hierarchical relationship. Transitions in HSMBehaviour may have a source or destination that is *any* state or HSMBehaviour within the entire hierarchy. This allows you to transition across HSM boundaries. When you do transition across boundaries, the stack of HSM invocations, leading to the source state, must be "unwound" until we reach the most direct common parent of the source and target states. As we do this, we must call *onEnd* for each behaviour along the way. Then, we must "drill-down" from the common parent to the target state, invoking *onStart* at each level.

The other mechanism that involves hierarchy is how the HSMBehaviour searches for a valid transition. It first looks at the transitions in the current state to see if any of their *trigger*s are true. If not, it checks the parent HSM to see if it has any transitions that are true. This continues until the root HSM is examined. If no transition can fire, the event is removed. The hierarchical examination of transitions allows you to easily create transitions that are defaults for all of the enclosing states. This is to the key to making reliable and robust state machines with no duplication of code in the states. State machines can be designed so that the interior states handle the "normal" case and the hierarchy handles exceptional conditions.

5. JADE's FSM Model

JADE's FSM model works as follows: the FSMBehaviour maintains a mapping from a state name to a Behaviour object. A second table maps a state name to a list of target states based upon an integer event number. A default transition may be registered to transition to another state in the event that there is no matching event number. One behaviour is registered as the initial state and one or more are registered as the final state. The basic execution model is implemented such that the current behaviour executes until it is done (i.e. the *done* method returns true). Upon completion, the *onEnd* method returns an event number as its value. The source state is indexed and then the event number is accessed to find the target behaviour. The target is then made current and the process continues.

Hierarchy is technically possible in the FSM model, since any behaviour (including an FSMBehaviour) can be placed as a state in a parent FSMBehaviour. However, that nested behaviour must completely finish execution before transitioning in the parent state machine. The sophisticated mechanism for handling hierarchical default values and *onEnd/onStart* processing in HSMBehaviour is not present in FSMBehaviour.

As we stated previously, the beauty of the FSM model is that any behaviour can be a state. That is literally **any** behaviour - not just a OneShotBehaviour, but ComplexBehaviours such as SequentialBehaviour, ParallelBehavour, and others. The only change that a user must make to a behaviour is to add it to the state machine and have the *onEnd* method return the correct event number.

6. Contrasting a State Machine in FSM and HSM

Using our HSMEditor tool based on the JGraph system [1], we have created an HSM with several nested state machines that is illustrated in Figure 5. On the left, the red state machine loops around with several states transitioning on assorted performatives. On the right, the blue state machine looks similar. What is interesting is the interaction between the two. When an "accept-proposal" message arrives, we switch between the red and the blue. No matter what state we are in, that message arrival forces this transition. Also, whenever a "failure" message arrives while in the red state machine or a "disconfirm" message appears

while in the blue states, the state machines transition to the final done state. The transitions in the middle (t5 and t15) allow movement from the inside of red to blue and back again depending on the appropriate message pattern.

The state chart model says that a hierarchical state machine can be translated into a flat machine. In order to make this state machine work in FSM, you would have to essentially go through that process. For example, you would have to take all of the hierarchical transitions and copy them into the child states. For example, the t6 transition from the red to the blue would be copied into all of the red states. That is, you would have to register a new transition from each child state, and then in each state you would search for the place that messages are received and add a new case looking for the "accept-proposal" performative and returning the new event number. Figure 6 shows a flat version of the same state machine as one would realize using FSMBehaviour. As you can see, the duplication of all of those hierarchical transitions makes the graph almost unreadable. We did take two liberties with this example. We assume that the *onStart* and *onEnd* code for the red and blue states would be duplicated inside each state's corresponding methods. We also assume that any "action" code that is specified on a transition is also duplicated inside of a state's action method (recall that this is because FSM transitions merely change state, all computation must occur within the behaviour). This duplication of code is required to have the same semantics that one receives by being able to encapsulate functionality into a state higher up in the hierarchy. From a software engineering view, this duplication of code is naturally quite problematic as the state machine evolves. One could possibly save some of this duplication by changing the structure so every transition on the left went through the red state before transitioning to either the blue state or the done state. A similar change would have to be required on the blue side. Effecting this change would require that relevant information be passed into the red state to indicate which transition to fire after the appropriate *onStart/onEnd* code was executed (i.e. it would have to determine if it should transition to blue or done). Another trick would be to clone the red state (again duplicating code) and then set one to transition to the blue and the other to transition to the done. As you can see the number of transitions grows and the number of states may grow as well. As you might imagine, the transitions that are fundamentally at a higher level in the hierarchy results in an FSM that increases in size and complexity and is thus the fundamental *scalability* issue within the FSM.

