Friendly Wizard: A Tool Kit for Building Sensor-Based Systems

By
Liyan Guo
B.S. Massachusetts Institute of Technology (2001)
Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology
September 23, 2002

Author_____
Department of Electrical Engineering and Computer Science
September 23, 2002

Certified by_____
Jeff Elliott
VI-A Company Thesis Supervisor

Certified by_____
Alex (Sandy) Pentland
M.I.T. Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Friendly Wizard: A Tool Kit for Building Sensor-Based Systems

By

Liyan Guo

B.S. Massachusetts Institute of Technology (2001)

Submitted to the

Department of Electrical Engineering and Computer Science

September 23, 2002

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

Abstract

Sensor-based systems are systems that use sensors as their inputs to carry out certain tasks. These systems share similarities but many developers of these systems write their own proprietary code. The ever-changing nature of research makes the development of proprietary code very difficult to maintain. Friendly Wizard, a rule-based forward inference engine, will be created to help developers build a scalable, easy-debugged and failure-tolerant sensor-based system.

Thesis Supervisor: Alex (Sandy) Pentland

Title: Academic Head, M.I.T. Media Laboratory

Table of Content

List of Figures

List of Tables

1. Introduction

Small devices like cell phones, hand-held computers, and watches have gained incredible amounts of processing power over the past 5 years. Due to the emergence of different sensors and advances in small-device technology, many technology companies started working on systems that enable sensors and small devices work together as a single unit. The system uses information from sensors and small devices together to accomplish larger scale tasks. Any system that carries out tasks based on inputs from its environment by using sensors is a sensor-based system.

1.1 Looking at The Past, Present and Future Of Sensor Based System

Sensor-based systems have been around for a long time. They are everywhere. People have used sensor-based systems to help them accomplish many tasks. Sensor-based systems tend to have some sort of intelligence that can be used to enhance our everyday life. For example, the smoke detector is a simple sensor based system. When the smoke sensor senses smokes in the air, it will beep to get attention. Here is another example: modern cars have many sensor-based systems installed. For instance, the engine can adjust the air/gas intake ratio by knowing the temperature from a temperature sensor and oxygen levels from an oxygen sensor. The engine can, therefore, burn fuel more efficiently and environmental friendly.

Currently, researchers tend to work on larger scaled sensor-based systems. For example, "Smart car". A Smart car can read traffic signs, detect road conditions and warn you when there is danger ahead. Smart car knows how to maintain

itself. It can remind you when it needs an oil change. Another example is "Smart house". Smart house knows your preferred room temperature and lighting. A smart house has a smart refrigerator that reminds you to get more milk when you are almost done with your current bottle. A smart TV in the house knows what program you want to watch before you even turn it on or better yet knows when to turn itself on. These systems have a higher level of intelligence and much more complicated structure than any other sensor-based system built before. But what will the sensor-based systems be like in the future? How complicated can they get?

Many people envision that in the future we will have sensor-based systems all over the place. With the speed of technology today sensor-based systems will be built into every object. Objects can talk to each other. They are sensor-based systems themselves and yet they are also building blocks of larger sensor-based systems. The smart car will be able to communicate with the smart house. The smart house will be able to talk to other sensor-based systems such as smart office, smart highway.

1.2 Examples of Sensor-Based System

Here is a detailed example of one popular sensor-based system in research today. It is a smart meeting room. The smart meeting room will serve as an example in this paper to show how Friendly Wizard can enhance the experience of building sensor-based systems in research today.

One of the examples of sensor-based system is a Smart Meeting Room. Smart Meeting Room can be defined as a room that assists humans in carrying out meetings. For example, the room could know when to dim the lights, when to turn on the slide projector and when to move a camera. The room could remind each

invitee about a meeting. It could determine who is talking in the meeting. The room might even capture slides and audio to a server to remove the burden of notes taking.

Here is one scenario to show how a Smart Meeting Room can help employees in their daily life. Angela works in IBM Yorktown research center. On typical day:

Table 1 Regular Meeting Room vs. Smart Meeting Room

Regular meeting room	Sensor enhanced meeting room
Angela looks up her calendar and remembers that she wants to attend a meeting with her manager Tina.	Angela gets to work and then she receives an email from the Smart Meeting Room that reminds her she has a meeting with her manager Tina.
During the meeting, Angela takes many pages of notes.	During the meeting, Angela only jots down some guidelines because she knows that the meeting audio and slides is recorded pervasively on a server. She can later synchronize her notes to the recording.
After the meeting she asks Tina if Tina can put some important slides for the meeting on-line because she was not able to write down everything on her notebook. Tina agreed and gave Angela the URL of the slides. Angela writes the URL down on her notebook.	After the meeting, Angela asks Tina about an URL Tina briefly mentioned during the meeting. Angela unnoticeably taps on a button on her notepad to signal the meeting room that she want the room to record what Tina is going to say and save the audio to her computer.

Smart Meeting Room can be helpful in our daily life. However, due to many problems not related to the technical aspect of meetings, developing a sensor-based system like a Smart Meeting Room is difficult.

2. Problem Description

To develop a sensor-based system, we would like to build a system that can:

- Process information from a large number and various types of sensors and devices. This means devices can automatically communicate with each other without much human intervention.
- Provide the developer with a way to debug and simulate the system.
- Minimize number of possible errors as more sensors are connected to the system.
-

However, the traditional way of plainly using a programming language to program such systems is likely to lead the developers to pitfalls. First, adding new types of sensors might be complicated. Second, debugging and exception handling of such systems can be difficult.

The first problem is as the number of sensors increases, adding a new type of sensor can be difficult or even disastrous. If a sensor-based system is experimental, it is likely the developers will start with building a smaller system first. Building a smaller system gives developers a better view of whether the actual system can be useful. Figure 1 shows the beginning of a typical project where the system starts with only few sensors.

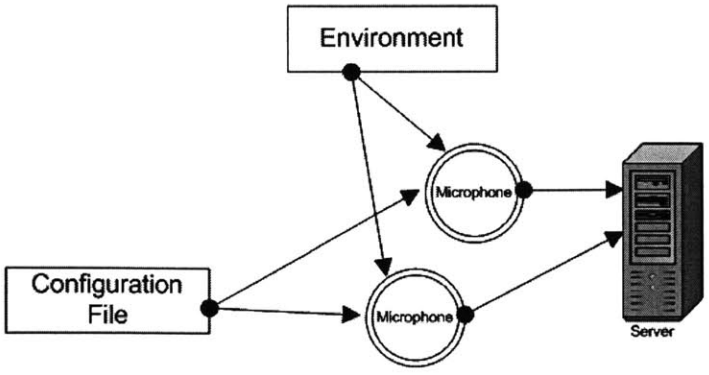


Figure 1 Beginning of building a sensor-based system.

Developers are likely to expand the smaller system they have built. That means new code will be added to the system. However the smaller system is intended as an indication of how the actual system will perform but is not designed to be expandable. Hence adding new sensors tends to make the system look like the system in Figure 2. Figure 2 shows the interactions within the system as the sensor-base expands. Each part of the system looks like a smaller version of Figure 1.

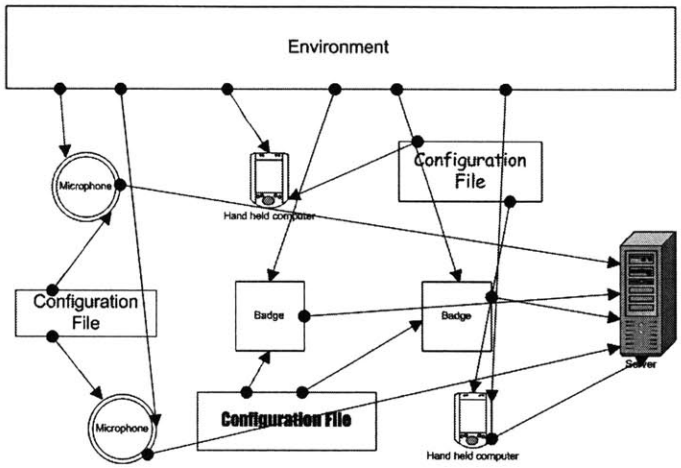


Figure 2 Interactions within a sensor-based system

The second problem is sensors that are connected to the system are different in the information they provide, their computational capability and communication protocol. The difference between sensors limits the salability of the system. If little thought has been put in to the development before the actual coding of the sensor-based system starts, the resulting system flow might be tightly coupled with different features of different devices. Failure of a device means failure of the whole system. The system needs to handle all the possible states properly. The debugging and simulating of such system can be very difficult, especially after new sensor is added. The newly added sensor might not be compatible with the rest of the system. The new sensor might require a different set of

configuration files, for example in Figure 2. This means developers need to change the interaction between different sensors and come up with new ways to debug the system. As the system gets bigger, debugging and simulating the system becomes more difficult to do.

These problems exist because of the lack of a standard way of programming sensor-based systems. If we can find a solution that abstracts and highlights the differences between sensors and formalizes how devices and sensors communicate with each other, we can build an easily maintainable, scalable, and flexible sensor-based system. A rule based system offers exactly the solution we need.

2.1 Available Solutions

Sensor-based systems that are developed in the past, such as smoke detector, elevator weight sensor, are small-scaled sensor-based systems. One or two people can easily understand them. There is normally not much work needed to make them work. Hence, the above problem does not apply to these kinds of sensor-based systems. However, there are only a limited amount of tasks can be accomplished by these kind of systems.

Commercial tools are also available today for developing very large-scale sensor-based systems. For example, Sun's technology enables unknown devices to communicate with each other on a network. Java Expert System Shell lets people use rules to manage large sensor based systems with artificial intelligence. Because these tools try to meet the needs of future sensor-based systems they are very difficult to use. There is a very steep learning curve associated with each technology. For example, a "hello world" program written in JINI can run over two pages.

2.2 Conclusion

What is lacking in sensor-based systems development today is a tool for systems that lay between the traditional small-scaled sensor-based systems and modern large-scaled sensor-based systems. Most of the projects in research are started with a small but complicated system. When such system is successful, it is then developed into a large-scale sensor-based system. Friendly Wizard is designed exactly for this early stage development of mid-sized sensor-based systems.

3. Friendly Wizard as a Solution

Friendly Wizard is a toolkit developed to enhance developers experience in making mid-sized experimental sensor-based systems. It has a flat learning curve, is flexible to changes in system design, easy to simulate, easy to debug and easy to interface. Friendly Wizard uses a forward inference rule engine with configurable rule syntax. Components that connect to Friendly Wizard, mainly sensors that provide information and devices that carry out tasks, will automatically configure themselves with Friendly Wizard's main engine. Developers can use rules to specify the behavior of the whole system. Three main features of Friendly Wizard are auto configuration, the rule-based engine and device/sensor simulation.

3.1 Rule-Based Inference Engine

Friendly Wizard is developed as a rule-based system. The main goal of Friendly Wizard is to help developers to use rules to construct logical relations between sensors, instead of plain code writing.

3.1.1 What is Rule-Based Engine?

Sensor-based systems draw conclusions based on sensor inputs. This behavior can be modeled by a rule-based system. Rule-based systems can associate simple rules, like "if... then..." clauses, the object being to model real-life scenarios. Rule-based systems in general have a knowledge base, a working memory and inference engine. The knowledge base is where the system stores

computer representation of knowledge. In this case, the knowledge is stored in some form of IF... THEN... rules. The working memory is used to store temporary variables so the inference engine can manipulate them and fire the right rule. The inference engine contains logic that can be applied to rules in the knowledge base and variables in the working memory.

A rule-based system is excellent in modeling decision-making systems like a sensor-based system. For example, let's say we want the Smart Meeting Room to dim the light if no one is in the room plus shut of the light after 5:00pm. We will store the above statement in the form of a rule, such as: "IF no one is in room THEN dim the light" and "IF time later then 5:00pm then shut the light off", and put this into the knowledge base of the rule-based system. The inference engine will first load a variable such as "how long the room has been empty" and "what time is it now" into the working memory. Different types of sensors supply these variable values. Then the engine goes to the knowledge base and looks through all the rules. If "what time is it now" is after 5:00pm, the inference engine will then know the rule "shut off light after 5:00" is true. The engine will then shut off the light in the room.

3.1.2 Why Rule-Based Engine?

The advantages of using rule-based systems for sensor-based applications is that they are:

- Flexible and scalable. Rule systems abstract and highlight the differences between sensors. All sensors are treated as variables in the working memory. More sensors present in a system just mean more variables in the working memory.

- Tolerant to device failure. Since devices are just a variable with a certain value in the working memory, dealing with device failure is much more generic and simple then traditional, plain coding.
- Simplify the management of complexity. Rule-based systems can use information within rules and variables to provide information on why it makes certain decision. This feature assists us in building a generic debugging system. The simulation can be carried out by simply changing the value of a variable in the working memory.
- Easy to understand. IF... THEN ... rules are easier for people to understand vs. computer programming language.
-

Now consider if we use a rule-based system for the setup in Figure 2. We can capture all the interconnected logics into a rule-based system. Then we will get Figure 3. Figure 3 shows a sensor-based system with a rule-based engine. Instead of having sensors communicate with each other, all the sensors communicate with the inference engine (rule-based engine). The rule engine draws decisions based on inputs from these sensors. IF and ELSE rules are used to describe inter-device logics in the rule engine. For every new sensor added, the developer only needs to write a driver for that sensor to connect to the rule engine. Hence, code maintaining for the whole system is broke up into code maintaining for individual sensors.

