MMI – AIBO Team

WatchDog

April 27, 2005

Silvia Oana Tanase

1. Contents

Preface

In this project I have received help from many people. I would like to thank the supervisor of the project professor Drs. Dr. Leon Rothkrantz, who was the main contributor of the concept of an AIBO watchdog. My project manager Siska Fitrianie has been given a lot of valuable help and feedback and also to all the members of the MMI - AIBO team for their support and collaboration.
I would also like to thank Prof. Dr. Lidia Sangeorzan from the “Transilvania” University of Brasov for making possible my coming to TU Delft.

Chapter 1 Introduction

1.1 Problem Setting

Violence in homes and on the streets worldwide devastates economies as well as lives The UN health agency warned in a report detailing how countries are spending billions a year dealing with the consequences. Some countries are devoting more than four per cent of their gross domestic product to arresting, trying and imprisoning violent offenders and providing medical and psychiatric care to victims of rape, child abuse and domestic violence, as reported by World Health Organization [1].

Worldwide, 1.6 million people die from violence each year, and millions of others suffer injuries, lingering physical, mental, sexual or reproductive problems, and lost wages and productivity, according to the WHO. It said violence remains a leading cause of death among people aged 15 to 44. In the U.S. alone, the statistics for violent crime are staggering. According to the FBI, on average a person is murdered every 22 minutes; someone is raped every four minutes, a robbery is committed every 26 seconds [2].

The result of increasing violence is that it has grown also our need of feeling safe wherever we are: on the street, at home, at work. Many people had started to find different ways of protecting themselves: installing alarms, surveillance cameras all over the house, hiring people to guard their belongings and their life, and getting dogs to watch them. Moreover society has gradually moved towards an extensive use of computers and automated support, both in everyday life and in work environments, the role and feasibility of autonomous robots has grown in importance. The concept of smart homes, with several computer-based systems making everyday life easier and safer is presently an active research and development subject, i.e. at Telenor [3]. Another aspect is the inclusion of electronics and robotics for fun and entertainment in homes such as PC and TV-games and the robot entertainment dog, AIBO. AIBO walks on four legs and commercial software makes it act like a small pet that walks around, sings songs or chooses to do nothing. AIBO owners also have the opportunity to program it to do other and maybe more useful things.

1.2 AIBO as a WatchDog

AIBO thought as a companion dog that entertains us and makes us smile could be also programmed to do more useful things such as protect us and announce us when he sees an intruder in our home. You will probably wonder why to buy an AIBO to protect you when you have a very nice dog that can do all this and he is even nicer than AIBO. The answer is very simple: AIBO could be a dog, a companion and a surveillance camera and he also has the capacity of being on duty day and night, 24 hours of 24 hours, 7 days a week.

Recent researches have been done on transforming the AIBO into a watchdog (see chapter 2). These researches transformed AIBO into a simple camera that has the capacity of barking at the moving objects and saving an image of the moving object which could be later seen by the owner.

Our project also inspires to “train” AIBO ERS-7 dog to be a watchdog. The AIBO will be able to detect motions and sounds, to bark and save images of moving objects, engage into investigations, and moreover will be able to “see” the intruder. He could even be used as a smoke detector: if he "smells” smoke he starts exploring to see if the house is on fire. AIBO

Watchdog will send an alarm via his wireless communication network if there is any threat. The AIBO Watchdog will act as a live trained dog and maybe in time he will replace the real ones.

1.3 Project Definition

The phases of this project are as follows:

1. **Study literature**. In this phase, we collect information, data, journals, papers and experimental reports about violence, about AIBO in general and about existing languages to program AIBO-ERS7

2. **Seek existing system**. In this phase, we compare and study some of the existing systems that were developed till this moment.

3. **Define and design new model.** In this phase, we design the global "ideal" architecture of a watchdog.

4. **Implementation.** In this phase, we use an incremental development approach.

5. **Analyze and test the prototype.** In this phase, we tested the watchdog and analyzed the results.

1.4 Report Overview

The structure of this report is as follows:

Chapter 1: Introduction provides general information about the project, background and motivations.

Chapter 2: Literature Study gives a view on what has been done in the field till now.

Chapter 3: Design describes the project context level design, class diagram, dynamic model and interface design.

Chapter 4: Implementation describes the implementation status.

Chapter 5: Testing presents a test plan for the" watchdog" project

Chapter 6: Conclusions and Recommendations evaluate and summarize the main results of this project and give directions for further work.

Appendix

Bibliography

Chapter 2 Literature Study

2.1 Security System

Security and safety is nowadays the main thing we think about. This is why the security system had evolved more and more from a simple alarm to a sophisticated security system with surveillance cameras and sensors for movement, fire and breaking glass. It is debated whether having an alarm system decreases the chances of a burglary. In theory, if a burglar is aware a house has a system, he or she might move on to another home. Even if the alarm system does not keep a burglar from breaking in, the burglar has a tendency to stay a shorter amount of time. This may decrease the number of items stolen and the extent of damage done. Most systems rely on a combination of contacts placed at doors and windows and motion sensors. Motion sensors, however, do not detect someone until they are already in the house. Glass break sensors are recommended for a good security system.

The problem with security systems is that they do not necessarily stop people from breaking in. The security system is only activated when the burglar has broken into the house. Also, by the time the intruder is detected and someone responds to the alarm, there could be enough time for the intruder to remove items and leave. If the system does not cause visible or audible alarms to flash or sound at the site, or there is no one nearby to see or hear these site alarms, the intruder can leave without being seen. Even if you have a sophisticated security system with surveillance cameras burglars may find a way of tricking the system because a camera is in a fix place and you could easily go unnoticed. However people tend to be much more relaxed when they have some security system installed in their home even if they know that nothing is sure now a days. The main problem of this kind of security system is that it has no mind of its own. This is why we propose using a robot to watch a house rather than having a security system. For example the robot could identify if the sound of breaking glass was made by an intruder or by the wind or whether the homeowner entered the house or it was an unauthorized person.

2.2 AIBO

AIBO, the robotic pet is an example of such an autonomous robot, developed and manufactured by Sony. Sony has introduced several different versions of AIBO since their first launch in 1999. AIBOs can walk, "see", and recognize spoken commands, and they are considered to be autonomous robots, since they are able to react to external stimuli from their owner or environment, or from other AIBOs and they have the ability to learn and to mature.

AIBO series:

- ERS-110, the 1st generation, has the ability to learn from its environment and express emotion.
- ERS-210 incorporates touch sensors and sound, voice and face recognition.
- ERS-311/ERS-312, new shape AIBOs: adorable LATTE and mischievous MACARON.
- ERS-220, AIBO with a new hi-tech robot look.
- ERS-7, improvement on interaction and wireless internet connectivity.

2.2.1. AIBO Anatomy

The AIBO ERS-7 from Sony has hardware features a faster CPU, a higher resolution camera and twice as much memory as its predecessors. User can interact via voice and tactile touch sensors, remotely access the robot and retrieve digital images on a PC via e-mail commands or an Internet browser (with Wi-Fi connection) [4].

Table 2.1 The features of the AIBO

Dimension:	180(W)x278(H)x319(D) mm
Weight:	Approx. 1.65kg (including battery and memory stick)
CPU:	576MHz, 64bit RISC Processor, MIPS R7000
Memory-SDRAM:	64MB
Program Storage Media:	Memory Stick - 1 slot, FAT 16
Moveable Parts:	Mouth (1 dof), Head (3 dof), 4 Legs (3 dof), 2 Ears (1 dof), Tail (2 dof)
Camera:	CMOS Image Sensor 350k pixels
Wireless LAN Card:	IEEE 802.11b (integrated)
Audio:	Miniature microphone Miniature Speaker 20.8mm 500mW MIDI Volume Switch
Built-in Sensors:	Temperature Sensor, Infrared Distance Sensor (head, body), Acceleration Sensor, Electric Static Sensor (head, back), Pressure Sensor (chin, 4 paws), Vibration Sensor
Power:	Approx. 7W consumption (Standard operation in autonomous mode) Approx. 1.5 hours operation times Approx. 2.5 hours charging time
LED:	Illume Face 28 LED (white 16, red 4, blue 4, green 4) Ear 2 LED left and right Head Sensor 2 LED white and amber Head (wireless LAN on/off) 1 LED (blue) Back Sensor 16 LED (white 8, red 3, blue 3, orange2) 28 multi-gradation expressions LED lights on the Illume Face by changing the pattern of the lights and their intensity of brightness.

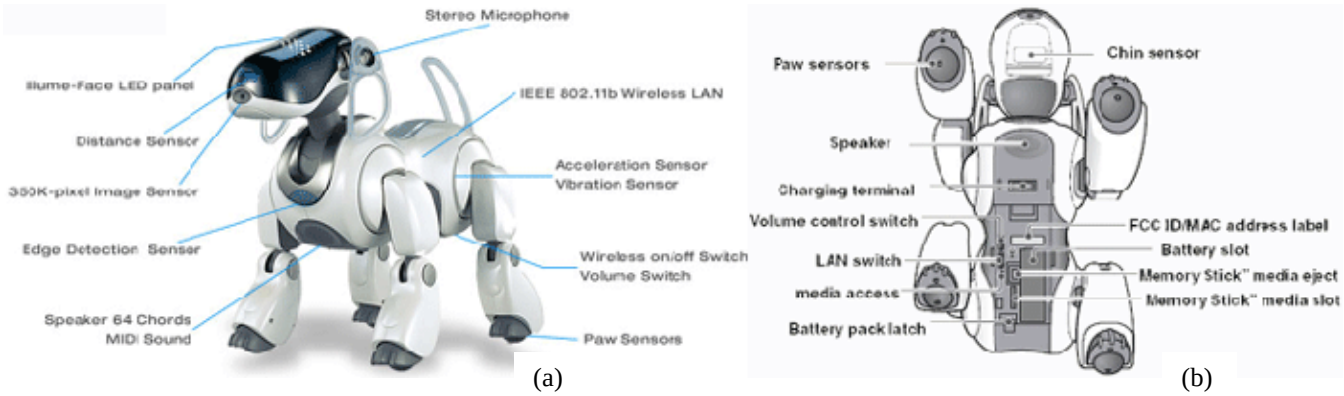

Figure 2.1 The anatomy of AIBO: (a) from the front side and (b) from bottom side

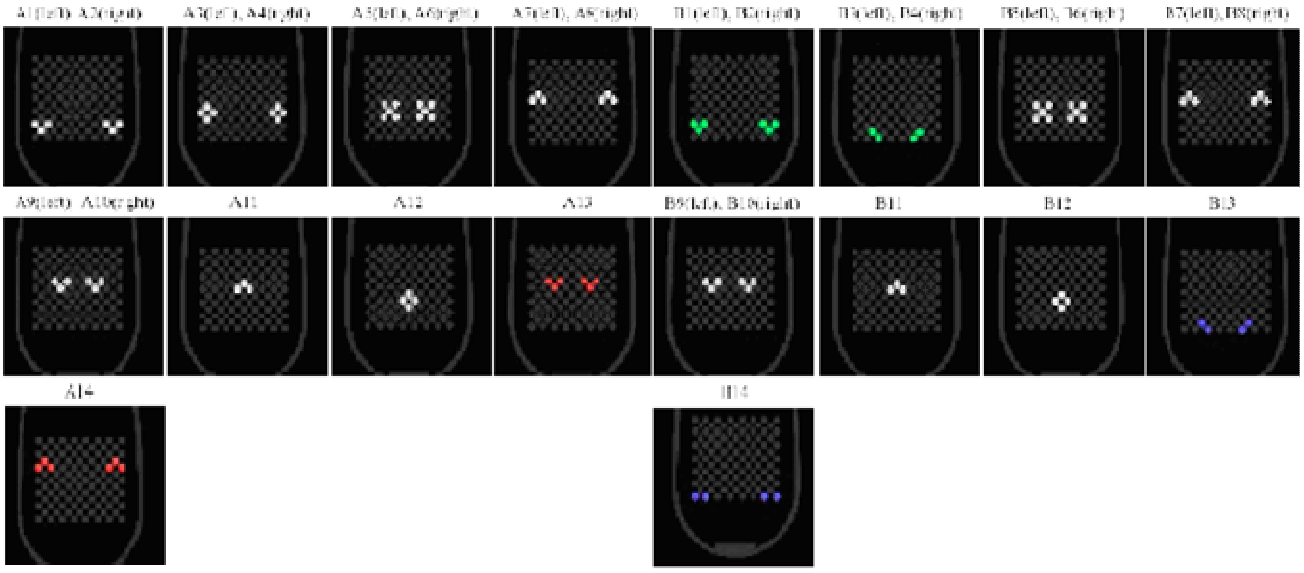

Figure 2.2 The combination of LED's

2.2.2. Programming the AIBO

Three frameworks are available to program the behavior of the AIBO: (1) OPEN-R SDK, (2) Tekkotsu and (3) URBI.

