SPEECH ARTICULATING SOFTWARE

Project report submitted in partial fulfilment of the requirement for the degree of Bachelor of Technology

in

Computer Science and Engineering

By:

Ankit Saini (141214)

Under the supervision of

Amit Kumar

to

Department of Computer Science & Engineering and Information Technology
Jaypee University of Information Technology, Waknaghat, Solan-173234, Himachal Pradesh

Candidate's Declaration

I hereby declare that the work presented in this report entitled **"Speech Articulating Software"** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from December 2017 to May 2018 under the supervision of **Amit Kumar**(Assistant Professor (Computer Science Department)). The matter embodied in the report has not been submitted for the award of any other degree or diploma.

(Student Signature)

Ankit Saini 141214

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

(Supervisor Signature)

Mr. Amit Kumar
Assistant Professor
Computer Science Department

Dated:

Acknowledgement

I owe my profound gratitude to my project supervisor M**r. Amit Kumar**, who took keen interest and guided me all along in my project work titled: **Speech Articulating Software** which is used to segregate the reviews whether it's true or fake, till the completion of my project by providing all the necessary information for developing the project. The project development helped me in research and I got to know a lot of new things this domain. I am really thankful to him.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

Giving command or to search something on the computer requires keyboard for input.An easy task but not for everyone even this is a barrier for people suffering from blindness or arms disability.Even after having potential they can never become a programmer because of there disabilities.Blind people can speak the commands, type documents and search flights and make reservation online through speech.And that is where the speech recognition helps to give them the means to achieve something they were initially not able to.But this is not the only advantage that SR has , even it has other potential like saving time, helps in learning new languages as it can act both as a translator and interpreter.Our project is divided into two parts one for speech recognition: Sphinx-4 and other for speech synthesis: FreeTTS library.

CHAPTER-1
INTRODUCTION

1.1 Introduction

Speech recognition(SR) which means the ability of computer to recognise or match a voice against a provided or acquired vocabulary and to perform the required task.The basic task of computer is to understand the spoken language and by understand we mean to react and convert the input speech into another medium example text.That is why it is also known as automatic speech recognition or computer speech recognition or speech-to-text(STT).

A Speech recognition needs a microphones that a person can speak into the SR software. A computer to understand and interpret the speech a good quality sound card is needed for a proper and good pronunciation input and output.In our project we will be using Sphinx 4 Speech Recognition system.The software is made with the collaboration of Carnegie Mellon University ,Sun Microsystems Laboratories and Mitsubishi Electric Research Laboratories (MERL).The platform for the software was chosen to be Java™ programming language and was built on it.It is made to be simple and flexible and that is why it is able to support every HMM-based acoustic models, language models and search strategies.The Algorithmic innovations in the software able it to use simultaneously multiple data streams.The software is made to be an open source project and was available publicly at SourceForge™ since its origination.

Every detail related to the project whether it is design, results or meeting notes all are publicly accessible.As the system has been made on Java, it has inherited all its advantages as well which includes its portability: once compiled the byte code can be used on any system that supports Java, to maintain memory management it has garbage collection which helps the system to from any memory considerations, it also provides means to write multithread applications for multiprocessor machines.Also it extracts the info the code and creates hypertext mark-up language files that provide documentation concerning the software system interface.

1.2 PROBLEM STATEMENT

To develop a programme which will perform the specified task victimisation by recognising command through speech and scrutiny it with the in-built word vocabulary. together with all this it additionally must offer output within the variety of speech for interacting with the user.

One of the biggest challenges in the research of Speech Recognition Systems is its accuracy.The accuracy is affected by many factors including noise, different speakers, different language, the size and domain of vocabulary.Even the design has its own challenges like varied kinds of Speech classes and Speech illustration, Speech Preprocessing stages, Feature Extraction techniques, info and Performance analysis.

1.3 OBJECTIVE

We have divided our task into two parts: -

Speech-to-Text

This is the tough part, the speech recognition, where through speech input terminals we'll attempt to acknowledge what the person is saying. For this we've to make the recognition grammar for the software package. we'll be recognising through Sphinx 4 library framework that Is written entirely in Java programming language so has inherent advantage of code maintenance.

Text-to-Speech

This is the half where when recognition and performing the task the system can output Speech as stating the completion or failure of the task and requesting following task. we are going to be achieving this through FreeTTS. FreeTTS could be a speech synthesis system written entirely within the Java programming language.

