Intelligent Agent Framework

- Some general requirements
  - It should be possible to add intelligent agents to existing Java programs
  - These agents should use AI techniques as appropriate
  - The agents should be practical and be useful in a wide range of applications
  - GUI interfaces should make design of agents simple
  - The design should be simple and understandable

Design Goals

- Agents should be able to add value to a single standalone application or a community of interacting agents that work with each other and with other applications
- The agent manager must provide a straightforward way to interface with the application program that will be using the agent
- The agent framework should be straightforward and easily understood
- Commercial grade software that handles all unexpected input and errors can be highly complex; to keep the framework simple and understandable, it will work as expected with good data but not be bulletproof
- Interagent communications must be supported
Functional Specifications

- It will be easy to add agents to existing Java applications
- A graphical construction tool will help compose agents
- The agents must have a relatively-sophisticated event processing capability
- If … then rules must be supported with either forward chaining or backward chaining
- Learning algorithms should allow agents to perform classification, clustering, and prediction
- Multiagent applications should support a KQML-like message protocol
- Agents must be persistent, in the sense they can be saved and reloaded later

Intelligent Agent Architecture - 1

- To provide autonomy and flexibility, the agents will run in separate threads; rather than calling agent methods directly, the application will create events that the agent responds to
- We will use Java Beans so that we can compose agents out of components
- Java provides a Delegation Event Model that uses EventObject and EventListener
- The Boolean rule-based system developed earlier will be extended to support sensors and effectors
Intelligent Agent Architecture - 2

- Classification will be provided by DecisionTree, clustering by BackPropNet, and prediction by KMapNet from the previous chapter on learning
- To provide KQML-like messaging we will use our existing rule capabilities along with the built-in Java event model
- We will use Java serialization to provide persistence for objects

Constructing Intelligent Agents

- We refer to our package as ciagent for constructing intelligent agents; we will use Java Beans whenever possible
package ciagent;
import java.io.*;

public class CIAgentState implements Serializable {
    public static final int UNINITIATED = 0;
    public static final int INITIATED = 1;
    public static final int ACTIVE = 2;
    public static final int SUSPENDED = 3;
    public static final int UNKNOWN = 4;
    private int state;

    public CIAgentState() { state = UNINITIATED; }

    public synchronized void setState(int state) { this.state = state; }

    public int getState() { return state; }

    public String toString() {
        switch (state) {
            case UNINITIATED : { return "Uninitiated"; }
            case INITIATED : { return "Initiated"; }
            case ACTIVE : { return "Active"; }
            case SUSPENDED : { return "Suspended"; }
            case UNKNOWN : { return "Unknown"; }
        }
        return "Unknown";
    }
}
CIAgentTimer

- The runnit Thread implements the run() method
  - After every sleepTime milliseconds, the agent’s method processTimerPop() is called
  - Events are queued up in an asynchronous manner and processed every asyncTime milliseconds
  - Printouts for debugging are provided
- Timer controls are
  - startTimer(), if necessary, creates the thread and starts it for the first time
  - stopTimer(), by setting timerEnabled to false, the processTimerPop() is not called
  - quitTimer(), causes the run() method to exit

```java
package ciagent;
import java.util.*;
import java.io.*;
public class CIAgentTimer implements Runnable, Serializable {
    private CIAgent agent;          // owner agent
    private int sleepTime = 1000;   // mSec - default to 1 second
    private boolean timerEnabled = true;
    private int asyncTime = 500;    // mSec - default to 0.5 seconds
    transient private Thread runnit = new Thread(this);
    private boolean quit = false;   // used to exit thread run() method
    private boolean debug = false;  // turn on to see timer vars

    // Creates a timer, specifying the agent that owns the timer.*
    public CIAgentTimer(CIAgent agent) { this.agent = agent; }

    // Sets the time (in milliseconds) that determines how often an agent performs
    // the autonomous behavior that is defined in its processTimerPop method.*
    public void setSleepTime(int sleepTime) {
        this.sleepTime = sleepTime;
    }

    // Retrieves the sleep time (in milliseconds). */
    public int getSleepTime() {
        return sleepTime;
    }

    // Sets the time (in milliseconds) that determines how often the agent processes
    // asynchronous events on its event queue.*
    public void setAsyncTime(int asyncTime) {
        this.asyncTime = asyncTime;
    }
}
```
public int getAsyncTime() { return asyncTime; }

