

# A Community of Expert-System Agents<sup>1</sup>

Tarmel Kennion  
NASA ACE and  
Department of Computer Science  
North Carolina Agricultural & Technical State University  
Greensboro, NC 27411  
[tkennion@ncat.edu](mailto:tkennion@ncat.edu)  
Dr. Albert Esterline, advisor

## *Abstract*

*This paper reports on the undergraduate research being conducted as part of the NASA – sponsored project “Formal Foundations of Agents.” This project concerns the development of a multi-agent expert system for information management. We outline the system’s original structure, discuss the current implementation using the Java Expert System Shell (Jess), and develop several extension possibilities for the future.*

## **1. Introduction**

As part of the NASA sponsored project “Formal Foundations of Agents,” students at North Carolina Agricultural & Technical State University are developing a multi-agent expert system. In 1996 Mr. Torri Murphy began development for his master’s project “Formalizing an Object-Oriented Agent Information Management System” [Mu97]. His system is the basis for the system currently being developed by Mr. Keith Ellis in his master’s project “An Agent Information Management System Using the Java Expert System Shell (Jess)”. This system provides a prototype for a schema-based model of reasoning. It is implemented in Jess, which is written in Java [DD98]. It is intended that the distributed nature of the agents in the Lights-Out Ground System (LOGOS) [LO97], being developed in the Automation Technology Section of NASA Goddard Space Flight Center (GSFC), be implemented in Jess. Our work involves extending this prototype system. We here report on extensions currently being implemented.

Section 2 of this paper introduces background on the system: work at GSFC, schema-based reasoning, and the structure of Mr. Murphy’s system. Section 3 reports on our adaptations of the Jess system to allow multiple agents executing on several threads. Section 4 concludes the paper with a summary of the current system, an evaluation, and possible future work.

## **2. System Structure**

### **2.1. Background**

Our prototype follows the multi-agent system representation of the Agent-based Flight Operations Associate (AFLOAT) system developed at GSFC. In AFLOAT [Ma96] each agent can be viewed as a miniature expert system in its limited domain. We also follow

---

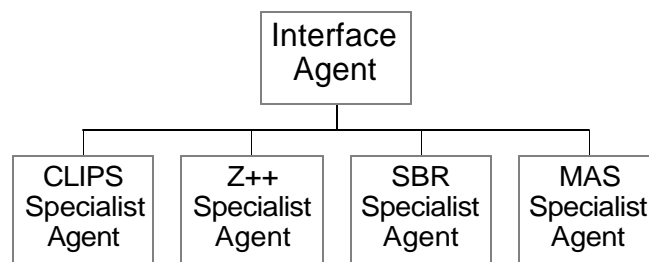
<sup>1</sup> This research was supported by grant NAG 5-4102, “Formal Foundations of Agents,” from NASA Goddard Space Flight Center.

the schema-based reasoning (SBR) approach [Tu94], which uses declarative knowledge structures to store an agent's problem-solving knowledge. Originally implemented in the C-Language Integrated Production System (CLIPS) [Gi93], our current prototype is implemented mainly in Jess, a clone of CLIPS. The system's specification is formalized in the Z++ object-oriented specification language [La95], following Mr. Murphy's project. Its concurrency and communications issues in particular are specified in the Real Time Logic (RTL) that is an integral part of Z++. It is a prototype system of agents for information management and report generation. This system interacts with the user through the Jess interface in order to determine the user's expertise and the domain(s) in which the user is interested. It then generates a list of references to one of the following domains: Z++, CLIPS, Multi-agents systems, and Schema-Based Reasoning

## 2.2. Schema -Based Reasoning

In Schema-Based Reasoning (SBR), declarative knowledge structures called schemas are used to store an agent's problem-solving knowledge. Procedural, contextual, and strategy schemas define the categories of schemas. Procedural schemas (p-schemas) are similar to hierarchical plans or scripts. They are interpreted by an agent's reasoning process as a list of commands or steps to be performed. Thus, a p-schema could be viewed as a sort of algorithm. Contextual schemas (c-schemas) act as guides in different situations. They determine which p-schema to use for each situation. Strategic schemas, or s-schemas, are domain-independent contextual schemas and hold information that defines the agent's problem solving strategies. In SBR the user finds an active goal using the current contextual and strategic schemas. A p-schema is applied to take action and achieve the goal. This process is repeated until all goals are satisfied.

