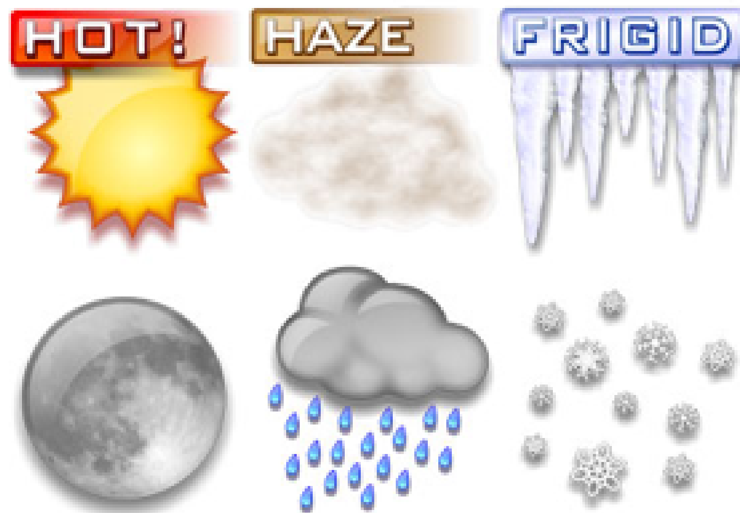
4005-750-01
Introduction to Artificial Intelligence
Environment Manager

Kurt Alfred Kluever (kurt@kloover.com)
Brian C. Renzenbrink (brenzenb@gmail.com)
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology

January 29, 2008



Abstract

A rule-based expert system has been developed that can autonomously govern a building's environment to optimize user comfort and energy consumption, whilst providing safety and monitoring functions. The expert system has been developed using the Java programming language and the Java Expert System Shell (JESS). Rules are stored as an external resource and can be modified in real time without requiring a rebuild of the entire project.

1 Executive summary

Nearly all modern buildings allow for control of the building's environment, but very few can adapt to the requirements of the occupants. This is mostly due to the fact that occupant preferences have not been recorded. Occupants who are unhappy with their work environment are less productive, and, in rare cases, can suffer from sick building syndrome and various other work related diseases.

While many recommendations exist, these standards do not capture the personal preferences of the occupants inside of the building. Instead, they are gross approximations of the general public. Furthermore, once chosen, the levels remain static and cannot react to changes in the occupants' preferences.

The latest generation of smart buildings are equipped with Personal Environment Modules (PEMs). PEMs allow the occupants to control the environment based on their own preferences. However, these systems are not without fault. Occupants tend not to use the PEMs if they perceive conflict with the other occupants sharing their work or living space. PEMs also do not allow for energy-conscious decisions made by the building operators.

Therefore, we have chosen to create an expert system to aid in monitoring several aspects of our building's indoor climate, as well as automating tools to adjust the current state to better meet the desires of all people inside the building and remain energy efficient. We have strategically placed wireless sensors around the building to monitor humidity, light, and temperature. In addition, we have also retrofitted the air conditioner, the heater, the humidifier, the dehumidifier and the digital light system to accept inputs from our expert system. The existing security system monitors when employees enter or exit the building by using RFID. The environment manager utilizes the security system to detect when employees are, or are not, present in the building. When no employees are present, the system will enter an energy saving mode.

2 Requirements

As mentioned in the executive summary, the problem is that no intelligent system exists that can avoid uncomfortable working conditions while still saving energy. Our solution is to create an energy-conscious, comfortable work environment for the occupants of the building, while providing alerts if a warning or emergency situation should arise. The system should meet the following requirements:

- Be fully autonomous and require no external human control
- Integrate the preferences of the occupants into the control decision
- Adapt to changes as occupants enter or exit the building
- Allow occupants to change their preferences
- Integrate readings of temperature, humidity, and illumination from the wireless sensors
- Reduce energy consumption when possible
- Execute an intelligent decision based on rules provided by domain experts

As an added benefit, the software will identify emergency situations such as fires, explosions, or floods, and issue emergency messages to the occupants and the appropriate emergency services. If a sensor detects a situation that may become an emergency, a warning message is issued to the occupants of the building.