public void startTimer() {
    timerEnabled = true;
    if (!runit.isAlive()) {
        runnit.start(); // start on first time only
    }
}

public void stopTimer() { timerEnabled = false; }

public void quitTimer() { quit = true; }

private void readObject(ObjectInputStream theObjectInputStream) throws ClassNotFoundException, IOException {
    // Restore the transient variables.
    runnit = new Thread(this);
    // Restore the rest of the object.
    theObjectInputStream.defaultReadObject();
}

public void run() {
    long startTime = 0;
    long curTime = 0;
    if (debug) {
        startTime = new Date().getTime(); // get start time in mSec
        curTime = startTime; // current time in mSec
    }
    if (sleepTime < asyncTime) { asyncTime = sleepTime; }
    int numEventChecks = sleepTime / asyncTime;
    if (debug) {
        System.out.println("sleepTime= " + sleepTime + " asyncTime= " + asyncTime + " numEventChecks= " + numEventChecks);
    }
    while (quit == false) {
        try {
            for (int i = 0; i < numEventChecks; i++) {
                Thread.sleep(asyncTime);
                if (debug) {
                    curTime = new Date().getTime(); // current time in mSec
                    System.out.println("async events timer at " + (curTime - startTime));
                }
                if (quit) { break; } // exit loop
                agent.processAsynchronousEvents();
            }
            if (timerEnabled && (quit == false)) {
                if (debug) {
                    System.out.println("timer event at " + (curTime - startTime));
                }
                agent.processTimerPop(); // call the timer processing method
            }
        } catch (InterruptedException e) {
            // swallow
        }
    }
}
CIAgent Overview - 1

- Interface implementations
  - CIAgentEventListener for communications purposes; we need a vector to hold all the registered listeners
  - Serializable so that agent is persistent; some data is transient and not stored in the serialized object

- Support for communications
  - `addCIAgentEventListener()` is used to register other agents that will listen to this agent
  - Any object can generate events; only objects implementing CIAgentEventListener can listen
  - There is also a `removeCIAgentEventListener()` method
  - `notifyCIAgentEventListener()` is used to send events
  - All three methods must be synchronized

CIAgent Overview - 2

- Naming the agent
  - The name of each agent is stored in a String with getter/setter methods available
  - PropertyChange events for Java Beans are used to signal configuration changes when the name changes

- Other methods
  - Putting agent in a known state: `initialize()` and `reset()`
  - `startAgentProcessing()`, `stopAgentProcessing()`, `suspendAgentProcessing()`, `resumeAgentProcessing()`
  - To build composite agents we have `addAgent()`, `removeAgent()`, and `getChildren()`
  - `readObject()` used in serialization must recreate transient data members
package ciagent;
import java.util.*;
import java.awt.*;
import javax.swing.*;
import java.beans.*;
import java.io.*;
public abstract class CIAgent implements CIAgentEventListener, Serializable {
    public static final int DEFAULT_SLEEPTIME = 15000; // 15 seconds
    public static final int DEFAULT_ASYNCTIME = 1000; // 1 second
    protected String name; // the agent’s name
    protected CIAgentState state = new CIAgentState();
    private CIAgentTimer timer = new CIAgentTimer(this);
    transient private Vector listeners = new Vector(); // list of listeners
    transient private CIAgentEventQueue eventQueue = new CIAgentEventQueue();
    transient private PropertyChangeSupport changes = new PropertyChangeSupport(this);
    private boolean traceOn = false;
    protected int traceLevel = 0;
    protected AgentPlatform agentPlatform = null; // agent platform reference
    protected Vector children = new Vector(); // contained agents (if any)
    protected CIAgent parent = null; // container agent (if any)
    /**
     * Creates a CIAgent instance.
     */
    public CIAgent() {
        this("CIAgent");
    }
    /* Creates a CIAgent with specified name. */
    public CIAgent(String name) {
        this.name = name;
        timer.setAsyncTime(DEFAULT_ASYNCTIME); // set asyncTime
        timer.setSleepTime(DEFAULT_SLEEPTIME); // set sleepTime
        state.setState(CIAgentState.UNINITIATED);
    }
    /*Sets the name of this agent.*/
    public void setName(String newName) {
        String oldName = name;
        name = newName;
        changes.firePropertyChange("name", oldName, name);
    }
    /* Retrieves the name of this agent.*/
    public String getName() { return name; }
    /**
     *  Sets the current state of this agent (UNINITIATED, INITIATED, ACTIVE,
     *  SUSPENDED, UNKNOWN).*/
    protected void setState(int newState) {
        int oldState = state.getState();
        changes.firePropertyChange("state", oldState, newState);
        this.state.setState(newState);
    }
    /*  Retrieves the current state of this agent (UNINITIATED, INITIATED, ACTIVE,
     *  SUSPENDED, UNKNOWN).*/
    public CIAgentState getState() { return state; }
/* Creates a CIAgent with specified name. */
public CIAgent(String name) {
    this.name = name;
    timer.setAsyncTime(DEFAULT_ASYNCTIME); // set asyncTime
    timer.setSleepTime(DEFAULT_SLEEPTIME); // set sleepTime
    state.setState(CIAgentState.UNINITIATED);
}
/*Sets the name of this agent.*/
public void setName(String newName) {
    String oldName = name;
    name = newName;
    changes.firePropertyChange("name", oldName, name);
}
/* Retrieves the name of this agent.*/
public String getName() { return name; }
/*
 * Sets the current state of this agent (UNINITIATED, INITIATED, ACTIVE,
 * SUSPENDED, UNKNOWN).*/
protected void setState(int newState) {
    int oldState = state.getState();
    changes.firePropertyChange("state", oldState, newState);
    this.state.setState(newState);
}
/* Retrieves the current state of this agent (UNINITIATED, INITIATED, ACTIVE,
 * SUSPENDED, UNKNOWN).*/
public CIAgentState getState() { return state; }
/* Sets the agent sleep time (in milliseconds) for autonomous processing. Note
* that this has no effect on a running agent. */
public void setSleepTime(int sleepTime) {
    timer.setSleepTime(sleepTime);
}