AIBO owners can teach their pet new behaviors by reprogramming them in Sony's special *R-Code* language or *Open-R SDK* for non-commercial use. AIBO Software Development Environment can create software that either executes on AIBO or executes on a PC or controls AIBO by using a wireless LAN. This SDE contains three SDKs (Software Development Kits) and a motion editor. The three SDKs are Open-R SDK, R-Code SDK, and AIBO Remote Framework. These development environments are provided free.

The Open-R SDK is a cross development environment based on gcc (C++) with which developers can make software that works on AIBO (ERS-7, ERS-210, ERS 220, ERS-210A, and ERS-220A). The R-Code SDK is an environment with which developers can execute programs written in R-Code, a scripting language, on AIBO (ERS-7). The AIBO Remote Framework is a Windows application development environment based on Visual C++, with which developers can make software that runs on Windows. The software can control AIBO (ERS-7) remotely via wireless LAN. AIBO Motion Editor can be used with the Open-R SDK, the R-Code SDK and the AIBO Remote Framework.

Open-R SDK provides a disclosed Open-R API of system layer ("level-2" interface). In this case, developers only can utilize some AIBO's functions such as:

- moving AIBO's joints
- get information from sensors
- get image from camera
- use wireless LAN (TCP/IP)

The Open-R SDK includes tools to make Open-R objects, sample programs, and memory stick images that must be copied to a AIBO programmable memory stick. Developers also may create programs with GNU Tools, e.g. gcc and cygwin. The developers should use the Open-R API to turn on/off LEDs manually

The developers also can make motions by synchronizing with sounds (MIDI, WAV) and LED patterns in AIBO Motion Editor. In this case, the MIDI and WAV data are played back by the PC. MIDI sounds and LED patterns cannot be played back with the Open-R SDK, but they can be used in the other two SDKs. WAV sounds can be played back with all three SDKs. Sound files and LED patterns cannot be created by AIBO Motion Editor. Other commercial application can be used to handle this, such as skitter. The R-Code SDK and the AIBO Remote Framework can recognize motion files that are made with AIBO Motion Editor.

Tekkotsu is a framework built on top of OPEN-R SDK[5]. This means, that in order to use Tekkotsu, the OPEN-R SDK also has to be installed. Tekkotsu offers a way to interface with WLAN. Joints, head movement camera etc. can be controlled via wireless LAN. The programming model with URBI also holds for Tekkotsu. (Tekkotsu server is also an object running on OPEN-R).

The advantage of Tekkotsu is that it offers higher level commands (instead of moving individual joints, one can issue commands like "walk"). Furthermore the Tekkotsu framework aids people who develop objects intended to work on the Aibo (with Tekkotsu) by adding a level of abstraction. So instead of having the to know the message passing details of ones object with other objects (like in the URBI) a Tekkotsu programmer can think in terms of behaviors (a term that is central in Tekkotsu programming framework tutorials).

Universal Robot Body Interface (URBI) is a scripted command language used to control robots (AIBO, pioneer)[5]. It is a robot-independent API based on client/server architecture. In the OPEN-R programming model the URBI server can be viewed as another object. The developer can make use of the URBI server in two ways: via a computer through the wireless LAN using the liburbi c++ (external client) or through direct inter-process communication using liburbi OPEN-R (onboard client).

In the case of an external client, the communication then takes place through a TCP/IP or IPC connection. When using URBI over TCP/IP messages are sending to the URBI object via telnet over port 54000. The URBI Object then sends the appropriate messages to the other objects on the OPEN-R system to accomplish the given command.

In the case of an onboard client, the client is also an OPEN-R object (containing doinit(), dostart(), etc.) that runs along in the OPEN-System and sends messages to the URBI server. Thus, URBI functionality can then be utilized by passing/receiving messages to/from the URBI object (which of course should also be running).

In the general case: when a message is sent to the URBI server (no matter if this is through an external client or through an onboard client), the server object will send message(s) to other objects as a result of this message, thereby hiding the internal message passing details to the client. Therefore, developers only need to concern themselves with the message passing details to and from their client to the URBI. Those passing messages that are not supported in URBI can be sent directly to OPEN-R system. Note however that in this case there will be a discrepancy between what is possible via Telnet and what is possible in the AIBO -for people who like to port their software to and from this can be a issue.

Since our implementation would be done under Java it is convenient to work using URBI. URBI has provided a library that connects between OPEN-R and Java, called liburbi Java.

2.3 AIBO as a Companion Dog

The work of Sony so far has given abilities to AIBO such as moving, "thinking", and displaying the lifelike attributes of emotion, instinct, learning, and growth. The Mind software consists in the following abilities:

- In default mode user can interact with a mature ERS-7 but can be reset it to a puppy stage.
- AIBO can understand and respond to 100+ words and phrases.
- AIBO will perform various autonomous behaviors based on recognition of owners face and voice. It should be noted however, that upon the experimentations using these commands, it was found that AIBO microphone is quite poor. Even under very quite conditions, it is difficult to generate responses from AIBO.
- AIBO has improved self-charging abilities and visual pattern recognition technology which enables it to respond to its 15 AIBO cards.
- AIBO can bring the AIBOne toy to the user on command and play with the Pink Ball. With integrated control over the operation of 20 joints in the AIBO body (20 degrees of freedom), the ERS-7 AIBO provides autonomous behavior and functionality.
- AIBO can better express its emotions and what it is thinking with the 49 multi-color LEDs. Finally, Illume-Face (using 28 of these LEDs) provides a completely new way for ERS-7 to show when it is happy, sad, angry, surprised, etc.
- AIBO allows for more organic interaction through the newly developed tactile touch sensors on the back, head, chin, and paw.
- Using wireless connection, AIBO can connect with other electronic devices, transmitting photos, sound files and messages. This controls AIBO's behavior and the applications can be used via PC or a mobile device.

AIBO owners cannot add new motions to the AIBO Mind Software. AIBO Motion Editor (available in the AIBO SDE) is the motion creation editor for AIBO (ERS-7). Motions that are created with AIBO Motion Editor can be used with Open-R SDK, and the AIBO Remote Framework

2.3.1. Emotion

The white LEDs, which are supposed to represent the various different states of AIBO are incomprehensible and are played too quickly to understand. Our experimental results shows that user interaction affects the emotional state of AIBO, which will be reflected some behavioral expressions [4]. The interaction does not affect, however, the overall instinctive state or behavior of AIBO. AIBO will result in momentary behavioral expression. The change in the emotional state, therefore, does not have any effects on the overall instinctive state of AIBO. The different instinctive states produce different behavioral expressions in AIBO. However, AIBO appear to have only momentous emotional states which are neither lasting nor remembered. AIBO makes 64-chord MIDI sounds to express its feeling. It cans also playback pre-recorded voice messages for you. If the user asks AIBO to dance, the funky music will sound.

AIBO incorporates five instincts and six emotions. The five instincts includes love, curiosity, movement, hunger (low battery), and sleep and the six emotions include happiness, sadness, anger, surprise, fear and dislike. These emotions are shown in AIBO's behaviors:

- Happy/joyful: a green LED pattern and wagging the tail sometime flip the ears or give a happy sound. It may happen when a user strokes its head and back sensor, playing with AIBOne or pink ball, when the owner response, or while and after charging.
- Anger: a red LED pattern happens when a user taps the third back light.
- Sad/Confused: a purple LED pattern happens when the owner ignore it. Dim cross white LEDs also means demur.
- Fear: a small white pattern.
- Hungry: a pair of small line white LEDs.
- Joking: one white LED (wink).
- Surprise: a pair of cross white LEDs.
- Find its charge station: a purple-white-green LED pattern.

2.3.2. Behavior

According to our observation AIBO will react always to encouragement or scolding by the user [4]. When AIBO is hungry, and is in search for the charge station, AIBO will ignore all other commands or interactions except for when reacting to user's encouragement or scolding. It will remain faithful in its aim to find the charge station. The AIBO card has priority over all other instincts or voice commands. AIBO will always follow what the AIBO card commands despite its instinct (except when hungry) and will ignore all other voice commands when carrying out what the card commands. AIBO will only sometimes to the voice commands, even when heard correctly, depending on its instinct.
Recent attempt in developing AIBO as a Companion Dog is currently on going within the MMI group at TUDELFT. This project proposes a new cognitive model for a companion dog. The final goal is to develop an AIBO that has human-like behaviors given some specific situations. As some Artificial Intelligence researches have put attempts on cognitive systems that model human personality, the project is also trying to accomplish by starting from applying existing known human cognitive models on the AIBO. In this project the researcher

tries to develop AIBO's characters that are not defined by what he learns or randomly receives from his internal cognitive system but by what the owner decides.

We start developing our own model from scratch trying to design a model that is as close as possible to a watchdog model. Since we start from scratch we cannot take benefit of all the sensors and multimodal functionalities SONY has invested the AIBO with. We will use for the project some abilities developed in the "companion dog" project such as: barking, turning the head, walking.

2.4. AIBO as a Watch Dog

Sony had developed a type of a watchdog in AIBO mind software 2 for the house sitting mode [10]:

- capture 15 pictures if object or sound detected
- send the capture image out via email
- record the sound
- turn on/off the mode using email, voice commands, or sensors
- adjust his head up an down

The Followings are a survey of current work in modeling of a watchdog. First, an example of a watchdog made in OPEN-R (WatchDog), and followed by an example of an autonomous watchman made in Teckkotsu. Finally, a comparison between the previous works and our proposal of a Watch Dog is presented in table 2.2.

Table 2.2 Comparison of the three projects

	WatchDog	**WatchMan**	**My WatchDog**
Features	o capture the image if object detected o bark at the moving objects	o makes a little barking sound if pink ball (seen as an intruder) is detected, but he has abnormal reactions to it, since it sees pink balls everywhere, especially in "noisy" surroundings (with several objects) or when there are almost pink objects around. The almost-pink objects can be, for example, an orange sofa or a red extinguisher. o plays a sad sound if the ball is lost after it was detected o walks in searching of the pink ball o stops and looks right	o Detects image and sound o If a sound is too loud then he will start to explore in search of unusual things o If an intruder is detected he will bark and make picture of him and maybe set alarm.

		and left for the ball o doesn't detect the walls o turn on/off the mode by pressing the central tail button	
Can only be activated on the station	o No. You can put the AIBO on the station for long time watching. You can also put the AIBO on the table or floor for short time watching as long as the battery lasts.	o No. In fact is necessary to put the AIBO on the floor because he will walk if in watch mode in searching of the pink ball.	o No. AIBO will explore the room in search of unusual sounds and image.
Max. captured images	o 300	o 0	o You could select how many pictures the dog will take.
Auto gain and shutter control	o Yes	o Not documented	o Not known
Access the captured images via WEB	o Yes. You can also set the password or change the configuration using the browser.	o No	o No
View what AIBO is watching via WEB	o Yes	o No	o No
Download the captured images via FTP	o Yes	o No	o No
Work with Mind/Mind 2	o No. You need a separate memory stick to run it as the limitation of OPEN-R SDK	o No	o No
Motion detection sensitivity	o Auto adjusted. Low in a light environment and high in a dark environment.	o No	o Not known

Background image re-initialization	o Auto	o No	o No
Head angle adjustment	o No. Move the head manually	o Adjusts the head left and right	o Adjusts the head left and right

2.2.1 WatchDog (For the ERS-7/ERS-7M2 model only)

With this program AIBO ERS-7 or ERS-7M2 dog can be turned into a watchdog [11]. If the AIBO detects any motions, it will:

- Bark at the moving objects, and
- Save the pictures (in JPEG format) in the memory stick.

The program includes a web browser that provides the following features:

- Basic authentication we have to input the user name and password for accessing the pictures.
- Web-based configuration settings. We can change the user name, password, watching image size, etc. settings on the web browser.
- Shows us the image that he is watching, and updates the image periodically and automatically.
- We can browse the captured images using the web browser.

More information about this watchdog can be found in Appendix A at the end of the report.

2.2.2 WatchMan

This project presents the possibility of automating watchman activities using simple and small robots such as AIBO's [6]. He demonstrates through design analysis and implementation how AIBO can be used as a watchman to keep areas under surveillance. The main focus in this study is safety, with risk analysis and some implementation as natural parts of the study to better comprehend the concept of software safety in critical systems. The main result of this project is the implementation and analysis of AIBO to walk and look around, and detects pink balls (as potential intruders). Several more requirements have been identified, but they have not been implemented. As not all requirements and no safety requirements have been implemented and the testing has been applied in a very informal manner. It would have been interesting to apply several more increments with implementation, testing and risk analysis, to give the implemented system more useful and accurate behavior.
The AIBO watchman will monitor a safety area and the correct performance of its action is therefore critical.