3 Specification

Our solution consisted of a Java framework wrapped around Jess, a expert system shell developed by Sandia National Laboratories in Livermore, CA. The framework models sensors and occupants and interacts with the rule-based expert system. Outputs from the expert system are used to control the environment.

The aforementioned wireless sensors are able to monitor three aspects of the environment:

1. temperature (S_t)
2. humidity (S_h)
3. light level (S_l)

A collection of these sensors is maintained and modified as sensors are added or removed from the system. These sensors provide our expert system with real time data about our environment. Individual sensor readings are asserted into the expert system, as well as an

aggregate value for all sensor readings. The aggregate sensor values are computed as follows:

$$\begin{aligned} SA_t &= \sum_0^k S_t/k \\ SA_h &= \sum_0^k S_h/k \\ SA_l &= \sum_0^k S_l/k \end{aligned}$$

where k is the number of sensors currently registered with the system.

All employees will be required to register their environment preferences with the system. The security system will track their entrance and exit from the building using RFID tags in their badges. As an employee enters the building, the security system will load their preferences from a database. The preferences of an individual contain their preferred levels for the temperature, humidity, and light level. In order to prevent occupants from unfairly influencing the resulting environment, experts have been consulted to determine logical bounds on the preferences. An occupant's preferences must meet the following requirements:

$$\begin{aligned} 60.0^\circ F \leq & \quad P_t \quad \leq 85.0^\circ F \\ 0.0\% \leq & \quad P_h \quad \leq 60.0\% \\ 0 \leq & \quad P_l \quad \leq 10 \end{aligned}$$

where P_t is the preferred temperature, P_h is the preferred humidity, and P_l is the preferred illumination level. The optimal environment is computed as follows:

$$\begin{aligned} OA_t &= \sum_0^k P_t/k \\ OA_h &= \sum_0^k P_h/k \\ OA_l &= \sum_0^k P_l/k \end{aligned}$$

where k is the number of occupants currently in the building. If there are no occupants in the building ($k = 0$), the system will enter Energy-Saving Mode. This mode will attempt to

save power by dimming the lights to level 1 and shutting off any environment regulators.

The difference between the current state (as computed by the aggregate of the sensors) and the optimal state (as computed by the aggregate of the occupants' preferences) will determine what environmental regulator to use and at which setting. For instance, if $SA_t = 72.0°F$ and $OA_t = 69.0°F$, the air conditioning unit will be activated and set to level 3 ($\lfloor SA_t - OA_t \rfloor = 3$). Note that no opposing environmental regulators will be activated (i.e., the heater will be turned off in the previous example). The regulators are bounded by integer values from 0 to 10.

4 Description of the domain problem

Most buildings do not adapt to meet the environmental preferences of their occupants. This can be attributed to the fact that occupant preferences are not usually recorded. Occupants who are unhappy with their work environment are less productive, and, in rare cases, can suffer from sick building syndrome and various other work related diseases. Gross approximations of the general public fail to capture the personal preferences of the occupants inside of the building. Also these approximations are wasteful of power when there are no occupants in the building. Therefore, we have decided to design and implement a system that can address these problems with the hope that more comfortable employees will be more productive.

5 An overview of the tool

The Java Expert System Shell (Jess) is a rule-based expert system that utilizes a rule engine and scripting language that is similar to LISP. Jess uses a declarative paradigm that continuously applies a collection of facts to a collection of rules. Rules can modify facts, which in turn can fire off more rules in a process called chaining. As each fact is entered or updated, all of the rules associated or chained to that fact will fire. This makes complicated, intricate systems easy to implement. Jess is also a very fast and lightweight tool. We chose to use Jess primarily because of our preference for the Java language.

6 Feasibility study

The following potential solutions were investigated prior to designing and implementing our solution:

1. Pay someone else to do our project for us.
2. Use expert knowledge to create a single, static system state that pleases most people.
3. Allow the occupants to access and adjust the controls themselves with no automated system.
4. Develop a rule-based expert system from the ground up.
5. Utilize an existing rule-based expert system and wrap a suitable framework around it.

Option 1 is a clear violation of academic dishonesty and we lacked funds for paying someone (we are poor college students). Option 2 is commonly used in buildings. However, it is a gross approximation of the general public and is not tailored to the needs of the occupants. Option 3 allows for conflict between occupants and can lead to wasted time and unsatisfactory conditions for the majority. Option 4 would be a grueling task and time did not permit us to choose this option in a 10 week course. Option 5 is feasible and meets our requirements and goals.