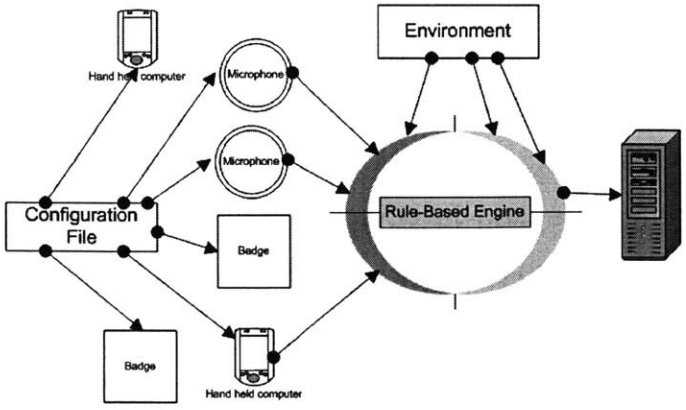


Figure 3 Structure of a sensor-based system using an inference engine

An added bonus of developing a rule-based engine is it can be used in most sensor-based systems. Among sensor-based systems, a lot of code shares similarities that are not yet compatible with each other. For example, codes for smart meeting room cannot be reused for smart lecture room because the smart meeting room interacts with certain a set of sensors differently than a smart lecture room. The rule engine abstracts out what is common among the development of sensor-based systems. It enables the developer to focus on the specifics of his system without worrying about the system structure for every sensor-based system.

3.1.3 Rule-Based System vs. Non Rule-Based System

I will present to you two solutions for building a smart meeting room for Angela's scenario. These two solutions will show you how a rule-based engine approach will help developers build a sensor-based system that is simple and fast.

In both examples, a team of programmers is required to implement the Smart Room. Due to the limitation on resources and time, their first goal is to automatically start/stop audio capture when a meeting starts/stops. The team is very successful with audio capture and they are also able to capture slides and video information because capturing these media is similar to capturing audio information. The manager is pleased with the result. He wants to make the system aware that a meeting is going on in certain rooms and who are the people present in these rooms. The system should be able to start/stop capture and store captured information based on invitees' demands. The team finds it difficult to incorporate the new type of sensor that senses presence of a person into the current system.

Table 2 Environment without Rule-Based System vs. Environment with Rule-Based System.

Solution I: Sensor Environment without Rule-based System.	Solution II: Sensor Environment with Rule-Based System.
The lead developer of the team restructures the old code so the new type of sensor can be added to the existing system. This means the team will spend more time on code maintenance and debugging. Two weeks later, the lead developer leaves the project. The project comes to a stop because no one else on the team knows how pieces of the project work together. People have to spend time on understanding what the lead developer did.	Instead of designing a new software structure, the team uses a rule-based tool. Now the programmers are able to focus on implement drivers for the sensors and let the rule-based engine handle all the interconnecting logic. Now the team spends less time on maintaining code. IF... THEN ... rules are easy to understand. When the lead developer leaves the project, the team was able to learn about rules fairly quickly.

So far the rule-based system does not seem to be much more of a time saver than straight coding. After all we might save some time on coding but we spend more time on learning how to program for the rule-based system. However, the rule-based system can also provide an easy and flexible programming interface, a suitable simulation environment, and a good rule-debugging interface. These add-ons for the rule engine will definitely minimize the time and difficulties in developing a sensor-based system.

For example, a programmer can simulate a device inside the rule engine before he actually starts to write the device driver. The rule-based engine has a built in fault tolerance. In case of a sensor malfunction, the rule-engine deals with

variables without assigned values. By default, if the expert system encounters a variable without assigned value, it won't stop the rule engine from making a decision. Maybe the decision won't be correct but at least the rule engine will keep running. Where as in traditional programming, developers need to take many possibilities into consideration to ensure their systems are fault tolerant. Last of all, rules are easily read. Imagine when a system performs an unexpected action, programmers need to trace through every step of the program execution to find out where the error is. In the case of the rule engine, the engine is able to tell you why it made certain decision.

Rules for the second example could look like:

Variables: /*Variables are values generated by incoming message*/

Subject, Action, Object,

Actions: /*Actions are names of functions outside inference engine*/

ButtonPress, enterRoom, recordForMeeting,

Rules:

Assign_Name:	IF Subject == Angela THEN Person=Angela)
Assign_Room:	IF Object == Room153 THEN Room_Name = 153
	IF Subject==Meeting AND Action==Start
	THEN MeetingStarts
Record_inMeeting:	IF MeetingStarts AND RoomName THEN
	recordForMeeting(RoomName)
EnterRoom:	IF Person AND RoomEntered THEN
	enterRoom(Person,RoomName)
	IF Person AND Action==ButtonPress Then
	buttonPress(Person)

3.2 Distributed Sensor Network with Auto Configuration

In the future, a sensor-based system will have hundreds if not millions of sensors. It is not likely that one or two people will develop all these sensors. It is also not likely there will be a central station that monitors and commands all the sensors available because that will be too much work. Sensor-based systems are designed to make our life easier but not meant to add complexity in our lives. Hence, it is likely that sensors and devices in a sensor-based system will have intelligence themselves to communicate with others through some kind of network.

Friendly Wizard is designed to build and control a large sensor network. In this network the designer describes what he wants the network to behave like by writing a set of rules. Individual devices and sensors and the network will automatically configure themselves to participate in this sensor-network. Friendly Wizard translates the rules written by the designer and controls all the devices and sensors on the network. Friendly Wizard will achieve this goal by enforcing an interface for all sensors and devices so that every sensor and device can be manipulated without interference. This abstraction barrier also enables all components on the network talk to each other using function calls provided by the interface.

3.3 System Simulation within Real-time Environments

Every one knows that no design is perfect, especially not the first trial. On the other hand research requires the best solution with little time. Therefore, to be able to find bugs and quickly debug the system is key in research. In fact, tools

that help in debugging a system are appreciated by almost all developers, in any area. Sensor-based systems are no exception to this rule.

The simulation package in Friendly Wizard enables developers to debug their systems before, during and after their systems are built.

Before a developer designs a system. He can use the simulation engine to construct a set of stubs of devices and sensors in his design. (A 'stub' is a Java class that has the same function calls as the real Java class, except the stub does nothing.) He can use the simulator to test if the set of rules that he wrote down give him the system behavior that he intended. In other words the simulator lets him see if the stubs behave the way he planed. This way most bugs in the rules will be waded out.

During construction of a sensor based system. The developer can always go back to the simulation engine and create stubs that represent the next component he is going to add to the system. The stub acts like another device on the network. It can interact with other components. The developer can construct new rules to test the feasibility of the new device. For example, the developer can find out whether the device conflicts with existing devices. The developer can run the old set of simulations to verify that the new device satisfies all the requirements.

After a system is built and running, there might be times that the system acts unexpectedly. Friendly Wizard is able to capture a detailed log of all the events triggered by the sensor-based system. Developers can look back into the log file and figure out what went wrong. Further more, because Friendly Wizard uses a forward inference engine to trigger actions, it can provide means for answers why certain actions are taken. It can also leave good hints of why certain actions were not taken.

For example, suppose Friendly Wizard is expected to turn off the light in the room when no one is around, however, Friendly Wizard turned the light on for two hours while no one is in the room. Instead of browsing through the entire log of a day, one can ask Friendly Wizard, "why do you turn on the light in the room, for example, yesterday 3:00pm?" Friendly Wizard would go through the log files and answer "Because according to rules_on_meetings_23, I turn on the light if a meeting is going on. Meeting_3342 was scheduled yesterday 3:00pm." With further inspection on Meeting_3342, the developer found out that Meeting_3342 was canceled. This could mean Friendly Wizard engine does not know how to deal with meetings that have been canceled.

4. Goals of Friendly Wizard

The primitive goal of Friendly Wizard is to build a rule-based system that provides maximum flexibility to developers. Maximum flexibility means the rule-based system can be reconfigured when rules are added or removed without recompiling or restarting the Rule Engine. This goal will be reached with a highly customizable rule engine and a robust configuration scheme, combined with a well-defined specification for interfacing with the rule engine.

The second goal of Friendly Wizard is to build a rule-based system that helps developers debug their applications in an easy way. The Rule-Based System will support different levels of simulations. The Rule-Based System also enables the user to simulate a stub sensor together with already developed sensors. This goal is backed up by a powerful rule engine that explains its action to the user, a developer UI that lets the developers simulate their systems visually and a well-designed rule engine that enables the developers to reach inside the engine without breaking it.

If the first two goals are accomplished on time, the third goal of Friendly Wizard is to build a distributed rule-based system. This implies that the rule-based systems can be distributed across multiple computers. The Friendly Wizard will be able keep data consistent and be tolerant to network failures.

5. Prototype

A prototype of Friendly Wizard is built for the smart meeting room given in previous examples. The smart meeting room has sensors that can detect who is in the room, who is speaking, as well as when a meeting has started. The smart room also provides access to equipment such as cameras, slides projectors and computers in the room. The smart meeting room generates events to the prototype rule engine such as: if the meeting has started/ended, who is in the room, who is speaking. The engine then triggers actions such as: start/stop audio recording and turn on/off lights.

The prototype sensor-based system will enable the smart room to display specific messages on a screen as invitees are coming into the room. The smart room will be able to start audio recording once a meeting is started in the room. When an invitee indicates that he needs to bookmark the meeting at certain time by pressing a button, the smart meeting room will store his bookmark as a time stamp that can be used to retrieve all recorded audio and captured slides. The smart meeting room is also able to recorde the meeting from different microphones based on who is talking at the time during the meeting.

6. Building Friendly Wizard

The entire toolkit is build based on Java. Java is chosen for several reasons. Java is a very high level language and Java is a cross platform language. It abstracts out many detailed low-level implementations for each platform Java supports. This is important for the developers. For example, developers would like to focus on how to make the system work instead of learning how some low level network stuff works differently under Linux and under Windows. Java applets have been used extensively on web pages over the Internet. Potentially these applets can be easily modified to be sensors that connect to Friendly Wizard.

Friendly Wizard consists of five modules, the inference module, communication module, I/O processing module, simulation module and debugging module. The inference module processes information; the communication module carries and distributes information; the I/O processing module deals with configuration and rule syntax; the simulation module simulates real live scenarios; and the debugging module logs debugging information.

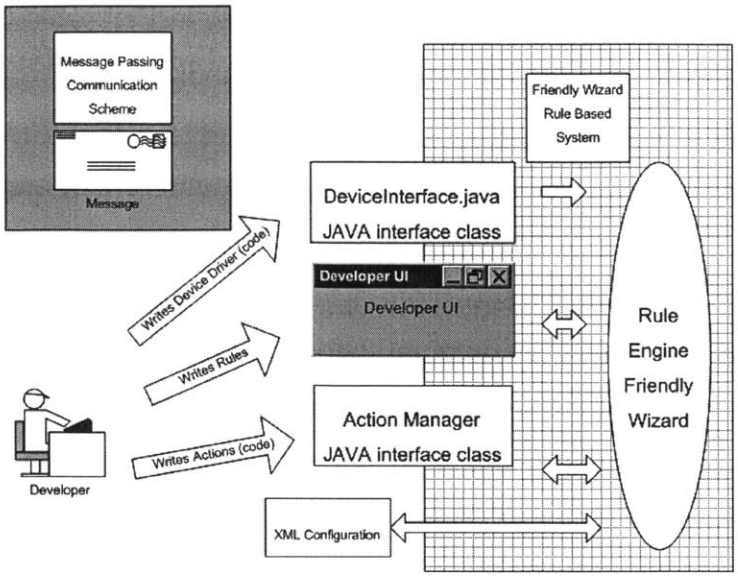


Figure 4 Building Friendly Wizard

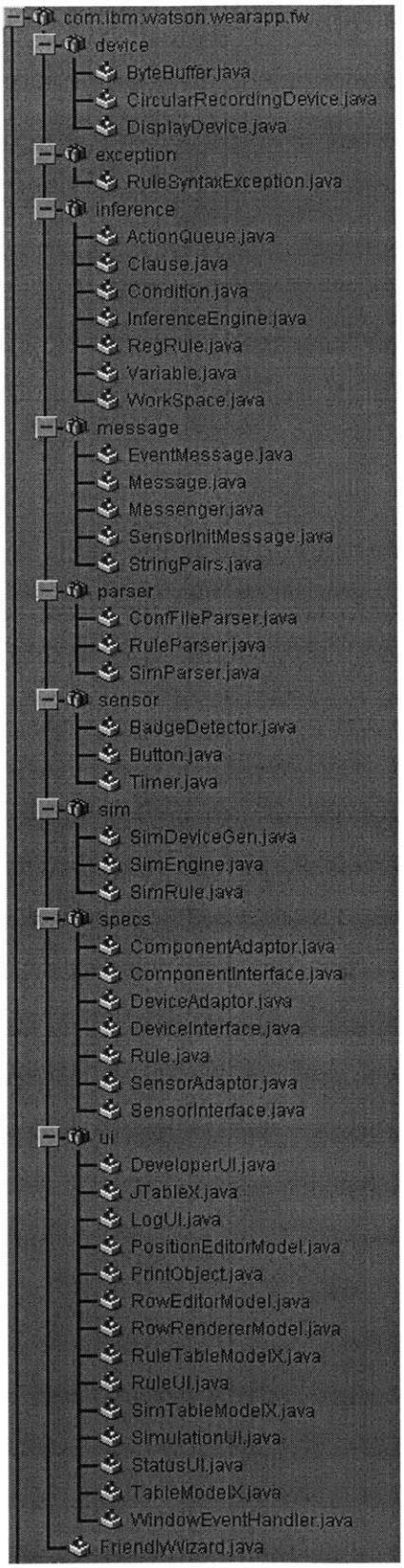
Friendly Wizard main engine resides on one machine. The main engine includes the inference module, I/O processing module, simulation module, debugging module. The communication module is used to connect the main engine to Friendly Wizard Components. Friendly Wizard components, later referred to as simply 'components' in this paper, includes Friendly Wizard sensors and Friendly Wizard devices. Friendly Wizard sensors, later referred to as sensors, are physical devices that provide information the main engine. Friendly Wizard devices, later referred to as devices, are physical devices that carry out actions requested by the Friendly Wizard main engine. Of course a physical device can be both a sensor and a device. However Friendly Wizard will treat the physical device as two separate devices, one is the sensor and another is the device. A physical device can also be divided into several sensors and several devices.