The watchman was developed through a set of phases.

- *In phase 1* the goal was to develop software that enables AIBO to autonomous walk down a corridor in an approximately straight line. If the watchman detects unauthorized personnel it will make a sound.

- *In phase 2* they extended the watchman's abilities to move. AIBO needed to be able to detect a turn and follow the curve of the turn, and be able to avoid other types of obstacles but at the end the watchman wasn’t capable to detect walls. This phase did also include the ability of AIBO to separate authorized from unauthorized personnel. The unauthorized personnel is the pink ball. AIBO does detect pink balls, but has abnormal reactions to it, since it sees pink balls everywhere, especially in "noisy" surroundings (with several objects,) or when there are almost pink objects around. The almost-pink objects can be, for example, an orange sofa or a red extinguisher. It can also detect pink balls in other areas, and seem more sensitive when there are moving objects around it. The source of this problem could be in how the AIBO codes different color or the camera calibration. A better recognition algorithm, with edge detection as well as color recognition, was recommended.[6]

Chapter 3 Design

3.1 Concept

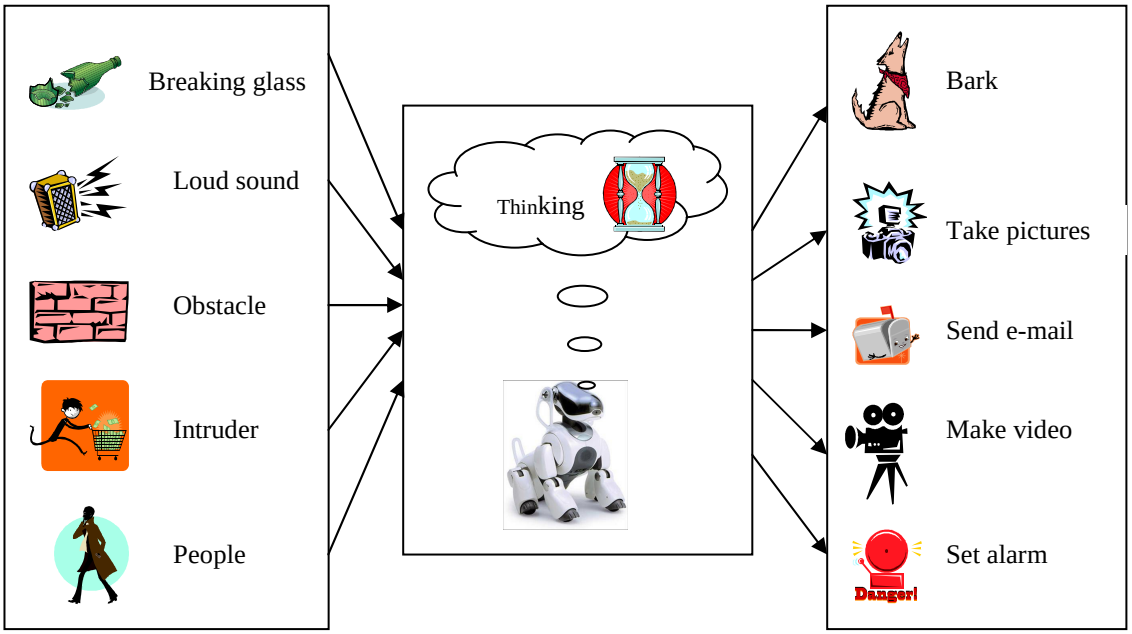

Figure3.1. AIBO model as an agent

The AIBO WatchDog will detect people and then he will try to identify if he knows the person or not. If he doesn't know the person then automatically he will assume that he is an intruder and he will take pictures and send them through mail to his owner, set the alarm and start barking angry. If the owner is at home he will be announced and if it's not home maybe the neighbors will hear it and announce the police. The purpose of setting the alarm is to announce the owner or his neighbors that they have an intruder in the house. On the other hand if he knows the person than he will let him/her know that he recognizes her and starts barking happy. If the person detected is the owner he will also be happy to see him/her.

He can also detect motions. When any kind of movement is detected he will take pictures and send them through e-mail at his owner.

We will use the priority value of tasks as used by the companion dog project within the MMI group at TUDELFT. That means that if the battery of the watchdog is empty or close to being empty the dog will not continue with his exploration but he will go and recharge the battery.

The watchdog will "watch" all the time so the user can see every thing the dogs sees in real time. The dog will be able to make as many pictures as the owner wants. The picture will be saved in JPEG format. We really think that the watchdog needs to be really similar to a real one and a real one sees everything that moves in the house and hears everything not only the

unusual thing for example if there are children playing in the house and they make a lot of noise he will go and see what happens and sees children playing and laughing and he knows that every thing is ok.

A robot sometimes can make a “mistake” and he can be easy tricked by men. This is why he will record every thing and the user can see every thing the dog sees not only the unusual things.

The AIBO watchdog is like a real dog as we said before but he will also be like a video camera that saves images of “unusual things” and later on send them through mail at his owner. So if the owner wants to see if something has been wrong in his house he will only need to check his mail. The AIBO will bark only if there is let say a minor thing like a glass broken. If an intruder is found he will bark but he will also take pictures and send them through e-mail and also if he detects movement he barks but send the picture with the movement to the owner so he could decide if it is a good or a bad thing happening in the house. It could probably be a glass falling on the floor or maybe it could be just the wind. But if the owner is in the house a bark is enough to announce him that something is not right and he will come and check or he will set the alarm or announce the police.

3.2 Object Model

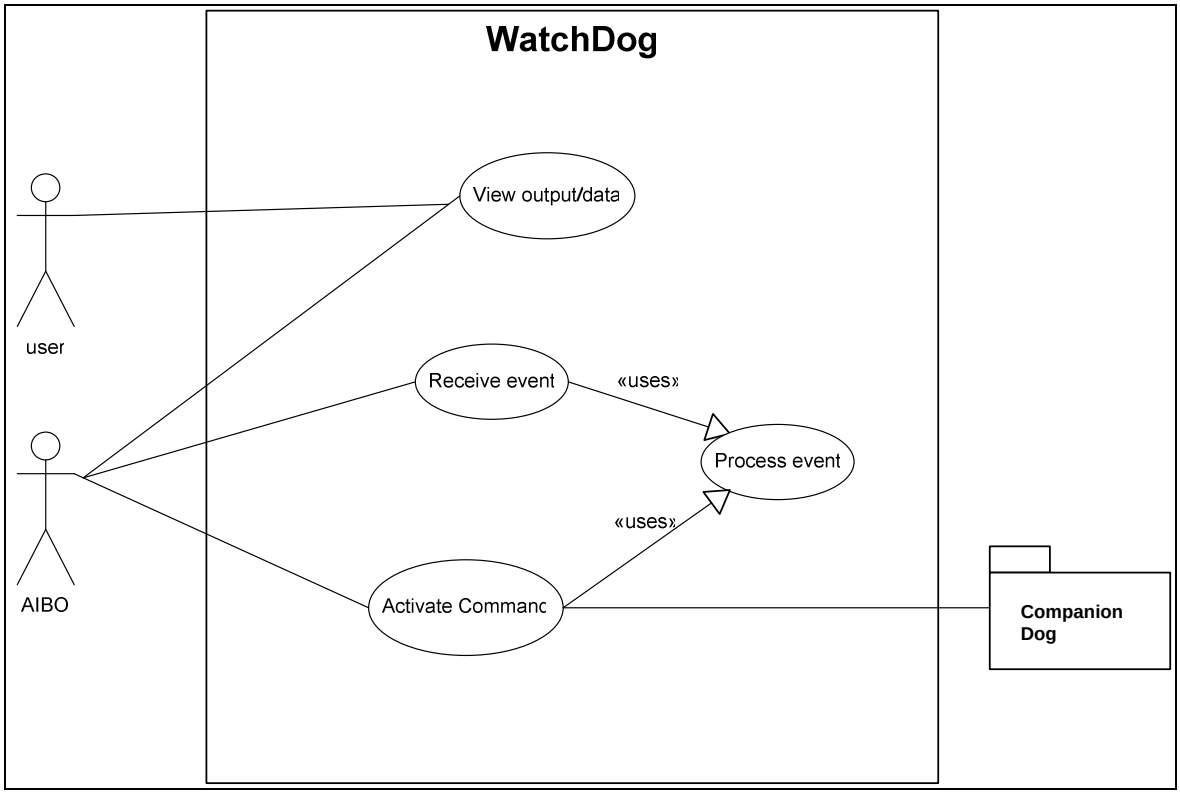

Figure 3.2 Use-Case Diagram

Table 3.1 Description of use case

1.	Name:	View output data
	Previous condition Post condition Exception Actors Description	Data has been stored. Image is showing/Mail was received. Data is empty. User and AIBO. User activates to show image or to check mail.
2.	Name	Receive event
	Previous condition Post condition Exception Actors Description	Input sensors have been processed. Sending event to “process event”. Event unknown. No event. AIBO. Receiving events from the process sensor.
3.	Name:	Activate command
	Previous condition Post condition Exception Actors Description	Decide what action the dog will execute. Send the command to the dog and the dog will execute it. No action. AIBO. Describes the series of commands the dog need to execute.
4.	Name:	Process Event
	Previous condition Post condition Exception Actors Description	Receive event. Decide the action to be taken by the dog. No action associate to this event. AIBO. Analyze the events received from “receive event” component.

A more detailed Uses case diagram is the one from Figure 3.3.

