p2nSpeech

A cognitive architecture approach to robot voice control and response

by

Siddtharth Patel

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Computer Science

at

Ivan G. Seidenberg
School of Computer Science and Information Systems
Pace University
May 2008

Thesis Signature (Approval) Page

I here by certify that this thesis, submitted by Siddtharth Patel, satisfies the thesis requirements for the degree of Master of Science and has been approved.

Dr. D. Paul Benjamin	Date
Thesis Advisor	
School of Computer Science and Information Systems	
Pace University	

Abstract

Future life pictures humans having intelligent humanoid robotic systems taking part in their everyday life. Thus researchers strive to supply robots with an adequate artificial intelligence in order to achieve a natural and intuitive interaction between human being and robotic system.

This project aims to explore the ways to command a robot using human voice and also enable the robot to exhibit cognitive behavior. This feature can be interesting with several industrial, laboratory and clean-room applications, where a close cooperation between robots and humans are desirable. The robot should be capable to understand things that are meant but not clearly stated in a particular voice command depending on the task at hand.

The effective use of robots to perform different tasks based on voice commands will depend upon the efficacy of interaction between humans and robots. The key to achieving this interaction is to provide the robot with sufficient skills for natural communication with humans so that humans can interact with the robot almost as though it were another human. Building a robot that can understand each and every aspects of human speech is a far too complex task and is yet to be archived.

This paper describes the design, implementation, and capabilities of p2nspeech system for pioneer2 series of robots which can be used (at very basic level) to enable the robot to understand few predefined keywords in voice commands. The capabilities required of the robot include voice recognition, command understanding, speech synthesis, processing the sensory information and above all exhibit intelligent behavior. These represent a small subset of potential capabilities humans utilize to perform some similar task in a complex environment.

Table of Contents

List of Figures

Introduction

This thesis is part of the ADAPT project. The ADAPT project (**A**daptive **D**ynamics and **A**ctive **P**erception for **T**hought) is a collaboration of three university research groups that is building a robot cognitive architecture that integrates the structures designed by cognitive scientists and linguists with those developed by robotics researchers for real-time perception and control.

The design of our robot architecture is based on the belief that language is central to human intelligence and thus should be used as a central organizing principle of an artificial intelligence. This means that language is not only used for communication, but also to represent and organize the robot's knowledge about itself and the world, and to structure the robot's reasoning and planning processes. Knowledge is organized according to units arising from the semantics of natural language: words, phrases, sentences, and discourse contexts.

The central goal of our work is to develop effective methods for robots to comprehend their environment. In our language-based architecture, this means developing effective methods for comprehending language. Our approach models language comprehension as a process of trying to recreate the observed speech by hypothesizing various sets of goals and beliefs for the communicating agents, generating their speech based on these assumptions and comparing it with the observed speech.

We believe that the comprehension requires *visualization*, and that the semantics of language requires visual representations. We view visualization as consisting of both a perceptual component and a reasoning component. The perceptual component is performed using the same perceptual mechanism that the robot uses to perceive its environment; the difference is that visualization perceives a simulation of the environment. Visual reasoning manipulates and superimposes representations that consist of a combination of symbolic knowledge and 3D animations.

ADAPT's virtual world is a multimedia simulation platform capable of realistic simulations of physical phenomena. It combines the various forms of map information found in most robots: topological, metric and conceptual information. ADAPT completely controls this virtual world, and can create arbitrary objects and behaviors in it, including nonexistent objects and behaviors that were not actually observed. Central to ADAPT's use of its virtual world is its ability to view these constructions from any point. This enables ADAPT to create visual representations with desired properties.

This thesis focuses on the implementation and testing of this virtual world. The virtual world we use is the Gazebo virtual robotics platform. The ADAPT architecture uses the Soar cognitive architecture to perceive and plan in this virtual world. In the following sections of this thesis, we describe these components and how they have been integrated.

p2nspeech is designed to work on Debian based Linux operating systems, it uses open source libraries for speech recognition, speech synthesis and artificial intelligence. The speech recognition library is called Sphinx, and we use Sphinx version 4 which is entirely written in Java. The speech synthesis library is called festival [version 1.95], it is developed in c programming language and it can run in server mode and listen for commands over the TCP/IP network. Finally we use SOAR for artificial intelligence part. Soar is a general cognitive architecture for developing systems that exhibit intelligent behavior.

Pioneer Robot

Pioneer robot is an intelligent mobile robot with solid rubber tires, a two-wheel differential, reversible drive system and a rear caster for balance. It also contains basic components for sensing and navigation in a real-world environment such as multiple sonar sensors, stereo camera, position-speed encoders. These equipments are all managed by an on board micro-controller and mobile-robot server software. Beside all basic components, Pioneer robot also comes with built-in on board computer with both wired and wireless Ethernet cards [1].

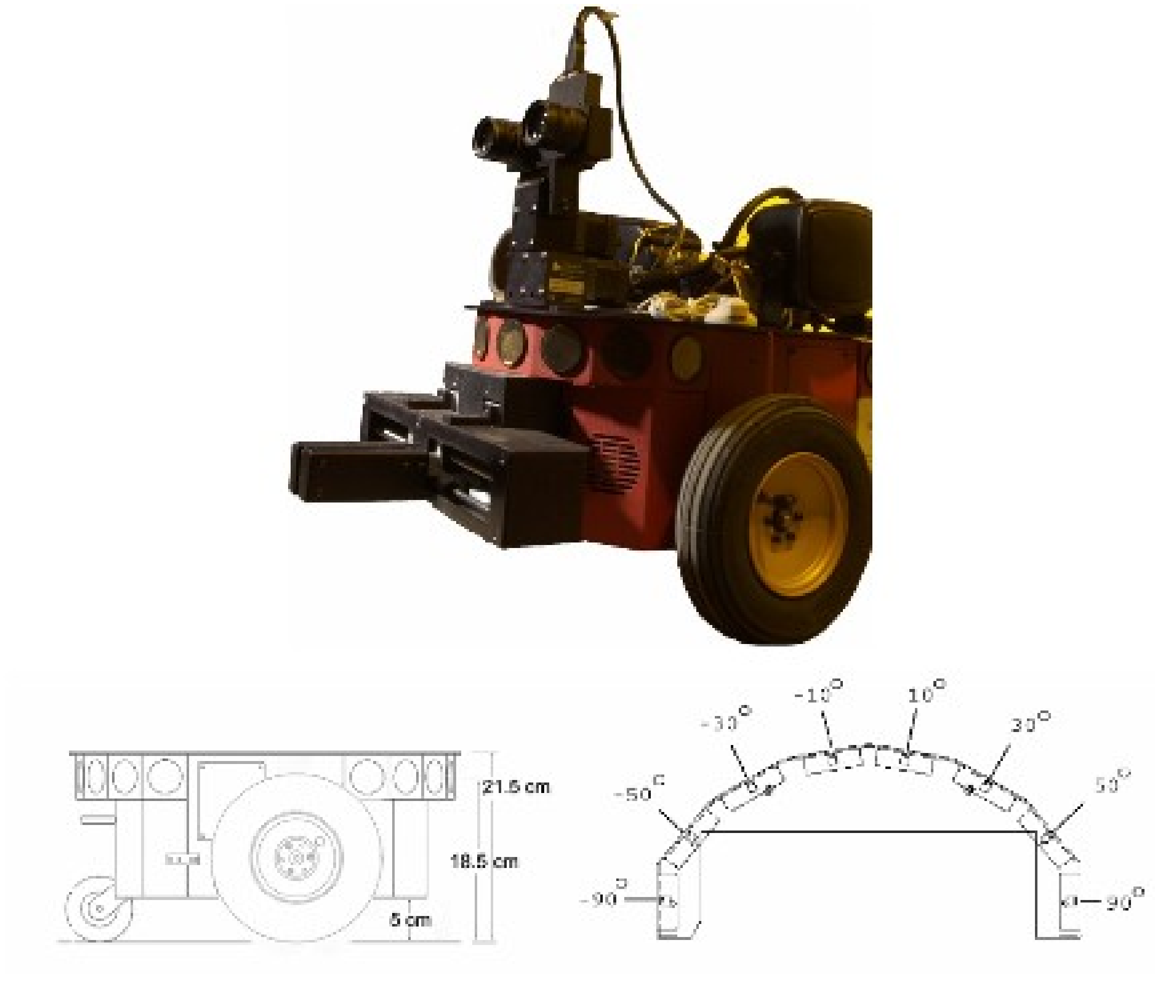


Figure 1: (Top) Pioneer robot. (Left) Dimension. (Right) Positions and angles of sonars

Sonar Sensors

Out Pioneer robot is equipped with two set of multiplexed sonar arrays, front and rear. Each array contains eight transducers, positioned as shown in the figure , that provide object detection and range information. The sonar acquisition rate is 25 Hz (40 milliseconds per sonar per array). Sensitivity ranges from six inches to nearly 16 feet [1].

Driving System

Driving system is based on high-speed, high-torque, reversible-DC motors, each equipped with a high-resolution optical quadrature shaft encoder for precise position and speed sensing and advanced dead-reckoning [1].

Batteries

Pioneer robot may contain up to three, hot-swappable, seven ampere-hour, 12 volts direct-current (VDC) sealed lead/acid batteries (total of 252 watt-hours), accessible though a hinged and latched back door [1].

Communicate with the robot

With a server program on the robot on board computer, communication with the robot can be done by sending/receiving binary streams through the Ethernet using basic socket programming over the wireless network. The client program sends a request to the robot for its status information at a fixed interval and then processes it and sends back a new command to the robot based on the robot sensory information received.

Soar

Introduction

Soar is a unified cognitive architecture, created by *John Laird*, *Allen Newell*, and *Paul Rosenbloom* at Carnegie Mellon University. It is both a view of what cognition is and an implementation of that view through a computer programming architecture for Artificial Intelligence (AI). Since its beginnings in 1983 it has been widely used by AI researchers to model different aspects of human behavior.

Soar provides the fixed computational structures in which knowledge can be encoded and used to produce action in pursuit of goals. It differs from other programming languages in that it has an embedded specific theory of the appropriate primitives underlying reasoning, learning, planning, and other capabilities which assumed to be necessary for intelligent behavior.

SOAR architecture can

- be used to build systems that work on the full range of tasks expected of an intelligent agent, from highly routine to extremely difficult, open-ended problems;
- represent and use appropriate forms of knowledge, such as procedural, declarative, episodic, and possibly iconic;
- employ the full range of problem solving methods;
- interact with the outside world; and
- learn about all aspects of the tasks and its performance on those tasks[2].

Unified Theory of Cognition

Soar started as an attempt to define a unified theory of cognition, to reconcile and unify the micro-theories of individual disciplines within the field of cognitive science. Newell described the human mind as a solution to a set of functional constraints (e.g., exhibit adaptive (goal-oriented) behavior, use language, operate with a body of many degrees of freedom) and a set of constraints on construction (a neural system, grown by embryological processes, arising through evolution)[3].

The SOAR cognitive theory says that cognitive behavior

- is goal oriented
- reflects a rich, complex, detailed environment
- requires large amount to knowledge
- requires the use of symbols and abstractions
- is flexible and a function of the environment
- requires learning form the environment and experience

Soar Architecture

Soar is a rule-based system based on the hypothesis that all deliberate goal-oriented behavior can be cast as the selection and application of operators to a state. A state is a representation of the current problem-solving situation; an operator transforms a state (makes changes to the representation); and a goal is a desired outcome of the problem-solving activity.

As Soar runs, it is continuously trying to apply the current operator and select the next operator (a state can have only one operator at a time), until the goal has been achieved. Soar has separate memories for descriptions of its current situation and its long-term knowledge. In Soar, the current situation, including data from sensors, results of intermediate inferences, active goals, and active operators is held in working memory. Working memory is organized as objects. Objects are described in terms of their attributes; the values of the attributes may correspond to sub-objects. Long-term knowledge specifies how to respond to different situations in working memory. Soar cannot solve any tasks without the long-term knowledge about it.

At this point one might think from perspective of cognitive architecture and intelligent system that Soar is quite similar to a typical rule-based system. Soar follows the basic three parts architecture of a rule based system: rule set, working memory, rule interpreter. In intelligent agent view, rules define an agent. In cognitive architecture view rules define long-term memory and working memory define short-term memory. Whatever be the point of view, the rules in a Soar rule set strongly resemble the rules in many rule-based system.

Soar Productions or Rules [long-term knowledge]

Rules encode the long-term knowledge in Soar and adding them is the way Soar is programmed, you do not change the way Soar creates states or learns, but you create rules. Working memory is where short-term information is stored and your rules will create and test structures in working memory. Rules are matched against the working memory to determine which rules will fire. When rules fire, they can make changes to working memory, as well as may perform simple actions such as printing messages in the debug window.

Rules encode appropriate knowledge in form of 'if-then' type of statements. Each production has three components: a name, a set of conditions (also called the *left-hand-side*), and a set of actions (also called the *right-hand-side*) For example, a very simple rule-based system, written in plain English, to determine whether or not to bring out an umbrella. In working memory, information about today's weather forecast can be put in by external world interpreter. The rules in this system may look like

R1: IF *the forecast says it will be raining today*
THEN *bring out an umbrella*

R2: IF *the forecast says it won't be raining today*
THEN *do not bring out an umbrella*

Then, the system examines all the rule conditions (*left-hand-side*) and determines a subset, the conflict set, of the rules whose conditions are satisfied based on information in the working memory. Of this conflict set, one of those rules is triggered (fired). Which one is chosen is based on a conflict resolution strategy. When the rule is fired, any actions specified in its clause (*right-hand-side*) are carried out. The *right-hand-side* may create some new working memory elements which in turn cause some other rules to fire.

Basic Syntax

Syntactically, each SOAR production starts with the symbol 'sp' *(Soar Production)* followed by the body of the production enclosed inside curly braces *'{'* and *'}'*. The body part starts with a name with no whitespace for the production that uniquely identifies the production; next is the *left-hand-side* or the conditions which are compared against the working memory to determine if the *right-hand-side* or the action part will be applied or not. On the *left-hand-side* all the conditions referring to the same object are grouped inside round brackets *'('* and *')',* similarly; for the *right-hand-side* all the actions referring to the same object are grouped inside round brackets *'('* and *')'*. The *left-hand-side* and the *right-hand-side* is separated by the symbol '-->' [4].