7. Graphical Creation of an HSM

When working with state machines that are more complicated than those with just a few states, you realize that you need a diagram of the states and transitions to help you design and understand the state machine. We realized this while working on our original HSM code and thus built a set of templates on top of the Microsoft Visio drawing tool. Our tool allowed you to draw diagrams and then the click of a button would generate Java code that would construct the state machine associated with the diagram that you designed.

Visio was a fine tool for doing this kind of work but was at times counter-intuitive. Doing small customizations to the diagrams by editing shape sheets or writing tricky Visual Basic code for the code generation was often difficult. In addition, since our plan was to release the HSM to JADE, many in the JADE community did not have access to the Visio software. So, we explored other options and came upon the JGraph [1] open-source graphing library. JGraph was developed to allow users to build editing tools with the concepts of shapes and wires linking the shapes together. We embarked on the task of creating a JGraph-based editor that would permit the development of HSM diagrams.

The HSMEditor application provides a simple method of creating the high-level control flow of an HSM. States and transitions can be connected and organized hierarchically much like the previous Visio-based implementation. Instead of using a code-generation approach, however, each node in the HSM references a class by name. Classes that derive from the Behaviour base class are referenced by each state, and classes that derive from the HSMTransition class are referenced by each transition. Parameters can also be stored with each graphical state so that the same user class can be used in different locations in the HSM by varying the parameters. (Parameters are passed to the constructor of the user class at runtime). This method avoids the problem of overwriting auto-generated code when updating the HSM that existed in the previous model. Since the HSM just contains named references to classes, the two remain independent until execution time.

In general, user code can be referenced only when necessary. When not specified, the HSM inherits the default behaviour of the HSMBehaviour class. HSMEditor also provides support for some basic HSM primitives that don't require user code to be referenced at all. For instance, the HSMPeformativeTransition class triggers on a specific performative that is set within the editor itself and persisted with the HSM to disk.

JGraph provides the core rendering and editing capability for the HSMEditor application. JGraph works with an abstract graph interface in such a way that the basic editing functionality (i.e. moving and resizing graph nodes, connecting graph edges, etc.) is implemented with no knowledge of the actual graph structure underneath. We implemented this graph interface for HSMBehaviour and were able to take advantage of the rendering customizations in both the editor and the visualization (see below).

State machines created in HSMEditor are persisted to an XML file format that describes the organization of the machine as well as references to user classes, layout information, etc. This XML file can then be loaded into a JADE

agent, which instantiates the appropriate user-defined Behaviour classes referenced by the HSM.

8. Visualization of an Executing HSM

Building on the HSMEditor, we also created a visualization tool that lets you watch the execution of an HSM. This visualization displays state machines that were designed with HSMEditor as well as those built programmatically. (Because programmatic HSMs lack the layout information that is saved in HSMEditor's XML format, a simple tabular layout is automatically generated for these HSMs). The visualization highlights the current state(s) and transitions as they fire. Figures 1-3 provide an example of the visualization.

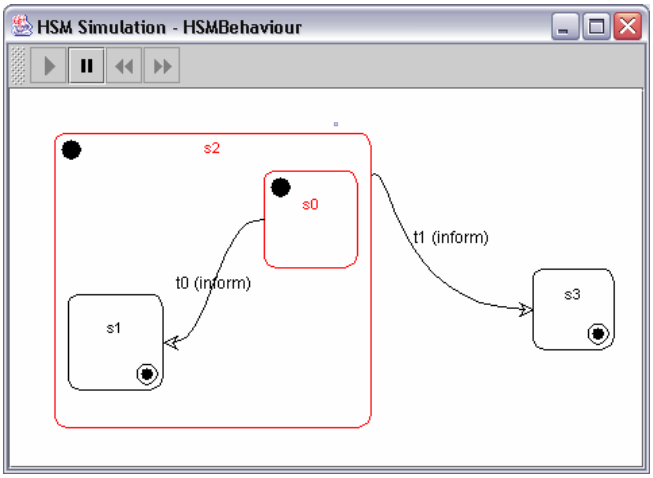

Figure 1. Visualization of a simple HSM. States s0 and s2 are active.