Friendly Wizard specifies set of Java interfaces and abstract classes to enforce how a developer can write his code to plug into Friendly Wizard. Friendly Wizard uses these java classes to specify its requirement of input sensors and configuration files etc. Developers can easily plug their systems into Friendly

Wizard by following the specification. For example, in Figure 4, DeviceInterface class is one of the Java interfaces specified by Friendly Wizard.

The inference module is a rule based forward inference engine. It makes decisions based on rules written by developers. See Figure 4. The rule engine is responsible for firing the correct rules in the correct order when it receives an incoming message. The rule engine loads up rules that are related to the message and fires them accordingly. The rule engine is also responsible for parsing rules, including identifying mistakes in rules, and gives developer a warning or shows debugging messages.

A developer can use Friendly Wizard through a graphical user interface that is called the developer UI. The developer UI includes four panels: a rule editing panel that includes every function related to rules, a simulation panel that lets the developer create simulations, a status panel that displays the state of the Friendly Wizard and last a log panel that displays recorded logs. The purpose of developer UI is to give the developer a visual presentation of what is going on in the system. Developers can edit rules and simulate unavailable sensors using this UI. The UI also gives messages that help the developer debug their rules.



6.1 Overall Program Architecture

The software structure of Friendly Wizard is divided into several java packages. Six of them correspond to the six modules of Friendly Wizard. The rest of the packages include accessory classes, examples and test cases.

The com.ibm.watson.wearapp.fw.inference package takes account of the inference module. The package includes classes that make a forward inference engine. The package also includes classes that enable the inference engine to communicate with other modules.

com.ibm.watson.wearapp.fw.message and com .ibm.watson.wearapp.fw.specs packages take account of the communication module. Communication is done by passing messages through a client and server like program.

com.ibm.watson.wearapp.fw.specs mainly contains interfaces and adaptor classes for Friendly Wizard Component. It is classified as a different module to facilitate development of the system. A developer can find all the interfaces, abstract classes and interface adaptors that they need in one folder.

Figure 5 Friendly Wizard Hierarchy

com.ibm.watson.wearapp.fw.parser package takes account of the IO processing module, it is able to parse XML rules and configuration files for the Friendly Wizard main engine. com.ibm.watson.wearapp.fw.exception package is an place holder for special exception messages created for Friendly Wizard. It is also part of the IO processing module.

com.ibm.watson.wearapp.fw.sim package takes account of the simulation module. The package can create virtual sensors and devices and assigned different action to each sensor or device.

com.ibm.watson.wearapp.fw.ui package takes account of the debugging module. Because debugging is mainly done through the developer UI, the debugging module is included in the UI package. The UI package also includes a front end graphical user interface for the Friendly Wizard main engine.

com.ibm.watson.wearapp.fw.sensor package and com.ibm.watson.wearapp .fw.device package take account of examples of sensors and devices in prototype. A developer can use them as a reference or a quick start to build their system.

A call to com.ibm.watson.wearapp.fw.FriendlyWizard will start the Friendly Wizard main engine.

6.2 Inference module

The Inference Module contains seven classes. The ActionQueue, Clause, Condition, InferenceEngine, RegRule, Variable, and workspace. Clause, Condition, Variable and RegRule are components that hold information for forward inference. See the Chart of Friendly Wizard's Hierarchy. The Inference

Engine carries out the actual inference. The workspace is a memory or storage space for the Inference Engine. When an action is ready to be executed, the inference engine sends the action to Action Queue. Action Queue is a separate thread that is responsible for making sure the specified Friendly Wizard device actually carries out the given action.

The most complicated class in the Inference Module is the Work Space Class. It is the repository for many types of information. It maintains lists of all the variables, clauses, conditions, and rules. It maintains current available devices and sensors. It also maintains a list of simulation devices and simulation rules. It knows which rule parser to use. Basically the Work Space class is a place to store and retrieve information. Work Space makes sure every piece of information is up to date and correct.

Variables, Conditions, Clauses, RegRules (Stands for Regular Rules, as we will see later there is also Simulation Rules, i.e. SimRule.) and Inference Engine together make a forward inference engine. Rule Parser parses input rules into variables, conditions, clauses and rules. Here is how everything works together.

Variable is the smallest unit. A variable contains a name and a value. To be safe, a variable also contains a list of possible values it can take. If it is assigned to a value other than possible values an error flag will be raised. Variable also contains a set of references that enable each variable to know all the clauses that use the variable. This set will be used in forward inference.

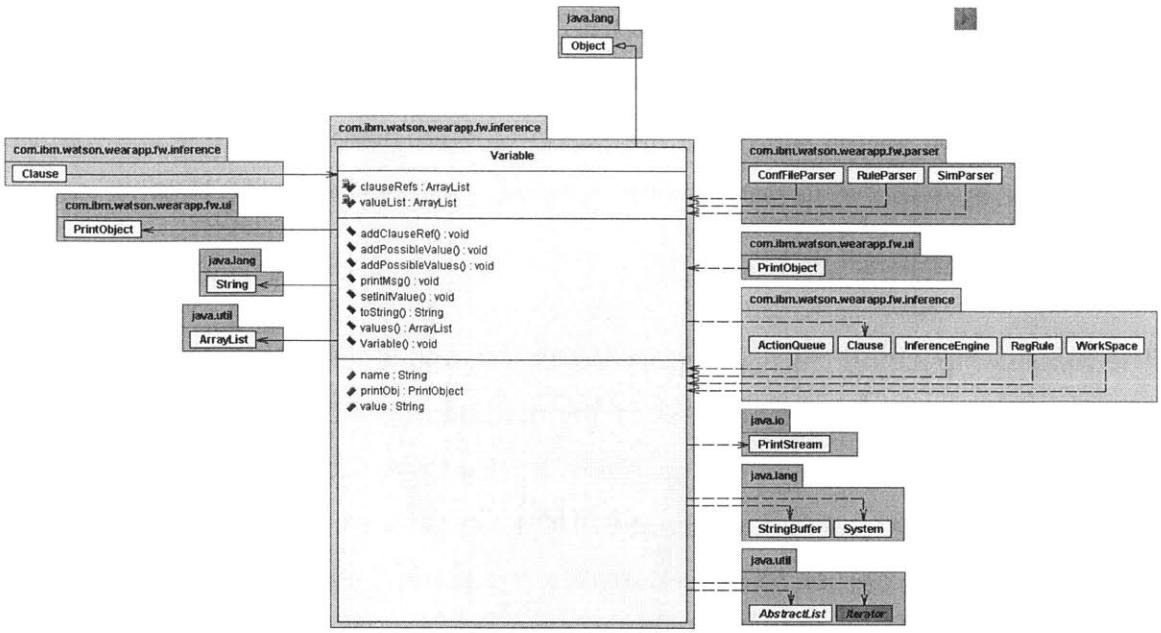


Figure 6 Class Variable

Conditions are used to compare two Variable instances and give a Boolean answer. For example, currently implemented conditions are: "equal to", "greater than" and "less than". Other conditions can also be implemented such as "earlier than", "older than".

A Clause contains a condition, a variable, a value for that variable and a Boolean Object. The reason that Boolean object is used in Clause instead of a boolean value is that a normal boolean can only have a value of true or false. A Boolean object can have value of true, false or null. The value of a clause is equal to the result of comparing the value held by the clause and the value held by the variable within the clause using the condition of the clause. See Figure 7 Class Clause. For example, we have the clause: (RoomID eq. R302). "RoomID" is a variable with name "RoomID". The variable "RoomID" contains a list of rooms that have an id. Room 302 is one of them. The id of room 302 is R302. "Eq" is the equality condition. “Eq” condition tests if a variable is equal to a value. The clause is evaluated to be true when the value of variable "RoomID" is equal to

"R302". Else the value of the clause is false. When "R302" is not a valid room id, the value of the clause is NULL. Clauses contain a set of references that enable each clause to know all the rules that use the clause. This set of references will be used in forward inference.

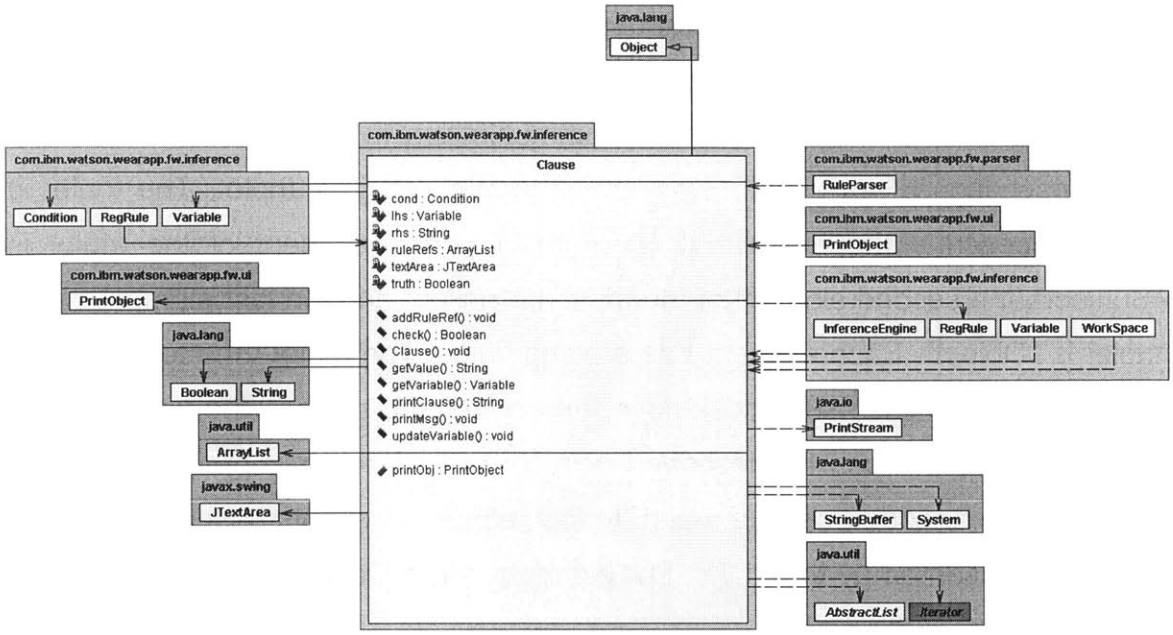


Figure 7 Class Clause

A rule consists of many clauses and one consequence. Clauses represent preconditions for the consequence. When all the preconditions are true then the consequence will be executed. There are two types of consequences. One type of consequence is Clause and another type is an Action. When the consequence is a clause then the value represented by the variable in the clause will be altered to the value held in the clause. If the consequence executed is an Action than then the inference engine will put the action in ActionQueue. ActionQueue will take over and make sure the action is triggered.

The prototype of Friendly Wizard does one inference at a time. Rules are fired only once in one inference. An action can trigger variables to change their values. In turn, because variables changed their values there might be more

actions triggered. Limits on the number of times an action can be triggered eliminate the possibility of deadlock. Because the number of rules is limited, the number of times actions can be fired is limited.

The inference engine starts inference when it receives events form Friendly Wizard sensors. Events are java objects that are sent over the network. A detailed description on how communication between the main engine and all the components will be explained in the communication module section. Here is a brief description of an event. Every event is encoded in a triplet. The triplet is used to describe the event that occurred. The first element of the triplet is "Subject", the second element of the triplet is "Action" and the last element of the triplet is "Object". Each element has a string value. The triplet tries to mimic the fact that most English sentences have three parts, the subject, the verb and the object. If an event can be described with a simple English sentence then that event can be encoded using a triplet. In fact, almost all events based on sensors can be represented by the triplet. For example, "Alan walked into the room" and "Rose is sitting in the sofa".

A typical action is triggered as following. When Friendly Wizard receives an event it first changes the value of the global default variables, which are "Subject", "Action" and "Object". These three variables correspond to the three elements in the triplet within the received event. When an event arrives at the main engine, the main engine will reassign the values from the event triplet to the three global variables. If the new value is different than the previous one, a variable will inform all the clauses that are affected by this change. A list of effected clauses is gathered. These clauses will check their truth-value. If the truth-value is affected by the change of variable value the clause will inform its change to all the rules that contain that clause as a precondition. A list of effected rules is gathered. As a result only rules that are affected by the incoming events are evaluated. When all the preconditions are satisfied in a rule, the consequent clause will be fired.

Here is an example. Suppose we have the rule that says. Rule 1: IF (Subject=Lisa) (Action=PressButton) (Object=MeetingRoom.) RecordMeeting (Subject=Lisa). Rule 2: IF (Subject=Lisa) (Action=Enter) (Object=MeetingRoom) THEN Greets(Subject=Lisa); Rule 3: IF (Subject=Sara) (Action=Enter) (Object=R302) THEN Greets(Subject=Sara);

Rule 1 says, if Lisa presses a button in the meeting room then the rule will start audio recording of the room. Lisa, in this case who triggered the action, can be passed in to the function RecordMeeting. This means the recording can be saved to Lisa's personal directory. Rule 2 says, if Lisa entered the meeting room the room will greet Lisa. Perhaps by displaying a welcome message or playing a piece of audio. Rule 3 says, if Sara enters room 302, room 302 will also greet Sara.