Requirements: -





Files Needed: -

1. Grammar File
2. Java File
3. Configuration File

1.4 METHODOLOGY

STEP 1: - Building the grammar file

A grammar in the Java Speech API can be defined as the speech that the recogniser listen to.Grammar are the set of words or tokens which a user say and also the sentences or patterns within which these tokens are said.

Rule grammar and dictation grammar are the two types which are supported by the Java Speech API.They are distinguished by the way the patterns of token are represented and their programmatic use.An application defines the rule grammar where as a recogniser defines the dictation grammar , also the dictation grammar is built int the recogniser.

An application provides the rule grammar to recogniser which defines the rules and restrict the speaker what he can say.Words or tokens defines rules and by references to other rules and logical comb.

CODE: -

#JSGF V1.0

grammar hello

public <greet> = (hy|hello|Power Options|blank |Program|calculator | Browser | Blue | Device Manager |Control | Player |task manager | Windows Security Centre)

public <command> = (Open | Close) (command| word | access |pad | paint |task manager)

public <action> = (start | stop) (excel |photo shop |nero |word pad|fire wall | recognise)

public <net> = (site) (face book | goo gall | mail)

STEP 2: - Building the Java File

Config manager is the main class that collects info regarding the machine and from the configuration file and additionally obtains resources from the machine.

Resources are allocated to recognizer. Recognizer provides the basic functionality of grammar management and the results which are produced if the active grammar matches to what the speaker has said.To provide this function the Recognizer interface extends Engine interface.

About to be mentioned are the list which provides the functionality and the ways in which these functions are specialised and is inherited by the javax.speech package from javax.speech.recognition package.

As the Engine interface is extended by Recognizer all the properties of speech engine are applied to the recognizer. Another class RecognizerModeDesc put together information regarding the dictation abilities of recognizer and people who trained the engine. javax.speech package's central class search, select and create the recognizers. Defined properties, selection and creation procedure helps in defining recognizer.

Basic state system of an engine in inherited by the Recognizer from engine interface which includes four states for allocation , the pause and resume state , the state monitoring methods and the state update events.

EngineListener interface is also extended by the javax.speech.recognition package as this helps the RecognizerListener to get the events specific to recognizers.

STEP 3: - Installing FreeTTS library

FreeTTS is based on Flite and written solely on the java language. Flite is a small run-time engine for speech synthesis which was produced at Carnegie Mellon University. Flite is produced by two universities the first is University of Edinburgh and uses its Festival Speech Synthesis System, the second is Carnegie Mellon University and uses its Fest Vox project.

FreeTTS uses and supports many voices:

US English voice ,an 8 kilo hertz diphone, male

male US English voice , a 16 kilo hertz diphone

male US English voice, a 16 kilo hertz limited domain

FreeTTS helps in Synthesis of speech by first breaking the input text into many sets of grammar and then converting the grammar into audible and clear speech.This is done through a procedure of successive operations on the text. An utterance structure than stores the cumulative results of every operation , also holds the complete analysis of it.

CHAPTER-2
LITERATURE SURVEY

2.1 LITERATURE REVIEW

1. Speech Recognition System: A Review 2015

Nitin Washani

M. Tech Scholar

DIT University

Dehradun (Uttarakhand, India)

Sandeep Sharma, Ph.D.

Head of Department of ECE

DIT University

Dehradun (Uttarakhand, India)

In this paper we are attempting to give a review regarding the Speech Recognition Technology and how this technology has improved in the previous years. There is no doubt regarding that the speech recognition is a much needed technology but that only makes it a difficult and challenging problem to handle.The performance of SR depends on many factors but the most important one and which has significant effect on it is the Signal Preprocessing Stage. It consists of many parts including an EPD, Filtering, Framing, Windowing, Echo Cancellation, etc. and any improvement in individual part can play a significant role in improving the overall performance of the system. More efforts should be put in FrontEnd for effective working for BackEnd. To counter noise one can use MFCC technique for Feature Extraction as it generates the training vectors by transforming speech signal into frequency domain.

2. Sphinx-4: A Flexible Open Source Framework for Speech Recognition 2014

Willie Walker, Paul Lamere, Philip Kwok,

Bhiksha Raj, Rita Singh, Evandro

Gouvea, Peter Wolf, Joe Woelfel

The Sphinx-4 framework is developed carefully and after that different implementations were created for every module in the framework.For example- MFCC, PLP and LPC is supported by the FrontEnd implementations.