/* Retrieves the agent sleep time (in milliseconds). */
public int getSleepTime() {
    return timer.getSleepTime();
}

/* Sets the time (in milliseconds) for asynchronous event processing. Note that
* this has no effect on a running agent. */
public void setAsyncTime(int asyncTime) {
    timer.setAsyncTime(asyncTime);
}

/* Retrieves the time (in milliseconds) for asynchronous event processing. */
public int getAsyncTime() {
    return timer.getAsyncTime();
}

/* Sets the trace level. */
public void setTraceLevel(int traceLevel) {
    this.traceLevel = traceLevel;
}

/* Retrieves the trace level. */
public int getTraceLevel() {
    return traceLevel;
}

/* Sets the agent platform for this agent. */
public void setAgentPlatform(AgentPlatform agentPlatform) {
    this.agentPlatform = agentPlatform;
}

/* Retrieves the agent platform. */
public AgentPlatform getAgentPlatform() { return agentPlatform; }

/* Retrieves the other agents running on the same agent platform */
public Vector getAgents() {
    if (agentPlatform == null) {
        return null;
    } else {
        return agentPlatform.getAgents();
    }
}

/* Retrieves an agent running on the same agent platform */
public CIAgent getAgent(String name) {
    if (agentPlatform == null) {
        return null;
    } else {
        return agentPlatform.getAgent(name);
    }
}

/* Retrieves a formatted string for display of this agent's current task. */
public abstract String getTaskDescription();

/* Retrieves a list of CIAgent objects contained by this agent. */
public Vector getChildren() {
    return (Vector)children.clone();
}
/* Sets the parent CIAgent for this agent. */
public void setParent(CIAgent parent) {
    this.parent = parent;
}

/* Retrieves the parent CIAgent object. */
public CIAgent getParent() {
    return parent;
}

/* Uses introspection on this bean to get the customizer class (if any). */
public Class getCustomizerClass() {
    Class customizerClass = null;
    try {
        BeanInfo beanInfo = Introspector.getBeanInfo(this.getClass());
        BeanDescriptor beanDescriptor = beanInfo.getBeanDescriptor();
        customizerClass = beanDescriptor.getCustomizerClass();
    } catch (IntrospectionException exc) {
        System.out.println("Can't find customizer bean property " + exc);
    }
    return customizerClass;
}