Following is a template for a Soar rules:

```
sp {rule*name
   (condition)
   (condition)
   …
-->
   (action)
   (action)
   …
}
```

Following are the rules written in Soar for the rain-umbrella example mentioned above.

```
sp {R1
   (State <s> ^forecast-info RAIN)
-->
   (<s> ^output BRING-AN-UMBRELLA)
}
sp {R2
    (State <s> -^forecast-info RAIN)
-->
   (<s> ^output DO-NOT-BRING-AN-UMBRELLA)
}
```

How Soar works?

Basic operations in Soar, without going into the details, is quite simple as explained above:

- ➔ Examine the left-hand-side of the rules.
- ➔ Match the conditions to the information in working memory.
- ➔ Fire the action on the right-hand-side of the matched rules.

The execution of a Soar Program proceeds through number of cycles. Typically each cycle has five phases:

- ➔ Input: New data comes into Soar working memory. In case of robot, new sensory data and in case of speech recognition, a new command that was just said.

- ➔ Operator Proposal: Productions fire and retract to interpret new data, propose operators for the current situation (operator proposal), and compare proposed operators (operator comparison).
- ➔ Decision: A new operator is selected, or an impasse is detected and a new state is created(sub-state).
- ➔ Operator Application: Productions fire to apply the operator. Because of changes from operator application, other productions may also match or retract. Productions fire and retract in parallel until there are no more additional complete matches or retractions of productions.
- ➔ Output: Output commands are sent to the external environment.

Operators

Operators are the decisions that Soar makes based on its rule firings. They answer the question, "What should I do next?" Operators may trigger an action internally or send a message to external software [5]. For example, in *p2nspeech*, operators tell the robot to perform an action such as move/turn/stop based on the sonar data from the robot or based on the voice command just said; also, operators tell how to respond to a voice command that dose not involve any robot actions. For example, when robot is asked "can you pick up the red box?", operator decide the response based on the information about the robot-world.

Operator Proposal: Proposal Phase

In this phase, Soar matches and fires all rule that match the working memory at that instance of time. This might cause some more rules to match and fire. Soar will continue to match and fire rules until no more rule can be fired.

Rules are matched and fired (conceptually) simultaneously in Soar, even though only one rule can be fired at a time in actual execution. To maintain this concept, Soar keeps the working memory consistent. When a fired rule changes something working memory which cause the left-hand-side conditions of the earlier fired rule to be false, Soar retracts the action(s) of the false rule. So at the end of the phase, it seems like all rules have been fired at the same time.

A common mistake made by Soar newbie is write rules whose *right-hand-side* invalidates its own *left-hand-side* there by causing an infinite loop in Soar [*infinite-loop: it fires as the left-hand-side matches the working memory, then retracts as the right-hand-side changes the working memory so that the left-hand-side no longer matches, then fires again because the left-hand-side matches again after the change made by the right-hand-side are retract and so on...*].

Following is the example of operator proposal rule, its usually done by attaching an operator

(^operator) to the root working memory and plus (+) indicates that the operator is acceptable.

```
sp {Propose*grab-umbrella
     (State <s> ^forecast-info RAIN)
-->
   (<s> ^operator <o> + =)
   (<o> ^name grab-umbrella)
}
```

Decision Phase

Because Soar matches and fires all possible rules that matched the working memory in the proposal phase, it is possible that more than one proposal get fired . But only one of the operator proposals will be selected and applied in each cycle based on the operator preferences. Preference can be binary, e.g. operator A is preferable over operator B, or unary, e.g. unary operator C is the best. Preference may also be expressed through separate rules in the rule set. In case of the above example equal sign '=' signifies indifferent preference, meaning that the operator can be randomly selected among all the operators with indifferent preference. Even though we will create an indifferent preference for these operators, the acceptable preference is still necessary and an operator will not be selected if it does not have an acceptable preference.

Application Phase

In this phase Soar applies the operator that got selected during the decision phase based on the preference. Soar dose this by applying ^operator attribute to the problem state now, due to this some more rule will fire as the now match the working memory. Below is an example of a rule that will fire if and only if the decision phase selects the operator named stop-now

```
sp {Apply*stop-now
   (State <s> ^operator <o>)
   (<o> ^name stop-now)
-->
   (write |Applying rules to stop the robot...|)
   ...
}
```

Application rule is processed similar to the other rules in the way that it will be matched and fired if the left-hand-side conditions are true and will be retracted if the conditions are false. But the difference is that the changes made by these rule to the working memory will persist even if the rule has been retracted.

Similar to proposal phase, Soar will match and fire all possible rules until no more rule can be fired. Then it begins the proposal phase of the next cycle.

Impasses and Learning

As mentioned in decision phase, preferences can be given at the creation time or later by other rules. It is possible that the given preferences are incomplete or inconsistent. (The preferences can be incomplete in that no acceptable operators are suggested, or that there are insufficient preferences to distinguish among acceptable operators. The preferences can be inconsistent if, for example, operator A is preferred to operator B in one rule, but operator B is preferred to operator A in another rule.)

Soar calls this situation as an impasse. There are several impasse types, but all of them mean that Soar does not know what to do next. The consequence action of trying to resolve the impasse problem, Soar *subgoals*.

By subgoaling, Soar create a substate, a new state and problem space with some information about the impasse, and begins its execution cycles on the substate. The main idea is that all rules, phases in this substate will lead to resolving the impasse in the superstate. Soar may create a hierarchy of superstates and substates in working memory if it hits impasses during execution in any substate.

In every state, there is a "^superstate" attribute which contain the name of the superstate. This allows programmer to keep track of what state they are working on. In the top most state value of "^superstate" attribute is "nil",

One advantage of this subgoaling using state/substate is that it provides a way of partitioning knowledge and limiting the operators to be considered in searching for a goal's desired state. We can compare this advantage to Object Oriented Programming. One state is one class, which has one main goal. During the execution of the class, we sometimes need to solve problems that belong to another class. So we subgoal to another class and return back with useful information. Then, we can continue working on the first class.

Soar's subgoaling mechanism also makes it possible for Soar to learn. When an operator impasse is resolved, it means that Soar has, through problem solving, gained access to knowledge that was not readily available before. Therefore, when an impasse is resolved, Soar has an opportunity to learn, by summarizing and generalizing the processing in the substate.

Soar's leaning mechanism is called chunking; it attempts to create a new rule called a chunk. The conditions of the chunk are the elements of the state that (through some chain of rule firings) allowed the impasse to be resolved; the action of the rule is the working memory element or preference that resolved the impasse (the result of the impasse). The conditions and actions are variable so that this new rule may match in a similar situation in the future and prevent an impasse from arising again [5].

In other words, Soar can learn by memorizing and summarizing left-hand-side the conditions and right-hand-side actions which occurred during impasses and subgoaling. As a result, it create a new rule which ready to be matched and fired. In the future, if the same situations occurred, instead of subgoaling, Soar can matches and fires this rule.

I-Support/O-support working memory element

Soar's working memory can be separated into two part, i-support and o-support working memory. It resembles the humans' short-term and long-term memory respectively.

I-support working memory is the memory space that Soar allocates for temporary working memory elements which will be wiped out when the creator rules are retracted. This came from the idea that humans do not memorize every pieces of information. We just retrieve them, process them, make use from the results from them, and finally forget them. For example, if someone tells us a phone number, we then process them by phone the number, and we finally forget the number.

O-support working memory is another memory space allocated for long term working memory elements. This kind of the working memory elements will not be wiped out with the retraction of the creator rules. Take a look at the same example above, if we often call that phone number, after a few times we would memorize it into the long-term memory part of our brain. Next time, we can recall the number out right away without need to read it from somewhere else.

Player-Gazebo

The Player Gazebo is a open source project that enables research in robot and sensory system. It is developed by an international team of robotics researchers and used at labs around the world. It follows the client-server architecture where Player works as the server for client program written in any programming language and Gazebo is a 3D robot simulator.

Player

Player runs on the robot and acts as a robot server providing a clean and simple interface to the robot's sensors and actuators over the network there by allowing the client program to be running on any machine that has network connection to the robot and can be written in any programming language that supports TCP sockets. Player was originally developed to support ActivMedia Pioneer2 family of robots, but now supports a variety of robot hardware and many common sensors.

Player can be used as server for a real robot or for a virtual robot simulator and allows the same client program to work with the real robot and with the virtual robot without need for any change.

Player uses a configuration file, usually with the extension **.cfg**, that instantiates the drivers needed to control the robot and tell the drivers how to access relevant hardware. The job of the configuration file is to map the physical devices to Player devices. Consider the following sample configuration file:

```
driver
(
  name "p2os"
  provides ["odometry:::position2d:0"]
  port "/dev/ttyS0"
)
driver
(
  name "sicklms200"
  provides ["laser:0"]
  port "/dev/ttyS1"
)
```

This file instantiates the p2os driver to access the robot's motors. This driver will talk to the robot over the serial port "/dev/ttyS0" and map its motors and encoders to the Player device "position2d:0". This file also instantiates the sicklms200 driver to access the SICK laser over the serial port "/dev/ttyS1" and present it as the Player device "laser:0".

Gazebo

Gazebo is a 3D multiple robot simulator with dynamics. It is capable of simulating a small population of robots with high fidelity in a three-dimensional world with realistic sensor feedback, object collisions and dynamics. Gazebo is normally used in conjunction with the Player device server; Player provides an abstracted, network-centric mechanism (a server) through which robot controllers (clients program) can interact with robot hardware. When used with Gazebo, Player provides *simulated* data in the place of real sensor data. In principle, client programs cannot tell the difference between real devices and the Gazebo simulation of those devices.

Gazebo can also be controlled through a low-level C API (libgazebo). This library allows developers to easily integrate Gazebo into their own (non-Player) servers or architectures.

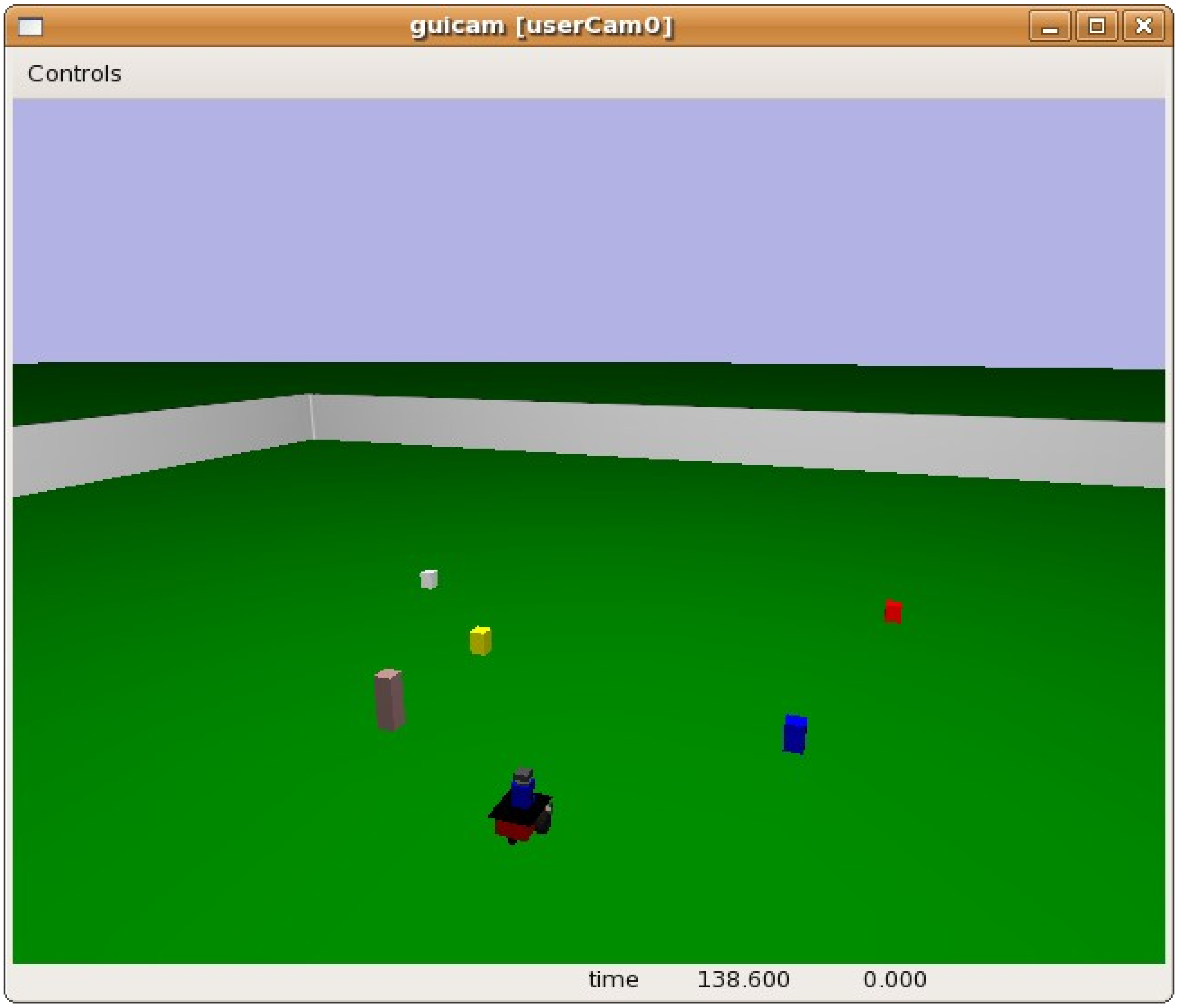


figure 2. Gazebo screen shot with Pioneer2 robot and few colored boxes[observercam 1]

While Gazebo can operate without a GUI, it is often useful to have a viewport through which the user can inspect the world. In Gazebo, such user interface components are provided through special models called the o*bservercam* that can be declared and configured in the world file. The observer cam can be attached any component of the world. In the above figure it is attached to fixed point in the world.