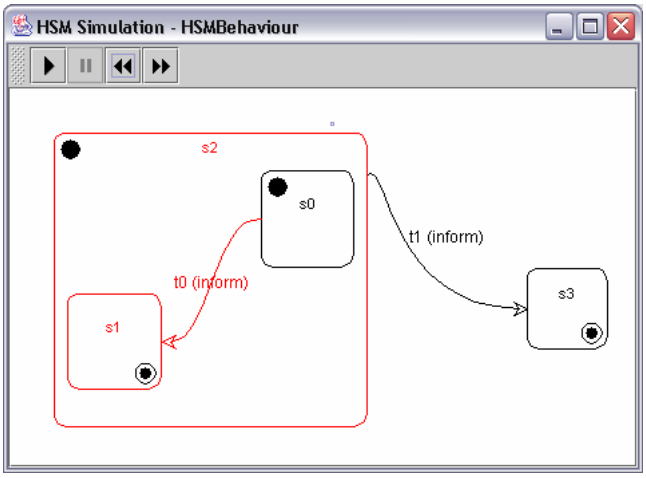

Figure 2. Following the first INFORM message. States s2 and s1 are active, and transition t0 has just been fired. The machine is paused and events are queued to be processed.

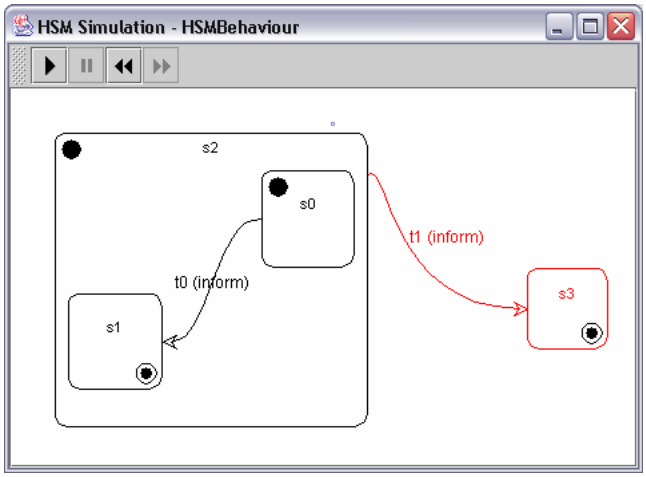

Figure 3. Following the second INFORM message. State s3 is active, and transition t1 has just been fired.

The visualization works much like a debugger; execution of an HSM can be paused at any time. Instead of blocking the agent from running, however, only the visualization is stopped. The behaviour itself continues to receive and process messages. Incoming events are placed on a queue to be processed when visualization is resumed. Programmers can then single-step (forward or reverse) through the received events. This is especially useful for machines that transition through a number of states very quickly, giving the developer the opportunity to trace the exact path through the machine visually to verify that is was correct.

9. Reverse Engineering

The JGraph-based framework that is used in both the visualization application as well as HSMEditor itself is able to generically traverse any HSM structure built with the HSMBehaviour class. Thus, it is possible to reverse engineer a programmatically-constructed machine into an editable XML-based HSM that is completely functional within HSMEditor.

Since each referenced class within an editor-based machine must be either a class derived from HSMBehaviour (for states) or from HSMTransition (for transitions), every state and transition in a programmatic HSM can likewise be referenced from HSMEditor. This allows existing HSMs that have been previously hand constructed and even compiled to be converted into visual form for editing and enhancement. The resulting machine can then be loaded directly into JADE (as described previously) and will function just like it did in programmatic form.