Just like FrontEnd the Linguist and Decoder supports different implementations, the Linguist supports language models like CFGs, FSTs, and N-Grams where as Decoder uses a variety of SearchManager implementations such as traditional Viterbi, Bushderby, and parallel searches.With the help of ConfigurationManager we were able to combine different implementations of modules in many ways and that supports our claim foe developing a flexible pluggable framework.The results seems to be fruitful as the performance regarding both speed and accuracy has improved.Various works like the parallel and Bushderby SearchManagers as well as a specialized Linguist that can apply "unigram smear" probabilities to lex trees are proving the Sphinx-4 framework to be "research ready" as it supports them easily.Well this is just the beginning and we expect it to support our future areas also.Now the sphinx-4 is available freely (BSD-style license).This gives permission to do research and develop it without any cost.

3. FreeTTS - A Performance Case Study 2012
   Willie Walker
   Paul Lamere
   Philip Kwok

Considering the native-C counterpart of FreeTTS - Flite we started our research of the performance characteristics of FreeTTS. We were expecting it to be equally fast with the Flite .But we were pleased to see the results as the FreeTTS was running two or three times faster by just simple optimizations and some of the aggressive optimizations which were performed by the Java HotSpot.
And now we are thinking to implement those optimisations in Flite to improve its performance . Clearly as the FreeTTS is running on java platform it has its benefits including the garbage collection and high performance collection utilities.However, these optimisations are time consuming for Flite considering programming point of view.

4. THE CMU SPHINX-4 SPEECH RECOGNITION SYSTEM 2015

Paul Lamere1, Philip Kwok, Evandro Gouvêa, Bhiksha Raj,
Rita Singh2, William Walker, Manfred Warmuth, Peter Wolf

1. Sun Microsystems Laboratories, USA
2. Carnegie Mellon University, USA
3. Mitsubishi Electric Research Labs, USA
4. University of California, Santa Cruz, USA

The main aspect of this paper is that we have described the salient features of Sphinx-4 speech recognition system related to its architecture.We have looked for the advantages of its modular design and the flexibility it provides in using different type of acoustic and language representations as well as also including the advantages that it gets due to its java platform.A new algorithm related to search which is Bush-derby algorithm is used by the Recognizer where as the traditional Viterbi search algorithm is also present.The algorithm is really helpful as it enables the software to add all the paths into the score for the unit.Not only that it allows the use of many sources of information including the audio and visual features or parallel streams. High-likelihood features contributes relatively because of the parameters associated with the Bush-derby algorithm also giving them an elegant way.Some of the Benchmarks shows that Sphinx-4 is faster than Sphinx-3 and is equally or better in recognition.But still not all tests are done and performed but as soon as they are completed the results will be shown to SourceForge.

5. Hands free JAVA (Through Speech Recognition) 2013

Rakesh Patel, Mili Patel
Department of Information Technology
Kirodimal Institute of Technology Raigarh

Our work primarily concerns for making the use of voice recognition easy for the programmers.Working for long hours on keyboard is really difficult to adapt and is always a problem for disabled people or for programmers those are suffering from repetitive strange injuries.

We provide a robust solution for speech recognition that uses java. The solution primarily focuses on finding different words having the same sound and finding the solution for it.-For example lets consider “for” the sound is similar to “four”, “far”.Now what we do is that all the spoken word is written by the software but if that is not the word which is desired by the speaker , the speaker can just say incorrect and the it will be removed. Also freeing the user from remembering the syntax we created a special program constructs for the user sake.We provide the system suitable for anyone whether he is disabled or not, having injuries and that is why our system is not limited to only textual programming but also for visual programming languages.