In the following screen shot the observercam gives the bird-eye view of the world. In this case the observercam is attached to the robot and move in the same way as the robot moves.

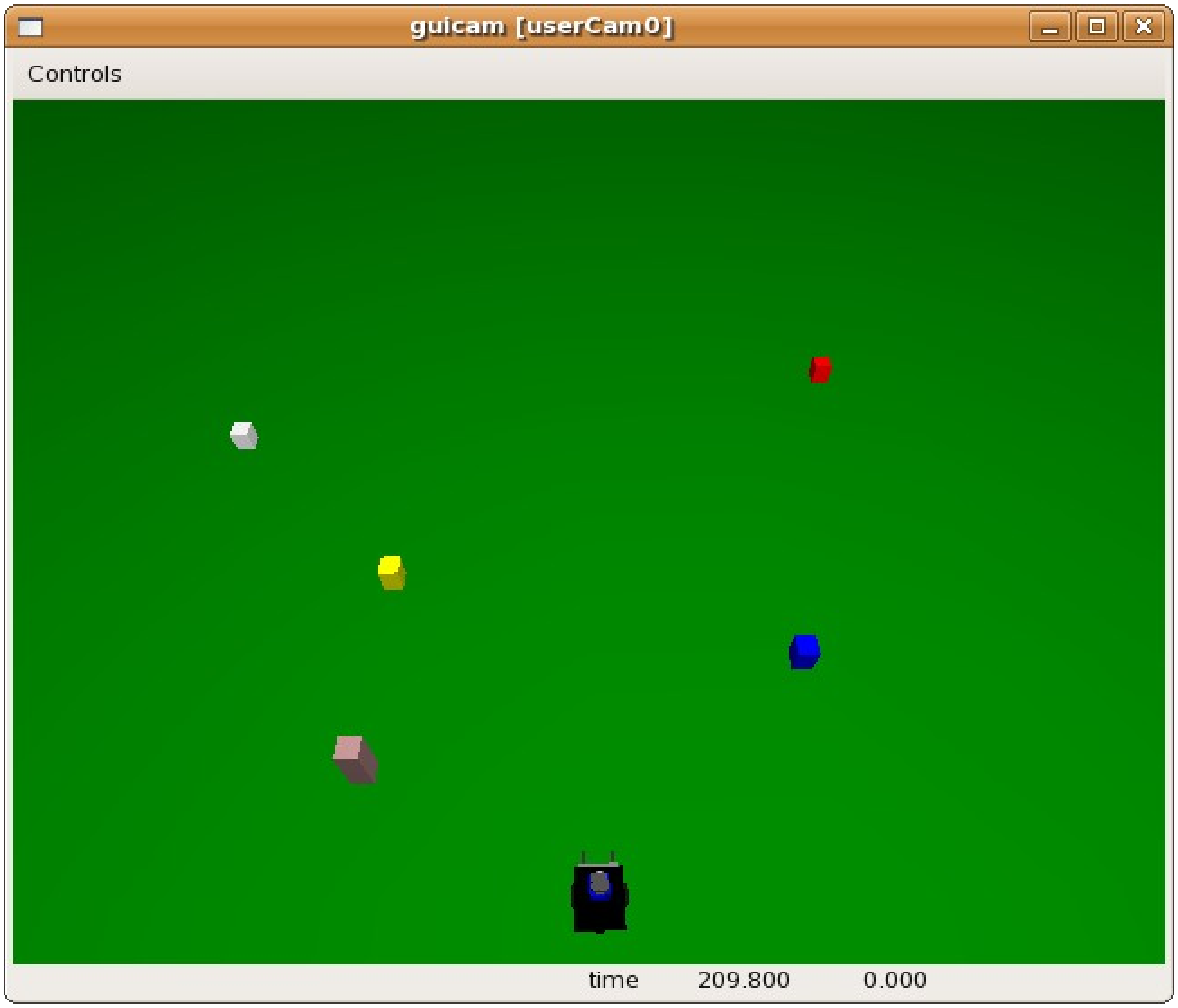
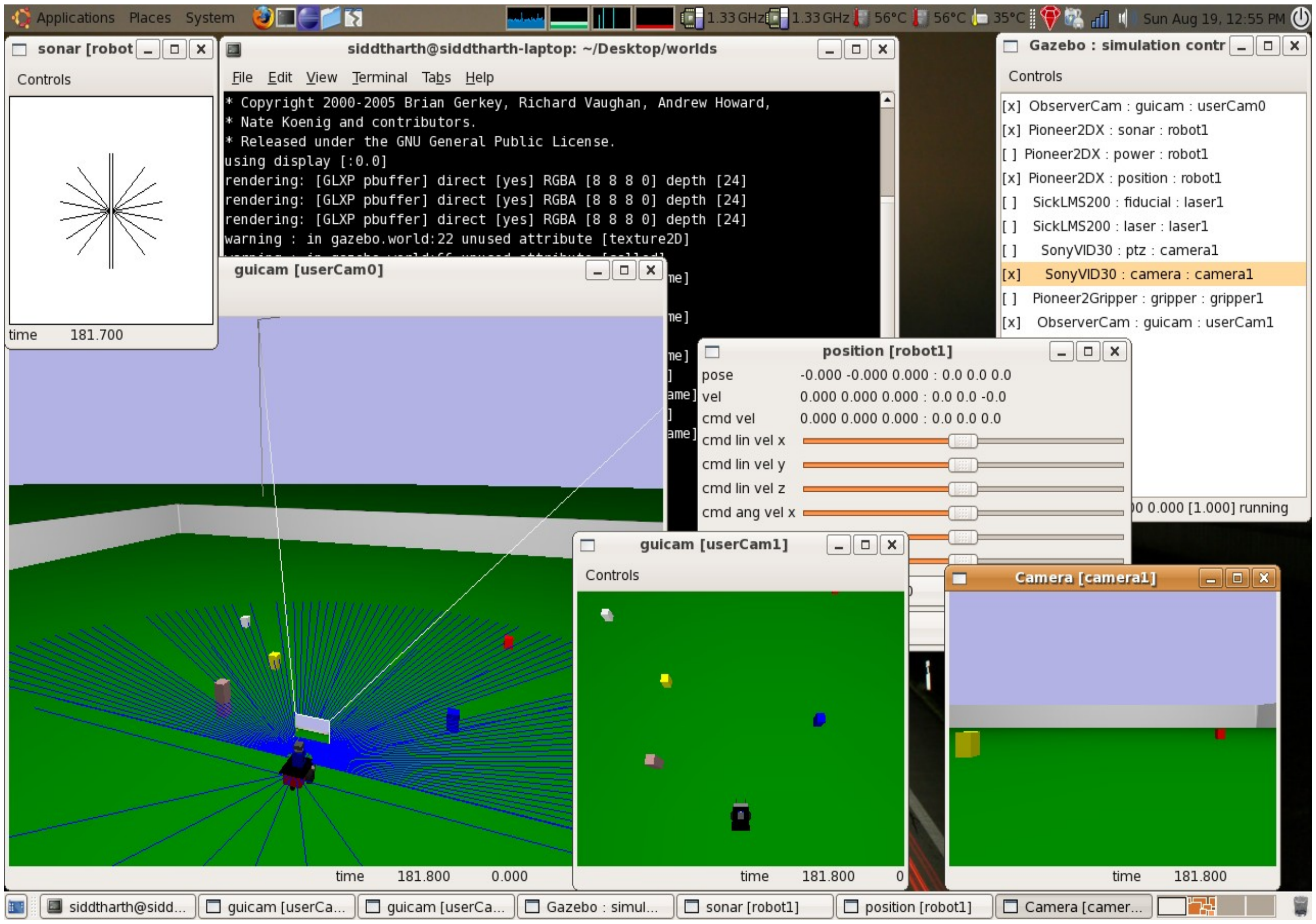


figure 3.: bird-eye view of the world from observercam attached to robot



```
* Copyright 2000-2005 Brian Gerkey, Richard Vaughan, Andrew Howard,
* Nate Koenig and contributors.
* Released under the GNU General Public License.
using display [:0.0]
rendering: [GLXP pbuffer] direct [yes] RGBA [8 8 8 0] depth [24]
rendering: [GLXP pbuffer] direct [yes] RGBA [8 8 8 0] depth [24]
rendering: [GLXP pbuffer] direct [yes] RGBA [8 8 8 0] depth [24]
warning : in gazebo.world:22 unused attribute [texture2D]
```

```
[x] ObserverCam : guicam : userCam0
[x] Pioneer2DX : sonar : robot1
[ ] Pioneer2DX : power : robot1
[x] Pioneer2DX : position : robot1
[ ]  SickLMS200 : fiducial : laser1
[ ]  SickLMS200 : laser : laser1
[ ]   SonyVID30 : ptz : camera1
[x]   SonyVID30 : camera : camera1
[ ]  Pioneer2Gripper : gripper : gripper1
[x]  ObserverCam : guicam : userCam1
0 0.000 [1.000] running
```

```
pose      -0.000 -0.000 0.000 : 0.0 0.0 0.0
vel       0.000 0.000 0.000 : 0.0 0.0 -0.0
cmd vel   0.000 0.000 0.000 : 0.0 0.0 0.0
cmd lin vel x
cmd lin vel y
cmd lin vel z
cmd ang vel x
```

figure 4: Gazebo screen shot showing a number of control panels and observercam views

We can have multiple observercam enabled at the same time as it can be see in figure 3. guicam[userCam 0] give the view of the world from the stationary observercam, guicam[userCam 1] gives the view from a observercam that is attached to the robot and Camera[camera1] is the view from the cameras on the robot. Besides this you can see various control panels that can be use to control or monitor robot motion and sensors.

Gazebo also uses a XML configuration file called the world file that contains the description of the world to be simulated. It describes the layout of robots, sensors, light sources, user interface components, and so on. The world file can also be used to control some aspects of the simulation engine, such as the force of gravity or simulation time step.

The world consists mainly of *model* declarations. A model can be a robot, a sensor, a static feature of the world or some manipulable object. For example, the following declaration will create a Pioneer2 robot model named *robot1.*

```
<model:Pioneer2DX>
      <id>robot1</id>
      <xyz>0 0 0.40</xyz>
      <rpy>0 0 45</rpy>

</model:Pioneer2DX>
```

Model I: Pioneer2 robot model named robot1

Associated with each model is a set of *attributes* such as the model's position <*xyz*> and orientation <*rpy*>. Models can also be composed. One can, for example, attach a scanning laser range-finder to a robot:

```
<model:Pioneer2AT>
      <id>robot1</id>
      <xyz>0 0 0.40</xyz>
      <rpy>0 0 45</rpy>
      <model:SickLMS200>
            <id>laser1</id>
            <xyz>0.15 0 0.20</xyz>
            <rpy>0 0 0</rpy>
      </model:SickLMS200>
</model:Pioneer2AT>
```

Model II: Pioneer 2 robot model with SickLMS laser

Here the <*xyz*> and <*rpy*> tags describe the laser's position and orientation with respect to the robot body. Once attached, robot and the laser form a single rigid body.

JavaClient API

Javaclient allows development of client applications for Player-Gazebo using the power of the Java programming language. The Java client implements all interfaces needed to communicate with player. Following is the link to the UML diagram for the client API [http://java-player.sourceforge.net/docs/javaclient_architecture_classUML.jpg]. A detail explanation about JavaClient is given in p2nspeech topic later in the document.

Sphinx Speech Recognition System

Sphinx is a state of the art Hidden Markov Model(*HMM*) based speech recognition system. It was designed at Carnegie Mellon University and is one of the worlds best and most versatile speech recognition system. An HMM-based speech recognition system, functions by first learning the characteristics (or parameters) of a set of sound units, and then using what it has learned about the units to find the most probable sequence of sound units for a given speech signal. The process of learning about the sound units is called *training*. The process of using the knowledge acquired to deduce the most probable sequence of units in a given signal is *called decoding,* or simply *recognition*.

The Sphinx system has several different decoders having different features. Roughly these can be described as follows.

Sphinx-2: Uses semi-continuous HMM's. It is the fastest decoder among all others, but since it employs an older technology, its accuracy is usually less than the other decoders. However, since it has been around for longer, it has features the others still lack, such as dynamic language model change. Models and incoming features are very rigidly defined.

PocketSphinx: This is a modernized version of Sphinx-2, specially optimized for embedded and hand held systems. It also consumes on average 20% less memory and 5-20% less CPU time than *Sphinx-2*.

Sphinx-3: Uses continuous HMM's. It can handle both live and batch decoding. Currently, it is the decoder most actively developed.

Sphinx-4: Uses continuous HMMs. It was written in the Java programming language. It provides high flexibility and great accuracy and speed for small tasks.

Sphinx-4

Introduction

Sphinx-4 is a state-of-the-art speech recognition system written entirely in the Java programming language. It was created via a joint collaboration between the Sphinx group at Carnegie Mellon University, Sun Microsystems Laboratories, Mitsubishi Electric Research Labs (MERL), and Hewlett Packard (HP), with contributions from the University of California at Santa Cruz (UCSC) and the Massachusetts Institute of Technology (MIT). Sphinx-4 started out as a port of Sphinx-3 to the Java programming language, but evolved into a recognizer designed to be much more flexible than Sphinx-3, thus becoming an excellent platform for speech research.

Features

Sphinx-4 is a very flexible system capable of performing many different types of recognition tasks.

- Live mode and batch mode speech recognizers, capable of recognizing discrete and continuous speech.
- Generalized pluggable front end architecture. The front end Provides a set of high level classes and interfaces that are used to perform digital signal processing for speech recognition.
- Generalized pluggable language model architecture. Includes pluggable language model for Java Speech API Grammar Format(JSGF).
- Generalized acoustic model architecture. Includes pluggable support for Sphinx-3 acoustic model.
- Generalized search management. Includes pluggable support for breath first and word pruning searches.
- Utilities for post-processing recognition results, including obtaining confidence scores, generating lattices and embedding ECMA Script into JSGF tags.
- Standalone tools, Includes tools for displaying waveforms and spectrograms and generating features from audio.