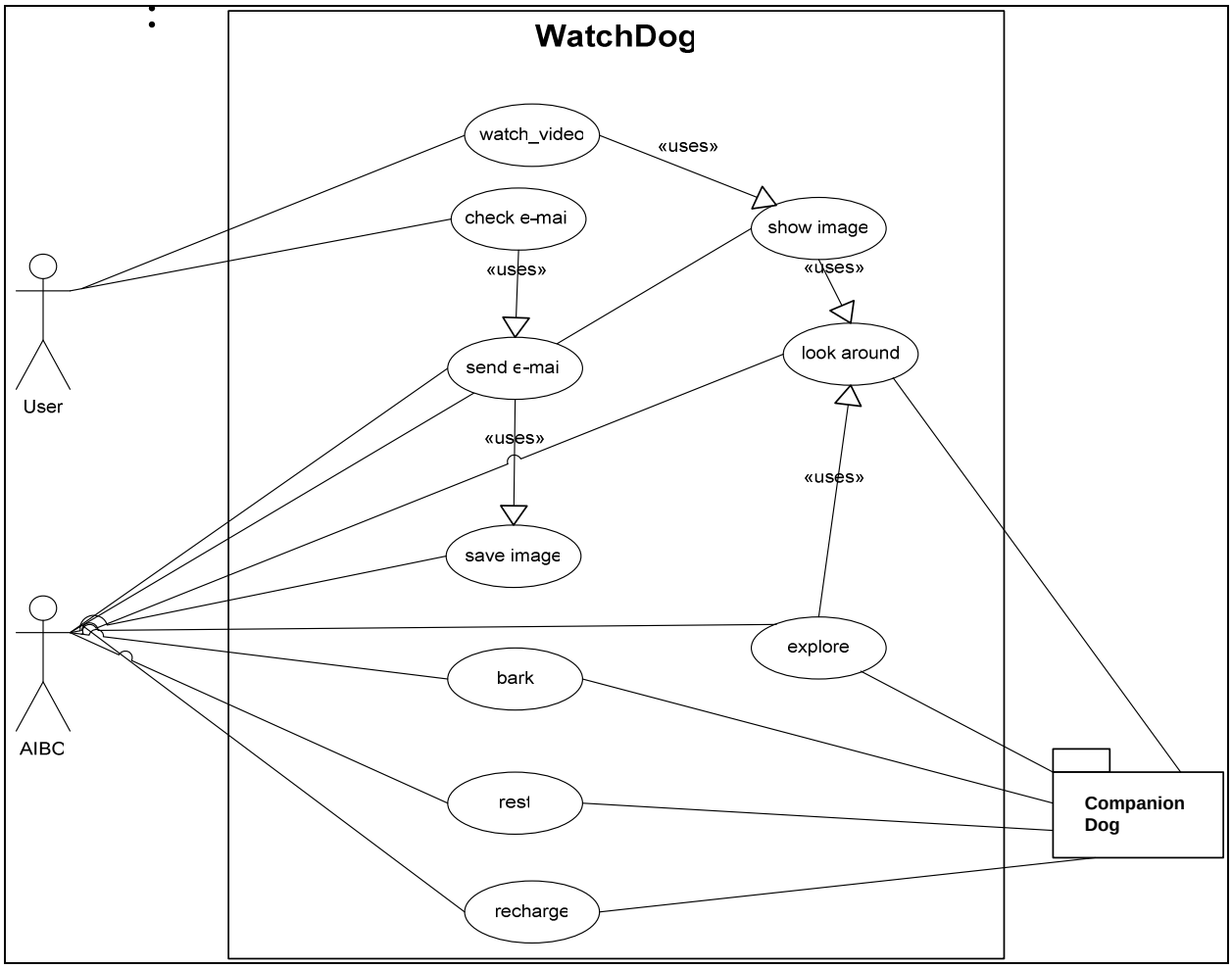

Figure 3.3 Uses Case Diagram

Class Diagram

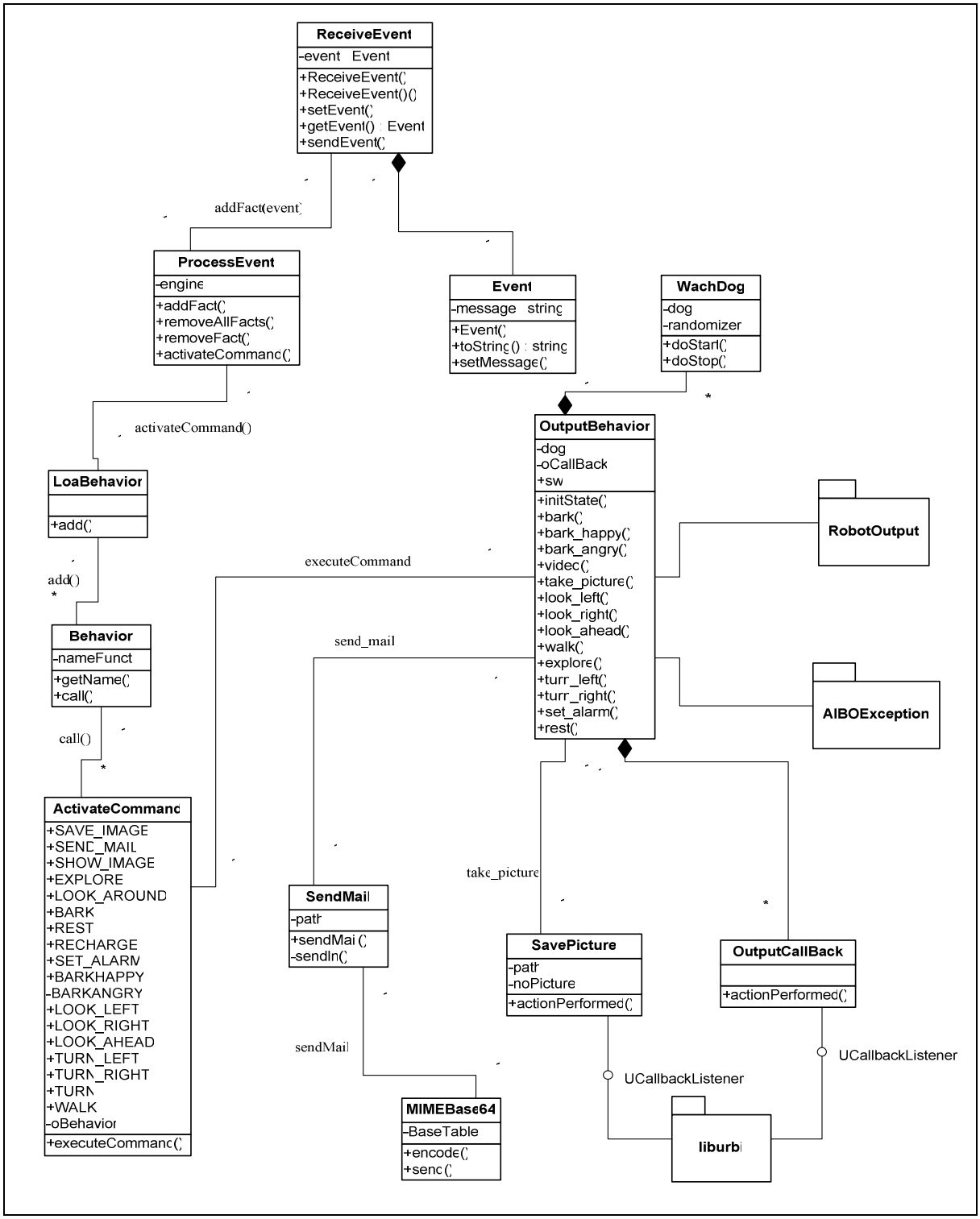

Figure 3.4 Class Diagram

Description of class diagrams

ActivateCommand Class deals with the execution of commands. By the commands the functions in OutputBehavior class are called.

OutputBehavior Class contains all the functions that are part of the behavior of the WatchDog. Many of these functions use functions from the packages RobotOutput and AIBOException witch are part of CompanionDog project.

Event Class is the event that the dog will receive.

ReceiveEvent class deals with the event received from the dog and send them to the ProcessEvent class.

ProcessEvent class is in fact the Rete engine. Here the facts are added to Jess knowledge base from an xml file and also the rules are loaded from a file and fired. We could say that this is the “mind “of the WatchDog.

Behavior class makes possible to add user-defined functions to Jess such as bark, walk, rest. This class implements the jess. Userfunction interface. We only need to implement only two methods: getName() and call(). Having written this class, we can then, in our Java main program, simply call Rete.addUserfunction() with an instance of our new class as an argument, and the function will be available from Jess code.

LoadBehavior class implements the jess. Userpackage interface. The jess.Userpackage interface is a handy way to group a collection of Userfunctions together; so that you don't need to install them one by one (all of the extensions shipped with Jess are included in Userpackage classes). A Userpackage class should supply the one method add(), which should simply add a collection of Userfunctions to a Rete object using addUserfunction(). Nothing mysterious going on, but it's very convenient because we can assemble a collection of interrelated functions which potentially can share data or maintain references to other function objects. We can also use Userpackages to make sure that your Userfunctions are constructed with the correct constructor arguments. All of Jess's "built-in" functions are simply Userfunctions, although ones which have special access to Jess' innards. Most of them are automatically loaded by code.

SendMail class deals with the sending of the mail to a specified mail address.

SavePicture class makes possible taking the picture and saving them in a folder image to a specified path. The picture saved will be than send by e-mail to the owner.

OutputCallBack class implements the interface CallBackListener from liburbi. We only have to implement the actionPerformed function. This function will be called every time a message marked with the corresponding tag is received from the server. The parameter of the method is a URBI event that contains all the necessary information about the received message. To resume the “actionPerformed” method is the action of the client depending to the URBI server message.

Packages **RobotOutput** and **AIBOException** witch are part of Companion Dog project. They are used to implement some parts of the behavior of the watchdog such as barking, crawling, looking left/right.

Liburbi package is the URBI library for Java providing an URBI client. It handles the TCP connection with the URBI server, and the dispatching of messages it sends and receives.

3.3 Dynamic Model

Figure3.5. shows a UML activity diagram which offers rich notation to show the sequence of activities the AIBO watchdog will do. It may be applied to any purpose (such as visualizing the steps of a computer algorithm), but is considered especially useful for visualizing processes, or use cases.

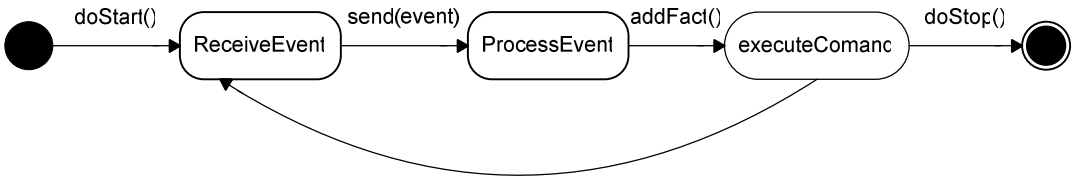

Figure 3.5 Watching activity

The activity diagram describes how the dog acts when he receives an event such as a loud sound, a man in the house or any kind of motion detection. First of all AIBO needs to be in the watching state by activate the doStart() function. Then he can receive the events and he passes then to the ProcessEvent component which will add the event as fact in the Rete engine and from there it will be decided what series of actions the dog will take. After taking action AIBO will be ready to receive another event or the watchdog can be stopped by calling the doStop() function.

3.4 Interface Design

Table 3.2 Description of interface

	Input	Description of process	Output
1.	Loud sound	Sends the message to the processing component which decides to explore and see what happened.	Exploring (walking, looking around, resting a few seconds)
2.	Obstacle	Sends the message to the processing component.	Sides step
3.	Intruder	Sends the message to the processing component which will decide the action to be taken.	Take picture Send them by e-mail

			Start the alarm Bark angry
4.	People unknown	Sends the message to the processing component which decides that if he does not recognize the person then it must be an intruder.	The same as in the case of an intruder
5.	People known	Sends the message to the processing component which decides to let the person know that he recognize him/here.	Bark happy
6.	Owner	Sends the message to the processing component and the dog will be glad to see his owner.	Bark happy
7.	Motion detected	Sends the message to the processing component which will decide the action to be taken.	Take picture Send them by e-mail Start the alarm Bark angry

Chapter 4 Implementation

4.1. Choice of language

- URBI – liburbi Java

  URBI is based on client/server architecture. An URBI server is running on the robot and a client is sending commands to the server in order to interact with the robot. The channel between the client and the server can be a TCP/IP connection or direct Inter Process Communication if the client and the server are both running on the robot. In that latter case, the latency is expected to decrease significantly. The robot is described by its devices. Each element of the robot that can be controlled or each sensor is a device and has a device name. From a programmer point of view, a device is an object. It has some mandatory methods and variables and a list of device-specific methods. Everything that can be done on the robot is done via the devices and the available methods associated to them. The main advantage of using this architecture is the flexibility it allows. The client can be a simple telnet client or a complex program sending commands over TCP/IP. This client can run on Linux, Windows or any other system and it can be programmed in C++, Java, LISP or any language capable of handling TCP sockets. For each new robot type, a new server has to be written. Once this server is running on the robot, it is straightforward to command the robot, whatever the robot is or how complex it is, as soon as one knows the list of devices and their associated methods. This list is made available in the documentation of the server and it is the only robot-specific piece of information required to know how to control a previously unknown robot. The syntax used to access the devices is designed with simplicity in mind.

  For the implementation of the watchdog we use URBI-java because of his main characteristics, which make it different from other existing solutions, and which are:
  - Ø URBI is a low level command language. Motors and sensors are directly read and set. Although complex high level commands and functions can be written with URBI, the raw kernel of the system is low level by essence.
  - Ø URBI includes powerful time oriented control mechanisms to chain commands, serialize them or build complex motor trajectories.
  - Ø URBI is designed to be independent from both the robot and the client system. It relies on TCP/IP or
  - Ø Inter-Process Communication if the client and the server are both running onboard.
  - Ø URBI is designed with a constant care for simplicity. There is no "philosophy" or "complex architecture" to be familiar with. It is understandable in a few minutes and can be used immediately.

  More information about URBI and liburbi Java will be found in the Appendix E at the end of the report.

- JESS

  More information about Jess in general can be found at the end of the report in Appendix F.

Ø Adding facts to JESS.

When the AIBO detects any kinds of event such as movement, a loud sound or an unknown person they will be registered in the XML file. The expert system reads the XML file and constantly adds facts to the working memory or knowledge base. Score Board is a collection of facts that have been inputted to it. In Jess, there are three kinds of facts: ordered facts, unordered facts and definstance facts.

Ordered facts are simply lists, where the first field acts as a sort of category for the fact. Here is one example of ordered facts:

```
(robo-mind man known friendly)
```

Unordered facts are useful but they are not structured. In object-oriented languages, *objects* have named as *fields* in which data appears. Unordered facts offer this capability (although the fields are traditionally called *slots.*). Rewriting the fact above to an unordered fact it would look like this:

```
(robo-mind (action man) (status known) (value friendly))
```

Before to create unordered facts we have do define the slots they have using deftemplate construct:

```
(deftemplate <deftemplate-name> [extends <classname>][<doc-comment>]
      [(slot <slot-name> [(default | default-dynamic <value>)]
                                          [(type <typespec>)]*))
```

The <*deftemplate-name*> is the head of the facts that will be created using this template. There may be an arbitrary number of slots. Each <*slot-name*> must be a symbol. The default slot qualifier states that the default value of a slot in a new fact is given by <*value*>; the default is the symbol nil. The *'default-dynamic'* version will evaluate the given value each time a new fact using this template is asserted. The *'type*' slot qualifier is accepted but not currently enforced by Jess; it specifies what data type the slot is allowed to hold. Acceptable values are ANY, INTEGER, FLOAT, NUMBER, SYMBOL, STRING, LEXEME, and OBJECT. As an example, defining the following template:

```
(deftemplate robo-mind
      (slot action)
      (slot status)
      (slot value))
```

If a fact is defined as above then we can add it to the inference engine of the expert system base like this:

```
(assert (robo-mind (action man) (status known) (value friendly) ) )
(assert (robo-mind (action man) (status unknown) ) )
```

If we don't provide a value for a slot and if he doesn't have a default value then the special value *nil* is used.