/* Uses introspection on this bean to get the display name (defaults to the */
/* class name). */
public String getDisplayName() {
    String name = null;
    try {
        BeanInfo beanInfo = Introspector.getBeanInfo(this.getClass());
        BeanDescriptor beanDescriptor = beanInfo.getBeanDescriptor();
        name = (String) beanDescriptor.getValue("DisplayName");
    } catch (IntrospectionException exc) {
        System.out.println("Can't find display name bean property " + exc);
    }
    if (name == null) {
        name = this.getClass().getName(); // default to class name
        return name;
    }
}

/* Resets this agent so that it is in a known state. */
public void reset() {}

/* Initializes this agent for processing. */
public abstract void initialize();

/* Starts the agent timer and asynchronous event processing and sets the agent */
/* state to ACTIVE. */
public synchronized void startAgentProcessing() {
    timer.startTimer();
    setState(CIAgentState.ACTIVE);
}

/* Stops the agent timer and asynchronous event processing and sets the agent */
/* state to UNKNOWN. */
public synchronized void stopAgentProcessing() {
    timer.quitTimer();
    setState(CIAgentState.UNKNOWN);
}

/* Temporarily stops the agent timer so that the autonomous behavior is */
/* suspended and sets the agent state to SUSPENDED. */
public void suspendAgentProcessing() {
    timer.stopTimer();
    setState(CIAgentState.SUSPENDED);
}
/* Resumes agent processing of the timer and asynchronous events and sets the agent state to ACTIVE. */
public void resumeAgentProcessing()
{
    // turn async event processing on
    // turn timer event processing on
    timer.startTimer();
    setState(CIAgentState.ACTIVE);
}

/* Provides the synchronous processing done by this agent. */
public abstract void process();

/* Provides the asynchronous, autonomous behavior of this agent that occurs periodically, depending on the sleep time for this agent. */
public abstract void processTimerPop();

/* Processes all events on the asynchronous event queue periodically, depending on the asynchronous event time for this agent. */
public void processAsynchronousEvents()
{
    CIAgentEvent event = null;
    while ((event = eventQueue.getNextEvent()) != null) {
        System.out.println("CIAgent: " + name + " dispatching an Async event");
        processCIAgentEvent(event);
    }
}

/* Performs synchronous event processing for this agent. */
public void processCIAgentEvent(CIAgentEvent event) {
}

/* Posts an event to this agent's event queue. */
public void postCIAgentEvent(CIAgentEvent event) { eventQueue.addEvent(event); }

/* Adds a listener for CIAgent events. */
public synchronized void addCIAgentEventListener(CIAgentEventListener listener) {
    listeners.addElement(listener);
}

/* Removes a listener for CIAgent events. */
public synchronized void removeCIAgentEventListener(CIAgentEventListener listener) {
    listeners.removeElement(listener);
}

/* Delivers the CIAgent event to all registered listeners. */
protected void notifyCIAgentEventListeners(CIAgentEvent e) {
    Vector l;
    synchronized (this) {
        l = (Vector) listeners.clone();
    }
    for (int i = 0; i < l.size(); i++) { // deliver the event
        ((CIAgentEventListener) l.elementAt(i)).processCIAgentEvent(e);
    }
}

/* Adds a listener for PropertyChange events. */
public synchronized void addPropertyChangeListener(PropertyChangeListener listener) {
    changes.addPropertyChangeListener(listener);
}

/* Removes a listener for PropertyChange events. */
public synchronized void removePropertyChangeListener(PropertyChangeListener listener) {
    changes.removePropertyChangeListener(listener);
}
public void trace(String msg) {
    // create a data event
    CIAgentEvent event = new CIAgentEvent(this, "trace", msg);
    // and send it to any registered listeners
    notifyCIAgentEventListeners(event);
}

/**
 * Adds a CIAgent to this agent and sets its parent member to point to
 * this agent.
 * @param child the CIAgent object to be added to this object as a child
 */
public void addAgent(CIAgent child) {
    children.addElement(child);
    child.setParent(this);
}

/**
 * Removes a CIAgent from this agent.
 * @param child the CIAgent object to be removed from this agent
 */
public void removeAgent(CIAgent child) {
    children.removeElement(child);
}

/**
 * De-serialize the object from the specified input stream by
 * re-initializing the object's transient variables, de-serializing
 * the object with defaultReadObject(), and then hooking up the
 * de-serialized stuff to the re-initialized stuff.
 * @param     ois the ObjectInputStream object from which this object
 *            is to be read
 * @exception ClassNotFoundException if any class file is not found
 * @exception IOException on any IO error
 */
private void readObject(ObjectInputStream theObjectInputStream) throws
    ClassNotFoundException, IOException {
    // Restore the transient variables.
    changes = new PropertyChangeSupport(this);
    listeners = new Vector();
    eventQueue = new CIAgentEventQueue();