Due to the requirements of the project, we decided that the chosen system must be autonomous (this eliminated Options 2 and 3). We then had to select between the following potential autonomous solutions:

- Instant automatic updates.
- Trying to reach the goal as quickly as possible (i.e. full blast AC until we have reached the goal temperature).
- Having the changes slow down as they approach their goal.

Instant changes are impossible in a large environment such as an office. A system that attempts to reach the goal state as quickly as possible would result in oscillation of the environment above and below the goal. This solution would also require an excess of energy. A system that used feedback control to gradually approach the goal state is the best solution.

The issue of zoning would be solved by implementing many individual expert systems; one for each zone. For simplicity of implementation, a single expert system is designed. If zoning is required, replicate the solution among every zone.

7 Implementation

For our system we have a dynamic number of sensors and occupants. As occupants are added to the working memory (representing that someone has entered the building) we update a running average of all occupants with the OccupantAggregator class. The OccupantAggregator is asserted into working memory and defines our goal state. Similarly, we average the sensor readings with the SensorAggregator to get our current state. However, each sensor is also added individually to working memory so that the rules can catch extreme readings. When an extreme reading is encountered, an appropriate warning or emergency message is issued containing the sensor's location.

We have used our real life expertise to set values that would correspond to emergency situations. For example, a temperature reading above 150°F issues a fire emergency alert while a temperature reading above 90°F but below 150°F issues only a warning message.

We used Jess to create a system of rules that compares the current state with the desired state and determines the best course of action for the environmental regulators. Jess is also used to fire off emergency alerts and warnings for individual sensors.

7.1 Rules

The following set of rules have been implemented in Jess and can be found in the rules.clp file.

7.1.1 Energy Saving Rule

1. If there are no occupants registered, dim the lights (illumination = 1) and turn off all systems to save energy (heater = 0, A/C = 0, humidifier = 0, dehumidifier = 0).

7.1.2 Intermediate Output Rules

1. If there is at least 1 occupant and 1 sensor registered and the occupants desire a lower temperature than the sensor readings, create a new TemperatureOutput object to be used for cooling.

2. If there is at least 1 occupant and 1 sensor registered and the occupants desire a higher temperature than the sensor readings, create a new TemperatureOutput object to be used for heating.

3. If there is at least 1 occupant and 1 sensor registered and the occupants desire a high humidity than the sensor readings, create a new HumidityOutput object to be used to increase humidity.

4. If there is at least 1 occupant and 1 sensor registered and the occupants desire a lower humidity than the sensor readings, create a new HumidityOutput object to be used to decrease humidity.

5. If there is at least 1 occupant and 1 sensor registered and the occupants desire a different illumination than the sensor readings, create a new IlluminationOutput object to be used to set illumination.

7.1.3 Final Output Rule

1. If we have a TemperatureOutput, a HumidityOutput and an IlluminationOutput asserted in memory, create a final OutputState that contains instructions for the Environment System.

7.1.4 Emergency Rules

1. If any sensor reports a temperature greater than 150°, create an EmergencyState object stating that there is a fire in the building at that sensor's location.

2. If any sensor reports a temperature greater than 150° and illumination greater that 10, create an EmergencyState object stating that there is an explosion in the building at that sensor's location.

3. If any sensor reports a humidity equal to 100%, create an EmergencyState object stating that there is a flood in the building at that sensor's location.

4. If any sensor reports a temperature less than 32°, create an EmergencyState object stating that it is freezing in the building at that sensor's location.

7.1.5 Warning Rules

1. If any sensor reports a temperature between 32° and 45°, create an WarningState object stating that it is near freezing in the building at that sensor's location.

2. If any sensor reports a temperature between 90° and 150°, create an WarningState object stating that it is very hot in the building at that sensor's location.

3. If any sensor reports a humidity less than 10%, create an WarningState object stating that it is very dry in the building at that sensor's location.

4. If any sensor reports a humidity greater than 90%, create an WarningState object stating that it is very wet in the building at that sensor's location.