If we received the event: (Subject=Lisa) (Action=Speaks) (Objec =MeetingRoom). This means our sensor detected that Lisa is speaking in the Meeting room. Change value in variable "Subject", "Action" and "Object" will trigger the evaluation of clause (Subject=Lisa), (Object=Meeting Room). Notice these two clauses appear in both Rule 1 and Rule 2 and they are the same clauses. Because the truth-values of these two clauses have changed, the rules that contain these two clauses will be evaluated. In this case, rule 1 and rule 2 will be evaluated. However, the Clause (Action=PressButton) in rule 1 and the clause (Action=Enter) in rule 2 have false truth-value. Hence, none of the rules are fired. In this case rule 3 is not evaluated at all.

In order to only evaluate the rules that are affected, we need to know the linkage between variables, clauses and rules. This complicates programming and increases memory spaces needed for rule evaluation. However the plus side is rule evaluation is faster and more efficient.

One of the advantages of using a rule engine is that if-then else rules are easily understood and every piece of a rule can stand-alone by itself. Rules are easily manipulated compared to programming code. In the first version of Friendly Wizard there are no nested "if and else" loops. That means every rule stands alone by itself. However, the developer can add customized rule parsers to the rule engine to parse more complicated rule structures into flattened standalone rules.

One of the weaknesses of a rule engine is when a lot of rules are added to the system, the conflicts between rules arise exponentially. Simple standalone rules lose their readability. And the whole system becomes hard to manage. To solve the problem, Friendly Wizard follows a very strict order when firing rules. See the appendix for Rule Specification details. Friendly Wizard allows the user to group rules together. This way the developers are able to treat each group of rules as one unit instead of treating them separately. Within one group Friendly Wizard lets the user specify priorities to each of the rules to resolve conflicts within a group. If for some reason a rule needs to be fired immediately regardless of the group firing order, it can be assigned to a higher system priority.

At initialization, Friendly Wizard loads rules and variables from a rule file to the memory of a rule engine. When an event is fired, a message will be sent to the Friendly Wizard's inference engine. Upon receiving the message, the engine extracts specified variables' value from the message. Rules that are related to these variables will be put in a firing queue. The inference engine orders these rules in a predefined way. There is always a unique way to order any set of rules. After each rule is fired, the inference engine might bring more rules into the queues. As each rule is fired the actions associated with each rule are executed by the rule server using different threads. Actions can modify variable values in the inference engine. Developers can also specify how many times an action can be fired.

6.3 Communication Module

To support a distributed system, an event-driven message-passing scheme is chosen for the communication module. Under this scheme Friendly Wizard can be extended to a multi-computer distributed system.

Ideally we would like to model a human that senses event changes in the environment and makes a decision of what he needs to do. Short of this, the perfect model will be a rule engine-polling model. In this model the rule engine is constantly polling the sensors for their value and constantly drawing inferences from these variables and making decisions. Just like a human is constantly aware of his environment. However, in real life the sensors we use rarely change their value compared to the speed with which a computer can pull. For example, we can have a sensor that senses if a door is opening. It's not likely a door will be opened and closed every 10 milliseconds. It's more likely the door is going to open once every 5 minutes. If we pull the door sensor every 10 milliseconds we will create lot traffic on the network and waste a lot of processing power. It is reasonable to give the sensor the ability to signal the rule engine when an event occurs.

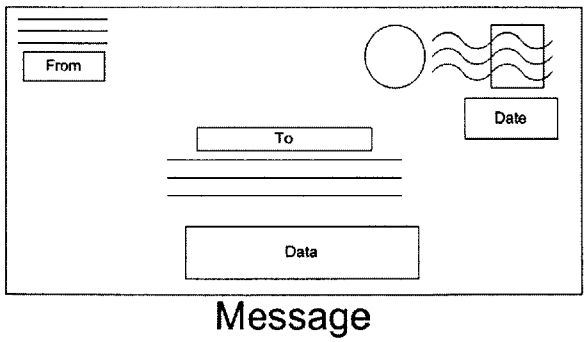


Figure 8 A message

A message-passing system will also be implemented to achieve a robust, scaleable and flexible system. Message passing systems are widely used in computer games where thousands of objects need to communicate with each other. Senders and receivers are encoded in the message. All messages can be sent to some special objects that process the messages and deliver the messages to the receivers. It is just like the US postal system where people can send their mail to the postal office and the postal office is responsible for the delivery of letters to other people. See Figure 8. The advantage is that every sensor in the system does not have to know where all other sensors are. On initialization, every sensor registers itself with a "Message Dispatch Table". It can then send and receive messages from other sensors. Most messages in Friendly Wizard will be messages that are sent from the sensors to the rule engine to indicate an event has been observed. Every message includes a sender field, receiver field, option field, time stamp and a data field. A data field can be any object in java. Developers can attach anything class in the data field of a message. Messages passed on a network might experience delays in the network. The time stamp in the message indicates the exact time when the interested event occurred. This can be used later to synchronize with all variables' time stamps so that correct variable value can be loaded to the inference engine of the Friendly Wizard system.

There are 5 classes in the com.ibm.watson.wearapp.fw.message package. They are Message, Messenger, EventMessage, SensorInitMessage and StringPair. See Figure 5 Class Hierarchy

Message class is a data type that holds all necessary information about a message. This information includes source of the message, destination of the message, when is the message sent, what object does this message contain and what kind of message it is. A message can be serialized to send across the network. There are five types of messages. Message receiver will decode each

message, especially the object within each message differently according to the message type.

The first type of message is DEVICE_INIT, device initialization message. A device sends this type of message when it is initialized. Whoever cares about a newly initialized device on the network, in this case the main inference engine, will look at this type of messages. The device initialization message contains information on what kind of device is it, where is it, what are some of its capabilities. Upon receiving such a message, the inference engine will decode this information and enable new rules that are specific to that device or enable user to load or write rules related to that device. For example, the Circular Recording Device will tell the inference engine that it is able to record 20 seconds of buffered audio and save it to a specific location. Now a user will be able to write rules such as, if Andy comes into the room then record 20 seconds of buffered audio to Andy's personal directory.

The second type of message is SENSOR_INIT message, sensor initialization message serves similar purpose. The difference between DEVICE_INIT message and SENSOR_INIT message is that the inference engine will parse the message and figure out what is the new information this sensor can provide to the user. For example, a door open or door closed sensor will tell the engine that now the user can get information on whether the door is closed or open from it.

The Third type of message is ENGINE_EVENT, engine event message. Sensors to the inference engine send this type of message. This type of message tells the engine that the state of sender has changed. Upon receiving such message the inference engine will perform a forward inference on all the loaded rules to determine if any action should be taken. If the change of state of a sensor, for example when a door is opened, caused a rule to be fired then the Inference Engine will send a TRIGGER_DEVICE, trigger device, message.

TRIGGER_DEVICE message is the fourth type of message. The corresponding device will be able to receive such messages and carry out specific action.

The last message type is GET_SENSOR_VALUE, get-sensor-value, message. This type of message is used to probe or when polling the state of a sensor. It is not used in current version of Friendly Wizard except during debugging state because the current version is completely event driven. However, in some cases where engine polling is more suitable to certain application, this type of message will be used. Basically the inference engine sends out a get-sensor-value message to a sensor and the sensor responds with another get-sensor-value message, from the sensor, which has a description of its state.

The messenger class is the heart of the communication module. It is responsible for providing an abstract barrier to higher-level applications. For example if a component wants to send a message to another, the component can simply pass the message to the messenger and the messenger will figure out the location of the destination component, in this case its network address, and makes sure the message gets delivered. On every machine or every Java Virtual Machine, there will be only one instance of messenger class. All components, for example, devices and sensors send and receive messages through the local messenger. The local messenger is responsible for communicating with other messengers. It's works just like the post office. Every town has a post office. The post office is responsible for delivering and receiving mail to its local community. It chooses the method which letters are delivered and makes sure all letters are delivered. The local messenger keeps a table of locations of other messengers. From the table it can figure out where each message should be delivered. When a new device that is on a separate machine is hooked up to Friendly Wizard, the Friendly Wizard main engine is responsible for announcing its presence to all messengers. Each messenger will be able to send and receive messages from the new machine. However in this version of friendly wizard, in order to reduce complexity, there is no direct inter-device communication. That means there is no

need for one device to talk to another directly. Inter-device communication is indirectly controlled by the inference engine, based on the rules currently loaded in Friendly Wizard. Direct communication between devices can save time and improve efficiency. The draw back is the system gets complicated and hard to scale.

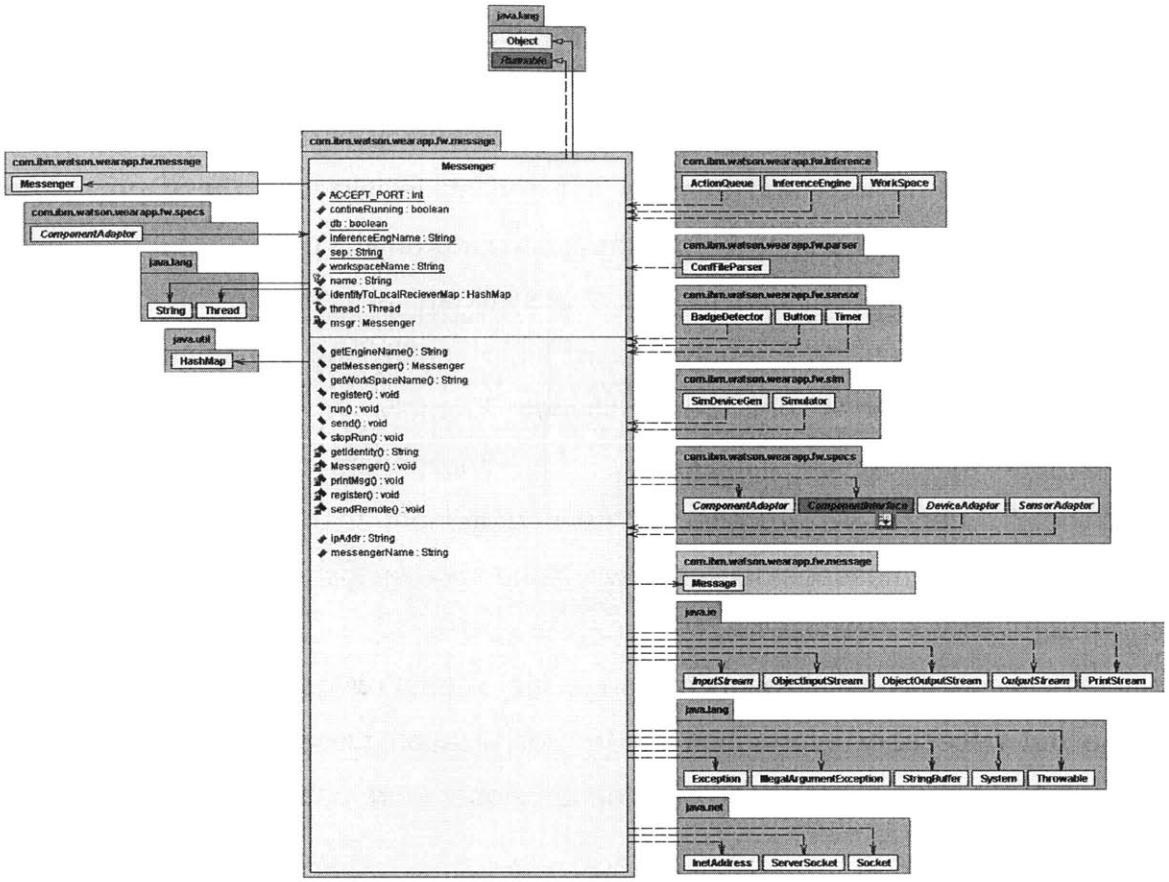


Figure 9 Messenger Class

Each messenger runs as a single thread on the host machine. Messenger acts like a server. It listens to all incoming messages. Once an incoming message is received, the messenger will accept the connection and create a separate thread to handle incoming messages. The messenger thread goes back and waits for more incoming connections. Messenger's class does not allow creation of duplicated messengers in one Java Virtual Machine. It is achieved by having a

private constructor for the messenger class. The only way to create a Messenger is to call the Static method, getMessenger, of the Messenger class. Once it is a Friendly Wizard component, the messenger can use the send message method of the messenger to send a message.

When a device or a sensor is first initialized, it is required to register itself to the local messenger. The local messenger will update its list of all registered local devices and sensors, which is another table that the messenger maintains. Then the device will be able to communicate with the Friendly Wizard main engine. It is required for all devices and sensors to send an initialization message to the Friendly Wizard main engine before they can send anything else. After the main engine receives the initiation message a symmetrical communication route is established between the Friendly Wizard main engine and the Friendly Wizard component. Messages delivered between Friendly Wizard and this new component will now be handled by local messengers that reside in each machine. If Friendly Wizard and this new component share the same physical machine, then all message deliveries will be handled by that messenger alone.

After receiving the initialization message the Friendly Wizard main engine will parse out necessary information for the initialization message, categorize the Friendly Wizard component and give the developer more options to write rules.

Here is an example of this process. Suppose a door sensor is started on the network. It is able to sense the state of a specific door. When a device is initialized, it calls the static method getMessenger to obtain a handle of the local messenger. Then it sends out the initialization message to the Friendly Wizard main engine. The messenger receives the request for delivering the initialization message. It checks for the sender to see if it is registered. If the sender is not registered it will peek into the initialization message and figure out enough information to register the device locally. Then the messenger will deliver the message. The initialization message describes what the door sensor can do,

what are the states it provides. It tells the main engine that the sensor is online now and ready for information. The initialization message of a sensor contains what information it can provide by providing different combinations of three words. The initialization message provides three arrays of strings represent an event triplet.