Architecture

Following it the architectural diagram of Sphinx-4 showing all its major components and how they all are connected to each other.

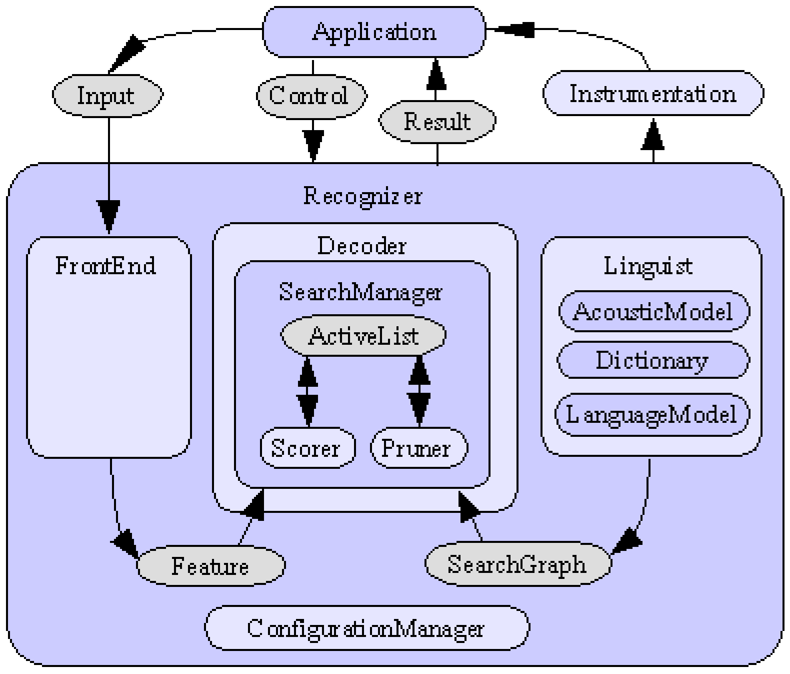


figure 5.: Sphinx-4 Architectural diagram

When the recognizer starts up, it constructs the front end which generates features from speech, the decoder, and the linguist (which generates the search graph) according to the configuration specified by the user. These components will in turn construct their own subcomponents. For example, the linguist will construct the acoustic model, the dictionary, and the language model. The decoder will construct the search manager, which in turn constructs the scorer, the pruner, and the active list.

Most of these components represents interfaces. The search manager, linguist, acoustic model, dictionary, language model, active list, scorer, pruner, and search graph are all Java interfaces. There can be different implementations of these interfaces. For example, there are two different implementations of the search manager. The Implementations to use is determined by the user through the configurations file, an XML base file that is loaded by the configuration manager. In this configuration file, the user can also specify the properties of the implementations. One example of a property is the sample rate of the incoming speech data.

Working:HMM-based Speech Recognition System

Sphinx-4 is an HMM-based speech recognizer, which is a type of statistical model. In HMM-based speech recognizers, each unit of sound (called a phoneme) is represented by a statistical model that represents the distribution of all the evidence (data) for that phoneme. This is called the acoustic model for that phoneme. When creating an acoustic model, the speech signals are first transformed into a sequence of vectors that represent certain characteristics of the signal, and the parameters of the acoustic model are then estimated using these vectors (usually called features). This process is called training the acoustic models.

During speech recognition, features are derived from the incoming speech signal in the same way as in the training process. The component of the recognizer that generates these features is called the frontend. These live features are scored against the acoustic model. The score obtained indicates how likely that a particular set of features (extracted from live audio) belongs to the phoneme of the corresponding acoustic model.

The process of speech recognition is to find the best possible sequence of words (or units) that will fit the given input speech. It is a search problem, and in the case of HMM-based recognizers, a graph search problem. The graph represents all possible sequences of phonemes in the entire language of the task under consideration. The graph is typically composed of the HMMs of sound units concatenated in a guided manner, as specified by the grammar of the task. As an example, a simple search graph that decodes the words "one" and "two". It is composed of the HMMs of the sounds units of the words "one" and "two":

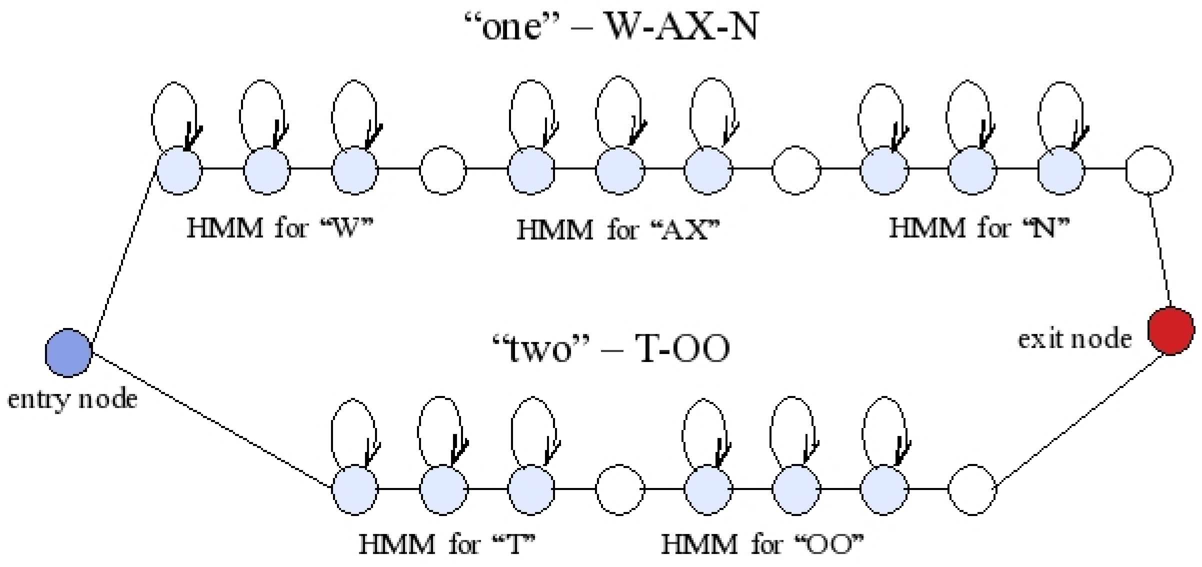


figure 6.: Search graph to decode words 'one' and 'two'

In Sphinx-4 the task of constructing a search graph is done by linguist. It requires a dictionary that maps the words to its phonemes("one" to the phonemes 'W', 'AX' and 'N' and "two" to the phonemes 'T' and 'OO'). The search graph also has information about how likely certain words will occur. This information is supplied by the language model. So in case of the above example if the probability of someone saying the word "two" is higher then someone saying the word "one", then the probability of translation from entry node to the first node of HMM of "T" will be higher then that for translation from entry node to the first node of HMM of "W".

Now during Speech Recognition the sequence of parameterized speech signal (i.e., the features) is matched against different path through the graph the best fit. In Sphinx-4, the task of searching through the graph for the best path is done by the Search Manager. As one can see from the above graph, a lot of the nodes have self transitions. This can lead to a very large number of possible paths through the graph. As a result, finding the best possible path can take a very long time. The purpose of the pruner is to reduce the number of possible paths during the search, using heuristics like pruning away the lowest scoring paths there by reducing the search time.

Sphinx-4 Configuration

The Sphinx-4 system is like most speech recognition systems in that it has a large number of parameters that control how the system functions. The performance of Sphinx-4 is highly dependent on how it is configured via the XML based configuration file. The configuration manager is responsible for reading through the configuration file and loading the appropriate components to be used by the system. It also determines the detailed configuration for each component[6]. The configuration file defines the following:

- ➔ The *names* and *types* of all of the components of the system
- ➔ The connectivity of these components - that is, which components talk to each other
- ➔ The detailed configuration for each of these components.

Consider the following sample configuration file that congifures the various aspects of the Sphinx-4 front end. The main design goal of the Sphinx-4 front end is flexibility. The front end is modeled as a pipeline of data processors, and the entire composition of the pipeline is configurable from the configuration file. Other design requirements include allowing multiple instances of the same data processor in the same front end, where each can be configured differently.

FrontEnd is a wrapper class for the chain of front end processors. It provides methods for manipulating and navigating the processors. The front end is modeled as a series of data processors, each of which performs a specific signal processing function. For example, a processor performs Fast-Fourier Transform (FFT) on input data, another processor performs high-pass filtering[7].

```
<component name="mfcFrontEnd" type="edu.cmu.sphinx.frontend.FrontEnd">
    <propertylist name="pipeline">
        <item>preemphasizer</item>
        <item>windower</item>
        <item>dft</item>
        <item>melFilterBank</item>
        <item>dct</item>
        <item>batchCMN</item>
        <item>featureExtractor</item>
    </propertylist>
</component>

<component name="preemphasizer"
      type="edu.cmu.sphinx.frontend.filter.Preemphasizer"/>

<component name="windower"
      type="edu.cmu.sphinx.frontend.window.RaisedCosineWindower"/>

<component name="dft"
      type="edu.cmu.sphinx.frontend.transform.DiscreteFourierTransform">
            <property name="numberFftPoints" value="512"/>
</component>

<component name="melFilterBank"
      type="edu.cmu.sphinx.frontend.frequencywarp.MelFrequencyFilterBank"/>

<component name="dct"
      type="edu.cmu.sphinx.frontend.transform.DiscreteCosineTransform"/>

<component name="batchCMN" type="edu.cmu.sphinx.frontend.feature.BatchCMN"/>

<component name="featureExtractor"
      type="edu.cmu.sphinx.frontend.feature.DeltasFeatureExtractor"/>
```

figure 7: Sample configuration file specifying a standard MFCC frontend

The current version of Sphinx-4 front end generates features that contains Mel Frequency Cepstral Coefficients (MFCC). In the above configuration file the first components defines the front end component of Java class type *"FrontEnd"* and *"mfcFrontEnd"* is the name used for the front end pipeline. It is then followed by a list of the data processors, using a *propertylist* called *pipeline.* The rest of the components specify each of the data processors in turn.

You can refer the API documentation for Sphinx-4 for further information about all components and their properties. You need to select and configure each component based on the task to get the best out of Sphinx-4.

Java Speech API Grammar Format (JSGF)

In *p2nSpeech,* Sphinx-4 uses the Java Speech API Grammar Format (JSGF) to perform speech recognition. The Java Speech Grammar Format (JSGF) is a BNF-style, platform-independent, and vendor-independent textual representation of grammars for use in speech recognition. Grammar is used to determine what the recognizer should listen for, and so describe the utterances a user may say*[8]*. The example below shows a grammar that generates basic control commands like *"move a menu please", "close file", "oh mighty computer please kindly delete menu thanks"*. It defines a public grammar rule called *"basicCmd"*. In order for this grammar rule to be publicly accessible, we must be declared it "public". Non-public grammar rules are not visible outside of the grammar file.

```
#JSGF V1.0

  public <basicCmd> = <startPolite> <command> <endPolite>;

  <command> = <action> <object>;
  <action> = /10/ open |/2/ close |/1/ delete |/1/ move;
  <object> = [the | a] (window | file | menu);

  <startPolite> = (please | kindly | could you | oh mighty computer) *;
  <endPolite> = [ please | thanks | thank you ];
```

figure 8: Command grammar that generates simple control commands.

The features of JSGF that are shown in this example above includes:

- ➔ using other grammar rules within a grammar rule.
- ➔ the OR "|" operator.
- ➔ the grouping "(...)" operator.
- ➔ the optional grouping "[...]" operator.
- ➔ the zero-or-many "*" operator.
- ➔ a probability (e.g., "open" is more likely than the others).

After the JSGF grammar is read in, it is converted to a graph of words representing the grammar called the grammar graph. It is from this grammar graph that the eventual search structure used for speech recognition is built. Shown below is the grammar graphs created from the above JSGF grammars. The nodes *"<sil>"* means "silence".

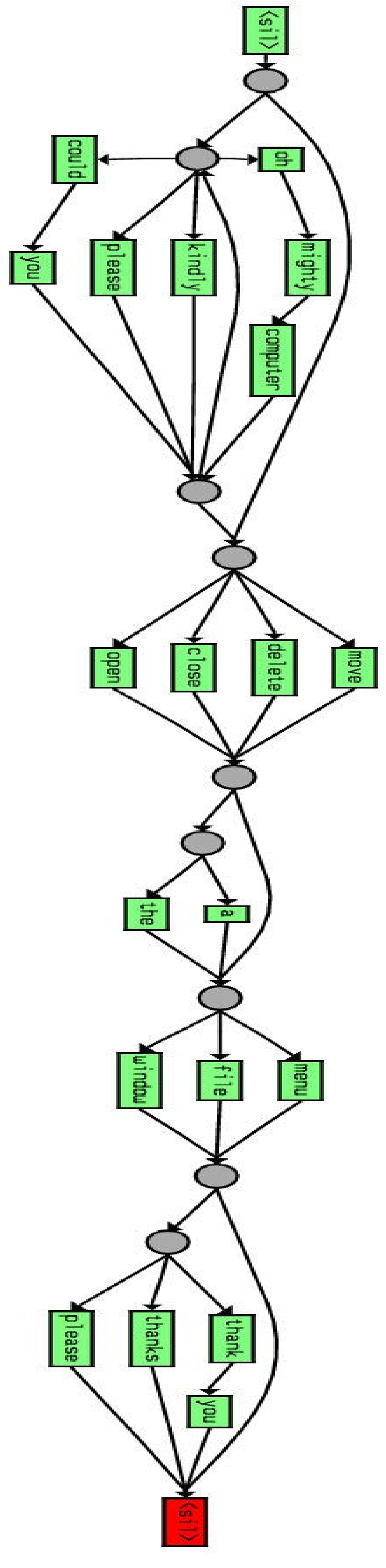


figure 9: Grammar graph created from the Command grammar.