Ø Deleting facts from Jess

After a rule had fired for a specific fact and a next event happened then we need first to delete the previous fact from the score board and add a new one.

To delete facts in Jess we must first know the fact id. As an example of deleting a fact is shown below:

```
(defrule is-friendly_intruder

  ?id <-(robo-mind (action ?X&:(eq ?X man ))(status ?Y&:(eq ?Y unknown))(value ?Z&:(eq
?Z friendly)))

     =>

 (retract ?id)

 (assert (robo-mind (action intruder) (status friendly) ) ))
```

Ø Rules in Jess

Rules in Jess can generate actions based on the contents of one or more facts. A Jess rule is something like an `if...then` statement in a procedural language, but it is not used in a procedural way. While `if...then` statements are executed at a specific time and in a specific order, according to how the programmer writes them, Jess rules are executed whenever their `if` parts (their *left-hand-sides* or *LHS*s) are satisfied, given only that the rule engine is running. This makes Jess rules less deterministic than a typical procedural program.
Rules are defined in Jess using the `defrule` construct. A rule in Jess looks like this:

```
(defrule intruder-detected
     (robo-mind (action ?X&:(eq ?X intruder))(status ?Y&:(eq ?Y nil))(value ?Z&:(eq ?Z
nil)))
               =>
     (behavior angry))
```

To have a more legible code we used a CPL file. In the CPL file we write all the rules and functions in Jess. The file is loaded with the *bach* command from the Java program. The CPL file can be seen well in Appendix C.

- XML

  The XML file is the database of the knowledge base. Every event the dog detects through his sensors is registered and put in the XML file.

  XML stands for eXtensible Markup Language and is a language to define structured information. It is very lightweight, as opposed to a traditional database, and can be edited relatively easy. All XML files need to obey to certain grammar rules of XML. Also the file has a predefined structure. These structures can be defined in Document Type Declaration file or DTD witch define what structures are allowed in the XML file. XML files use just as HTML tags to separate the structure from data. A tag indicates that information will follow, at the end of the information needs to be a closing tag. Everything from the beginning of a tag till its end it is called element. Every element has to be defined in the DTD and it defines how each element is build up.

  For our system we will use a XML file containing the possible scenarios that can appear while the AIBO is watching the house. The DTD file is defined as follows:

```
<!ELEMENT scenariolist (scenario*) >
<!ELEMENT scenario (scenario_name,status*,value* )>
<!ELEMENT scenario_name (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ELEMENT value (#PCDATA)>
```

  A valid XML file has an element *scenariolist*, which is the main structure. The *scenariolist* can have zero or more *scenario* as indicated by *. Each *scenario* has a *scenario_ name*, zero or more *status* and zero or more *value*. The scenario_name and value are defined as #PCDATA witch is Parsed Character Data and means it can contain an arbitrary string of characters. For a complete overview of the used XML file see Appendix B.

To read the information from the XML file we used a class called MyXMLReader, and create String from the information we get out of it. After that we insert them into the Rete engine. Below is shown how we get the facts from the XML file and put them in the Rete.

```
public void addFact(Event event)
    {   try
        {   engine.reset();
            String jessCode = "";
            String []action = event.getEvent();
            jessCode += "(assert (robo-mind (action "+ action[0]+")";
            if (action.length > 1)  jessCode +="(status " +action[1]+")";
            if (action.length > 2)  jessCode +="(value " +action[2]+")";
            jessCode += "))";
            engine.executeCommand(jessCode);
            activateCommand();
        }
        catch (JessException je)
        {}
}
```

In the event we put the String read from the XML file. For the moment we have more scenarios in the XML file and every 30 seconds we choose one randomly. In time in the XML file will be put all the information gathered from the environment and send to the dog through his sensors.
To have a closer look on how we read the XML file you can look at Appendix D at the end of the report.

4.2 The framework of the AIBO watchdog

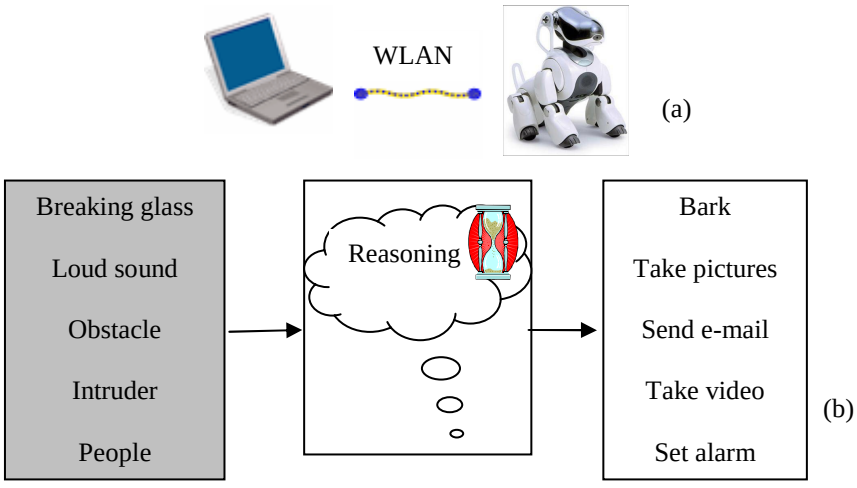

Figure 4.1 Framework of AIBO watchdog

Because we used liburbi Java we do not have an autonomous robot. The program will be on the computer and we only connect to the AIBO and send him commands. We can say that the dog will only execute the commands we send to him. The software we develop can control AIBO remotely via wireless LAN.

The input parts in the figure 4.1. (b) are not implemented in the project. In table below is described the implementation status.

Table 4.1 Implementation status

Requirement	Implementation status
R.1 Start WatchDog	Finished, it is able to start the watchdog, although the procedure could be modified, because it will be desiderated that the WatchDog starts automatically not only by pushing a button.
R.2 Stop WatchDog	Finished, the WatchDog can easily be stopped.
R.3 Move inside the house without bumping into obstacle	Unfinished, is dependant on R.5.
R.4 Walk forward and backwards	Finished, but it needs improving because now the WatchDog has a way of walking we called "crawl" but the motion is bumpy and some of the sensors (camera) are not so accurate. For that we need a smoother motion.
R.5 Recognize walls and obstacles	Unfinished, the robot does not detect walls or other obstacle.
R.6 Discover intruder	Unfinished, the robot does not detect intruder.
R.7 Barking	Finished. The dog has 3 ways of barking: a normal bark, a happy one and angry one.
R.8 Save picture	Finished. It could save picture in jpeg format on the computer to a specified path.
R.9 Send mail	Finished. It could send mail with the pictures taken to a specified address.
R.10 Set alarm	Finished. It could set an alarm consisting on playing a beep sound a couple of times.
R.11 Respond to intruder and the other threats as motion, loud sound	Finished. The dog response to the intruder with a bark, an angry one and takes pictures and sends them through mail at the owner. And if a loud sound is detected then he will go to explore and see what is going on. If a motion is detected he will bark and take picture and also sends mail.
R.12 Reasoning	Finished. We modeled the "mind" of the WatchDog. Of course this mind could suffer improvements.

The current implementation focused on reasoning therefore the input module will be developed in other project. The project was designed as modular as possible so that it will be easy to connect with input sensing module. The only thing the input sensing module will need to do is to transmit some parameters that will be written in the XML file and later on process them by the reasoning module.

For developing the reasoning module we used JESS and for the output behavior we used liburbi Java.

4.3. Client server communication

Client/server is a computational architecture that involves client processes requesting service from server processes.

URBI is a robot-independent API based on client/server architecture [4]. We can make use of the URBI server in two ways: via a computer through the wireless LAN using the liburbi java (external client) or through direct inter-process communication using liburbi OPEN-R (onboard client).

In the case of an external client, the communication then takes place through a TCP/IP or IPC connection. When using URBI over TCP/IP messages are send to the URBI object via telnet over port 54000. The URBI Object then sends the appropriate messages to the other objects on the OPEN-R system to accomplish the given command.

In the case of an onboard client, the client is also an OPEN-R object (containing doinit(), dostart(), etc.) that runs along in the OPEN-System and sends messages to the URBI server. Thus, URBI functionality can then be utilized by passing/receiving messages to/from the URBI object (which of course should also be running).

In the general case: when a message is sent to the URBI server (no matter if this is through an external client or through an onboard client), the server object will send message(s) to other objects as a result of this message, thereby hiding the internal message passing details to the client. Therefore, we only need to be concerned with the message passing details to and from our client to the URBI server. The URBI server is on the memory stick on the AIBO and the client is on the computer. The client communicates with the server through the wireless LAN.

Chapter 5 Testing

5.1 Testing results

Testing ensures that the system is according to the systems specification, and there should be a recognizable relationship between requirements and the planned tests. Testing of the product is necessary to verify that the system's functionality matches the functionality described in the requirements document of the system.

Many of the requirements such as discovering the intruder and other threats and recognizing walls have not been implemented and we had to simulate them and because of this the results of the testing is not conclusive. Moreover due to the limitation of the time we had no time to test all the things we had implemented. Future testing and validation of the requirements should at least contain the tests outlined in Table 5.1.When all the requirements would be implemented a more formal testing should be made.

Table 5.1: Suggested tests for robot watchdog

Requirement	Test	Repetitions
R.1 Start watchdog	The robot starts operating when the start watching button is pushed.	5
R.2 Stop watchdog	The robot is paused when the Stop watching button is pressed.	5
R.3 Move inside the house without bumping into obstacle	The robot is able to move in an approximately straight line in 8 meters	10
R.4 Walk forward and backwards	The robot is able to walk forward	5
R.5 Recognize walls and obstacles	The robot detects a wall and an obstacle when placed in front of it	5 x 5 different angles
R.8 Discover intruder	The robot discovers an intruder when an intruder is present in the robot's area of vision	10
R.7 Barking	The robot barks.	5
R.8 Saving pictures	The robot saves picture on the computer to a certain path.	5
R.9 Send mail	The robot sends a mail to an certain address mail	5
R.10 Set alarm	The robot sets an alarm consisting on playing a beep sound a couple of times.	5
R.11 Respond to intruder and any other threats	The robot response to the intruder and any other threats that he knows such as motion and loud sound	10
R.12 Reasoning	The robot must decide what action should be taken according with the event received	10

Chapter 6 Conclusion and Recommendation

In this chapter we evaluates and summarizes the main results of this projects and give directions for further work

6.1 Discussion

In this section we discuss some of the choices made during the project.

The most discussable choice in this project is the choice of the URBI Java framework. This is because using URBI we do not have yet the possibility of creating an autonomous robot but the developers of URBI promised that in time this will not be a problem and we can create autonomous robot. For now the robot will be manipulated with the help of the computer. The server is on the robot and the client runs on the computer. Another inconvenience of using URBI is that it is very new and it has not yet implemented many of important features like walking, turning and we had to use the one implemented in the Companion dog project. Motion commands used cause jerky movements in the joints when the robot pauses and then starts again and when walking the quality of the video is not very good. URBI is a Universal Robotic Body Interface and allows you to control any robot with a powerful script language that you can interface with any of your favorite programming language (C++, java, matlab …) and OS (Windows, Mac, Linux). URBI is designed with a constant care for simplicity. There is no "philosophy" or "complex architecture" to be familiar with. It is understandable in a few minutes and can be used immediately. URBI is a low level command language. Motors and sensors are directly read and set. Although complex high level commands and functions can be written with URBI, the raw kernel of the system is low level by essence.
Using OPEN-R could have been another possibility but in OPEN-R less functionality would have been possible to. Another possibility would have been Tekkotsu. But with Tekkotsu we also have the same problem with the motion commands (at least WalkMC) which can cause jerky movements in the joints when the robot pauses and then starts again. And moreover Tekkotsu and OPEN-R are designed only for programming AIBO and not for any other robot such as URBI.