10. Example Application

Min Yin and Ryan Delucchi were both early adopters of the HSM JADE extension. Min Yin has leveraged the HSM within two intelligent agents: A Personal Location agent and a Personal Travel agent. Ryan Delucchi, on the other hand, used the HSM as a foundation for his intelli-

gent Meeting Arranger agent. Figure 4 illustrates the structure of the Meeting Broker HSM. The Meeting Arranger was actually implemented as a stand-alone service agent. Once an outside agent sends a REQUEST to arrange a meeting for a set of potential participants, an instance of the meeting broker HSM is created and activated.

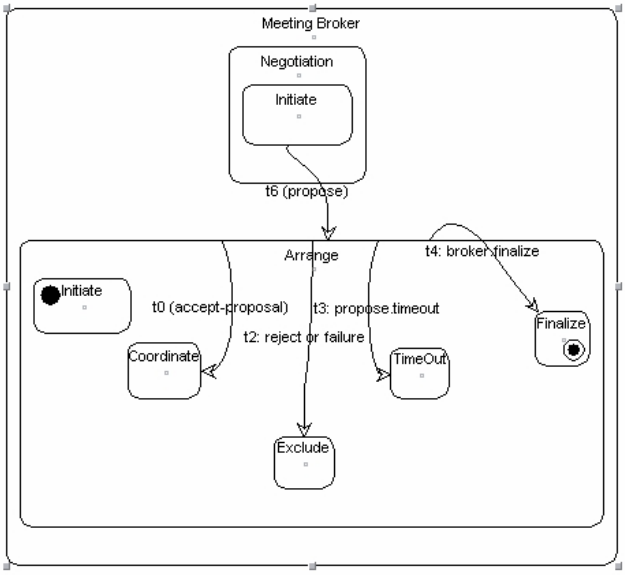

Figure 4. Meeting Broker HSM.

As one might expect, multiple instances of the Meeting Broker HSM can coexist independently (thus allowing multiple meetings to be "arranged" asynchronously as well as in-sequence). Once a broker for a particular request is created, the HSM begins with the initiate behaviour, which is nested within the Negotiation HSM. This behaviour accepts the request and sends a PROPOSE to all potential participants of the meeting (as dictated by the initial REQUEST message).

This PROPOSE message is also sent to the originator itself, where a unique conversation id ensures that the message makes its way to the appropriate Meeting Broker HSM instance. This propose causes the Arrange HSM and consequentially the inner Initiate sub-behaviour to become active.

At this point, the Initiate sub-behaviour waits for

- ACCEPT_PROPOSAL / REJECT_PROPOSAL messages from the potential participants, or
- HSMEvent.TIMEOUT for the "propose.timeout" timer.

When an ACCEPT_PROPOSAL is received, a transition activates the Coordinate state and that participant is coordinated with other meeting participants that have also sent an ACCEPT_PROPOSAL.

When a REJECT_PROPOSAL is received, a transition activates the Exclude state and that participant is excluded from the meeting. If the active-time duration of the Initiate, Coordinate, or Exclude states exceeds that of the Timer interval, the Timeout transition will cause the TimeOut state to become active (which results in the entire meeting arrangement session to end with all the meeting participants that have accepted the proposal).

When all participants have responded or a timeout has occurred: a CONFIRM is sent to all the actual participants of the meeting (if a meeting can be scheduled to accommodate enough of the potential participants). Finally, a "broker.finalize" message is posted to the event queue and the meeting broker session terminates.The model of the HSM is directly reflected in the Java class library implementation as shown below:

- Arrange class
  - Initiate inner-class
  - Coordinate inner-class
  - Exclude inner-class
  - Timeout inner-class
  - Finalize inner-class

This is somewhat of a “different” state machine model as there is only one real state-to-state transition (t6). The others go from the HSMBehaviour Arrange to the substates. The beauty of this approach is that the arrival of each appropriate event selects the appropriate state, which does its job and then waits for the next event. Thus, this state machine is similar to a *case* or *select* control construct in traditional programming languages.