ECMAScript Action Tags for JSGF

Sphinx-4 allows grammars written in the Java Speech API Grammar Format to use the JSGF tagging mechanism together with ECMAScript to specify a transformation from an utterance to information that is meaningful to the application. The information is returned in the form of ECMAScript values, such as strings and sets of attribute-value pairs (ECMAScript objects).

Need for ECMASCRIPT

ECMAScript is intended to address the following technical challenges in developing and using speech recognition applications.

- Simplify the paraphrase problem: When a grammar allows many forms of an utterance to have the same meaning to the application, the action tags provided to the application should be the same.
- Internationalization: Whenever possible, action tags can be language-neutral so that the application is less sensitive to the spoken language and so that new grammars can be developed for new languages with minimal changes to the application.
- Enhance documentation and maintainability: Because syntax and semantics are jointly defined, modifications can be made simultaneously and documentation can be co-located.

JSGF grammar allow an application developer to specify the legal utterances-sequences of words that the user may say. However, typically the sequence of words in is not by itself very useful to an application. Consider the following examples:

Utterances	**Application need**
eight thousand five hundred and twenty four	8524
August 24th 2007 the last day of summer session	“08/24/2007” or {year:2007, month:08, day:24}
I want to fly from Boston to Chicago. Hikoki-de, Boston-kara, Chicago-made ikitai	{action:”fly”, from: “Boston”, to: “Chicago”}

The above table illustrates two kinds of values that are useful to applications: simple values such as numbers or strings, and sets of attribute-value pairs. Simple values are useful for example in grammars for basic types such as numbers, dates, and times. Simple values are also useful in simple command & control applications and in directed-dialog applications in which the user is asked a question and is then expected to supply a single piece of information. Sets of attribute-value pairs are useful in more complex command & control applications and in more sophisticated dialog applications, in which any utterance may simultaneously provide several pieces of information to the application. In case of p2nspeech we use attribute-value pair method. For, example the utterance *“can you go near the red ball [sphere]”* has following attribute-value pair:

attribute	**value**
task	go-near
color	red
object	ball

Following is the part of the grammar file that leads to utterances similar to one above:

```
<task> = (pick up){this.value="pick-up"}| (go near){this.value="go-near"}|
              (push){this.value="push"}| (go around){this.value="go-around"};
 <color> = red{this.value="red"}|blue{this.value="blue"}|black{this.value="black"}|
                yellow{this.value="yellow"}|white{this.value="white"};
<smallobject> = /1/sphere{this.value="ball"}|/6/box{this.value="box"};

public <command>=(can you <task>{this.task=$task.value} the <color>{this.color=$color.value}
                             <smallobject>{this.object=$smallobject.value});
```

figure 10: Sample from p2nspeech Grammar file

Sphinx-4 API provides tools for processing JSGF grammar using ECMAScript action tags. These can be found under the package “edu.cmu.sphinx.tools.tags”*[9]*.

Festival:Speech Synthesis System

Festival offers a full text-to-speech support through number of API's. It is completely developed in C++ and uses the Edinburgh Speech Tools Library for low level architecture and has a Scheme (SIOD) based command interpreter for control [Festival sits on top of the Edinburgh Speech Tools Library, and uses much of its functionality]. Festival is multi-lingual (currently English, Welsh and Spanish) though English is the most advanced. The most current version of festival is 1.95.

The distribution includes:

- Full English (British and American) text to speech
- Full C++ source for modules, SIOD interpreter, and Scheme library
- Lexicons based on CMULEX and OALD
- Edinburgh Speech Tools, low level C++ library
- HTS hidden markov model based synthesis engine 4 HTS American English Voices (From CMU)
- Multisyn general purpose unit selection engine
- Various multisyn voices (More to follow)
- British English diphone database
- 2 American English diphone databases
- 4 voices (1 British male, 2 American male and 1 American female)
- Castilian Spanish diphone database
- Full documentation (html, postscript and GNU info format)

Festival can be used in a variety ways for speech synthesis. But, for p2nspeech we only use it for simple text-to-speech. We run festival in client-server mode where festival is the server and p2nspeech is the client. Festival server keeps listening for a text input over the TCP/IP network. The details about how festival and p2nspeech communicate is explained later in the document.

p2nspeech

The main aim of this project is to enable enable the robot [we call it p2] to listen, process and understand few basic voice commands. p2nspeech consists of four main blocks:

1. *SoarRobot* -> The Soar Cognitive Architecture
2. *Sphinx-4* -> The speech recognition system
3. *Festival* -> The speech synthesis system
4. *Player-Gazebo* -> The robot simulator

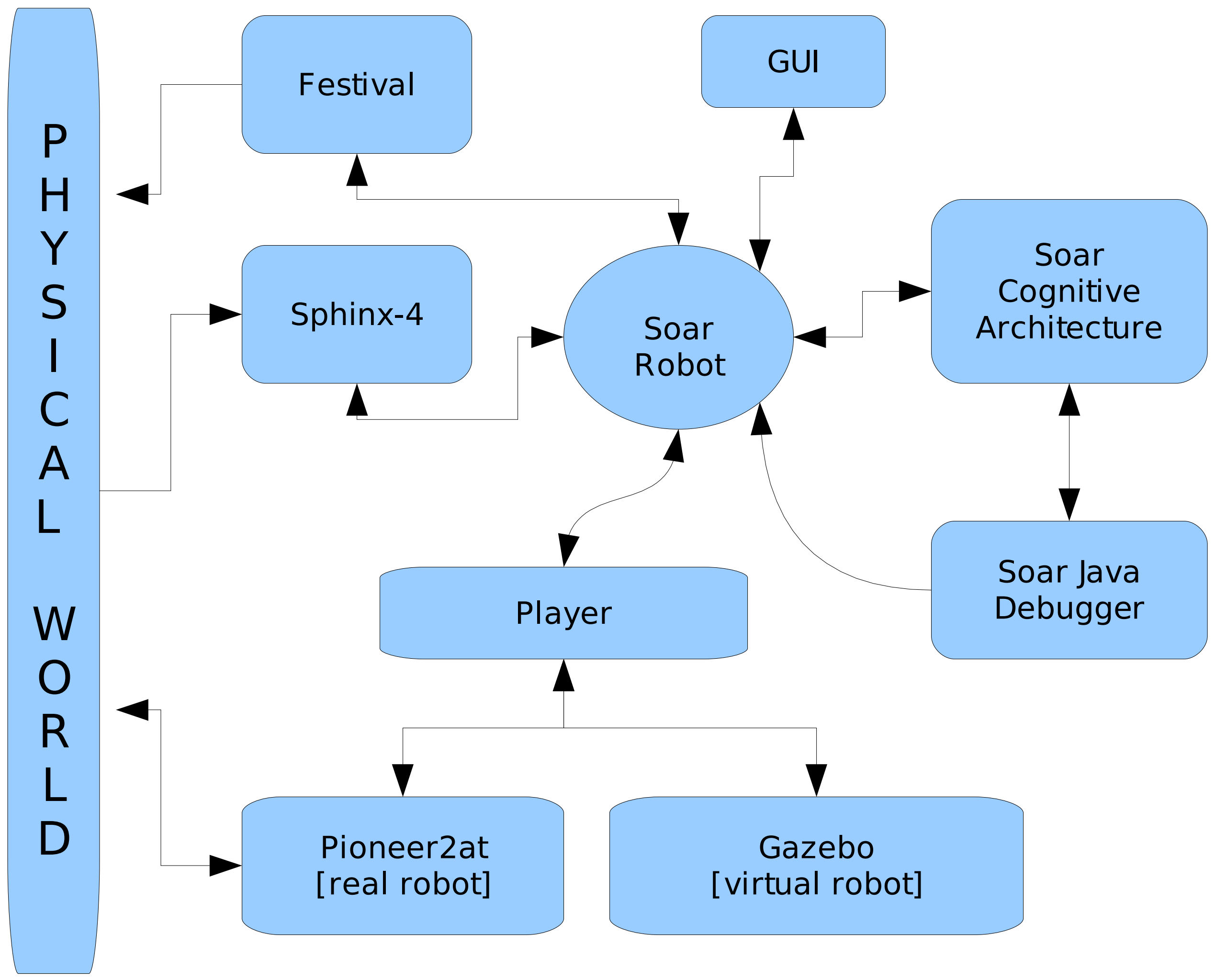


figure 11: p2nspeech block diagram

How it all works?

The above figure shows the block diagram of p2nspeech and how each block is connected to each other. As sceen from the figure above the three blocks that interact with the real physical world are the robot it-self, sphinx-4 and festival. The Soar block [Soar Cognitive Architecture] is responsible for making all the decisions based on facts and the current data about the world. All other blocks are connected to Soar through SoarRobot. The real robot and the virtual robot both are connected to SoarRobot through player so, the same code can work with the real robot and the virtual robot with minimum change to the code.

SoarRobot:SoarRobot_PS

SoarRobot is the main thread for the whole application. It starts Sphinx-4, GUI, Soar in a new thread and connects to Festival and Player over TCP/IP network. It also starts the Soar Java Debugger which connects directly to Soar kernel over TCP/IP network too.

SoarRobot and Player is connected via SoarRobot_PS class; this allows SoarRobot to be used with the real robot and the virtual robot by just changing one line in SoarRobot which is as follows.

```
SoarRobot_PS gz = new SoarRobot_PS(“gazebo”);
```

This will create an instance of SoarRobot_PS that will connect to Gazebo over the network and now all the method call will send information to and receive informaion from Gazebo.

Simillarly the follwing line will crearte an instance of SoarRobot_PS that will connect to the real robot

```
SoarRobot_PS gz = new SoarRobot_PS(“p2”);
```

In case where SoarRobot_PS is used to communicate with gazebo we use the JavaClient API that was mentioned above. The JavaClient API implementents all the interfaces needed to connect to the palyer. The general architectur for connecting to the player via JavaClient is as follows[10]:

```
1: Connect to robot by constructing a PlayerClient object
2: Create devices that are to be used in the program by requesting them from
      the PlayerClient object.
3: while(someConditionToFinishIsNotTrue) {
4:    Read the data from devices
5:    Based on received data determine actions
6: }
```

figure 12: JavaClint General Structure

For better understanding of the concept consider the following code snippest:

```
1: import Javaclient.src.*;
2: public class JAVACLIENT_EXAMPLE {
3:  public static void main(String[] args) {
4:    PlayerClient pc = new PlayerClient("localhost",6665);
5:     Position2DInterface p2di= pc.requestInterfacePosition2D(0,
                           PlayerConstants.PLAYER_OPEN_MODE);
6:     while (true) {
7:         pc.readAll();
8:         ppd.setSpeed(100, 30);
9:    }
10: }
11: }
```

figure 13: JavaClient example code snippest

In the above figure at line 1 we import the *JavaClient* library. At line 4 the PlayerClient object is being created. The two parameters that are needed to connect to the player are the ServerName ("localhost" means connect to machine that executes the program) and PortOfConnection (6665 default port number for the connection). The PlayerClient is the main Javaclient class and it contains methods for interacting with the player device. Line 5 creates/request a Position2D device. The two parameters it needs are the index of the device(in this case it is zero) and the access mode[10]. Lines 6-9 describe a "life cycle" of the program. At line 7 a readAll() method is being called, which reads data for every created device. Thus, after line 7 user can access new data returned from the server.

Line 8 set the speed for the robot.

Festival:Festival_Client.java

Once connected to the robot, SoarRobot creates an instance of festival_client. This class is responsible for connecting to the festival server that running as a serepate process. It does all the initialization needed. It dose so by importing two libraries namely “festival.client” and “festival.scheme”. These two libraries have all the methods needed to connect to the festival server, to do initializations and to register for events which are fired when a request for text-to-speech is send to the festival server.

One thing to note about festival_client is that it can either run in test mode or real mode, that is when running in real mode it acctually connects to the festival server and dose text-to-speech. While in test mode it dose not connect to the festival server and dose no text-to-speech instead it prints out the text on standard output. This is very useful while testing or debugging the code or if one needs to disable text to speech. All that needs to be done is just change on line in SoarRobot that creates a new instance of festival_client class.

```
i]        festival_Client tts=new festival_Client(false);

ii]   festival_Client tts=new festival_Client(true);
```

figure 14: Starting festival client in test mode and real mode

The above figure show how to start the festival_client in test mode and in real mode. The only difference between the two statements is the parameter that is used while creating the new object. Line one causes festival_client to start in test mode while line two causes it to start in real mode. If the festival server is not running or if there is some problem while connecting to it in real mode, festival_client uses the the default mode that is the test mode[festival server needs to be manually started from the shell].

When a new instance of festival_client is created all the client side default initialization and configuration is take care of automatically. As festival server listens for commands over TCP/IP network, we can have the server and the client running on serepate machine. In case of p2nspeech we run it on the same machine. By default festival server listens on port 1314 and festival client tries to connect to the server on the same machine on port 1314. If festival server needs to be run on a different machine all that needs to be changed it the IP address.

Sphinx-4:Speech_Reco.java

This class is responsible for detecting the audio hardware, configuring it and loading all the necessary modules needed for speech recognition depending on the configuration file explained above. In general Sphinx first dose all the initialization needed and then keeps running in an infinite loop processing the audio data and trying to detect if any thing was said. So in p2nspeech the main thread SoarRobot starts Sphinx in a separate thread. Sphinx uses custom events to notify the main thread when it recognizes some command.

Similar to Festival_client sphinx can also be started in either test mode or real mode. When run in test mode the user can type-in the commands from the GUI. This is useful while debugging and testing. The following code fragment show how to start sphinx in test mode and real mode respectively:

```
i] spreco=new Speech_Reco(true); //true = testmode

ii] spreco=new Speech_Reco(false); //false = realmode
```

figure 15: Start Sphinx in test mode and real mode

As it can be seen from the above code sample, similar to Festival_client, sphinx can be started in test mode or in real mode by just changing one line in SoarRobot.java.