6.2 Conclusion

The watchdog project aimed to “train” AIBO ERS-7 dog to be a watchdog. The current dog is able of receiving randomly possible events from the environment. He has a reasoning module that makes him capable of “thinking” and “analyzes” the threats we had “trained” him to recognize and take a certain action. The selection of the reactions is done in real time. The reasoning part is the focus of the project. He is capable of barking, starts the alarm, saves pictures and sends them through mail at a certain e-mail address. We can see also everything the dog sees. If the user have a center of surveillance in home he can watch from there what happens inside the house. He could also walk or better say crawl inside the house and explore the area but he will bump into walls or any other obstacles because he is not yet capable of recognizing walls and obstacles.

We developed watchdog system using Java to extract inputted events written in XML messages. Using XML as an input, we expect the system flexible to be coupled with any input fusion module. This input fusion module should link to some recognition modules, such as

object recognition, distance recognition and sound recognition, to process different possible input from environment using AIBO's sensors, such as: sound, image and gesture. The input fusion processes the received recognitions and sends to our developed watchdog system using an XML format. The current watchdog system will process the input as a new event and send it to the reasoning module.

Jess engine [7] was used to infer the rules to select appropriate behavior. This rule-based reasoning was developed by listing possible events in a house environment as possible facts and associating with possible reactions of a real watchdog. The rule-based approach opens up opportunity to develop the reasoning module incrementally.

URBI [9] has been chosen to provide the system with a library to activate the behavior of AIBO, such as: motion generation, sound generation, and picture capturing. Client-Server connection has been used to establish communication between the program and the URBI server on AIBO through AIBO wireless network using TCP/IP connection. The capabilities of current developed AIBO are:

Till this moment AIBO has been "trained" to perform well different tasks such as entertain, watch-dog, rescue dog [12]. These entire tasks have in common the physical/hardware components of an AIBO. While some might argue that a watchdog does not need personality, moods or emotions it is our believe that all these task orientated AIBO's need to have in common also a complex personality model [12]. This is why in the future it will be desirable that all projects developed in MMI group at TUDELFT should be combined in one framework

6.3 Recommendation

Future work will be required in order to have a real input mechanism from a developed input fusion module. This work will be developed on the project of developing an AIBO framework within the MMI group at TUDELFT, which combines the WatchDog, the Companion dog, and the Cooperate dog. As mentioned earlier, a certain mechanism to reason environment and behavior for each task for the dog should also be designed and developed. In this case, we will use task modes with priority. This moment the dog has the thinking and the behavior of one surveillance dog but not the "seeing" and "hearing".

Furthermore, field test with real users, real environment and real situations are necessary to gather the data about how people might react to AIBO and vice versa, how AIBO might react towards real events, and how people experience this. Therefore the design can cover more requirements in real context use.

The implementation status of the different requirements is given in Table 4.1. Formal testing has not been carried out, due to time limitations. The requirements that are unfinished, or can be improved are discussed. Examples of extensions to the current functionality of the AIBO watchdog are as following:

R.6 Walk forward and backward

At the current stage the robot is able to walk or better said to crawl forward and backward. However, the motions are rather bumpy; he does not stand on his paw he only stands on his

elbows. But with this kind of walk picture and video quality needs to be improved and therefore we need a smoother walking algorithm. This is a subject for future work.

R.5 Recognize walls and obstacle

The robot doesn't detect walls or any other obstacle. This also can be made by integrated in the project the pattern recognition project and also using the distance sensor of the AIBO. This would be a subject for future work.

R.3 Move inside the house without bumping into obstacle

This requirement depends on the functionality described in the functional requirement R.5. An algorithm for planning where to walk is necessary if the robot shall move in complex areas. This is not yet implemented. The robot should be able to turn around corners once a path finding algorithm and wall recognition has been successfully implemented.
As an improvement the robot could be supplied with the plan of the house. This would extend the feasibility of the watchdog. With the plan the dog will be able to move inside the house without bumping into walls so he only needs to recognize obstacles.

R.6 Discover intruder

The dog does not discover the intruder this also because the project of facial recognition is under construction. This will also be a subject for future work.

R.11 Response to intruder and other threats

The dog is able to respond to the intruder and to other threats that he knows. In the future the dog should be able “to learn” new threats and from his behavior to choose one that he “thinks” is more suitable. If a new threat appears he would first discover the similarities with other threat that already exists in his memory and after that decides what kind of behavior to choose. This it will be very helpful because we do not have to train him from the beginning to recognize all threats. He will know the basic ones and when he discovers a new one he just add it to his database and acts as the one who is most similar to. This way we could say that the AIBO watchdog is able to learn.

R. 12 Reasoning

This part was made using a rule-based expert system. The rules are very clear but they are not flexible and if an unknown event happens the dog will not react to it in any kind. I think another expert system will be much better to use for example a fuzzy expert system or maybe Neural and Adaptive Systems. This will be also a subject of research for future trail of watchdog prototype.

Chapter 7 Appendix

Appendix A

The WatchDog stages:

- **Start**

To start the watchdog you need to copy the program on the memory stick and after that insert the AIBO memory stick into AIBO. Then boot AIBO.

- **Background initialization stage**

After booting, AIBO will first needs to initiate his background image. During the initialization period, AIBO shows the face from the picture. You and any moving objects should not be viewed by the AIBO during this period. After AIBO finishes the background initialization stage he will turn off the light and move on to the watching stage.

- **Watching stage**

While in watching stage if AIBO detects any moving objects, he will bark at the objects and show his face as the following picture shows.

He will save a picture in the /OPEN-R/MW/DATA/P directory. The captured pictures will be saved in JPEG format and the file name format will be CAP_NNN.JPG, where the NNN is a serial number from 000 to 299. If the serial number reaches the maximum number, it will then begin counting from 0 again. He will also save the last captured picture in the CAP_LAST.JPG file.

- **Background-re-initialization**

AIBO will go into the background re-initialization stage if:

- There are lots of image changes. This case may be caused by the turning on/off the lights, or moving his head/body.
- He has continuously barked for more than 10 times.
- Someone touches his back sensor (rear) for 1 second.

- **Auto gain control**

This program will automatically change the camera gain (low, middle, or high) based on the current environment. If you put the AIBO in a dark environment he cannot detect motions.

- **Download the pictures via FTP**

You can download the picture saved on the memory stick with ftp but first the AIBO wireless functions should be enabled. And you need to change the directory to `DATA/P.` The pictures can be downloaded using the **`get`** command.

- **WatchDog web functions**

This function enables you to browse the captured pictures via a web browser. First the AIBO wireless functions should be enabled and on the web browser input **`http://<AIBO_ip_address>.`** The home page will show the current watching image, and the image will be updated periodically and automatically.

Appendix B

The XML file wd.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE wdog [
   <!ELEMENT scenariolist (scenario*) >
   <!ELEMENT scenario (scenario_name,status*,value* )>
   <!ELEMENT scenario_name (#PCDATA)>
```

```
   <!ELEMENT status (#PCDATA)>
   <!ELEMENT value (#PCDATA)>
]>
<scenariolist>
   <scenario>
      <scenario_name>intruder</scenario_name>
      <status>friendly</status>
   </scenario>
   <scenario>
      <scenario_name>intruder</scenario_name>
      <status>unfriendly</status>
   </scenario>
   <scenario>
      <scenario_name>loud_sound</scenario_name>
      <status>behind</status>
   </scenario>
   <scenario>
      <scenario_name>loud_sound</scenario_name>
      <status>front</status>
   </scenario>
   <scenario>
      <scenario_name>loud_sound</scenario_name>
      <status>right</status>
   </scenario>
   <scenario>
      <scenario_name>loud_sound</scenario_name>
      <status>left</status>
   </scenario>
   <scenario>
      <scenario_name>motion_detected</scenario_name>
   </scenario>
   <scenario>
      <scenario_name>obstacle</scenario_name>
   </scenario>
   <scenario>
      <scenario_name>no_obstacle</scenario_name>
   </scenario>
   <scenario>
      <scenario_name>nothing</scenario_name>
   </scenario>
   <scenario>
      <scenario_name>man</scenario_name>
      <status>known</status>
      <value>friendly</value>
   </scenario>
   <scenario>
      <scenario_name>man</scenario_name>
      <status>known</status>
      <value>unfriendly</value>
   </scenario>
   <scenario>
      <scenario_name>man</scenario_name>
      <status>unknown</status>
```

```
      :
      :
      :
      :
      :
      :
      :
            <value>unfriendly</value>
         </scenario>
         <scenario>
            <scenario_name>man</scenario_name>
            <status>unknown</status>
            <value>friendly</value>
         </scenario>
         <scenario>
            <scenario_name>man</scenario_name>
            <status>unknown</status>
         </scenario>
      </scenariolist>
```

Appendix C

The rule.cpl file:

```
load-package wdog.LoadBehavior)

(deftemplate robo-mind (slot action) (slot status) (slot value
))

(deftemplate robo-state (slot do_action) (slot next_action))

(deffunction start-bark (?bark_type)
     (bark ?bark_type)
    )

(deffunction be-unfriendly ()
         (save_picture)
         (start-bark angry)
         (set_alarm)
           )

(deffunction explore()
               (look around)
               (crowl_front))
(deffunction turning()
      (turn right)
      (turn right) )

(deffunction behavior(?bark_type)
    (save_picture)
   (set_alarm)
   (bark ?bark_type) )

(defrule is-owner
    (robo-mind (action ?X&: (eq ?X owner)))
      =>
    (start-bark happy))

(defrule friendly-intruder
   (robo-mind (action ?X&:(eq ?X intruder))(status ?Y&: (eq ?Y
friendly))(value ?Z&:(eq ?Z nil)))
         =>
    (behavior normal))


(defrule unfriendly-intruder
   (robo-mind (action ?X&:(eq ?X intruder))(status ?Y&: (eq ?Y
unfriendly))(value ?Z&:(eq ?Z nil)))
         =>
   (be-unfriendly))
```

```
:
:
:
:
:
(defrule intruder-detected
   (robo-mind (action ?X&:(eq ?X intruder))(status ?Y&:(eq ?Y
nil))(value ?Z&:(eq ?Z nil)))
         =>
    (behavior angry))

(defrule is-friendly_intruder
    ?id <-(robo-mind (action ?X&:(eq ?X man ))(status ?Y&:(eq
?Y unknown))(value ?Z&:(eq ?Z friendly)))
         =>
   (retract ?id)
    (assert (robo-mind (action intruder) (status friendly) )
))

(defrule is-unfriendly_intruder
   ?id <-(robo-mind (action ?X&:(eq ?X man )) (status ?Y&:(eq
?Y unknown))(value ?Z&:(eq ?Z unfriendly)))
         =>
    (retract ?id)
   (assert (robo-mind (action intruder) (status unfriendly) )
))

(defrule is-intruder
   ?id <- (robo-mind (action ?X&:(eq ?X man ))(status ?Y&:(eq
?Y unknown))(value ?Z&:(eq ?Z nil)))
         =>
    (retract ?id)
   (assert (robo-mind (action intruder))))

(defrule is-unfriendly
   (robo-mind (action ?X&:(eq ?X man))(status ?Y&:(eq ?Y
known))(value ?Z&:(eq ?Z unfriendly)))
         =>
   (bark normal))

(defrule is-not-intruder
   (robo-mind (action ?X&:(eq ?X man))(status ?Y&:(eq ?Y
known))(value ?Z&:(eq ?Z friendly)))
         =>
   (bark happy))

(defrule motion-detected
   (robo-mind (action ?X&: (eq ?X motion_detected)))
         =>
   (behavior normal))

(defrule obstacle-detected
     (robo-mind (action ?X&: (eq ?X obstacle)))
           =>
     (look right)
    (turn right))

(defrule no-obstacle
```

```
   (robo-mind (action ?X&: (eq ?X no_obstacle)))
         =>
   (crowl_front))

(defrule nothing
   (robo-mind (action ?X&: (eq ?X nothing)))
         =>
   (default_state))

(defrule sound_detected_front
   (robo-mind (action ?X&: (eq ?X loud_sound))(status ?Y&: (eq
?Y front)))
         =>
   (look ahead)
   (explore))

(defrule sound_detected_right
   (robo-mind (action ?X&: (eq ?X loud_sound))(status ?Y&: (eq
?Y right)))
         =>
   ;(look right)
   (turn right)
   (explore))

(defrule sound_detected_left
    (robo-mind (action ?X&: (eq ?X loud_sound))(status ?Y&:
(eq ?Y left)))
         =>
   ;(look left )
   (turn left )
   (explore))

(defrule sound_detected_behind
   (robo-mind (action ?X&: (eq ?X loud_sound))(status ?Y&: (eq
?Y behind)))
         =>
   (turning)
   (explore))
```

Appendix D

Here is the code of MyXMLReader where we read from XML file and transform the elements into String.