11. Conclusion

This new version of our HSM system is a vast improvement over the FSM model in JADE and our previous version. It is completely integrated into JADE such that JADE behaviours are now states. It allows you to easily construct very complicated and very robust state machines. Since hierarchy really comes into play when you have complicated state machines, building a visualization tool was essential. Our HSMEditor aids you in drawing hierarchical state machines and then generates an XML realization of the machine that can be loaded and run inside of any JADE agent. The visualization engine lets you watch the execution of any HSM (including HSMs created without the HSMEditor) and pause, rewind, and replay the execution. Again, as state agent complexity increases, this tool helps you to track down problems in your design. Lastly, we added the ability to reverse-engineer an HSM that was created in Java code back into a state machine that is XML loadable. This permits a user to start with a very simple machine and then as it evolves decide to switch over to a graphical version for editing and revision.

There are still several enhancements that we have planned. One is to build on the ParallelBehaviour to allow concurrent state machines. Another plan is to write a translator that can take in the XML specification of an HSM and generate a Java code realization. Lastly we are currently working with the JADE team to release the HSMBehaviour

for use by all JADE users. We are currently investigating whether we could create an FSMBehaviour compatibility mode that would allow existing agents developed with FSMBehaviour to directly use HSMBehaviour and thus permit the replacement of FSMBehaviour with HSMBehaviour. If that is not feasible, we will release HSMBehaviour as an add-on package to JADE.

Acknowledgement We wish to thank Min Yin for her role as a beta tester and incorporation of HSMBehaviour in her Personal Location and Travel agents.

12. References

[1] G Alder, Design and Implementation of the JGraph Swing Component. Also see http://www.jgraph.com

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

[3] G. Booch, J. Rumbaug and I. Jacobson, The Unified Model-ing Language User Guide, Addison-Wesley, 1999

[4] S. Fonseca, M. Griss, R. Letsinger, An Agent Mediated E-Commerce Environment for the Mobile Shopper, Hewlett-Packard Laboratories, Technical Report, HPL-2001-157

[5] M. Griss, S. Fonseca, D. Cowan, and R. Kessler Using UML State Machines Models for More Precise and Flexible JADE Agent Behaviors HPL 2002-298(R), July 2002 (AAMAS AOSE workshop, Bologna, Italy, July 2002)

[6] M. Griss, S. Fonseca, D. Cowan, and R. Kessler, SmartAgent: Extending the JADE agent behavior model HPL 2002-18, Jan 2002 (AOSE workshop, SCI SEMAS Workshop, Orlando Florida)

[7] M. Griss and R. Letsinger, Games at Work - Agent-Mediated E-Commerce Simulation, Workshop, Autono-mous Agents 2000, Barcelona, Spain, June 2000. (Also HP Laboratories Technical Report, HPL-2000-52.)

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

[9] H. Nwana, D. Nduma, L. Lee, and 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/).