Every Friendly Wizard Component has a unique identity string. The identity string is generated based on three criteria: the name give to the component, which is easily readable by humans; the component count with in the Java Virtual Machine, which can be used by a computer to identify a component locally; the component and the network address of the physical machine that hosts the component, which can be used to resolve the component globally. When a Friendly Wizard component wants to send a message it will get the messenger by calling the static getMessenger method from Messenger class. Then it will call send message method on the messenger. The messenger resolves the destination by checking the destination component's identity string. The messenger then delivers the message to the destination. If the delivery failed the messenger is able to send a special message back to the sender to indicate the failure. Again, Friendly Wizard engine, devices and sensors are all Friendly Wizard components.

6.4 I/O Processing Module

This IO processing module has three major tasks. First, the IO module parses Regular rules into variables, clauses and rules. Second, the IO module parses configuration files for the main engine. Last, the IO module parses Simulation rules for the simulation module. The three major tasks are accomplished by three classes in the com.ibm.watson.wearapp.fw.parser package. See Figure 10 for

relations between parser package and other packages. Parsers are isolated as another package enables developers to supply their own parsers. Parsers agree with other modules, for example the main engine, on the format of its output. As long as the developer supplies parsers that output the same format he can use his own parser. The developer can make his own rule input syntax. He can also add his own configuration parameters while not having to worry about other modules. The three parsers supplied are simple but effective. Just to show how parsers can be different: Configuration parser and Simulation demonstrated how to parse XML files. RuleParser demonstrated how to parse a format specific file.

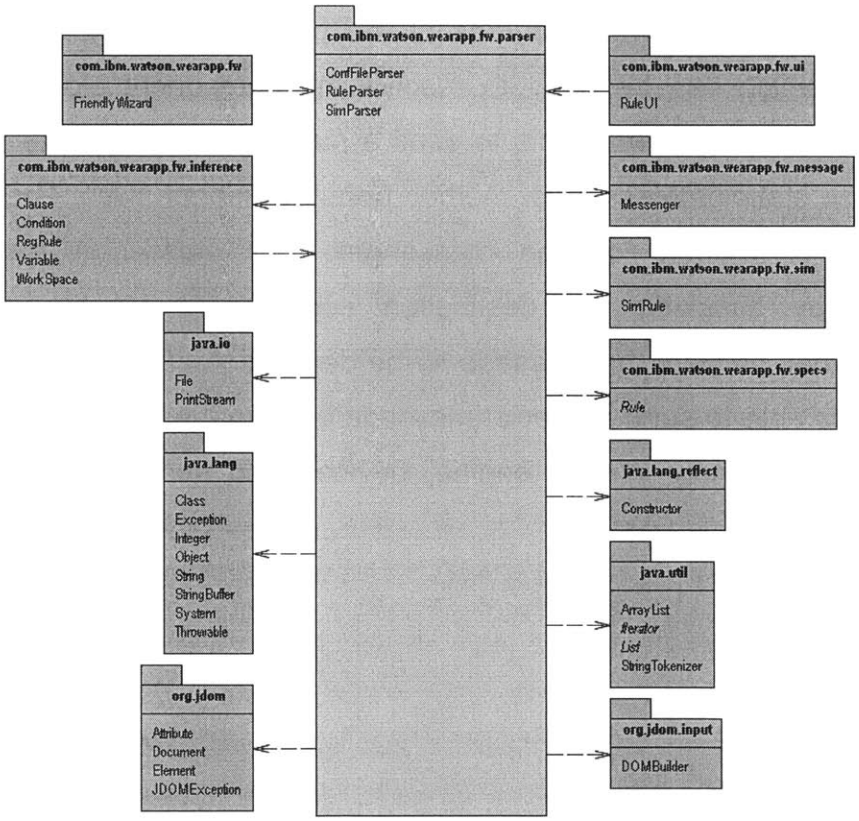


Figure 10 Parser Package

Configuration file parser and the Simulation Rule parser parse an XML file. XML parser JDOM is used to parse XML. There is no particular reason that JDOM is

used other than it is a JAVA XML parser. As stated above the developer can make their own parser hence chose whatever tools they want.

The syntax of the rules for the prototype is fairly simple. The keywords are IF, AND, OR, THEN. Arguments that were parsed to the consequent are separated by ":". Here is an example:

Rule2 IF Subject=Sara AND Action=Enter THEN GreetPerson:Action:Subject".

"Rule2" is the name of the rule. It is used to identify the rule. Names are unique to a rule. They can be automatically generated. Rule names enable Friendly Wizard to communicate with the user. "Subject=Sara" and "Action=Enter" will be parsed as clauses. Everything after the keyword THEN is consequent. In this case "GreetPerson" is an action that is registered with the Friendly Wizard main engine. "Action:Subject" are the arguments that will be passed to the main engine. In this case, "Sara" will be passed to the registered action in place of Subject. "Enter" will be passed to the registered action in place of Action.

Configuration contains information that is used to configure the Friendly Wizard main engine. There are three sections to configuration file. They are variables, devices and sensors. See Appendix A for an example of configuration file.

Normally, when a Friendly Wizard sensor registers with the Friendly Wizard main engine, the main engine creates variables inside its workspace according to what information the sensor can provide. However, sometimes this is not enough. For example, a developer would like to the GreetPerson action to greet a person coming to the room as well as display how many people are already in the room. The developer can either make a Friendly Wizard sensor that keeps track of how many people have entered and left the room. Or the developer can, as we previously discussed, write a rule with a clause as consequent. In this case the developer needs to create a variable that holds the value of how many people

are currently in the room. Hence, the developer needs a way to input variables directly in to friendly wizard rule engine. Of course, he can create the variable when he enters the rules but he can also create variable inside configuration file.

The sensor section and device section of the configuration file have similar functions. Although in most of the cases the sensors and devices start by themselves and they will register with the Friendly Wizard main engine, a developer can use these sections to ask Friendly Wizard to initialize and start sensors and devices whenever the configuration file is parsed. To start a Friendly Wizard component, the friendly wizard engine needs to know the name of the component, a string description of the component and where to locate the class file that starts the component. The prototype version of the Friendly Wizard main engine is only able to start sensors and devices that are connected to the same physical machine.

Simulation parser also parses a XML file. See appendix B for an example of simulation XML file. There are four sections of the simulation file, variables, devices, sensors, and simRules. The variable section of the simulation XML file is exactly the same as the variable section in the configuration file. The devices and sensors sections of the simulation XML file are similar to the devices and sensors section of the configuration file. What is different is that these devices and sensors are virtual devices and virtual sensors. The simulation XML file contains different information about devices and sensors. For example, instead of knowing what class file to use for a device, the simulation engine only needs to know the name of the device. On the other hand, knowing what class file to use is not enough for a sensor because the simulation engine has to inform the Friendly Wizard main engine what the sensor dose.

Just like regRule, every simRule has a name. SimRule also contains the name of the device it is simulating. Each simRule corresponds to an event that will occur

in the future. It has a triplet record of what event will be occurring and a WaitTime field that records when the event will been simulated. For example:

<rule name="rule2" sensorName="SenseStuff" waitTime="5" subject="brad" object="R302" action="Enter" />

"Rule2" is the name of the simRule. This simRule says, at time slot number "5", i.e. 5 seconds in to simulation, the Friendly Wizard will receive an event "Brad enter R302" by a sensor named "SenseStuff".

SimRule and RegRule are subclasses of Rule. The Rule class is placed under package com.ibm.watson.wearapp.fw.specs because it is an abstract class.

Workspace plays a very important role in the background of the IO Processing Module. As we mentioned before, Workspace is the place that all the data and structures for the inference engine is stored. The main function of Workspace is to supply information for the inference engine. Rules, variables, clauses are stored in separated lists. Workspace makes sure that variables, clauses, and rules correctly reference each other. The simulation engine also resides in the workspace. The purpose is to hide the simulation engine from the developer.

Besides providing services for the rule engine and simulation engine, the workspace is also a housekeeper. It knows and is responsible for every detail in the system. It keeps records of the sensors that are currently online: What state information can each sensor provide? Which rule parser to use? Which simulation parser to use? What are devices available to perform an action? What information do these devices need to perform its action?

The workspace only acts as an information provider. Other classes get information from the workspace by calling different functions of workspace. For

example, getSensorList() function returns a list of currently available sensors. AddOneRule() function adds a new rule to the workspace.

6.5 Simulation Module

There are three classes in the Simulation Module. SimDeviceGen class, SimEngine class and SimRule classes.

As we have seen in the IO processing module. Simulation rules consist of the rule name, sensor name, TimeToFire and a triplet that represents what event will be sent to the Friendly Wizard main engine when TimeToFire arrives. When a set of simulation rules is loaded by a developer, the rules are parsed and stored in Workspace. SimDeviceGen will generate new Friendly Wizard sensors if necessary according to the simulation rule loaded. For example, if the simulation rule uses a real device but phony message, virtual sensors do not need to be created. If the simulation rules use sensors that have not yet developed, virtual sensors will be created by SimDeviceGen. SimDeviceGen makes sure every new sensor has a unique name and is correctly registered with messengers on the local machine. For example, if we want to simulate an event from a door sensor, SimDeviceGen will make sure the phony message is sent to the inference engine as if it is coming from the door sensor by temporarily registering itself as the door sensor and sending the message. Once all the simulation rules are in place, SimEngine runs as a separate thread to ensure accurate timing. Simulation engine arranges rules in the order of TimeToFire and fire events one by one at the right time.

There is a graphical user interface developed to help the developers use the simulation engine. For detailed information see chapter Using Friendly Wizard.

6.6 Debugging Module

Debugging Module is located in the com.ibm.watson.wearapp.fw.ui package. The debugging module is basically a log file, but a log file with intelligence. There are two levels of intelligence of the debugging module. The prototype only implemented the first level.

The first level of debugging intelligence comes form the forward inference engine. As described in the inference engine module, forward inference is done using a data structure that consists of variables, conditions, clauses and rules that all have pointers to each other. This data structure enables Friendly Wizard provide a high level description of what is going on. This is an ability inherited from knowledge-based systems.

A knowledge-based system is able to ask users simple questions and give the user a suggested action. For example, a person cannot start his car so he uses a car-wizard knowledge based system to help him diagnose the car. The system can ask the person questions such as, "Did the engine lights turn on when you turned the key?", "Did the battery light turn on when you turned the key?" and "Did you drive last night?" When the person has finished answering the questions, the car-wizard knowledge based system might tell he that he might have forgotten to turn off the light yesterday when he last drove his car." The system might also tell the person that because the last time he drove was around sunset, he might have had his lights on but did not notice when he left his car.

Friendly Wizard acts more or less like the car-wizard system. They make decisions of what questions to ask and what actions to suggest based on their internal data structure and how they are programmed. However, these systems can translate the data structures within to some form of sentences that human

can understand. The first level of Friendly Wizard's debugging intelligence uses this characteristic in the knowledge-based system to make a higher-level log file. Instead of traditional Web Server logging where mostly numbers are recorded, the Friendly Wizard log system can provide information higher-level in string format. For example, you might see the following in the log file of Friendly Wizard:

"On July 23rd. 3pm. “Meeting Room Door” says Door for “the Meeting Room” opened; ID sensor for Bill says Bill is in the area around the meeting room; according to Rule4, I ordered the greeting board to display greeting message for Bill in the meeting room."

The second level of debugging intelligence is to enhance this feature by adding filters on top of the log file. The reason is if many rules are fired, even with high-leveled human readable log files, a developer still has to read thousands lines. Filters on the log file help the developer see and search for what he wanted. For example: if the greeting display message was not able to display correct message, the developer might want the log file to display any information that is related to the greeting display device only. The developer might also want to choose the level of detail of the log file. For example, instead of wanting to know why Friendly Wizard made a decision on the per-rule level, the developer might want to know when a rule was valid to fire on a per-clause level. In other words a developer might want a message like this:

"On July 23rd. 3pm. Rule 4 is filed because, (Subject=Bill) (Action=enter) (Object="MeetingRoom");"

The developer can in the future inquire why (Subject=Bill), the Friendly wizard will respond:

"On July 23rd. 3pm. Id sensor send event: (Bill; Enter; R302);

7. Using Friendly Wizard

7.1 System Configuration

Configuration specifications are stored in a text file. The developer can easily read and change the system configuration. Friendly Wizard can also dynamically load up configuration files and configure itself. This ability helps developers debug their applications quickly and easily without shutting down the system. Variable values in the configuration file can be edited using any text tool. However, one difficulty is that a text file is hard to parse. There is a trade off between flexibility of text input and complexity of the text file parser. As a result, Friendly Wizard uses an XML parser, JDOM, to parse XML configuration files.

The configuration file specifies: locations of classes for all the sensors are stored so they can be loaded dynamically; locations of the rule files that need to be loaded in to configuration; name and values for certain user defined variables. For each device/sensor, the configuration file lets the developers define its attributes like name and constructor parameters. Later on, configuration file might even support call back functions to enable developer add customized code in initialization.