```
package wdog;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.ParserConfigurationException;

import org.xml.sax.SAXException;

import java.io.File;

import java.io.IOException;

import org.w3c.dom.Document;

import org.w3c.dom.*;

import java.util.Vector;


public class MyXMLReader

{

 private Document document;

 public MyXMLReader(String filename)

 {

  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

  try

  {

   DocumentBuilder builder = factory.newDocumentBuilder();

   document = builder.parse( new File(filename) );

  }

  catch (SAXException sxe)
```

```
    {
      // Error generated during parsing
      Exception  x = sxe;
      if (sxe.getException() != null)
          x = sxe.getException();
      x.printStackTrace();
    }
    catch (ParserConfigurationException pce)
    {
      // Parser with specified options can't be built
      pce.printStackTrace();
    }
    catch (IOException ioe)
    {
      // I/O error
      ioe.printStackTrace();
    }
  }
 /**
  *
  * @return the number of scenarios
  */
  public int getScenario()
  {
   NodeList s = document.getElementsByTagName("scenario");
   return s.getLength();
  }
```

```
/**
 * @param sNumber - represent the number of the scenario to return
 * @return an array of Strings with the scenarios from the XML file
 */
public String[] getScenario(int sNumber)
   {
   NodeList s = document.getElementsByTagName("scenario");
   Node s1 = s.item(sNumber);
   NodeList tags = s1.getChildNodes();
   Vector v = new Vector();
   int j=0;
   for (int i = 1 ; i<tags.getLength(); i+=2)
   {
      v.add(tags.item(i).getFirstChild().toString());
      j++;
  }
   String[] tagString = new String[v.size()];
   for (int i = 0; i < tagString.length ; i++)
      tagString[i] = v.get(i).toString();
   return tagString;
 }
```

Here is the code of MyXMLWriter where we write in the XML file:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
```

```
import org.xml.sax.SAXException;
import java.io.File;
import java.io.IOException;
import org.w3c.dom.Document;
import org.w3c.dom.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

public class MyXMLWriter
{
        private Document document;
        public MyXMLWriter(String filename)
        {
          DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
          try
          {
            File f = new File(filename);
            DocumentBuilder builder = factory.newDocumentBuilder();

            document = builder.parse( f );

          }
          catch (SAXException sxe)
          {
             // Error generated during parsing
             Exception  x = sxe;
```

```
            if (sxe.getException() != null)
                x = sxe.getException();
            x.printStackTrace();
        }
        catch (ParserConfigurationException pce)
        {
            // Parser with specified options can't be built
            pce.printStackTrace();
        }
        /* catch(TransformerConfigurationException tco)
            {
             tco.printStackTrace();
            }
        catch(TransformerException te)
            {
             te.printStackTrace();
            }*/
        catch (IOException ioe)
        {
            // I/O error
            ioe.printStackTrace();
        }
    }
    public void write(String scenarioName, String status, String value)
    {
        Element scenarioList = document.getDocumentElement();
```

```
  Element scenario = document.createElement("scenario");
  scenarioList.appendChild(scenario);

  Element sName = document.createElement("scenario_name");
  scenario.appendChild(sName);
  sName.appendChild(document.createTextNode(scenarioName));
if (!status.equals(""))
{
  Element sStatus = document.createElement("status");
  scenario.appendChild(sStatus);
  sStatus.appendChild(document.createTextNode(status));
}

if (!status.equals(""))
{
  Element sValue = document.createElement("value");
  scenario.appendChild(sValue);
  sValue.appendChild(document.createTextNode(value));
}
try {
 // Prepare the DOM document for writing
 Source source = new DOMSource(document);

 // Prepare the output file
 File file = new File("wdog\\watch.xml");
 Result result = new StreamResult(file);
```

```
            // Write the DOM document to the file
            Transformer xformer = TransformerFactory.newInstance().newTransformer();
            xformer.transform(source, result);
        } catch (TransformerConfigurationException e) {
        } catch (TransformerException e) {
        }
    }
}
```

Appendix E

1.URBI language

The main part of URBI is the URBI language which defines how commands can be sent to the robot, what kind of scripting features are available and the syntax associated. The working cycle of URBI is to send commands from the client to the server and to receive messages from the server to the client. Commands can be written directly in a telnet client on port 54000, where the messages will also be displayed or, as we will see later, using a program and a library.

1.1 Getting and setting a device value

As we said in the introduction, each element of the robot is called a device and has a device name. For example, in the case of AIBO, here is a short list of devices: legFL1, neck, camera, speaker, micro, head sensor, accelX, pawLF, and ledF12...The first thing that can be done with a device is to read its value. This is done with the "val "method. For example, it can be done with a telnet client opened at port 54000 on the robot. We start every line with the">" sign to show the command prompt but this is not visible in a normal telnet session, nor is it part of the command syntax. Other lines without">" are messages from the server:

> neck.val;

[036901543: notag] 15.1030265089

The message returned is composed of a first part between brackets displaying a timestamp in milliseconds (from the start of the robot) and a command tag. In this case, the command tag is "notag" since no tag has been specified with the command. The tag can be specified before a ":", preceding the command. With the command tag, it is possible to retrieve the associated message later, possibly in a flow of other messages from the server:

> mytag: neck.val;

[041307845: mytag] 15.0040114317

This tagging feature is an essential part of URBI and the URBI Java library where callback functions can be associated to any tag, enabling asynchronous message handling. Error messages are tagged with the "error" tag when they are not related to a specific command. This is useful when one wants to implement an error handling function, which will be triggered each time a message tagged with "error" is received. Warning messages also exists.
The second part of the message is the response of the server. In the case of our example, it gives the value of the AIBO neck device which is the position of the neck motor in degrees. The "val" command can be used with any device. The type of data returned depends on the device and can be checked with the "unit"instruction:
> unit neck;
[041447411: notag] "deg"
Available units are, among others, "deg", "bool", "lum" (luminosity),"cm", "Pa" (pressure), "m/s^2 "(acceleration).
The minimal and maximal range is also available with any device, using the"maxrange" and "minrange" instructions. For example, in the case of the headPan device:
> therangemax: rangemax headPan; therangemin: rangemax headPan;

[057845441:therangemax] 91.0000000
[057845441:therangemin] -91.0000000
Note that the “valn” method also exists and is similar to “val” except that it handles a normalized value between 0 and 1, computed from the minrange and maxrange values of the device. This is useful when one does not know a priori the unit and range of a device or to write programs which are to certain extend”robot-independent “.
The “val” method can also be used to set a particular device value. If the device is a motor, it is going to move to the specified value. In the case of a LED, this will switch it to the corresponding illumination (between 0 and 1):
> motoron;
> headPan.val = 15; headTilt.val = 20;
The “motoron” command is necessary at the beginning to start the motors of the robot. The two next commands set the position of the head pan and tilt. Note that, by default, a setting command does not produce any return message. This can be modified with options associated to the tag at the beginning of the command, like the “+report” option which tells the server to be in “verbose” mode. The client will be prompted when the command starts and when it finishes. With the “error” tag, the server will notify the client if any error occurs (obstruction on the motor movement for example).
> mycommand +report: legLF2.val = 15;
[058477124: mycommand] *** start
[058477156: mycommand] *** stop
Note how meta information regarding the command is prefixed by ***.This is to make clear that the message is not a return value and has no type. Command specific error messages and warning messages are also prefixed by ***.The tag “mycommand” is compulsory since if one uses only “+report” it would be difficult in general to know what the “start” and “stop” refers to when several commands are started at the same time.

1.2. Modificators

The value specified by a “val” command is reached as quickly as the hardware of the robot allows it. It is however possible to control the speed and other movement parameters using modificators. The following example commands the robot to reach the value 80 deg for the motor device headPan in 4500 ms and the value 40 deg for headTilt with a speed of 12.5 degrees per seconds:
> headPan.val = 80 time: 4500;
> headTilt.val = 40 speed: 12.5;
The speed or time modificators are always positive numbers It is possible to specify a speed without giving a targeted final value by setting the desired value to infinity (inf) or minus infinity (-inf).For example, in the case of a wheeled robot, one might need to control the right wheel speed with:
> wheelR.val = inf speed: 120;
If the range of the device is not infinite, the command will stop when the value reaches maxrange or minrange. Another interesting modificator is “accel” those meaning is to control the acceleration, and “sin” which tells the robot to reach the value in a specified time and in a sinusoidal way. This is useful when a circular movement is required. In future versions of URBI, it is planned to provide a way to define custom modificators, based on any motion profile and not only sinusoidal trajectories. This will be particularly useful to describe walk sequences. Modificators can be combined, “time” being prioritary over “speed” which is prioritary over “accel”:
> wheelR.val = 150 speed: 120 time: 2000;
This command means that the value 150 must be reached at speed 120.This speed must be reached in 2000ms (this sets an implicit first phase of acceleration of 60 deg/s ^{2}).The priority

between modificators tells which modificator is modifying the others in case of a combined use.
An important point about modificators is that it is not only available to set devices but for any kind of variable (variables of a device or global variables).
Considering the following example:

```
> myvariable = 0;
> myvariable = 50 time: 10000;
> myvariable;
[001410040: notag] 2.45471445
> myvariable;
[001412020: notag] 12.35471445
> myvariable;
[001442020: notag] 50.00000000
```

The first affectation sets the variable to zero and the second one tells the robot to reach the value 50 in 10 seconds. When the value of “myvariable” is checked over time, we see that it is evolving from 0 to 50 during this time interval. This is a unique and powerful feature of URBI compared to other existing languages and which makes it a fundamentally asynchronous and time-oriented language. It allows creating a dynamics for parameters, useful in many situations like for example in the design of a “walk” sequence for a legged robot.

1.3. Binary data sending and receiving

Some devices like cameras, speakers or microphones are handling binary data. In the case of AIBO, the camera device is called camera and its value (the current image) can be retrieved with a “val” command, just like any other device:

```
> camera.val;
[004757741: notag] BIN 5347 jpeg 208 160

############################################################
########### 5347 bytes of binary data ##############...
```

Binary data always starts with the word “BIN” immediately followed by the number of bytes that will be received. Extra information follows. In the case of an image, it is the image encoding type (jpeg or YCbCr for AIBO) and the image height and width. After the carriage return, the binary data starts. The system switches back to ASCII mode after the last byte is sent. It is possible to specify in which format the image should be sent (jpeg or YCrCb or, in future versions, mpeg) with the “format” method of the camera device. It is also possible to set the jpeg compression factor, the camera gain, white balance or shutter speed. Here is an example session:

```
> camera.format = 0;
> camera.jpegfactor = 80;
> camera.gain = 1;
> camera.wb = 2;
> camera.shutter = 0;
> camera.resolution = 0;
```

See the specific AIBO URBI server documentation for the precise meaning of the values. This example is interesting here because it shows how device-specific variables can be set using other methods than “val” or “valn”. This is illustrating the ease of use of the “device.method” format chosen for URBI and how extensible it can be to support any kind of device with particular methods that might exist in future robots. This is an important point if URBI is to become a standard. In the case of sound, for the AIBO micro device for example, one can request the incoming sound for 500ms with the following command:

> micro.val(500);
[004857788: notag] BIN 2048 wav 16000 16 2
##
########## 2048 bytes of binary data ##################
##...

If the parameter 500 is omitted, the sound is streamed without interruption, in small samples of 32ms (it can be stopped with a “micro.stop” command).The extra information after “BIN 2048” is the sound encoding (wav file in the case of AIBO), sample rate in Hz, encoding depth in bits and the number of channels. To send sound, the syntax is similar:
> speaker.val = BIN 2048 wav;
##
########## 2048 bytes of binary data ##################
##...

Note that since the server is waiting for a command terminated by a semicolon, it is right after the semicolon that the binary data starts. Of course, these binary transfers are not usable when the client is a telnet client and it is necessary to use a programming interface like the C++URBI Library to make a full use of these features. As we will see, many functions are provided to easily and transparently send or receive binary data. Note that it is possible to play a specific file on the robot with speaker.play (”myfile.wav”) sent from a telnet client.