CHAPTER-3
SYSTEM DEVELOPMENT

3.1 ALGORITHM

1. Get resources information from the config file.
2. Create an object of ConfigurationManager class.
3. Create an object of Recognizer class.
4. Create an object of Microphone class.
5. Allocate resources to object of Recognizer class.
6. IF microphone starts recording then

   Print Say: (Command | Program| Browser | Bluetooth | Device Manager |Power Options |Cal | Control | Player |task manager | Windows Security Center)

   Print Say: (open word | open photoshop|open Access|start Excel|start nero |start fire wall| open Pad |open Paint)

   Else

   Print can’not start microphone

   Deallocate resources.
7. Print What can I do for you Sir

   Call the method recognize of Recognizer class recognizer.recognize()
8. Get best result using result.getBestFinalResultNoFiller()
9. IF resultText.equalsIgnoreCase("hy")||resultText.equalsIgnoreCase("hello") then

   Try String[] greet = {"Hello there"

   ,"Hi","What is it"}

   Speak greet
10. IF resultText.equalsIgnoreCase("command")

    Try Process p

    Speak Opening command prompt please wait

    p = Runtime.getRuntime().exec("cmd /c start cmd")

    resultText = result.getBestFinalResultNoFiller()

    Print ("You said: " + resultText + "\n")
11. IF resultText.equalsIgnoreCase("close command")

    Try Process p

    p = Runtime.getRuntime().exec("cmd /c start taskkill /im cmd.exe /f")

    resultText = result.getBestFinalResultNoFiller()

```
Print ("You said: " + resultText + "\n")
```

```
12.  IF (resultText.equalsIgnoreCase("Power Options"))
                Try Process p
                 Speak ("The Power Options are on the screen sir")
                p = Runtime.getRuntime().exec("cmd /c powercfg.cpl")
                 resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
13. IF resultText.equalsIgnoreCase("Blue")
                Try Process p
                p = Runtime.getRuntime().exec("cmd /c fsquirt")
                resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
14. IF resultText.equalsIgnoreCase("start photo shop")
                Try Process p
                p = Runtime.getRuntime().exec("cmd /c start photoshop")
                resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
15. IF resultText.equalsIgnoreCase("calculator")
                Try Process p
                p = Runtime.getRuntime().exec("cmd /c start calc")
                resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
16. IF resultText.equalsIgnoreCase("Windows Security Center")
                Try Process p
                p = Runtime.getRuntime().exec("cmd /c wscui.cpl")
                 resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
17. IF resultText.equalsIgnoreCase("Player")
                Try Process p
                p = Runtime.getRuntime().exec("cmd /c start wmplayer")
                resultText = result.getBestFinalResultNoFiller()
                System.out.println("You said: " + resultText + "\n")
```

```
18. IF (resultText.equalsIgnoreCase("Program"))
                {
                  Try Process p
                  Speak ("Opening Programmes please wait")
                  p = Runtime.getRuntime().exec("cmd /c start appwiz.cpl")
                  resultText = result.getBestFinalResultNoFiller()
                  System.out.println("You said: " + resultText + "\n")
19. IF resultText.equalsIgnoreCase("Control")
                        Try Process p
                        p = Runtime.getRuntime().exec("cmd /c control")
                        resultText = result.getBestFinalResultNoFiller()
                        Print ("You said: " + resultText + "\n")
20. IF resultText.equalsIgnoreCase("open paint")
                       Try Process p
                       p = Runtime.getRuntime().exec("cmd /c start mspaint")
                      resultText= result.getBestFinalResultNoFiller()
                      Print ("You said: " + resultText + "\n")
21. IF resultText.equalsIgnoreCase("close paint")
                       Try Process p
              p = Runtime.getRuntime().exec("cmd /c start taskkill /im mspaint.exe /f")
               resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
22. IF resultText.equalsIgnoreCase("close calculator")
                       Try Process p
              p = Runtime.getRuntime().exec("cmd /c start taskkill /im calc.exe /f")
                     resultText = result.getBestFinalResultNoFiller()
              Print ("You said: " + resultText + "\n")
23. IF resultText.equalsIgnoreCase("Browser")
                       Try Process p
                       p = Runtime.getRuntime().exec("cmd /c start chrome.exe")
                      resultText = result.getBestFinalResultNoFiller()
                      Print ("You said: " + resultText + "\n")
24. IF resultText.equalsIgnoreCase("close Browser")
                      Try Process p
```

```
p = Runtime.getRuntime().exec("cmd /c start taskkill /im chrome.exe /f")
resultText = result.getBestFinalResultNoFiller()
25. IF resultText.equalsIgnoreCase("open task manager")
Try Process p
p = Runtime.getRuntime().exec("cmd /c start taskmgr.exe")
resultText = result.getBestFinalResultNoFiller()
26. IF resultText.equalsIgnoreCase("Adobe")
Try Process p
p = Runtime.getRuntime().exec("cmd /c start acrord32.exe")
resultText = result.getBestFinalResultNoFiller()
Print ("You said: " + resultText + "\n")
27. IF resultText.equalsIgnoreCase("site face book")
Try Process p
p = Runtime.getRuntime().exec("cmd /c start chrome www.facebook.com")
resultText = result.getBestFinalResultNoFiller()
Print ("You said: " + resultText + "\n")
28. IF resultText.equalsIgnoreCase("site go gall")
Try Process p
p = Runtime.getRuntime().exec("cmd /c start chrome www.google.com")
resultText = result.getBestFinalResultNoFiller()
Print ("You said: " + resultText + "\n")
29. IF resultText.equalsIgnoreCase("site mail")
Try Process p
p=Runtime.getRuntime().exec("cmd /c start chrome https://mail.google.com")
resultText = result.getBestFinalResultNoFiller()
Print ("You said: " + resultText + "\n")
30. IF resultText.equalsIgnoreCase("close task manager")
Try Process p
p = Runtime.getRuntime().exec("cmd /c start taskkill /im taskmgr.exe /f")
resultText = result.getBestFinalResultNoFiller()
Print ("You said: " + resultText + "\n")
31. IF resultText.equalsIgnoreCase("open pad")
Try Process p
p = Runtime.getRuntime().exec("cmd /c start notepad")
```

```
                resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
32. IF resultText.equalsIgnoreCase("close pad")
                Try Process p
        p = Runtime.getRuntime().exec("cmd /c start taskkill /im notepad.exe /f")
                resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
33. IF resultText.equalsIgnoreCase("open word")
                    Try Process p
         p = Runtime.getRuntime().exec("cmd /c start winword")
                resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
                Speaker ("What type of document do you want")
34. IF resultText.equalsIgnoreCase("blank")
                    try
                    {
                        Robot robot = new Robot()
                      // Simulate a key press
                      robot.keyPress(KeyEvent.VK_ENTER)
                      robot.keyRelease(KeyEvent.VK_ENTER)
                    Speak ("Please speak the text to enter in the document")
                    while(true)
                     {
                          Result result1 = recognizer.recognize()
                           if (result1 != null)
                           {
                              String text = result1.getBestFinalResultNoFiller()
                   StringSelection stringSelection = new StringSelection(text)
          Clipboard clipboard = Toolkit.getDefaultToolkit().getSystemClipboard()
                clipboard.setContents(stringSelection, stringSelection)
                 robot.keyPress(KeyEvent.VK_CONTROL)
                 robot.keyPress(KeyEvent.VK_V)
                robot.keyRelease(KeyEvent.VK_V)
                robot.keyRelease(KeyEvent.VK_CONTROL)
```