A description of the structure of the configuration file can be found in the IO processing module section. Here is an example of a configuration file: (See appendix A for a detailed example of configuration file.)

```
<fw>
 <varibles>
  <varible name="rooms" init="20-153">
   <value name="Jeff's room"/>
```

```
    <value name="20-153"/>
    <value name="noi's office"/>
   </varible>
  <sensors>
   <button>
    <properties name="a button" className ="Button" sim="true"/>
    <location loadpath="com.ibm.watson.wearapp.fw.sensor"/>
   </button>
  </sensors>
  <devices>
   <displayMessage>
    <properties name="display message" className="DisplayDevice" sim="true"/>
    <location loadpath="com.ibm.watson.wearapp.fw.device"/>
   </displayMessage>
</fw>
```

The first section of configuration file is the variable section. The variable section lists a set of variables that are initialized by the engine. These variables can be used as a temporary value that holds values from result of an action. A variable element in variables section are enclosed by <variable></variable> tag. Each variable must have a name and an initial value. There parameters a specified as attribute to <variable> tag. I.e. <varible name="rooms" init="20-153">. A variable also contains a list of possible values. Each value is identified by the tag <value> attributes to the <value> tab are name of the value. I.e. <value name="true" />. When a variable is specified in the variable section, Friendly Wizard main engine will initialize their variables whenever it is reconfigured.

The second section is sensors section. Sensors section contains a list of sensors enclosed by tag <sensor></sensor>. Each tags within sensor section becomes a sensor. The name of the tag becomes the name of the sensor it represents. For example, the tag <button></button> tells us there is a sensor called "button" and information about that sensor is enclosed in <button></button> tags.

Friendly Wizard uses the reflection Java package to start sensors. In order for Friendly Wizard engine to initialize a sensor, the Friendly wizard must know the name of the Java class to initialize and the location of the Java class. Location of the sensor class is specified in Java package format. Hence, the path of the sensor class must be in the ClassPath environment variable when Friendly Wizard is initialized. The location of the sensor is indicated by the tag <location> and attribute "loadpath". For example <location loadpath= "com.ibm.watson.wearapp.fw.sensor"/>. The name of the sensor class is specified in the <properties> tag with attribute "className". For example, <properties name="a button" className ="Button"/>. Besides the name of the sensor class, Friendly Wizard also requires a simple description of the class. The description is assigned to the "name" attribute of the properties tag. Friendly Wizard can use this description to create a more meaningful debugging message.

The devices section is very similar to sensors section. Friendly Wizard can also initialize device classes. Later, Friendly Wizard will also be able to initialize sensors or devices over the network or at least try to send a message to a remove machine to start the sensors or devices. The prototype of Friendly Wizard cannot initialize sensors or devices over the network.

7.2 Connect to Friendly Wizard

Friendly Wizard connects to developers' code through java interfaces and abstracted classes. Five major interfaces in Friendly Wizard and their designated functions are listed in the table below.

DeviceInterface.java An interface that enables developers to write their own sensor controller that can be integrated into Friendly Wizard.

ConfFileParser.java An interface that can be used to build customized configuration file parser.

RuleParser.java An interface that can be used to build a customized rule parser.

ActionTable.java An abstract class that enables the developer to connect their action functions to the Friendly Wizard. OnInit() function in ActionTable must be implemented to load values for each variables in the rule engine at the beginning of each inference.

Action.java An interface that specifies the requirement on actions that developer wants to trigger when rules fire.

DeviceInterface.java class specifies how developers can connect their software with Friendly Wizard. An abstract class Device.java has been implemented to give the developers a quick start with their sensors.

Device.java can be extended by developers to customize classes. Device.java has all the functionality DeviceInterface.java requires, namely every sensor needs to be able to register to a message dispatch table. It also needs to be able to receive and send messages.

ConfFileParser.java specifies an interface for Friendly Wizard configuration file parser. A default parser is supplied with Friendly Wizard. The purpose of the configuration file parser is to give developer the option of writing their own configuration file parser to parse their preparatory configurations. For example developers can combine the configuration file of any part of the system they are developing with the configuration file of Friendly Wizard. For detailed information on how configuration files are parsed by Friendly Wizard please see the configuration module section.

RuleParser.java enables the developer to change the syntax of the rules that can be used in Friendly wizard. Developers can create a stricter set of rules that have a specification that is a subset of the current specifications. A rule parser can also increase the functionality of rules in the system. For example, a developer can create a rule parser that parsers nested IF THEN ELSE rules and flattens them into the rules Friendly Wizard requires. This way the developer can build more complicated rule structures with better syntax.

ActionTable.java and Action.java are java classes that specify what is needed for an action, such as a call back function, to be enabled in the rule engine. The configuration file tells the rule engine where all the actions classes are located, the rule file tells the rule Engine what the names of actions are that will be used. ActionTable.java is a place to store all the actions. It also handles updating variable values for the inference engine.

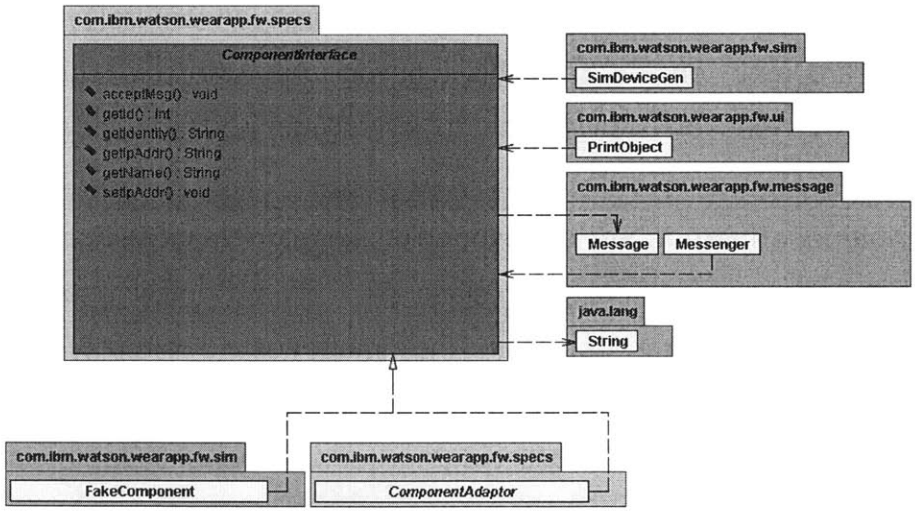


Figure 11 Component Interface

Friendly Wizard enables the developer to integrate sensors and devices to friendly wizard by implementing some interfaces. This restricts the developer when designing specific function calls but reduces the chance of error and

increases the flexibility, compatibility, scalability and stability of the whole system. This way Friendly Wizard can build another layer of abstraction to free itself from dealing with developers' code. And lets the developers be more focused on how each component will work instead of how everything works together.

There specifications are all under the "spec" package. It was called interface but interface is a keyword in java. The interface all Friendly Wizard components have to implement is the Component Interface. See Figure 11 for Component Interface UML Diagram. Basically every component has to be able to accept messages. It must have an ID. It also has an identity string that uniquely identifies the component itself. It knows the network address of the machine it runs on. It must have a name for a developer to identify it.

There are two sub interfaces for the component interface. They are Device Interface for devices, basically those components that carry out actions; and Sensor Interface, basically those components that provide information on current state of the environment. Device Interface requires an onFire() function call. Friendly Wizard main engine calls this function when the rule engine fires an action. When this function is called, the device will carry out specific tasks.

For each interface there is are corresponding adaptor classes. For example for ComponentInterface.java there is a class file called ComponentAdaptor.java. Adaptor classes are packaged with the interfaces to simplify developer's experience. Each adaptor class is a minimum functional implementation of the corresponding interface class. UML diagram for ComponentAdaptor is show in Figure 12.

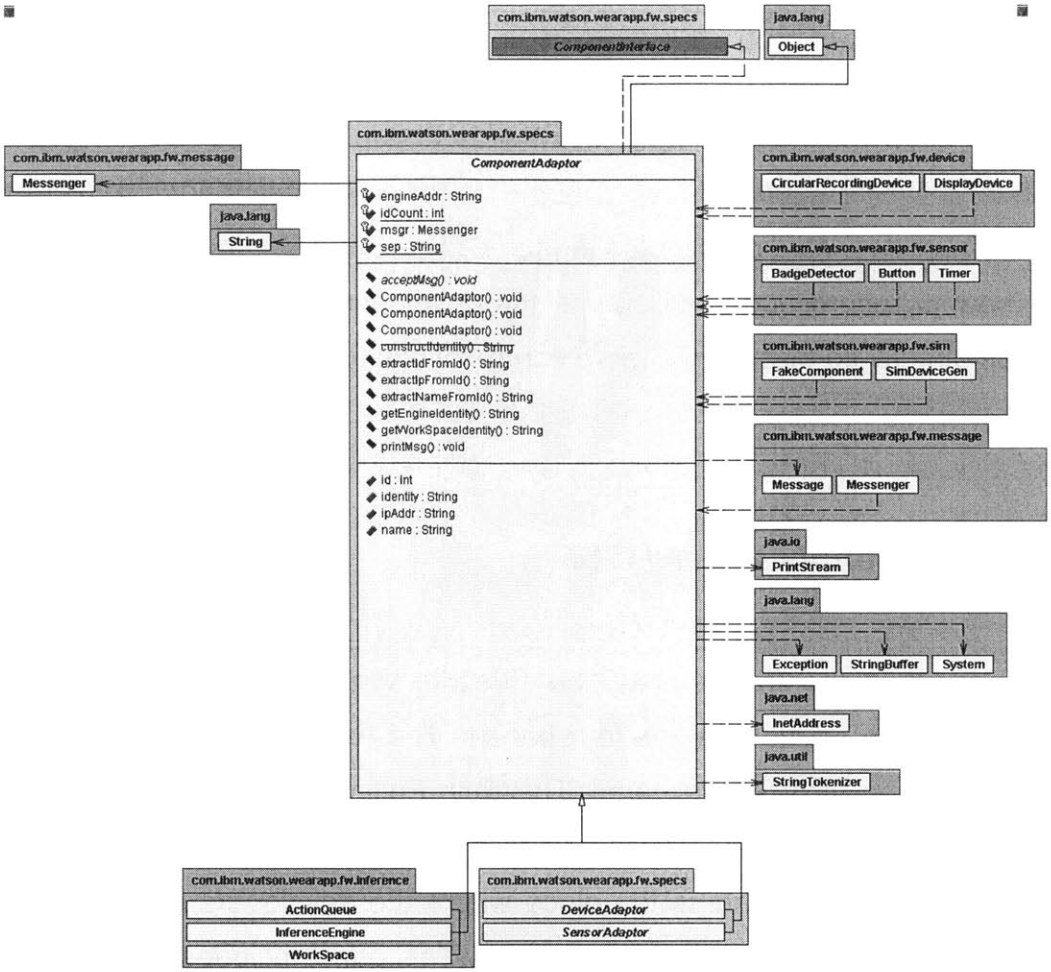


Figure 12 Component Adaptor

For example a SensorAdaptor represents a Friendly Wizard sensor that does nothing. In other words if the developer used an instance of SensorAdaptor without any modification, he is able to plug this instance right into Friendly Wizard toolkit. When SensorAdaptor starts, it will send an initialization message to the Friendly Wizard main engine to register itself as a sensor that does nothing. The developer will be able to identify the sensor by its name but not able to use it. However, the developer can extend the function of SensorAdaptor to make a new sensor. This way the developer can concentrate on coding what the sensor does instead of worry about how to meet all the requirements by the Sensor Interface.

The developers can also make something working in minutes. To make a device that records audio, the developer only needs to implement the OnFire() function and extend DeviceAdaptor Class. OnFire() is called by the Friendly Wizard main Engine when an recording event is triggered. The OnFire() function only needs to know how to record audio. Of course the developer can always implement the interface directly so that he has more control of what his sensor does.

7.3 Using Friendly Wizard GUI

Developer UI is the front-end of the Friendly Wizard main engine. It has a graphical user interface that has four panels. The RuleEditing Panel, Simulation Panel, Log Panel and Status Panel. The Rule Editing Panel is used for loading, viewing, inputting and editing rules plus displaying any information that related to rules. The simulation panel is the place where the developers can simulate his ideal setup before he actually implements it. This panel includes anything that is related to setting up and viewing a simulation. The log panel is responsible for keeping and displaying a log of events that is generated by Friendly Wizard. Future versions of Friendly Wizard will be able to control the level of detail of the log displayed. The purpose of the log is providing the developer with a way to evaluate the quality of the rules and the interaction between devices. More importantly, the log panel provides very important debugging information when the system that has been developed does not act as intended. For example, the log panel displays why a rule is triggered or why a rule is not triggered, what are the conditions that triggered a certain rule, what are the messages it received and what are the messages it sent out. The status panel is just another tool for debugging. It displays the current status of the entire system by sections. For example, what are the devices on the network, what can they do, what are the sensors on the network and what information can they provide, where are all

components located, what rules have been loaded to the current system, what are the elements of these rules.