When Sphinx is started in real mode [a new thread is created in either case] it calls the method "Init_reco()" which dose all the initialization needed and finally runs the infinite loop listening to the audio data. Speech_Reco.java imports several Sphinx-4 classes following are the three main classes that are must for performing speech recognition:

```
import edu.cmu.sphinx.recognizer.Recognizer;

import edu.cmu.sphinx.result.Result;

import edu.cmu.sphinx.util.props.ConfigurationManager;
```

figure 16: Sphinx-4 must needed library

The *Recognizer* is the main class any application should interact with (refer also to the architecture diagram above). The *Result* is returned by the Recognizer to the application after recognition completes. The *ConfigurationManager* creates the entire Sphinx-4 system according to the configuration specified by the user.

```
try{
      URL url;
      url=Speech_Reco.class.getResource("p2world.config.xml");
      cm = new ConfigurationManager(url);
      recognizer = (Recognizer) cm.lookup("recognizer");
      mic =(Microphone) cm.lookup("microphone");
}catch(Exception e){
      System.err.println("Error starting speech recognizer");
      e.printStackTrace();
}
```

figure 17: Sphinx:Speech_Reco.java.Init_reco()

Consider the above code for "Init_reco()", The first few lines creates the URL of the XML-based configuration file. A ConfigurationManager is then created using that URL. The ConfigurationManager then reads in the file internally. Since the configuration file specifies the components "recognizer" and "microphone" (for details refer to the API doc[7]), we perform a *lookup()* in the ConfigurationManager to obtain these components. The *allocate()* method of the Recognizer is then called to allocate the resources need for the recognizer. The Microphone class is used for capturing live audio from the system audio device. Both the Recognizer and the Microphone is configured as specified in the configuration file.

Following is the part of the "Init_reco" that dose the recognition and also also fires a custom event named "Event_new_voice_command" every time it recognizes some thing. The program first turns on the Microphone (`mic.startRecording()`). After the microphone is turned on successfully, the program enters a infinte loop that repeats the following. It tries to recognize what the user is saying, using the `recognizer.recognize()` method. Recognition stops when the user stops speaking, which is detected by the endpointer built into the front end by configuration. Once an utterance is recognized, the recognized text, which is returned by the method `result.getBestFinalResultNoFiller()`, is printed out and the new voice command event is fired to notify all the listener. If the Recognizer recognized nothing (i.e., result is null), then it will print out a message saying that. Finally, if the program cannot turn on the microphone in the first place, the Recognizer will be deallocated, and the program exits. It is generally a good practice to call the method `deallocate()` after the work is done to release all the resources.

```
if(mic.startRecording()){
      while(true){
            Result result = recognizer.recognize();
            if(result!=null){
                  String you_said = result.getBestFinalResultNoFiller();
                  if (you_said.isEmpty()){
                        continue;
                  }
                  System.out.println("you said: " + you_said);
                  //Get the confidence Score for each word
                  ...

                  fire_event_new_voice_command(you_said);

            }else{
                  System.out.println("Sorry cannot hear you properly
                                          \n please try again");
                  }
      }
}else{
        System.out.println("cannot start microphone");
        recognizer.deallocate();
        System.exit(1);
}
```

The event (Event_new_voice_command) basically contains only a string variable that hold the voice command that was just recognized so that the listeners of this event can get the data they need. When SoarRobot starts Sphinx in a new thread, it registers itself as a listener for the custom event. So, from now onwards it is notified every time Sphinx recognizes a command. Following is the code fragment that dose that:

```
spreco.add_NewVoiceCmd_Listener(new LISTNER_new_voice_command(){
      public void A_new_voice_command(EVENT_new_voice_command evt){
            set_str_new_voice_cmd(evt.NewVoiceCmd);
      }
});
```

figure 18: Register SoarRobot for new voice command event

So, every time Sphinx fires new voice command event , the method `A_new_voice_command()` is called. It then calls the method `set_str_new_voice_cmd()` which stores the new voice command in SoarRobot class variable. The reason for doing this will be made clear later in this document.

SoarRobot(explained further)

Once, SoarRobot is done with initialization of all the components of p2nspeech it starts its main work *loading, initializing and running SOAR* . It first starts Soar Kernel in a new thread and starts the Java Soar debugger that connects to the soar kernel directly via sockets. The soar debugger helps in getting various information about the current state of the soar kernel which includes information about the working memory, operators proposed, operator selected and so on. It then creates a new agent named “SoarSphinx” and load all the productions from “soar.sphinxagent.soar” file. Following is the code fragment that does all this and some further work explained later.

```
try{
      kernel = Kernel.CreateKernelInNewThread("SoarKernelSML");
}catch(Exception e){
      System.err.println("Exception while creating kernel: " + e.getMessage());
      System.exit(1);
}

if (kernel.HadError()){
      System.err.println("Error creating kernel: " +
      kernel.GetLastErrorDescription());
      System.exit(1);
}

agent = kernel.CreateAgent(AGENT_NAME);
boolean load = agent.LoadProductions(SPHINX_AGENT);
if (!load || agent.HadError()) {
    throw new IllegalStateException("Error loading productions: "
                          + agent.GetLastErrorDescription());
}

CurrentRobotID = agent.CreateIdWME(agent.GetInputLink(), "currentrobotdata");

timer = new Timer();
timer_task= new TimerTask(){
public void run(){
      getRobotData=true;
}
};
timer.schedule(timer_task, 200,200);
registerForUpdateWorldEvent();
```

figure 19: Soar Initialization

As seen from the above code if any error is encountered during soar initialization then the system exits right a way. After it is done with creating the agent it creates a dummy working memory element on the input link named "currentrotodata". A new currentrotodata WME is created every time during the update world cycle. Later it creates a new timer task that repeatedly sets the boolean variable "`getRobotData`" after every 200 millisecondes which is later used while the update world cycle and is reset there. Finally SoarRobot registers itself for update event. It is similar to registering for new voice command event explained earlier. The only difference here is that here the SoarRobot will be notified by the Soar Kernel after it has finished its current execution cycle. During this the method `runEventHandler()` will be called and in case of p2nspeech will further call the method updateWorld() method that reads all the new commands from the output link and sends it to appropriate components [commands for robot to SoarRobot_PS, Response to Sphinx input to Festival_Client] and check if it is time to update the robotdata; if it is then it request all the data from the robot and attaches it to "rotodata" WME under input-link, also if there are any new voice commands then it attaches them too to "sphinxdata" WME under input-link also. Following is a general code structure of the methods `runEventHandler()` and `updateWorld()` that p2nspeech follows:

```
public void runEventHandler(int eventID, Object data, Agent agent, int phase){
try{
// We have a problem at the moment with calling Stop() from arbitrary threads
// so for now we'll make sure to call it within an event callback.
if (m_StopNow)
{
      m_StopNow = false ;
      kernel.StopAllAgents() ;
      System.exit(0);
}
      updateWorld() ;
}
catch (Throwable t)
{
      m_StopNow=true; //uncoment it later
}
}
```

figure 20: SoarRobot runEventHandler()

As one can see from the code above every time SoarRobot is notified about the completion on an execution cycle, it checks if it is time to stop Soar Kernel execution or continue with updating the data. It is done this was because at present stopping soar kernel while it is in its execution cycle is a little buggy, it might crash. Hence, we stop it after it finishes it execution cycle.

```
private void updateWorld(){
      //check if new command on agent's output link
      if (agent.Commands()){
          ...
          //process all the commands
      }

      //check if it is time to update data from the robot
      if (getRobotData)
      {
         getRobotData = false;
         ...
         //create a new rotodata
         //delete old currentrobotdata
         //make currentrotodata point to robotdata
         //read data from the robot and attach it to robotdata
         ...
      }


      //if there is any new voice command
      if (!str_new_voice_cmd.isEmpty()){
      //there is a new voice command now parse it and put it make new WME
      //this is done with ActioTagParser
            //create new sphinx data
            //delete currentsphinxdata
            //create new currentsphinxdata and
            //make it point to sphinxdata

            //parse the new voice data received for
            //ECMAScript tags
            try{
                  parseit(str_new_voice_cmd);
            }catch(Exception e){
                  System.out.println("sphinx data partailly updated");
            }

            //set it to null so that u dont
            //put the same info back again
            str_new_voice_cmd=null;
      }else{
            System.out.println("no new voice info to be updated");
      }

      //commite all the changes just made
      agent.Commit();
}
```

figure 21: SoarRobot updateWorld method structure

All updates to the Soar working memory can only be done between execution cycles hence, all the major work related to updating soar data and reading data from soar is initiated from the updateWorld method.

Soar Agents for p2nspeech

As mentioned at the beginning of the document that Soar is similar to a rule base system and it stores all the long-term knowledge or facts in rules, and short-term knowledge in working-memory. It then compares these facts against the working memory, during each execution cycle, to determine which facts are true for the current situation and applies them and retracts those that are now false; which results in making changes to the working memory which consequently results in some more rules to be fired and retracted.

So, we can call a set of such facts or rule as a agent for a particular task. Generally, soar agents are separated in files and have a ".soar" extension for better understanding. In case of p2nspeech we have the following soar agents:

- sphinxagent.soar
- wait.soar
- P2.soar
- move.soar
- makeworld_new.soar
- sphinx.soar
- sphinx_whereis.soar
- sphinx_gonear.soar
- sphinx_context.soar
- sphinx_can-you.soar

Here the last file "sphinxagent.soar" is like a parent agent file that will load all the other agents that are mentioned above. The Soar Kernel treats all the rule from all the soar agents file similarly. It dose not know that the are grouped together in separate files. It just for better understanding. In following sections explains in detail all the Soar rule from the above mentioned files.

sphinxagent.soar

This is the parent soar file that contains some configuration information for the Soar Kernel and all the Soar agent files that will be loaded when this file is loaded. It just contains statements like “source soar_agent_file_name.soar “ for each agent file.

wait.soar

As the name suggest this file contains soar rule (wait operator) that make Soar wait during some situations. It is useful in almost every task. The wait operator is proposed whenever there is a state no-change impasse. A state no-change impasse arises when no operators are proposed. This can happen when there is no new data for few execution cycles which results in no new operators to be proposed and applied, which in turns result in no change in working memory. If there were no wait operator, Soar would generate a cascade of state no-change impasses that ultimately causes Soar to halt. Following is the English explanation followed by Soar rule for the wait operator.

English explanation:

If the state has **^attribute state** and **^choices none** and has no wait operator proposed **then**, propose it.

Soar Rule:

```
#Wait operator
sp { propose*wait*operator
   (state <s> ^attribute state
              ^choices none
             -^operator.name wait)
   -->
   (<s> ^operator <o> + <)
   (<o> ^name wait)
}
```

In Soar the attributes [**^attribute state** and **^choices none**] on the state are created when no operator can be selected for a state – a state no-change impasse. The key to the wait operator is that it tests to see that a wait operator is not selected. Thus, once one is selected, the proposal no longer matches and retracts, but then will get reproposed during the next proposal cycle. The wait.soar file contains only the wait operator.

P2.soar

This file contains rules specific to Pioneer2AT robot. At present it only contains rules regarding the sonar position, but more rules related to sonar or other robot sensors can be added to this file. On P2 each sonar sensor has a number uniquely identifying it. Based on this sonar-number the rules from P2.soar file elaborate the position of the sonar on the robot that is, is the sonar is locate on the left, right, front, or back side of the robot. This information is specially useful while robot is moving . The soar rule responsible for moving the robot depends on theses rules in order to stop the robot if there is a wall in front of it or in other similar situations.

Elaboration rule:

English Explanation:

If there is new data on the input-link and this is the top state **then** attach attributes [^left-sonar 0 1, ^right-sonar 6 7, front-sonar 2 3 4 5] to state <s>. These attributes mean that sonar number 0,1 are no the left side of the robot, sonar number 6,7 are on the right side of the robot and sonar number 2,3,4,5 are in the front side of the robot [at present we don't use the sonars that are on the back side of the robot hence, there is no point in elaborating their numbers and location].

Soar Rule:

```
sp { elaborate*left-sonar
(state <s> ^io.input-link.currentrobotdata <c>
           ^superstate nil)
-->
(<s> ^left-sonar 0 1)
}


sp { elaborate*front-sonar
(state <s> ^io.input-link.currentrobotdata <c>
           ^superstate nil)
-->
(<s> ^front-sonar 2 3 4 5)
}


sp { elaborate*right-sonar
(state <s> ^io.input-link.currentrobotdata <c>
           ^superstate nil)
  -->
(<s> ^right-sonar 6 7)
}
```

move.soar

As the file name suggest this file has soar rules relate to robot motion. This file has few elaboration about the if anything is near on the side or front, about the robot motor status and so on. These basic elaboration are further used in other soar rules that decide the robot command to be sent to the robot depending on the voice command and the working memory created by these elaborations. Unlike the above explained files this file has more number of rules and some of it depends on the working memory elements created by the rules explained earlier.

Elaboration rule:

Following are the elaboration rules along with their explanation from movearound_sphinx.soar.

English explanation[Rule:near distance]**:**

If it is the top state **then** create **[^close-distance-front 500, ^close-distance-side 500].** This rule is fired only once at the beginning and it creates two working memories that specify what is near respective to the data from the sonars.

Soar rule:

```
sp { elaborate*init
(state <s> ^superstate nil)
-->
(<s> ^close-distance-front 500
     ^close-distance-side 500)
}
```

English explanation[Rule:robot stopped]**:**

If input-link has new data from the robot that says that robot velocity is zero **then** attach attribute **[^condition robot-stopped]** to the top state.

Soar rule:

```
sp { elaborate*robot*stopped
(state <s> ^io.input-link.currentrobotdata.Vel 0.0)
-->
(<s> ^condition robot-stopped)
}
```

English explanation[Rule:robot moving**]:**

If input-link has new data from the robot that says that robot velocity is not equal to zero **then** attach attribute **[^condition robot-moving].**

Soar rule:

```
sp { elaborate*robot*moving
(state <s> ^io.input-link.currentrobotdata.Vel { <v> <> 0.0 } )
-->
(<s> ^condition robot-moving)
}
```

English explanation[Rule:object near by in front**]:**

If there is new data from the robot and the front sonar data[distance] is less then the value for the attribute **[^close-front-distance] then** attach attribute **[^condition object-nearby-in-front]** to the top state. This rule will match for each front sonar once. This rule depends on the working memory that was created by the front-sonar elaboration[it determines which sonar are locate in the front of the robot].