7.2 Code Structure

We started off by creating a superclass called SystemState that would be extended by our Sensor and Occupant classes. A SystemState is an object that contains a value for humidity, temperature, and illumination. SystemState provides accessors for these values. The Sensor subclass provides mutators for the environment values as well as a sensor identification number. The Occupant subclass also provides mutators for the environment values, but it has limits on the values that can be chosen. If an occupant tries to request an out of bounds value for humidity, temperature, or illumination the method will throw an InvalidPreferenceException. These preference boundaries, along with default values, can be found in the static PreferenceLimits class.

The SensorAggregator and OccupantAggregator classes handle the collection of Sensor and Occupant objects. Both of these classes extend SystemState. SensorAggregator also manages the number of sensors currently registered in the system. It is used the addSensor() method to update the running average of all sensor data. Similarly, the OccupantAggregator manages the number of occupants and the running average of the preferences.

The next group of classes were designed to model intermediate outputs. There are intermediate output objects for humidity, temperature, and illumination. All of them have an accessor and a mutator that limits the value to 10. These objects are used to create the OutputState class that contains the instructions for the Environment Manager.

There is a MessageOutput class that provides basic functionality for the subclasses: EmergencyState and WarningState. These two classes are used to form the output necessary to alert the Environment Manager to the location and nature of the problem. They provide accessors and mutators for their data fields.

8 Users guide

The Environment Manager can be run using the following command:

```
java -jar EnvironmentManager.jar
```

By default, the demo is run using 10 sensors and 10 occupants. The occupants all request random environment preferences and the sensors read random environment values. Alternately, you can specify the number of sensors and occupants to create on the command line. For example:

```
java -jar EnvironmentManager.jar 200 5000
```

This will run the simulation with 200 sensors and 5000 occupants. The simulation will most likely report that warnings or emergency situations exist. This is because random data is being used to simulate the sensor readings.

Below is a sample output of running the Environment Manager with no parameters:

```
Adding the Occupant Aggregator: Temp=69.08111031350295F Humid=50.140395727159% Illum=6
Adding the Sensor Aggregator: Temp=72.7933973465817F Humid=42.74003568825808% Illum=5
----------
Warning/Emergency Messages:
WARNING Message: Sensor 1 is reporting 'Temperature is near freezing in the building.'
WARNING Message: Sensor 3 is reporting 'It is very wet in the building.'
WARNING Message: Sensor 7 is reporting 'It is very dry in the building.'
EMERGENCY Message: Sensor 0 is reporting 'Fire in the building!'
EMERGENCY Message: Sensor 2 is reporting 'Explosion in the building!'
EMERGENCY Message: Sensor 2 is reporting 'Fire in the building!'
EMERGENCY Message: Sensor 3 is reporting 'Flooding or raining in the building!'
EMERGENCY Message: Sensor 5 is reporting 'Temperature is at or below freezing in the building!'
EMERGENCY Message: Sensor 8 is reporting 'Temperature is at or below freezing in the building!'
EMERGENCY Message: Sensor 9 is reporting 'Temperature is at or below freezing in the building!'
----------
Instructions:
Set the A/C to level            3
Set the heater to level         0
Set the humidifier to level     7
Set the dehumidifier to level   0
Set the lights to level         6
```

9 Development Process Documentation

9.1 Development Process

Originally, we planned to investigate the Numenta Platform for Intelligent Computing (NuPIC). After an initial analysis, we unanimously decided against it due to our lack of familiarity with Python and time constraints.

Prior to starting the implementation, we had multiple design meetings. The first step was develop a list of rules that we ultimately wanted to incorporate into the system. Next the Java framework to model our occupants and sensors was built. Finally, the Java framework was connected to Jess and the rules were written.

After the initial solution was implemented, testing and documentation began. The documentation continued to evolve as the testing improved the system. One last meeting was held to finalize the documentation and submit the project.

9.2 Shared Responsibilities

- Brainstorm designs for the system
- Develop rules in psuedo code
- Learn JESS and convert the rules to JESS
- Finalize the documentation

9.3 Kurt's Responsibilities

- Prototype a Java solution without JESS
- Design a framework for JESS to work with
- Incorporate the framework with JESS

9.4 Brian's Responsibilities

- Research the feasibility of NuPIC
- Write documentation for the system
- Testing of the system