7.3.1 The Rule Panel

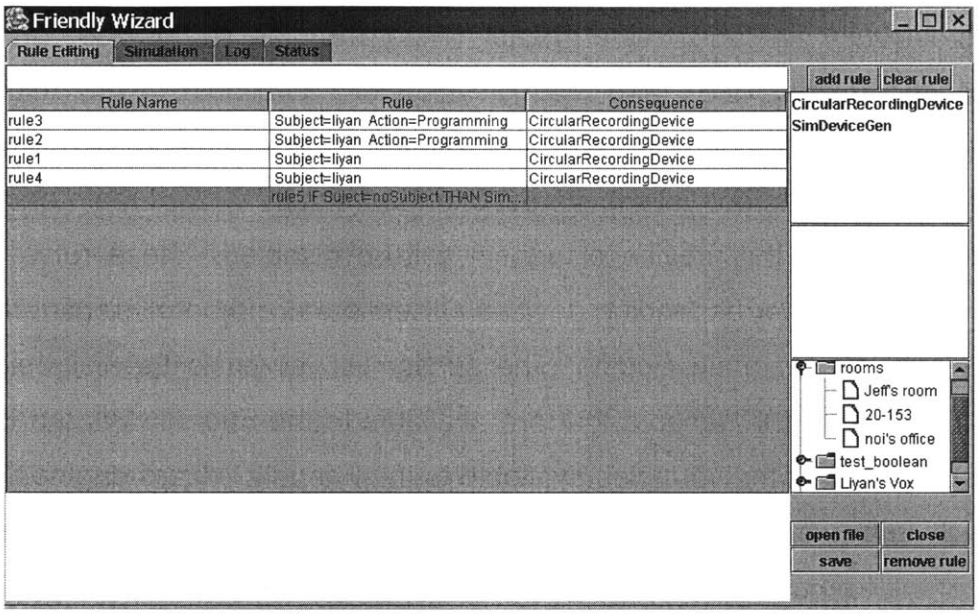


Figure 13 The Rule Panel

The center Table is used to display rules. There are three columns to the table, rule name, rule and consequence. If a rule is parsed without any mistakes, the name column will display the name of the rule, and the rule column will display clauses or preconditions of the rule and the consequence column will display the action that will be taken if the rule is fired. The user can edit the rule directly within the table. If a rule is parsed incorrectly, the incorrect rule will be displayed in the rule column. The row will have dark gray background color and red foreground color. The developer can edit the incorrect rules within the table. Incorrect rules are not filed.

There is a white message panel at the bottom of the table. Messages related to rules will be displayed in side the message panel. For example, an error message will be displayed in the message panel if the incorrect rule is highlighted. The message will give the developer some suggestions of what is wrong with the rule.

On the right side of the rule table, there are three lists. The top one lists all the devices currently available to the Friendly Wizard Engine. In this case, there are "CircularRecordingDevice" and "SimDeviceGen", available. The rule table disables rules that use invalid device or sensor names. If the device or sensor does not exist or has not yet registered with the Friendly Wizard main engine, all the rules that use that device or sensor will be disabled. These rules will be enabled when the corresponding devices or sensors are properly registered with the Friendly Wizard main Engine. The sensor list, which is the list below the device list, lists all the sensors that are available to the Friendly Wizard Engine. The last list lists all the variables available to the Friendly Wizard engine. The list is actually in tree structure. Names of variables are the branches of the tree. The leaves of the tree are the possible values for each variable.

The purpose of these lists is to help the developer to write a rule. Developers do not need to remember things like what are the names of a light sensor in the room two doors down. If the light sensor is running and properly registered with Friendly Wizard engine, then its name will be show in the sensor list. This feature is very convenient and often used when there is a typo in the names of a device, sensor or variables.

On the very top of the Simulation Panel there is a text field the developer can use to enter the rules from keyboard. On the right side of the text field there is a "add button" to add the rules the developer just entered to the rule table. The "clear rule" button clears all the text in the text field.

There is a control button panel located at the lower right hand corner of the rule panel. "Open file" button lets the developer choose a text file to parse. The "close" button will close the current rule file and the "save" button will save all valid rules in the rule table to a file. "remove rule" button will remove the currently highlighted rule from the rule table.

7.3.2 The Simulation Panel

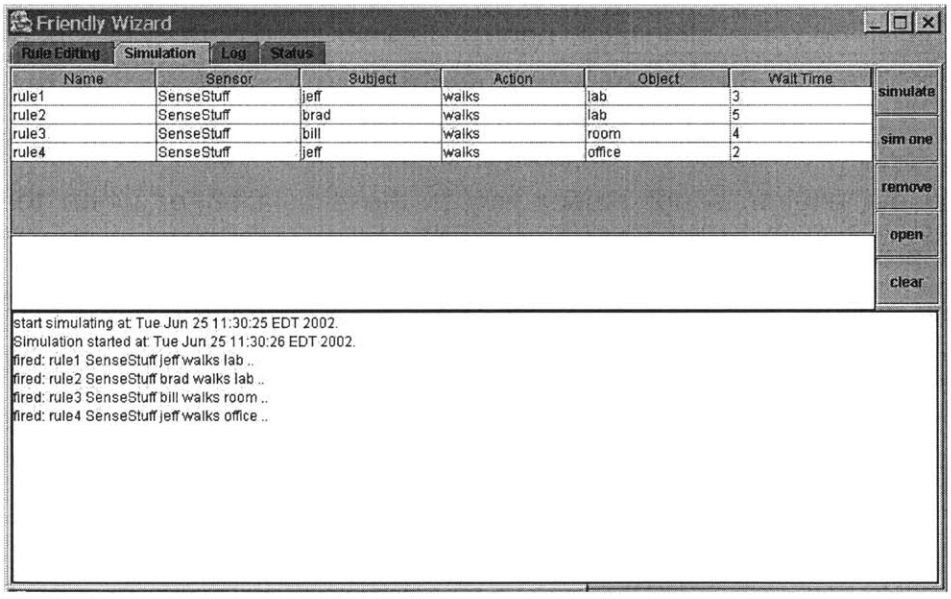


Figure 14 The Simulation Panel

The simulation panel also has a simulation rule table that displays six fields of simulation rules. "Name" column indicates the name of the rule, the "Sensor" column indicates the name of the sensor that the Simulation Engine is simulating. "Subject", "Action" and "Object" represent the event triplet. Wait time is the time

to wait before the simulation engine sends the event triplet to the Friendly Wizard main engine.

The text area at the bottom of the Simulation rule table is used to display messages that are related to the simulation. For example, an error message can be displayed while parsing simulation rules. The buttons on the right side of the simulation rule table are the control buttons for the simulation panel. The "Simulate" button causes the simulation to start. "Sim one" button causes the highlighted rule in simulation table to be simulated. "Remove" button removes one row from the simulation rule table. "Open" button opens a pre-written simulation rule and the clear button clears all the rules in the simulation table.

The bottom half of the simulation table is the simulation status display area. As simulation started, the simulation engine displays its progress in this area. For example, whenever a simulation rule is fired, Errors that occurred during the simulation are also displayed in this text area. For example, when the Friendly Wizard main engine is not responding to the simulation or when there is a network problem, the developer can restart the simulation by clicking on the "Simulate" button.

7.3.3 The Log Panel and The Status Panel

Log panel and Status panel are display boards that display different information about the Friendly Wizard main engine.

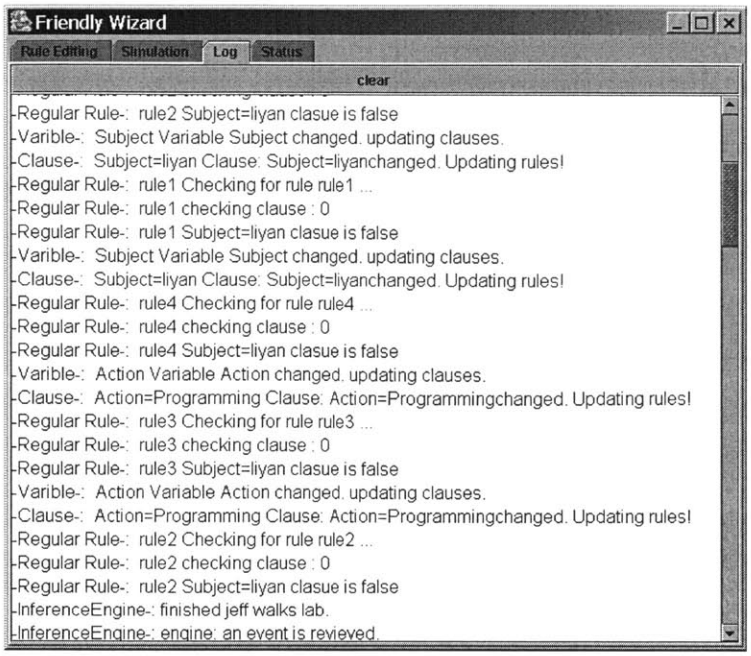


Figure 15 The Log Panel

The log panel displays a log file of the Friendly Wizard main engine. A log file includes every decision Friendly Wizard makes upon receiving an event. A developer can use this log file to examine the steps the Friendly Wizard main engine took during the inference process. Log Panel displays a log message in a human readable form. For example, the message "Subject Variable Subject changed. Updating clauses." Means the subject field in the event triplet changed the variable "Subject". As a result, the rule engine is checking any clauses that might be affected by the change.

Figure 16 The Status Panel

The status panel displays static information about the inference engine. For example, detailed information on all sensors and devices that are currently connected to the Friendly Wizard main engine. The detailed information includes: the IP address of the physical machine, a sensor or a device that is on. When is the last time the sensor and device communicated with the Friendly Wizard main engine? How long has a device or a sensor been on the network? Other information includes: what are the rules available? How many times each rule has been fired. What are the variables available? What are the clauses that use these variables?

The log panel and the status panel help developers to put their systems together as well as debugging their systems. More features can be added to make these two text panels more interactive. For example the filter feature described above in the debugging section. However, in the early version of Friendly Wizard only text panels are provided to the developer.

8. Future of Friendly Wizard

8.1 Updates

By evaluating the prototype of Friendly Wizard, two updates of Friendly Wizard can be performed to increase it efficiency and flexibility.

The first problem is that network actions consume a lot of processing. This is because every time a message is passed between the Friendly Wizard main engine and its components a socket is opened and closed. This scheme works well when sensors vary slowly. For example a door night open once in 5 minutes. However, for a sensor that samples a target that changes very frequently, much processing power and overhead will be used to setup and shutdown each connection. One solution to this problem is to implement a persistent socket in the communication module. Devices or sensors will use the persistent socket to connect to Friendly Wizard, or at least have the option to use the persistent socket to connect to the Friendly Wizard main engine. For example, every component of Friendly Wizard should open one connection to the Friendly Wizard main engine and keep this connection open until it dies. This means the messenger classes will only be used when a component has just started or a connection between a component and Friendly Wizard main engine is dropped. This will reduce over 90% of the overhead for the communication module.

The second problem is the communication module is complicated and needs a layer of transparency to better aid developers to implement Friendly Wizard components using different computers. In the prototype, to start a device on a Friendly Wizard network, the developer has to know the physical IP address of the Friendly Wizard main engine. In order to determine whether an event is

routed to the Friendly Wizard correctly, a developer has to look in the Java code for Messenger class. As a result, an abstraction layer needs to build on top of the Messenger class to provide enough flexibility to the developer and an abstract barrier over the Messenger class. One solution to this problem is to use RMI and develop a broad case protocol.

RMI is Java Remote Method Invocation. This is a standard way to abstract out network communication between Java objects on different machines. RMI uses stubs to hide away network implementations between two java objects. This way, developers no longer need to seek into the Messenger Java class in the communication module for communication problems. The stubs can also implement a network broadcasting protocol so that a stub can query the network for an IP address of the Friendly Wizard main engine. This way a developer can truly not worry about any code other than the part that makes his sensor or device work.

8.2 New Features

The log panel and static panel of the Friendly Wizard can be improved. A filter, or rather a complicated filter can be placed over the log message. This filter will help the developer to sort out the information he wanted for the Friendly Wizard main engine. The developer should be able to adjust the amount of detail that is reported by the filter. For example, a developer can ask, "display for me all information about the door sensor #1 between 3:30pm and 4:00pm today. I want the information on decisions the Friendly Wizard engine made only." A even more complicated sensor can answer the following question, "Why did the Friendly Wizard engine not fire rule#1 yesterday at 3:00pm." Knowing why the

Friendly Wizard did not do something is often more important than knowing why it did something.

The second improvement in the future can be making the Friendly Wizard a truly distributed system. Currently Friendly Wizard main engine is located on one machine. If the machine is powered down the whole system will not function. If Friendly Wizard main engine is allowed to locate on different machine, two solutions exist to make Friendly Wizard a distributed system. The first solution is each Friendly wizard main engine can synchronize its rules with the result of the Friendly Wizard main engines. Events will be dispatched by the messenger class to different Friendly Wizard main engines. The second solution is rules will be divided between each Friendly Wizard. Events will be passed to every Friendly Wizard main engine, but only one Friendly Wizard will perform the entire inference. Other Friendly Wizard engines will abort the processing once it discovers that it does not have the correct rule.

9. Future Development

There are three directions in which friendly wizard can be improved. These directions are not included in the original description of Friendly Wizard. Each of the directions has advantages and disadvantages. These three directions will be discussed here to give the developer some ideas on how Friendly Wizard can be further modified. These three directions are: building Friendly Wizard using agent technology, modifying Friendly Wizard's rule engine to be a fuzzy logic rule engine, and using a Bayesian network in place of Friendly Wizard's inference engine.

9.1 Friendly Wizard Based on Intelligent Agents

9.1.1 What is an Agent?

The term Agent can be used to describe a piece of software component that has certain capability such as a web form wizard or a network music finder. This piece of software can perform certain tasks by itself. It can also collaborate with other software or agents to perform more complex tasks.

Agent oriented programming originated from research in distributed artificial intelligence. Because an agent is only a piece of software component, it can be platform independent while the software running the agent may be platform dependent. Multiple agents can be distributed to achieve certain tasks. Applications can integrate different agents and their functionalities together to achieve more complex goals. Further more, an Agent can be reused. Today, agent based applications are developed by many research institutions. To summarize, some key characteristics of agents are:

- Automation
- Provides abstraction barriers above object oriented programming
- Flexibility
- Assumes imposing minimum requirement on the operating environment
- Ability to collaborate
- Suitable for distributed applications

9.1.2 Agent and Rule-Based Engines

Agents can be used to model reasoning behavior just like a rule-based engine. An agent can be goal directed or event driven. The structure of such agent consists of the following:

- A set of knowledge
- A set of events it will responds to
- A set of goals that it desires to achieve
- A set of plans to achieve these goals and handle events that might occur

Compare this to a the structure of a rule based engine:

- A knowledge Base
- An Inference Engine
    - Inference Engine receives events
    - Inference Engine fires actions

Conceptually, agents and rule engines are very similar. They both have a knowledge base that they used to make their decisions. They both react to incoming events. In fact, an agent's goals and plans can be encoded into rules.