    // Restore the rest of the object.
    theObjectInputStream.defaultReadObject();
}
CIAgentEvent Overview

- JavaBeans requires this be derived from the built-in Java EventObject
- The constructors
  - A single parameter specifying the source
  - The source plus an event argument object
  - The source, an action String that is the name of a method in the target object, and an event argument object
- The “event argument” object
  - We cannot know in advance the subclass for this object, so it is a generic object
  - Getter methods are provided for the action and argument properties

```java
class CIAgentEvent extends java.util.EventObject {
    private Object argObject = null;
    private String action = null;

    /* Creates an extremely simple CIAgentEvent object. */
    public CIAgentEvent(Object source) {
        super(source);
    }

    /* Creates a more complex CIAgentEvent object that includes an object related to the event. */
    public CIAgentEvent(Object source, Object argObject) {
        this(source);
        this.argObject = argObject;
    }

    /* Creates a more complex CIAgentEvent object that includes an object related to the event and an action. */
    public CIAgentEvent(Object source, String action, Object argObject) {
        this(source);
        this.action = action;
        this.argObject = argObject;
    }

    /* Retrieves the object related to this event. */
    public Object getArgObject() {
        return argObject;
    }
}
```
CIAgentEvent - 2

CIAgentEvent - 2

CIAgentEventListener Interface

• The following methods are required by the EventListener parent class

```java
class CIAgentEventListener extends java.util.EventListener {
    public void processCIAgentEvent(CIAgentEvent e) {
        // Process the event
    }

    public void postCIAgentEvent(CIAgentEvent e) {
        // Post the event
    }
}
```
CIAgentEventQueue Overview

- `addEvent()` puts a new event on the end of the queue
- `getNextEvent()`
  - Returns null if the queue is empty or
  - Removes the first item and returns it
- `peekEvent()`
  - Returns null if the queue is empty or
  - Returns the first item in the queue but does not remove it

```java
package ciagent;
import java.util.*;
import java.io.*;

public class CIAgentEventQueue implements Serializable {
    private Vector eventQueue;

    /* Creates an event queue. */
    public CIAgentEventQueue() {
        eventQueue = new Vector();
    }

    /* Adds an event to the end of the queue. */
    public synchronized void addEvent(CIAgentEvent event) {
        eventQueue.addElement(event);
    }

    /* Retrieves the first element from the queue (if any) after removing it from * the queue. */
    public synchronized CIAgentEvent getNextEvent() {
        if (eventQueue.size() == 0) {
            return null;
        } else {
            CIAgentEvent event = (CIAgentEvent) eventQueue.elementAt(0);
            eventQueue.removeElementAt(0);
            return event;
        }
    }
```
CIAgentEventQueue - 2

```java
/**
 * Retrieves the first element from the queue (if any) without removing it.
 * @return the first event on the queue
 */
public synchronized CIAgentEvent peekEvent() {
    if (eventQueue.size() == 0) {
        return null;
    } else {
        return (CIAgentEvent) eventQueue.elementAt(0);
    }
}
```

Lifetime of a CIAgent

1. Construct the instance of CIAgent and set to UNINITIATED
2. Set the JavaBeans properties programmatically or using a Customizer
3. Call the initialize() method; state is INITIATED
4. Call startAgentProcessing(); this starts the event queue thread and any event/timer processing, state is ACTIVE
5. Use the agent by calling process() direct or by sending events to be processed asynchronously
6. Processing can be suspended and restarted using suspendAgentProcessing() and resumeAgentProcessing()
7. Call stopCIAgentProcessing to stop the associated eventTimer thread; state is UNKNOWN
Start-up Collaboration Diagram

CIAgent action collaboration diagram
Sensors and Effectors

• One way to make our agents intelligent is to add BooleanRuleBased enhancements based on sensors and effectors in the application

• if sensor(sensorName, RuleVariable) then
effector(effectorName, parameters)

• The SensorClause makes a call to the sensor method and registers it with the RuleBase

• At runtime, RuleBase looks up sensorName and calls the method for the registered sensor object