In our system there is a specialist agent designated for each of the four domains. For each specialist agent, we specify one or two c-schemas and provide several p-schemas. We do not use s-schemas. The c-schemas determine how the user is questioned about topics of interest. The questioning is done by p-schemas that navigate through a hierarchy of topics. A specialist agent records each topic noted by the user and broadcasts it to the other specialist agents and the interface agent. Since there is overlap in the domains of the specialist agents, each such agent inspects each topic recorded by the other agents. If the topic falls within its domain, then it responds appropriately. The interface agent receives all the topics noted by the specialist agents and compiles a document that gives page references to sources in the literature for each topic. The figure below shows a domain representation of our system.



## 3. Jess Implementation

### 3.1. Concurrency in Jess

In the earlier, CLIPS system, communication and concurrency were simulated by hand. Each time an agent was to broadcast a topic, it output this topic on its window and paused. Similarly, each time an agent might receive a topic, it paused. Topics were copied from the windows where they were output to windows waiting for input. Then all agents resumed. This was useable but was not the desired implementation. In fully automating the system, we take advantage of Java's concurrency and communications capabilities. We accomplish this by implementing Mr. Murphy's system in Jess. Concurrency in Jess is accomplished through the use of Java threads. We use multiple threads of control within a single program running on a single platform. Java threads are single logical executable paths that allow for multiple flow of execution within a process. Threads are given a priority and then time-sliced. That is, each thread is given a limited amount of time called a quantum to execute on the processor. When that time expires, the thread is made to wait while all other threads of equal priority get their chances to use their quantum. This continues in a round robin fashion.

It should be noted that different operating systems and thread packages implement different scheduling policies. Jess is intended to be platform independent. The Jess implementation on Windows 95/NT uses the underlying Win32 thread scheduler, which is time-sliced. On Solaris and other UNIX platforms, the 1.0.2 JDK uses a custom package developed by Sun called Green Threads, which is not time-sliced. In the future, the Solaris JDK will likely use the Solaris thread package (which is not time-sliced).

### 3.2. Concurrency using pipes

Our implementation for multithreading uses pipes to establish communication between threads. There are three steps to adapting the Jess system to our needs. In the first step, we extend the `java.applet.Applet` class to implement the `Runnable` interface:

```
public class jessThreads extends java.applet.Applet
    implements Runnable {...}
```

This class defines the parent thread, which is the hub for the specialist agents: CLIPS, MAS, SBR, and Z++. The child thread represents the interface agent. Class `jessThreads` calls the `run()` method. An instance of the class can then be allocated, passed as an argument when creating a `Thread`, and started. To create a thread, we have the option to subclass `Thread` so that it defines its own `run()` method or to pass a `Runnable` object to the `Thread` constructor. The interface specifies the `run()` method that is required for use with the `Thread` class. By implementing the interface `Runnable`, we declare our intention to run on a separate thread. This class, `jessThreads`, is the class of the client thread, which is the hub for the CLIPS, MAS, SBR, and Z++ specialist agents. It is also the class of the server thread, in which the interface agent runs.

Initially, Jess used thread priority to manage its interface:

```
Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
```

On most single processor machines, threads are usually mutually exclusive, so thread priority is important. `Thread.currentThread().getPriority()` will get the priority of the top-level application thread, and it can set successive priorities which are typically lower. Setting the priority to `MIN_PRIORITY` or “low” ensures that the user and other I/O processing are not conflicted with. This type of thread usage is similar to background processing.

We, however, use piped communication to govern how threads share the processor. The second step, then, in adapting the Jess system to our needs involves the methods used in thread processing. The `init()` method initializes the screen and defines piping by way of `PipedOutputStream` and `PipedInputStream`. Also, `init()` creates all the new threads and passes them to the pipe. The `run()` method is used to synchronize the activity between the threads as they pass back and forth from the pipes. The `start()` method starts the parent and child thread groups. The `stop()` method closes open threads and associated pipes. The `boolean action()` method is the event handler for the user when he presses the "Start" and "Stop" buttons for the `JessThreads` applet. Finally, the `main()` method is used to create, initialize, and start the threads and the window frame. The following code illustrates how the `main()` function is used to manipulate threads.

```
public static void main(String args[]) {
    //create the frame
    Frame f1 = new Frame("Standalone Thread Processor for JESS");

    //create the threads
    jessThreads s1 = new jessThreads();
    s1.init();
    s1.start();
    f1.add("North", s1);
    f1.resize(600, 500);
    f1.show();
}
}
```