In practice, agents have many advantages over monolithic rule-based engines. Agents are distributed. Agents are flexible. For example, one agent can contain a rule-engine itself or agents can collaborate to carry out tasks as a rule engine. Agents Oriented Programming is more directly mapped to Object Oriented Programming which developers are more familiar with. Agents can be platform independent and agents can be reused.

However, the trade off is complexity. To achieve flexibility, ability to be distributed and the other properties of an agent, requires careful design and consideration. It is complicated enough that there are even commercial platforms available for agent development. For example the commercial software JACK Intelligent Agent comes with its own agent language and agent compiler.

Complexity is the reason that the Friendly Wizard prototype did not implement the rule engine as an agent oriented rule engine. However, if there is great need for flexibility and the ability to be distributed, a developer could possibly convert the Friendly Wizard rule engine to an agent based rule engine. This requires reimplementation of com.ibm.watson.wearapp.fw.inference package and com.ibm.watson.wearapp.fw.parser package. The reimplementation is possible because Friendly Wizard is a JAVA based language, which is object oriented. Most Agent Oriented Programming can be directly mapped to Object Oriented Programming. For example, JACK Intelligent Agent is a JAVA based agent-programming platform. Many parts of Friendly Wizard can be reused.

Making Friendly Wizard into an agent oriented programming plan from a rule-based platform is not recommended. If a developer needs to take advantage of agent oriented programming, other tools like JACK Intelligent Agent should be used.

9.2 Friendly Wizard with Fuzzy Logic Rule Engine

Fuzzy logic can be thought of as approximate reasoning. Fuzzy logic enables uncertainties to be encoded into a knowledge base. For example, fuzzy logic enables rules to encode information such as "maybe" or " almost certainly". Fuzzy logic can also encode relative information such as "large" or "far". Fuzzy logic is very useful because much human knowledge involves uncertainty. Fuzzy logic enables and expands the ability of our knowledge base to encode these rules and provide smoother transactions between events.

Fuzzy logic contains the following parts:

- A knowledge base that contains data and rules
- An inference engine
- A fuzzification interface
- Defuzzification interface

A fuzzification interface is used to translate values of inputs to fuzzy rule sets that can be later used by the inference engine. A defuzzification interface will translate the inferred fuzzy result to a non-fuzzy action that will be carried out by the inference engine.

Friendly Wizard inference engine can be easily upgraded to a fuzzy logic engine. Current prototype of Friendly Wizard already has a knowledge base and an inference engine. The only component that needs to be implemented is the fuzzification interface and defuzzification interface. The upgraded version preserves all the property of current Friendly Wizard but with the ability to represent approximate information. Fuzzy logic is a one of the ways Friendly Wizard can be further improved.

9.3 Friendly Wizard Based using a Bayesian Network

A Bayesian network is a graphical model that encodes probabilistic relationships among different objects. For example Bayesian Network can encode the rule "IF door opened AND Brad is detected THEN display "welcome Brad" to the graph below, Figure 17. Further more, probability can be assigned to each branch to indicate the likely hood that branch is taken.

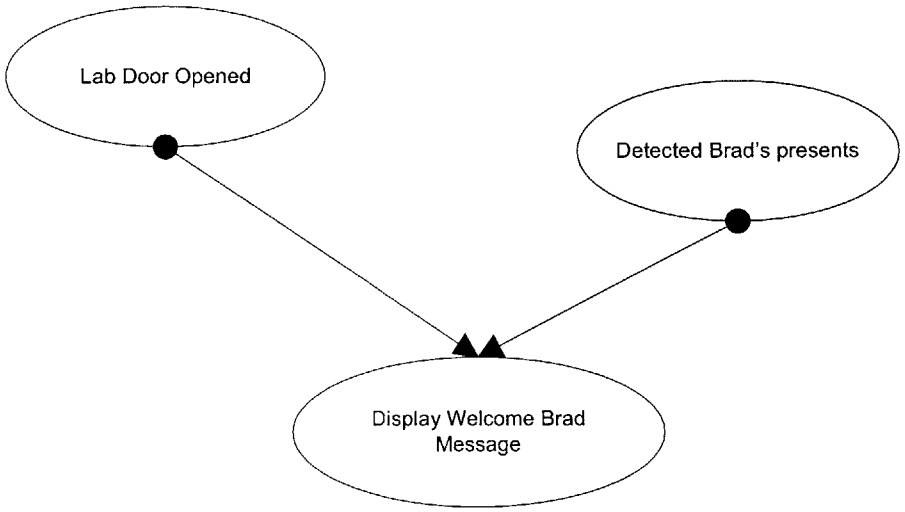


Figure 17 A Simple Bayesian Network

The ability to assign probability to each branch in the graph enables rules with uncertainty to be used in inference. Just like fuzzy logic, Bayesian Networks can be used to expand the scope of rule engine in Friendly Wizard.

Another characteristic of a Bayesian network is that it can "learn" the true probability of each branch. For example, if we have a rule that says "IF every person starts to speak during a meeting THEN 70% chance the meeting is over." This rule can be described by Bayesian's Network as Figure 18.

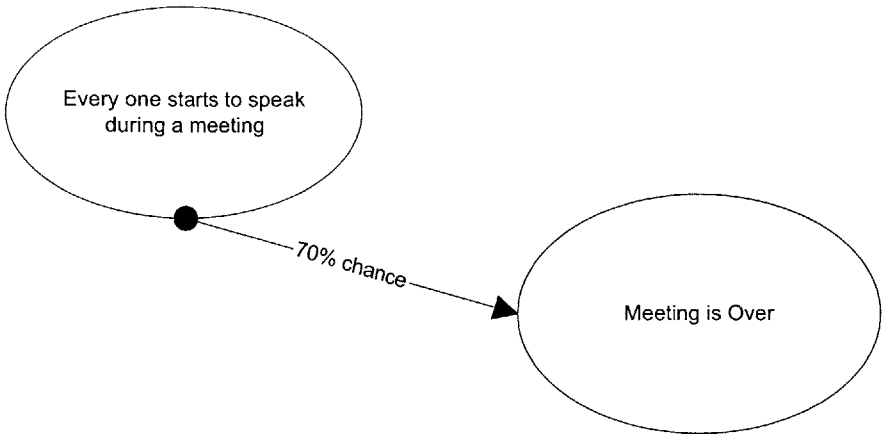


Figure 18 Edge Assigned with Probability in a Bayesian Network

We initially assign 0.7 as the probability of the branch, but if we use this rule for many meetings, a Bayesian Network can replace the probability of 0.7 with a more realistic probability. In a sense, the Bayesian Network is learning.

Implementing Friendly Wizard using a Bayesian Network is not difficult. A new Rule parser can be supplied to parse rules with a certain probability into graphs. An inference engine needs to be modified to "learn" the real probability over time. The plus side is the developer can now get a visual view of the Rule Base in the rule engine. They can also edit rules using a graphic interface.

What the developer needs to consider is that a Bayesian learning network might not be the best representation of their sensor-based system. The Bayesian Network model is ideal for representing causal and probabilistic relations. However, it might not suit applications that do not have causal and probabilistic relation between nodes. For example, if a developer's Bayesian network consists mostly of locally connected nodes like the figure 19, he is probably better off using plain rules for simplicity.

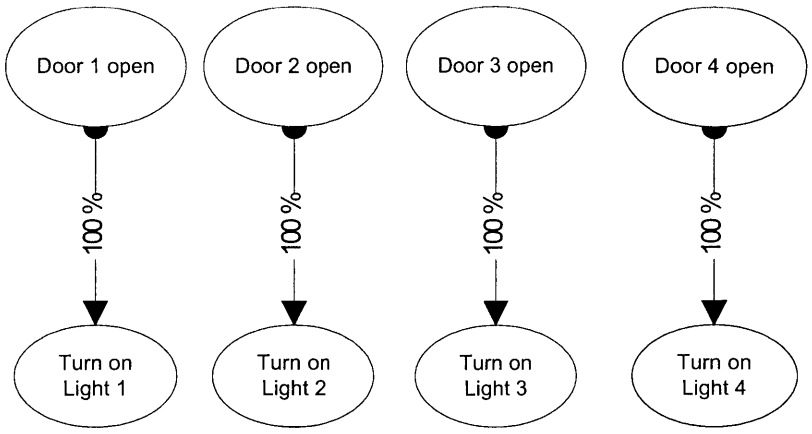


Figure 19 Bayesian Network with Little Interaction

10 Conclusion

This paper provided a description of sensor-based systems. It introduced expert knowledge systems and proposed a solution that uses expert knowledge systems to solve certain problems in current sensor-based systems. Later this paper presented a tool called Friendly Wizard. Friendly Wizard solved the problem in sensor-system development today: that a tool is needed for systems that lay between traditional small-scale sensor-based systems and modern large-scale sensor-based systems. Friendly Wizard gives the developer a forward inference engine and a set of Java interfaces to help them deploy their sensor-based project faster.

At the end, this paper discusses possible updates and improvements for Friendly Wizard. The paper also points out three directions Friendly Wizard can take. They are: building Friendly Wizard using agent technology, modifying Friendly Wizard's rule engine to be a fuzzy logic rule engine, and using a Bayesian network in place of Friendly Wizard's inference engine. In conclusion, agent based technology introduces great complexity but it has many advantages over monolithic rule-based engines. A fuzzy logic rule engine enables rule engines to use rules with uncertainty. Fuzzy logic rule engines are easily build using the current Friendly Wizard infrastructure. A Bayesian Network enables graphical representation of uncertainties, however, it might not be suited for every kind of application that uses a sensor-based system.

There are still many issues in developing sensor-based systems that are not resolved. Hopefully more tools like Friendly Wizard will be available to developers in the years to come.

11 Reference

[1] Joseph Giarratano and Gary Riley, Expert System – 2nd ed., Boston, MA: PWS Publishing Company, 1998.

[2] John Durkin, Expert Systems Design and Development, New York, NY: Macmillan Publishing Company, 1994.

[3] Mark Deloura, Game Programming Gems, Rockland, MA: Charles River Media, Inc. 2000.

[4] Edmund C. Payne and Robert C. McArthur, Developing Expert Systems, New York, NY: John Wiley & Sons, Inc. 1990.

[5] Joseph P. Bigus and Jennifer Bigus, Constructing Intelligent Agents with Java, New York, NY: John Wiley & Sons, Inc. 1998.

Appendix

A: Sample Configuration File

```
<!-- configuration.xml -->

- <fw>
  - <varibles>
    - <varible name="rooms" init="20-153">
        <value name="Jeff's room" />
        <value name="20-153" />
        <value name="noi's office" />
      </varible>
    - <varible name="test_boolean" init="true">
        <value name="true" />
        <value name="faulse" />
      </varible>
    - <varible name="Liyan's Vox" init="not speaking">
        <value name="speaking" />
        <value name="not speaking" />
      </varible>
    - <varible name="noi's hand reco" init="no activity">
        <value name="no activity" />
        <value name="write down sym" />
        <value name="write down stuff" />
      </varible>
    </varibles>
  - <sensors>
      - <!--
          <button>
```

```
        <properties name="a button" className
    ="Button" sim="true"/>
        <location
    loadpath="com.ibm.watson.wearapp.fw.sensor"/>
      </button>
      <badgeDetector>
        <properties name="a badge detector"
    className="BadgeDetector" sim="true"/>
        <location
    loadpath="com.ibm.watson.wearapp.fw.sensor"/>
      </badgeDetector>

        <voxSensor>
        <properties name="an other badge"
    className ="Vox" sim="true"/>
        <location
    loadpath="com.ibm.watson.wearapp.fw.sensor"/>
      </voxSensor>
  -->
</sensors>
- <devices>
  - <!--
      <displayMessage>
        <properties name="display message"
    className="DisplayDevice" sim="true"/>
        <location
    loadpath="com.ibm.watson.wearapp.fw.device"/>
      </displayMessage>
  -->
- <circularRecording>
```

```
        <properties      name="circular      recording"
      className="CircularRecordingDevice"
      sim="true" />
<location loadpath="com.ibm.watson.wearapp.fw.device" />
    </circularRecording>
  </devices>
</fw>
```

B: Sample Simulation File

```
<!-- sim.xml -->
- <sim>
  - <variables>
    - <variable name="Location">
        <value name="YKT" />
        <value name="Howthorn" />
      </variable>
    </variables>
  - <devices>
      <device name="Detective" />
      <device name="Detective2" />
    </devices>
  - <sensors>
    - <sensor name="SenseStuff">
      - <action>
          <value name="walks" />
          <value name="enters" />
          <value name="plays" />
        </action>
      - <subject>
          <value name="jeff" />
          <value name="brad" />
          <value name="bill" />
        </subject>
      - <object>
          <value name="lab" />
```

```
        <value name="office" />
        <value name="room" />
      </object>
    </sensor>
  </sensors>
- <simRules>
    <rule name="rule1" sensorName="SenseStuff"
    waitTime="3" subject="jeff" object="lab"
    action="walks" />
    <rule name="rule2" sensorName="SenseStuff"
    waitTime="5" subject="brad" object="lab"
    action="walks" />
    <rule name="rule3" sensorName="SenseStuff"
    waitTime="4" subject="bill" object="room"
    action="walks" />
    <rule name="rule4" sensorName="SenseStuff"
    waitTime="2" subject="jeff" object="office"
    action="walks" />
  </simRules>
</sim>
```