Soar rule:

```
sp { elaborate*object-nearby-in-front
(state <s> ^io.input-link.currentrobotdata <c> ^front-sonar <num>)
(<c> ^sonar <son>)
(<s> ^close-distance-front <cl>)
(<son> ^number <num> ^reading { <v> < <cl> } )
-->
(<s> ^condition object-nearby-in-front)
}
```

English explanation[Rule:no object near by in front**]:**

This rule is similar to the rule explained above except that it fires only if there is no object near in the front. [This rule is coded differently as it was only written for testing earlier]

Soar rule:

```
sp { elaborate*no-object-nearby-in-front
(state <s> ^io.input-link.currentrobotdata <c> )
(<c> ^sonar <son1> <son2> <son3> <son4> <son5>)
(<s> ^close-distance-front <cl>)
(<son1> ^number 1 ^reading { <v1> > <cl> } )
(<son2> ^number 2 ^reading { <v2> > <cl> } )
(<son3> ^number 3 ^reading { <v3> > <cl> } )
(<son4> ^number 4 ^reading { <v4> > <cl> } )
(<son5> ^number 5 ^reading { <v5> > <cl> } )
-->
(<s> ^condition no-object-nearby-in-front)
(write (crlf) |no object near by in front|) }
```

English explanation[Rule:object near by to the right or left**]:**

This rule is also similar to the 'object near by in front' rule except is fire when the sonar locate on the right or left side of the robot report distance data that is less then the working memory for close distance on the side.

Soar rule:

```
sp { elaborate*object-nearby-to-right
(state <s> ^io.input-link.currentrobotdata <c> ^right-sonar <num>)
(<c> ^sonar <son>)
(<s> ^close-distance-side <cl>)
(<son> ^number <num> ^reading { <v> < <cl> } )
-->
(<s> ^condition object-nearby-to-right)
}


sp { elaborate*object-nearby-to-left
(state <s> ^io.input-link.currentrobotdata <c> ^left-sonar <num>)
(<c> ^sonar <son>)
(<s> ^close-distance-side <cl>)
(<son> ^number <num> ^reading { <v> < <cl> } )
-->
(<s> ^condition object-nearby-to-left)
}
```

English explanation[Rule:copy voice command**]:**

If there is new sphinx data on the input link and it has a new voice command **then** attach attribute[**^voicecmd**] to the top state and further attach attribute[**^cmd]** white the new voice command as its value to attribute **[^voicecmd]** just created.

Soar rule:

```
sp { elaborate*copy*voice-command
(state <s> ^io.input-link.currentsphinxdata <c>
           ^superstate nil)
(<c> ^voicecmd <vcmd>)
-->
(<s> ^voicecmd <v>)
(<v> ^cmd <vcmd>)
}
```

Propose/Apply rule:

Following are the proposal rule and their application rule from move.soar. These rules depend on the elaborations just explained above. These rules are responsible for issuing robot motion commands based on the sonar information and the voice command.

Proposal rule:

English explanation[Rule:propose move forward**]:**

If the robot is not already moving forward and there is no object in front of the robot and the new voice command say "move forward" **then** propose to move forward and specify the desired robot velocity.

Soar rule:

```
sp { propose*move*forward
   (state <s> ^condition robot-stopped)
   (<s> ^condition no-object-nearby-in-front)
  - (<s> ^action moving-forward)
    (<s> ^voicecmd.cmd move-forward)
-->
   (<s> ^operator <o> + = )
   (<o> ^name move-forward)
   (<o> ^target-velocity 150)
}
```

Remark:In the above soar rule the minus sign "-" is similar to 'NOT'. In this case it means that there is **no** attribute**[^action moving-forward]** attached to top state <s>.

Application rule:

English explanation[Rule:apply move forward**]:**

If there is a output link and the operator to move forward is selected **then** send command to the robot to set its velocity to one specified by the operator through the output link and also attach attribute **[^action moving-forward]** to the top state.

Soar rule:

```
sp { apply*move*forward
   (state <s> ^io.output-link <ol> ^operator <o>)
   (<o> ^name move-forward)
   (<o> ^target-velocity <vel>)
-->
   (<s> ^action moving-forward)
   (<ol> ^setVel <com>)
   (<com> ^param1 <vel>)
}
```

Proposal rule:

English explanation[Rule:propose turn left/right**]:**

The proposal rule[also application rule] for turning the robot to left or right are similar to each other.

If there is a new voice command that says turn left [turn right] and there is no object near to the robot on the left side [right side] and there robot is not already turning left [right] **then** propose an operator to turn left [right].

Soar rule:

```
sp { propose*start*turn*left
   (state <s> ^voicecmd.cmd turn-left
        -^condition object-nearby-to-left
        -^action turning-right
        -^action turning-left)
-->
   (<s> ^operator <o> + =)
   (<o> ^name start-turn-left)
}

sp { propose*start*turn*right
   (state <s> ^voicecmd.cmd turn-right
          -^condition object-nearby-to-right
          -^action turning-right
          -^action turning-left)
-->
   (<s> ^operator <o> + =)
   (<o> ^name start-turn-right)
}
```

There are few more rules in move.soar that stops the robot from hitting anything while its moving forward or turning right or left. These rules are quite similar to the ones explained above. For example the rule to stop the robot if there is some obstacle in front while its moving forward will be as follows:

```
If the robot is moving forward and there is a object near by in front
  then propose a stop operator.
If the stop operator is selected and there is a output link
  then set the robot velocity to zero through the output link.
```

Similarly the rule to stop the robot while its turning right or left and if there is a object on that side then propose an operator to stop the robot and when selected the application rule will set the robot velocity to zero.

makeworld_new.soar:

All the rules in this file and the from the files that will be explained after this are mostly related to gazebo, the virtual robot simulator. The rules from this file create working memory elements that represent all the objects present in the virtual world of the robot. Due to the limitations of gazebo at present only static information about the objects is available. All the object information is attached to the top state <s> and has the following structure:

```
state <s> ^object <obj>
<obj> ^called object-name
          ^mass object-mass
          ^size object-size
          ^xyz object-location
```

The attributes size [^size] and location [^xyz] both further contain attributes [^x, ^y, ^z]. In case of size they represent the size of the object in x,y and z direction respectively and in case of location they represent the location of the object in x,y,z coordinate system. Due to the limitations of gazebo we can only retrieve real time data from the robot, we do not get any information about the world. Hence, when p2nspeech starts for the first time it parses the world file [xml base world file explained earlier] to get the initial information about the world and attaches it on the input-link. So when soar kernel starts running, rule from makeworld_new.soar will be fire and creating working memory elements for each object and attach it to the top sate.

Proposal rule:

English explanation:

If there is new information about a object and information about that object is not already attached to the top state **then** propose a operator to create an new WME for that new object.

Application rule:

English explanation:

If there is a operator proposed to create new WME for a particular object and all the information about that object is present on the input-link **then** create a new WME for that object on the top state and copy all the information about the object form the input-link to the newly created WME.

sphinx.soar

As the name suggest this file contains rule related to Sphinx [speech recognition]. This file contains rules that help in responding to question about if there is a particular object present in the virtual world, for example: 'is there a red box'. The answer for this type of questions is either 'yes' or 'no' and depends on the information that is created by the rules from makeworld_new.soar. This project [p2nspeech] is not capable of parsing a sentence and determining what the sentence mean. In p2nspeech we represent the information that we can derive form the sentence like the one mentioned above in a fixed structure on the input-link. This helps in simplifying the soar rules and besides that extracting valuable information from a sentence is far too complex and beyond the scope of this project. There is another project called NLSoar [11], It is a system built using the SOAR general cognitive architecture and aims to provide incremental real-time natural language capabilities.

Proposal rule:

English explanation:

If there is some new data from speech recognition and it is a particular type of question which is still not answered and question is of type 'is there a particular object?' and has all the information needed to identify the object in the world and if there a object in the virtual world that has the same properties as the object questioned about **then** propose a operator that will create answer as 'yes' to the question.

In case if there no object that has the same properties then propose a operator that will answer as 'no' to the question.

Soar rule:

```
#yes
sp { propose*qid11*yes
    (state <s> ^io.input-link.currentsphinxdata <sd>
       ^object <obj>
      -^answered <sd>
       ^superstate nil)
    (<sd> ^qid 11
          ^called <called>
          ^color <color>)
    (<obj> ^called <called>
           ^color <color>)
-->
   (<s> ^operator <o> + =)
   (<o> ^name q11ans
        ^ans yes
        ^called <called>
        ^color <color>
        ^sd_id <sd>)
(write (crlf) |propose yes|) }
#no
sp { propose*qid11*no1
    (state <s> ^io.input-link.currentsphinxdata <sd>
            -^answered <sd>
             ^superstate nil)
    (<sd> ^qid 11
          ^called <called>
          ^color <color>)
    (<s> -^object (^called <called> ^color <color>))
-->
   (<s> ^operator <o> + =)
   (<o> ^name q11ans
        ^ans no
        ^called <called>
        ^color <color>
       ^sd_id <sd>)
(write (crlf) |propose no|)
}
```

Application rule:

English explanation:

If the answer operator is selected and there is a output-link and the operator has all the information about the question including the answer for that question **then** attach all this information to the output-link under the attribute sphinxdata and mark the question as answered.

Soar rule:

```
   sp { apply*qid11
      (state <s> ^operator <o>
          ^io.output-link <ol>
          ^superstate nil)
      (<o> ^name q11ans
          ^ans <ans>
          ^called <called>
          ^color <color>
          ^sd_id <sd>)
   -->
       (<s> ^answered <sd>)
       (<ol> ^sphinxdata <sd1>)
       (<sd1> ^qid 11
             ^ans <ans>
       ^called <called>
       ^color <color>)
       (write (crlf) <color> | | <called> |: | <ans> )
   }
```

sphinx_whereis.soar

This file contains rule that respond to questions of type 'where is a particular type of object ?' for example, “where is the red ball ?” and the response is like “to front right”. It means that the object addressed is in front of the robot and is also to left of the robot. This information is very useful in robot motion , when the robot is asked to go near a particular object, its motion depends on the answer to this type of question. These rules uses sub-goaling to generate the response.

Proposal rule:

English explanation:

If there is some new sphinx data on the input-link and it is a question of type 'where is the red box' and it contains all the required information about the object that is needed to answer the question **then** propose a operator with all the information about the object.

Soar rule:

```
sp {propose*go-near-qid12
    (state <s> ^superstate nil
               ^io.input-link.currentsphinxdata <csd>
               ^object <anyobj>
            -^answered <csd>)
    (<csd> ^qid 12
           ^color <anycolor>
           ^called <anycalled>)
    (<anyobj> ^color <anycolor>
              ^called <anycalled>
              ^objid <objid>)
-->
   (<s> ^operator <o> + =)
   (<o> ^name whereis
        ^qid 12
        ^objid <objid>
        ^sd_id <csd>)
}
```

There is no application operator for this rule, hence it subgoals. Now in the sub state depending on the position of the robot and the position of the object operators are proposed and applied. The apply operator make changes to the top state <s> and creates new working memory regarding the position of the object. This results in retraction of the sub-goal as the answer to the question is found. There are four operator that can get proposed in the sub state depending on the position of the robot and the object. They are propose that object is in front and to right of the robot, propose that the object is in front and to the left of the robot, propose that the object is behind and to the right of the robot and propose that the object is behind and to the left of the robot.

Proposal rule:

English explanation:

If there is new sphinx data and its regarding the question of type 'where is a particular object' and it contains all the necessary information about the object **then** propose a operator with all the information available about the object.

Soar rule:

```
sp {propose*go-near-qid12
    (state <s> ^superstate nil
               ^io.input-link.currentsphinxdata <csd>
               ^object <anyobj>
              -^answered <csd>)
    (<csd> ^qid 12
           ^color <anycolor>
           ^called <anycalled>)
    (<anyobj> ^color <anycolor>
              ^called <anycalled>
              ^objid <objid>)
-->
   (<s> ^operator <o> + =)
   (<o> ^name whereis
        ^qid 12
        ^objid <objid>
        ^sd_id <csd>)
}
```

This propose operator has no apply operator hence soar will subgoal to find a solution and will retract where there is any apply rule match the working memory. In this case soar will subgoal to find the location of the object and in doing so it will propose operators in sub state which when selected and applied will make changes to the top state, which in turn will cause some rule to match in the top state and hence it will retract the sub goal.

Proposal operator:

English explanation:

If the current state is a sub-state and it is caused due to a operator with name “whereis” [that is this this sub state due a question similar to 'where is the red box'] and it has all the information about the object **then** compare the position information of the object with the position information of the robot and propose the position of the object with respect to the robot.