The third step involves the maintenance of all new threads. This essentially involves establishing a hub for maintaining and coordinating the specialist-agent threads and for interfacing these threads with the interface-agent thread. The `newThread` class is the extension of the initial thread class. In this class, piping is established along with a `run()` method to keep track of the current thread while others are set to come into the pipe. The following code is our implementation of class `newThread`.

```
class newThread extends Thread {
    String jessTalk[] =
```

```

    {
        "This is the child Thread",
        "Initializing for the Interfaceagent Agent",
        "Connection Established"
    };

    TextArea AreaC = null;
    int dialog = 0;
    int sleepInterval = 2000;
    PipedOutputStream PipeOutC;
    PipedInputStream PipeInC;

    public newThread(PipedInputStream PipeIn, PipedOutputStream
        PipeOut, TextArea ChildArea) {
        this.PipeOutC = PipeOut;
        this.PipeInC = PipeIn;
        this.AreaC = ChildArea;

        try {
            PipeOutC.connect(PipeInC);
        } catch ( IOException e ) {
            return;
        }
    }

    public void run() {
        try {
            while ( true ){
                Thread.currentThread().sleep(sleepInterval);
                AreaC.appendText(jessTalk[dialog] + new String("\n"));
                Thread.currentThread().sleep(sleepInterval);
                PipeOutC.write(dialog);
                dialog++;
                if (dialog > 2)
                    dialog = 0;
            }
        } catch ( IOException e ) {
            return;
        } catch ( InterruptedException e ) {
            return;
        }
    }
}

```

## 4. Conclusion

The system we have described consists of a hub of four communicating specialist agents, all executing on their own threads, that communicates with an interface agent, which executes on its own thread. Interagent communication is via pipes, which are also responsible for synchronization. The specialist agents are expert systems that execute the same Jess program but have their own rule bases. Each of these agents interacts with a user to navigate through a taxonomy of topics. The hub of these agents has the role of a client. The interface agent is also an expert system; it has its own Jess program and rule

base. The interface agent has the role of server and receives information from the specialist agents, from which it compiles a report.

We have found that pipes are appropriate for synchronizing multi-agent systems with extensive communication because threads suspend while waiting for input. Also, an effective way to structure multi-agent systems is to group clients together in a hub that communicates with a server agent. This architecture generalizes to cases where several servers may interact with a hub of clients and where there are several hubs of clients.

In the future we hope to implement multi-agent systems that include several servers with different functionalities (such as producing rule bases or interpreting data, besides compiling reports) and several hubs of clients. We also intend to produce a prototype that implements agents on separate platforms using Java's networking capabilities. This will involve sockets instead of pipes, but the hub and client-server architecture should still apply. In the more distant future, we anticipate implementing mobile agents, which move between platforms.

## References

- [DD98] Deitel, H. M. and Deitel, P. J., *Java: How to Program* (2<sup>nd</sup> ed.), Prentice Hall: Upper Saddle River, NJ, 1998.
- [Fr98] Friedman-Hill, E., *Jess, The Java Expert System Shell*, SAND98-8206, Version 4.0, Distributed Computing Systems, Sandia National Laboratories, Livermore, CA, 1998.
- [Gi93] Giarratano, J.C., *CLIPS User's Guide* (CLIPS Version 6.0). NASA L.B. Johnson Space Center, Information Systems Directorate, Software Technology Branch, 1993.
- [La95] Lano, K., *Formal Object-Oriented Development*. London: Springer-Verlag, 1995.
- [LO97] LOGOS Development Team, Automation Technology Section, GSFC, *LOGOS Requirements & Design Document*, NASA/Goddard Space Flight Center, Greenbelt, Md., 1997, <http://groucho.gsfc.nasa.gov/agents/products.html>.
- [Ma96] Maynard-Reid, P., *Architectural, Knowledge Representation, and Formalization Issues in AFLOAT*. Automation Technology Section, NASA Goddard Space Flight Center, 1996.
- [Mu97] Murphy, Torrii, "Formalizing an Object-Oriented Agent Information Management System," Master's Project, Department of Computer Science, North Carolina A&T State University, 1997.
- [Tu94] Turner, R., *Adaptive Reasoning for Real-World Problems: A Schema-Based Approach*. Hillsdale, NJ: Lawrence Erlbaum, 1994.