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

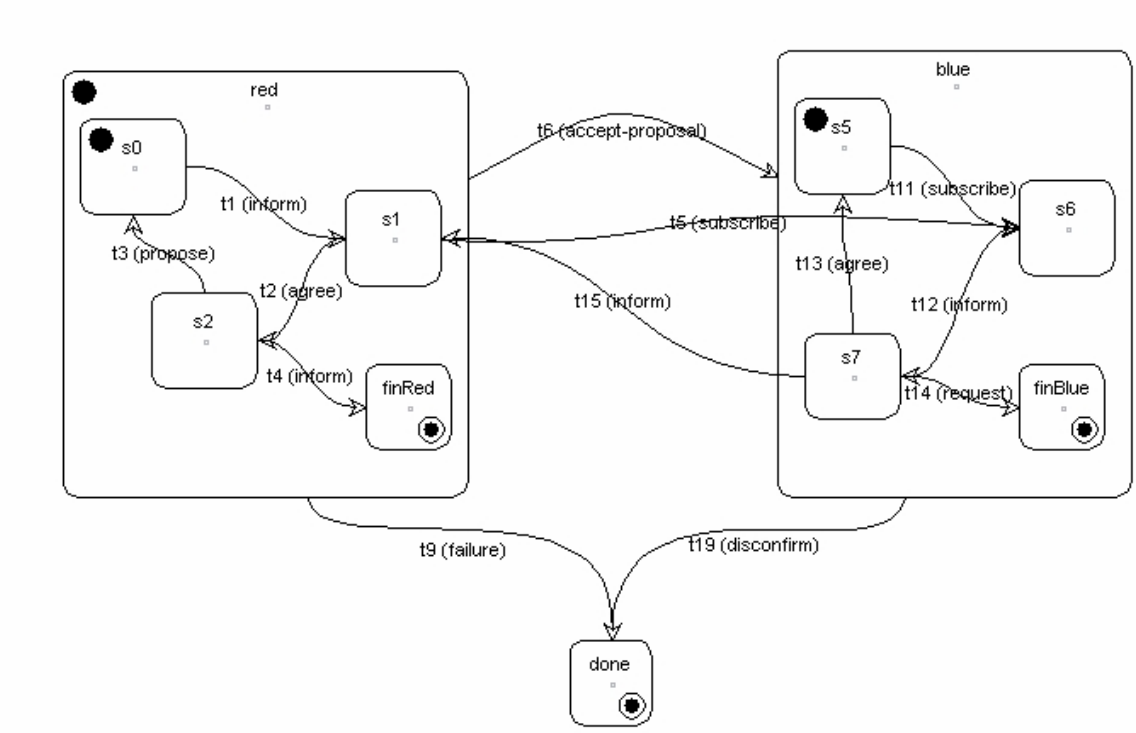

Figure 5. A Hierarchical State Machine created with the HSMEditor.

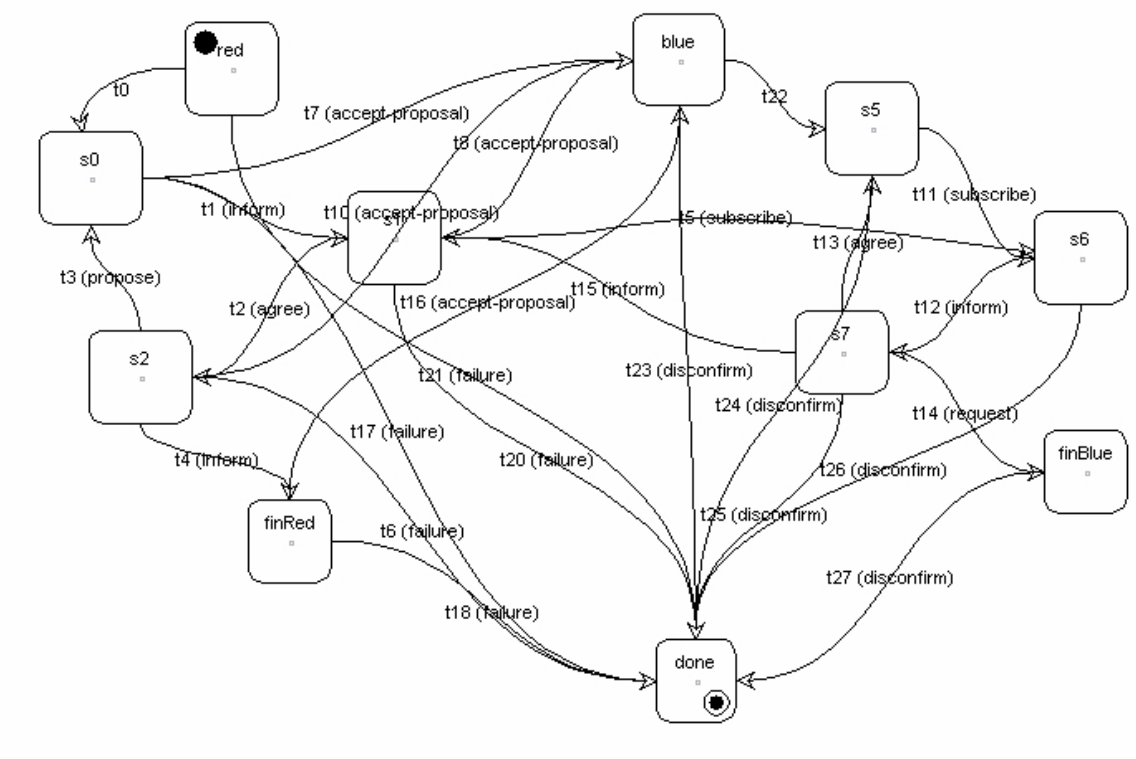

Figure 6. An FSM version of the HSM created in Figure 5.