• The effector works in a similar manner

• Rule.display() no longer works because the lhs, condition, and rhs data items are not used; to overcome this difficulty, display methods are added for sensors and effectors

Sensor Interface

```java
package rule;
import java.util.*;
import java.io.*;

public abstract interface Sensor {
    /* Defines the sensor method that must be implemented by
     * every class that can be called as a sensor. */
    public Boolean sensor(Object obj, String sName, RuleVariable lhs);
}
```

• The obj is a sensorClause, the String is the name of the sensor, and the RuleVariable is used to store the results
package rule;
import java.util.*;
import java.io.*;

public class SensorClause extends Clause {
    Sensor object;
    String sensorName;

    /* Creates a SensorClause object with the given parameters. */
    SensorClause(String sName, RuleVariable Lhs) {
        lhs = Lhs;
        cond = new Condition("=");
        rhs = " ";
        lhs.addClauseRef(this);
        ruleRefs = new Vector();
        truth = null;
        consequent = false; // must be antecedent
        sensorName = sName;
    }

    /* Displays information about this sensor clause. */
    public String display() {
        return "sensor(" + sensorName + "," + rhs + ") ";
    }

    /* Checks the truth value returned from the sensor. */
    public Boolean check() {
        if (consequent == true) {
            return null;
        }
        if (lhs.value == null) {
            BooleanRuleBase rb = ((Rule) ruleRefs.firstElement()).rb;
            object = (Sensor) (rb.getSensorObject(sensorName));
            truth = object.sensor(this, sensorName, lhs);
        }
        return truth;
    }
}
Effector Interface

```java
package rule;
import java.util.*;
import java.io.*;

public abstract interface Effector {

  /* Defines the effector method that must be implemented by
   * every class that can be called as an effector. */
  public long effector(Object obj, String eName, String args);
}
```

- The `obj` is an `effectorClause`, the `String` is the name of the effector, and the `String` is the arguments when the effector is called

EffectorClause

```java
package rule;
import java.util.*;
import java.io.*;

public class EffectorClause extends Clause {

  Effector object;      // object to call
  String effectorName;  // method to call
  String arguments;     // parameters to pass

  /* Creates an EffectorClause object with the given name
   * and argument string. */
  public EffectorClause(String eName, String args) {
    ruleRefs = new Vector();
    truth = new Boolean(true);  // always true
    consequent = true;  // must be consequent
    effectorName = eName;
    arguments = args;
  }

  /* Returns a display string for the effector. */
  public String display() {
    return "effector(" + effectorName + "," + arguments + ") ";
  }
}```
Adding Facts

- A Fact can be an assignment of a value to a RuleVariable, a sensor call, or an effector call
- Facts are registered in the RuleBase
- Before the inferencing cycle in BooleanRuleBase is called, an initializeFacts() method is called to set the Facts
- By using sensors and effectors, we are no longer confined to predefined rules in the rule base; rather we call outside methods defined by the application to test when a rule can fire, and we call outside methods from the application to carry out the desired action

```
package rule;
import java.util.*;
import java.io.*;
import java.awt.*;
import javax.swing.*;
public class Fact {
    BooleanRuleBase rb;
    String name;
    Clause fact;    //only 1 clause allowed
    Boolean truth; // states = {null=unknown, true, or false}
    boolean fired = false;
    /* Creates a Fact object as part of the given rule base
     * with the specified name and the fact clause. */
    Fact(BooleanRuleBase Rb, String Name, Clause f) {
        rb = Rb;
        name = Name;
        fact = f;
        rb.addFact(this); // add self to fact list
        truth = null;
    }
    /* Adds display information about this fact to the given text area. */
    void display(JTextArea textArea) {
        textArea.append(name + " : ");
        textArea.append(fact.toString() + "\n");
    }
}
```
Fact - 2

// assert the fact

/* Asserts the fact in the given rule base. */
public void assert(BooleanRuleBase rb) {
    if (fired == true) {
        return; // only assert once
    }
    rb.trace("\nAsserting fact " + name);
    truth = new Boolean(true);
    fired = true;
    if (fact.lhs == null) {
        // it's an effector
        ((EffectorClause) fact).perform(rb); // call the effector method
    } else {
        // set the variable value and update clauses
        fact.lhs.setValue(fact.rhs);
        // now retest any rules whose clauses just changed
    }
}