Soar rule:

```
sp {propose*object-position*front-left
    (state <s> ^name whereis
               ^superstate <ss>
               ^objid <objid>)
    (<ss> ^object <obj>
          ^io.input-link.currentrobotdata <crd>)
    (<obj> ^objid <objid>
           ^xyz <xyz>)
    (<xyz> ^x <ox>
           ^y <oy>)
    (<crd> ^pos <pos>)
    (<pos> ^x { <rx> < <ox> }
           ^y { <ry> < <oy> })
-->
    (<s> ^operator <o>)
    (<o> ^name objat
         ^loc front-left)
#    (write (crlf) |##########object front left##########|)
}

sp {propose*object-position*front-right
    (state <s> ^name whereis
               ^superstate <ss>
               ^objid <objid>)
    (<ss> ^object <obj>
          ^io.input-link.currentrobotdata <crd>)
    (<obj> ^objid <objid>
           ^xyz <xyz>)
    (<xyz> ^x <ox>
           ^y <oy>)
    (<crd> ^pos <pos>)
    (<pos> ^x { <rx> < <ox> }
           ^y { <ry> > <oy> })
-->
    (<s> ^operator <o>)
    (<o> ^name objat
         ^loc front-right)
#    (write (crlf) |##########object front right##########|)
}

sp {propose*object-position*behind-right
    (state <s> ^name whereis
               ^superstate <ss>
               ^objid <objid>)
    (<ss> ^object <obj>
          ^io.input-link.currentrobotdata <crd>)
    (<obj> ^objid <objid>
           ^xyz <xyz>)
    (<xyz> ^x <ox>
           ^y <oy>)
    (<crd> ^pos <pos>)
    (<pos> ^x { <rx> > <ox> }
           ^y { <ry> > <oy> })
-->
    (<s> ^operator <o>)
    (<o> ^name objat
         ^loc behind-right)
#    (write (crlf) |##########object behind right##########|)
}

sp {propose*object-position*behind-left
    (state <s> ^name whereis
               ^superstate <ss>
               ^objid <objid>)
    (<ss> ^object <obj>
          ^io.input-link.currentrobotdata <crd>)
    (<obj> ^objid <objid>
           ^xyz <xyz>)
    (<xyz> ^x <ox>
           ^y <oy>)
    (<crd> ^pos <pos>)
    (<pos> ^x { <rx> > <ox> }
           ^y { <ry> < <oy> })
-->
    (<s> ^operator <o>)
    (<o> ^name objat
         ^loc behind-left)
#    (write (crlf) |##########object behind left##########|)
}
```

Application rule:

English explanation:

If the current state is a sub state and there is a top state and if a operator is proposed regarding the object position **then** create new working memory on the top state with position information about the object.

Soar rule:

```
        sp {apply*object-position
            (state <s> ^name whereis
                       ^io.output-link <out>
                       ^operator <o>
                       ^sd_id <csd>
                       ^superstate <ss>)
            (<o> ^name objat
                 ^loc <anyloc>)
        -->
            (<ss> ^answered <csd>)
            (<out> ^sphinxdata <sd>)
            (<sd> ^qid 12
                  ^ans <anyloc>)
        }
```

sphinx_gonear.sphinx

This file contains rule that deal with robot motion. It responds to voice commands like 'go near the red box'. This file has one of the most complex rule as I depends on all the working memory elements that is created by the rules explained earlier and all the information that is received from the robot and then accordingly create new working memories and generate commands to control the robot motion until it reach near the desired object. In general it decides weather the robot should keep moving forward or take a turn based on the working memory elements that are present at that time.

When the new voice command is to go near some object, we create a new working memory element that represents that the robot is moving towards some object. We do this because it will take some time for the robot to finally reach near the requested object, hence in the mean time if something else is asked for example "is there a red box?" then the robot can answer to that question and still remember that it was doing some task. This example helps demonstrating performing simultaneous activities just like humans. It will attach all the new working memory generated based on the current information available to this newly created working memory that represents the task. All this information will be helpful in deciding the next command for the robot.

Proposal rule:

English explanation:

If it is the top state and there is a new data on the input link and it say to go near some object and it also has all the information about that object and there is no other same task **then** propose a operator to create a new working memory that will hold all the information while the robot is going near the object.

Soar rule:

```
     sp {propose*go-near-rid1
   (state <s> ^superstate nil
              ^io.input-link.currentsphinxdata <csd>
              ^object <anyobj>
             -^gonear-actions <act>
             -^answered <csd>)
   (<csd> ^rid 1
          ^color <anycolor>
          ^called <anycalled>)
   (<anyobj> ^color <anycolor>
             ^called <anycalled>
             ^objid <objid>)
-->
  (<s> ^operator <o> + =)
```

```
   (<o> ^name gonear
        ^objid <objid>
        ^sphinxid <csd>)
}
```

Application rule:

English explanation:

If there is a operator proposed to go near a particular object **then** create a new working memory attached to the top state and create sub-working memory that will hold all the specific information about the task at hand like, error-distance will hold information about how far is the robot from the object in both x and y direction.

Soar rule:

```
sp {apply*go-near*actions
    (state <s> ^operator <o>)
    (<o> ^name gonear
         ^objid <objid>
       ^sphinxid <csd>)
-->
    (<s> ^gonear-actions <act>)
    (<act> ^objid <objid>
           ^update yes
         ^status stoped
         ^robotheading <rh>
           ^objectatheading <oh>
           ^error <err>
           ^sphinxid <csd>)
(write (crlf) | ####apply go near actions#### |)
}
```

This new newly created working memory will cause more soar rules to be matched and fired and this will cause soar to sub goal where some more rules will be triggered that will generate all the information and attach it to the top state under the task working memory. This will then cause more rules to match and issue new commands to the robot and this will cause the robot data to change and hence soar will again subgoal to generate further new information. This loop will continue until the robot reach the desired object. There is a whole bunch of soar rules in sphinx_gonear.soar file that are responsible to generate all the information. They all quite easy to understand.

sphinx_context.soar and sphinx_can-you.soar:

These files contains rule that relate to context awareness. For example consider the following conversation:

- I am in New York city.
- I at time square.
- Is 1 Pace Plaza near to 163 William Street [CSIS building] ?

If we look at the whole picture, we are talk about the whole New York city, which is pretty big and when we say "Is 1 Pace Plaza near to 163 William Street ? " we are talk about a smaller part in the New York city which we can say that it is close to each other comparatively. Now lets say,

- I am at 163 William street.
- I am in the Robot Lab on the second floor.
- Is 1 Pace Plaza near to 163 William Street [CSIS building] ?

Now in the above conversation we are talking about just the CSIS building and the robot lab which is in 163 William street. So when we say "Is 1 Pace Plaza near to 163 William Street ?", now the answer can be No as we are talking about just the CSIS building and the robot lab which are in the same building and 1 Pace Plaza is a completely different building, few streets away from CSIS building. So the answer to the same question depends on the topic or the context we are talking about. Determining the context from the conversation and responding to questions based on that is a very complex process from AI perspective, which we humans do all the time with ease.

In p2nspeech we have few rule that respond to some straight forward questions about if a object is near or not based on the context set by the previously asked question. This project cannot determine the context from a conversation but we manually set the context based on the question and then the soar rules uses that context while responding to a question. For example, If we ask "is there a red box ?" then the answer can be yes or no depending on whether there is a red ball in the virtual world or not. But this will set the context to "entire virtual world" because the question meant that "is there a red box in the entire virtual world or not ?" So if the next ask that "is the red box near ? " then the robot might say that "it is near" as we are talking about the entire virtual world.

Now say, we ask "can you pick up the red box?" , it might say it can or it cannot depending on the weight of the box. But now the context narrows down to the red box only and not the entire virtual world. Now if we ask again "Is the red box near ? " , this time robot might say 'No' even if the robot is closer to the object then it was in the previous case. It is because now we are talking only the red box and not the entire virtual world. The context is now completely different, hence near-distance is different now then it was earlier when we meant entire virtual world. Following are the soar rules that make this possible, when soar first starts it set “pick-up” as the default context, that is we are talk about picking up some object.

Proposal rule:

English explanation:

If there is a new sphinx data on the input link and it has information about the context and if it different then the current context **then** propose a operator to change the context.

Soar rule:

```
sp {propose*set*context*from-sentence
    (state <s> ^superstate nil
               ^io.input-link.currentsphinxdata.task <anytask>
               ^current-task <ct>)
    (<ct> -^task <anytask>)
-->
    (<s> ^operator <o> + =)
    (<o> ^name change-task
         ^task <anytask>)
}
```

Application rule:

English explanation:

If there is a operator proposed to change the context **then** delete the working memory that represent the current context and create a new working memory to represent the new context.

Soar rule:

```
sp {apply**set*context*from-sentence
    (state <s> ^operator <o>
               ^current-task <ct>)
    (<ct> ^task <crrtask>)
    (<o> ^name change-task
         ^task <anytask>)
-->
    (<ct> ^task <anytask> <crrtask> -)
}
```

As mentioned above at present p2nspeech associates context with only with one type op question “is a particular object near or not”, that is only with what what is meant by 'near'. How far can a object be from the robot to be said that it is near. So, in p2nspeech context is associated with near distance. So different if the context change then the near distance change.

Hence, if we talk about the entire virtual world then near-distance is say, 10 units, so anything that is within 10 units from the robot is said to be near to the robot; and now if we talk about picking up something the context changes and focuses on just one object hence, near-distance is now say 3 units, so the same object that was considered to be near earlier is now not near anymore. It is simple as that. We have a bunch of elaborations that set the near distance based on the context information available. They all a quite simple and are as follows:

Soar rule:

```
sp {elaborate*near-distance*pick-up
    (state <s> ^current-task <ctsk>)
    (<ctsk> ^task pick-up)
-->
    (<ctsk> ^near-distance 1)
}

sp {elaborate*near-distance*go-near
    (state <s> ^current-task <ctsk>)
    (<ctsk> ^task go-near)
-->
    (<ctsk> ^near-distance 4)
}

sp {elaborate*near-distance*go-around*OR*push
    (state <s> ^current-task <ctsk>)
    (<ctsk> ^task << go-around push >>)
-->
    (<ctsk> ^near-distance 3)
}

sp {elaborate*near-distance*find
    (state <s> ^current-task <ctsk>)
    (<ctsk> ^task find)
-->
```

```
            (<ctsk> ^near-distance 8)
        }
```

So, if we ask about if something is near or not then it completely depends on how far it is and whats the current near distance set to, which in turn depend on what we are talking about. It is quite similar to how humans decide if any physical object is said to be near them or not. But in case of humans it depends on a lot other things too for instance how tired you feeling ? or how lazy you are feeling etc.

Conclusion

With all the SOAR rules and Speech Recognition turned off the robot doesn't do much; The server program running on the robot keeps waiting for a command from the client. When we send a command to move forward that is enable motors for both the wheel and set the velocity the starts moving forward and keeps moving in the same direction until a command is send to stop or turn. In this situation while the robot is moving forward and if there is some obstacle in front of it the robot sonar sensors will detect is but but the robot will keep moving as there are no SOAR rule processing this sonar information from the robot and also there is no speech recognition enabled which enables the user to tell the robot to either turn or stop. The robot will hit the obstacle unless we sent a command to stop or turn.

When both SOAR and Speech Recognition are turned on we can ask the robot to move forward. [Voice Command: Move/Go Forward]. In this case the SOAR rules will decide weather the robot should move forward or not depending on weather there is any obstacle in front of it. If its path is clear then it will start moving forward and if the sonars report of any obstacle close in front of the robot, SOAR rules to turn the robot will fire causing the robot to turn. Besides that the user its self can ask the robot to turn in either directions or stop by just saying the appropriate voice command.

When SOAR is turned off the robot is unable to interpret the sonar information. But when we have SOAR turned on the robot can interpret the sonar data and can send appropriate commands to the robot based on the SOAR rules that fire. Further with speech recognition also enabled the user can speak out the voice command and once the speech recognition application finishes its processing it sends the voice command to SOAR. This causes rules to match and fire thereby creating new working memories which in turn will cause more rules to fire. Depending upon the voice command received and present sonar information SOAR sends appropriate command to the robot.

Future Possibilities:

At present the SOAR rule's are fairly simple and straight forward. It can process the sonar information coming for the robot and generate command's for the robot in order to avoid collisions. For the real robot the only way to get information about the surrounding world is through the sonar sensors. The sonar data is not very useful in picturing the surrounding world. The robot can use it to detect the presence or absence of some object, but it cannot use it to identify the object. This limits the information that we can get into SOAR.

But with the Player-Stage we can go a step further. Here we can get information about the robot surrounding world. Its not very detailed information but it is more that what we can get from the sonars in case of the real robot. With Player-Stage as mentioned earlier in this document, the virtual world is build from a XML configuration file which contains a fair amount of information about the virtual world. So, we parse this XML file and extract the virtual world information and push it into SOAR. Hence in case of Player-Stage we can ask the robot about the presence or absence of some object, how far a particular object is and so on. But again the information about the virtual world is partly static which limits the use of this simulator.

Humans rely on five types of sensing, sight, hearing, taste, smell, and tough. We combine all sensing information together to get the best solution for every situations. It would be a big forward step if a robot can learn how to combine different type of information together to solve the problem. In future we can have a vision system that can process the vision information from the cameras mounted on the robot in real time and generate appropriate SOAR memories similar to what we do now form the XML configuration file for the virtual world. The information from the XML configuration file is static but with the real time vision system we can have dynamic information about the surrounding world and we can use the same SOAR rule to question about surrounding objects. Building a such a real time vision system is very complex. Also such a vision system will need a lot more computational resources as it will have to process large amount of data in real time. The vision system will have to be capable of doing object recognition and a lot more other stuff. As long as the vision system can create similar working memories as we have at present, we can use this same application with the real robot too.

Besides this the robot can only understand simple voice commands but in future we can add more SOAR rules to enable the robot to understand more commands. We can also have SOAR rules that can do a fairly decent amount of language processing there by enabling the robot to understand complete sentences instead of simple voice commands. Efforts are being made in developing a SOAR based Natural Language Processing system called NL-SOAR.

Soar learning capability is one of the reasons that we base our research on it. Once we have much enough rules implemented based various basic ideas, we will be able to take more advantage from Soar by letting it learn and create rules for more complicate concepts by itself.

Bibliography

1: ActiveMedia Robotics, LLC., Pioneer 2 / People Bot Operation Manual, 2002

2: John E. Laird and Clare Bates Congdon, The SOAR users manual, Edition 1, 2006

3: Richard L. Lewis, Cognative Theory, SOAR, 2001

4: John E. Laird, The Soar 8 Tutorial Part 1, 2006

5: Robert I. Follek, SoarBot: A rule-based system for playing poker, 2003

6: Carnegie Mellon University, Configuration Manager for Sphin-4, 2004

7: CMU, Sphinx 4 API doc, ,

8: , Class API JSGFGrammar,

9: , API doc for Package edu.cmu.sphinx.tools.tags, ,

10: Maxim A. Batalin, Java Client for Player/Gazebo,

11: , NL-SOAR, ,