1.4. Serial and parallel commands

One key feature of URBI is the ability to process commands in a serial or parallel way. When separated by the “&” separator, two commands will be executed in parallel. Moreover, they will start at exactly the same time:
> headPan.val = 15 & headTilt.val = 30;
This will move the head pan and tilt together, with both motors starting at the same time. In the same way, it is possible to serialize commands by separating them with a pipe. In that case, the second command will start just after the first one is finished, with no time gap.
> headPan.val = 15 | headTilt.val = 30;
This will move the head pan to 15 degrees and only when this value has been reached, and just after, it will start to move the headTilt motor.
Two commands separated by a semicolon have almost the same time semantics as the serial ”|”: the second will start after the end of the first, but the time gap between the end of the first and the beginning of the second is not specified. Most of the time, URBI commands will be separated by semicolons. The main difference in using semicolons is that commands are interpreted as they arrive, whereas with pipes the chain of commands cannot start until it is completely received (and terminated by a semicolon for example). Finally, two commands can be separated by a colon. In that case, the time semantics is close to the parallel operator “&” except that the two commands don’t necessarily start at the same time. The meaning of a colon terminated command is simply to start the command as soon as possible. In particular, as soon as the command is in the buffer, it will be executed, whereas with “&” the chain of commands must be integrally received before execution.
The following relationships represent those time dependencies:
a;b : b.start >= a.end
a,b : b.start >= a.start
a&b : b.start == a.start
a|b : b.start == a.end
The operator priority is the following: ; , & |

These time sequencing capabilities are another specificity of URBI and are very important features to design and chain complex motor commands or behaviors.

1.5. Loops, conditions, event catching

Several control structures are available, like the classical “for”, “while” and “if”. Some extra control structures like “loop” which is equivalent to “while (true)”or “loopn (n)”equivalent to “for(i=0 ; i <n; i++)” are also provided for convenience.
The syntax of”for”, “while” and “if” is the same as in JAVA. Here are some examples:

```
if (ledF11.val == 1)
{
if (ledF12.val == 0)
ledF12.val=0.254 speed: 0.1;
ledF11.val = 1;
} else
ledF11.val=1 time 1454;
for (i=0;i<100;i++) {
headTilt.val = i/100 sin: 124;
ledF10.val = i/100;
};
while (legLF1.val < 12)
legRF1.val = lefLF1.val;
```

As a specificity of URBI, event catching control structures like “whenever” and “at” are also available. The instruction “at (test) command “will execute the command only once at the moment when the test becomes true. It is possible to set a hysteresis threshold associated to the test so that the test has to be false n times before it can trigger the command again when it becomes true. This is done with the tilde separator in the test. The following example let the head move in circles, except when an object is detected in the 25cm short range for an AIBO:

```
period=2500;
at (distanceNear.val > 25 ~ 3)
scanning: loop {
{ headTilt.val = 90 sin:period |
headTilt.val = -90 sin:period }
&
{ headPan.val = 90 sin:period |
headPan.val = -90 sin:period }
}
else
stop scanning;
```

In this example, the hysteresis threshold is set to 3, which means that the test must be false 3 times before it can trigger the “loop” command again. The meaning of the “else” part is symmetrically identical. The “stop” command, followed by a tag name, means that any command with this tag will be stopped. This is used here to stop the head circular sweeping. Note how the serial and parallel operators are used to specify the circular head movement. The number after the tilde operator can also be a time period. In that case the time unit must follow the number, like”50ms “. The instruction “whenever (test) command” will execute the command as long as the test is true. When the test becomes false, the command is not restarted once it is finished and the “whenever” instruction waits for the test to become true again. Without entering into details, we can say that the mechanism used by the URBI kernel

to process commands involves command substitution in the command stack. When executed, the command “if (test) command1 else command2 “is transformed into “command1” if (test) is true, and “command2” otherwise. In the same way, when the command “while” is executed, it is immediately replaced by the following command:

```
while (test) instruction;
<=>
if (test) {
instruction;
while (test) instruction;
}
else noop;
```

Note that “noop “is an instruction which take a cycle to execute and does nothing. This substitution mechanism is computationally efficient and, as a side effect, gives a precise way to describe the semantics of the instructions. Here is the semantics of “at” and “whenever”as described in the kernel:

```
whenever (test) instruction;
<=>
if (test) {
instruction;
whenever (test) instruction
}
else {
noop;
whenever (test) instruction
}
at (test) instruction; <=>
if (test) {
instruction;
at (!test)
at (test) instruction;
}
else {
noop;
at (test) instruction;
}
```

We are currently working on a more formal mathematical description of the URBI semantics, but the above description will be used as a reference. Like “else” or the “if “instruction, there is a “else” part for whenever and a “onleave” part for “at”. The semantics of “else” and “onleave” is symmetrically defined compared to the main body of the instruction.

2 URBI library

2.1 Instantiating a URBI Client

UClient is the main class of the LibURBI Java Project. If you create an instance of UClient, passing the name of a URBI server host as the first parameter and optionally the port as the second parameter, the object will directly connect to the server.

UClient client = new UClient(“192.169.1.27 ”);

2.2 Sending data

To send a command, you can either use the method ‘send ’ to append the String message to a Send buffer or the method ‘effectiveSend ’ which immediately sends data through the socket.

client.send(“motoron;”);
client.effectiveSend(“speaker.val;”);

2.3 Receiving data

Some precise messages received from the URBI server are the results of a command previously sent by a client. The mechanism of URBI tags enables to link a message to its reply : each sent command is associated with a tag (if it is not precise in the String sent message, the URBI server associated its reply with the ‘notag ’ tag) and this tag is repeated in the reply message from the server. LibURBI Java uses the performance of the ‘java.nio ’ API introduced in JDK 1.4. A selector associated with the socket channel handles the reception of those (binary or not) messages.

2.4 Implementing Callbacks

One of the most important features of the LibURBI Java Project consists in the ability of UClient instances to register Callbacks in their Callback container. A Callback object is associated with a tag and implements the UCallbackListener interface, defining the ‘actionPerformed ’ method. This function will be called each time a message marked with the corresponding tag is received from the server. The parameter of the ‘actionPerformed’ method is a URBI event. An URBIEvent object contains all the necessary information about the received message(Considering an URBIEvent named ‘event ’):

•event.timestamp: the timestamp of the received message

.•event.tag The tag which associates the received message to the corresponding Callback.

•event.type: the type of the received message. (can be “jpeg ”, “raw ”, “wav ”)

•event.binBuffer: the binary received buffer. (is null if the received message does not contain binary data)

•event.size: the size of the binary data.

•event.cmd: the received command.

•event.height: the height of the binary received data if this one is an image byte array.

•event.width: the width of the binary received data if this one is an image byte array.

•event.sampleRate: the sample rate of the binary received data if this one is an audio byte array.

•event.nbOfBits: the sample size of the binary received data if this one is an audio byte array.

•event.nbOfChannels: the number of channels of the binary received data if this one is an audio byte array.

•event.signed: the signed value of the binary received data if this one is an audio byte array.

To register a Callback and associate it with a tag, you simply need to call the 'setCallback 'method of the UClient.

```
UCallbackListener call  = new UCallbackListener();

client.setCallback(call, "call ");
```

To remove a Callback, use the 'deleteCallback'with the corresponding tag.
client.deleteCallback("call ");
We implemented the Callback interface in our classes.
Below we gave an example of such a class that makes the saving of the pictures:

```
public class SavePicture implements UCallbackListener
{       private int nopictureMade;
        private String path = "";
        private int noPicture = 100;
        public SavePicture(String path,int noPicture)
        {
            this.path = path;
            this.noPicture = noPicture;
            nopictureMade = 0;
        }
        public SavePicture(String path)
        {
            this.path = path;
        }
        public SavePicture()
        {}
        public void   actionPerformed(URBIEvent event)
        {
            if (event.getBinBuffer() == null) return ;

            int       width = event.getWidth();
            int       height = event.getHeight();
            ImageUtilities.setWidth(width);
            ImageUtilities.setHeight(height);
            Image im = ImageUtilities.blockingLoad(event.getBinBuffer());
            if (im != null){
```

```
                        if (nopictureMade>noPicture)
                            OutputBehavior.sw = false;
                        else
                            nopictureMade++;
                        int [] pixels = new int[width * height];
                        PixelGrabber pg = new PixelGrabber(im,0,0,width,height,pixels,0,width);
                        try
                        {
                            pg.grabPixels();
                        }
                        catch(InterruptedException ie)
                    {
                            ie.printStackTrace();
                        }

                        buffer.setRGB(0,0,width,height,pixels,0,width);
                        //Encode as a JPEG and creates an directory image in the given path
                        try
                        {
                            File dir = new File(path+"\\image");
                            FileOutputStream fos = null;
                            if (dir.isDirectory())
                                fos = new FileOutputStream(dir+"/out"+nopictureMade+".jpg");
                            else
                            {
                                boolean succes = dir.mkdir();
                                if (succes)
                                    fos = new FileOutputStream(dir+"/out"+nopictureMade+".jpg");
                                else
                                    System.out.println("The directory "+dir+" couldn't be created");
                            }
                            JPEGImageEncoder jpeg = JPEGCodec.createJPEGEncoder(fos);
                            jpeg.encode(buffer);
                            fos.close();

                        }
                        catch (Exception e)
                        {
                            System.err.println(e.getMessage());
                        }
                        buffer.flush();
                        buffer = null;
                        im.flush();
                        im = null;
                    }
                }
            }
```

Appendix F

Jess

Jess-Java Expert System-is a fast, powerful rule engine for the Java platform [7]. Jess supports the development of rule-based systems which can be tightly coupled to code written in the powerful, portable Java language. We use Jess to develop our rule based expert system. The most important modules that make up a rule-based system are shown in the Figure below:

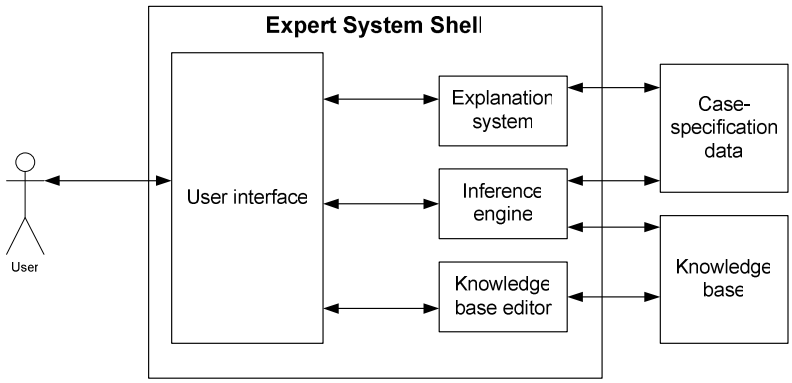

Figure 4.1

The user can interact with the system through a *user interface*. An *inference engine* interacts with both *knowledge base* and data specific to the particular problem being solved. The *expert knowledge* will typically in the form of a set of IF-THEN rules. The *case specific data* includes both data provided by user and partial conclusion based on the data. In a simple forward chaining rule-base system the *case specific data* will be the elements in the *working memory*. Almost expert systems have an *explanation subsystem* which allows the program to explain its reasoning to the user. Some expert systems have a *knowledge base editor* which is helpful to update and check the knowledge base.

Instead of representing knowledge in a relatively declarative, static way (as a bunch of things that are true), rule-based system represent knowledge in terms of a bunch of rules that tell you what you should do or what you could conclude in different situations. A rule-based system consists of a bunch of IF-THEN rules, a bunch of facts, and some interpreter controlling the application of the rules, given the facts.

There are two broad kinds of rule system: forward chaining systems, and backward chaining systems. In a forward chaining system you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts. In a backward chaining system you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new sub goals to prove as you go. Forward chaining systems are primarily data-driven, while backward chaining systems are goal-driven.

Chapter 8 Bibliography

[1] WILLIAM J. KOLE, WHO REPORTS ON VIOLENT CRIME

[2] AP, 20 MAY 1994

[3] http://www.fremtidshuset.com/eng/

[4] CHI HYUN ANGELA LEE, NOVEMBER 2004, “ABOUT AIBO”

[5] ZHENKE YANG, JANUARY 2005, PROGRAMMING AIBO

[6] Ingeborg Strand Friisk, November 28, 2003,” Autonomous AIBO watchman-TDT4735 Software Engineering, Depth Study”,http://www.idi.ntnu.no/grupper/su/fordypningsprosjekt-2003/fordypning2003-Ingeborg-Strand-Friisk.pdf

[7] http://herzberg.ca.sandia.gov/jess/docs/70/

[8] JEAN-CHRISTOPHE BAILLIE, “URBI: A UNIVERSAL LANGUAGE FOR ROBOTIC CONTROL “

[9] BASTIEN SALTEL, EDITION DECEMBRE 15, “THE LIBURBI JAVA PROJECT” 2004

[10] http://www.sony.net/Products/aibo/

[11] http://www.aibo.a0soft.com/

[12] Dobai, I., Rothkrantz L. and Charles van der Mast, year 2005, "Personality model for a companion AIBO"