35. IF resultText.equalsIgnoreCase("close word")

Try Process p

p = Runtime.getRuntime().exec("cmd /c start taskkill /im winword.exe /f")

resultText = result.getBestFinalResultNoFiller()

Print ("You said: " + resultText + "\n")

36. IF resultText.equalsIgnoreCase("start word pad")

Try Process p

p = Runtime.getRuntime().exec("cmd /c write")

resultText = result.getBestFinalResultNoFiller()

Print ("You said: " + resultText + "\n")

37. IF (resultText.equalsIgnoreCase("stop word pad"))

Try Process p

p = Runtime.getRuntime().exec("cmd /c start taskkill /im wordpad.exe /f")

resultText = result.getBestFinalResultNoFiller()

Print ("You said: " + resultText + "\n")

38. IF resultText.equalsIgnoreCase("start Excel")

Try Process p

p = Runtime.getRuntime().exec("cmd /c start excel")

resultText = result.getBestFinalResultNoFiller()

Print ("You said: " + resultText + "\n")

39. IF resultText.equalsIgnoreCase("stop Excel")

Try Process p

p = Runtime.getRuntime().exec("cmd /c start taskkill /im excel.exe /f")

resultText = result.getBestFinalResultNoFiller()

Print ("You said: " + resultText + "\n")

40. IF (resultText.equalsIgnoreCase("start firewall"))

Try Process p

p = Runtime.getRuntime().exec("cmd /c start firewall.cpl")

resultText = result.getBestFinalResultNoFiller()

Print ("You said: " + resultText + "\n")

41. IF resultText.equalsIgnoreCase("close fire wall")

Try Process p

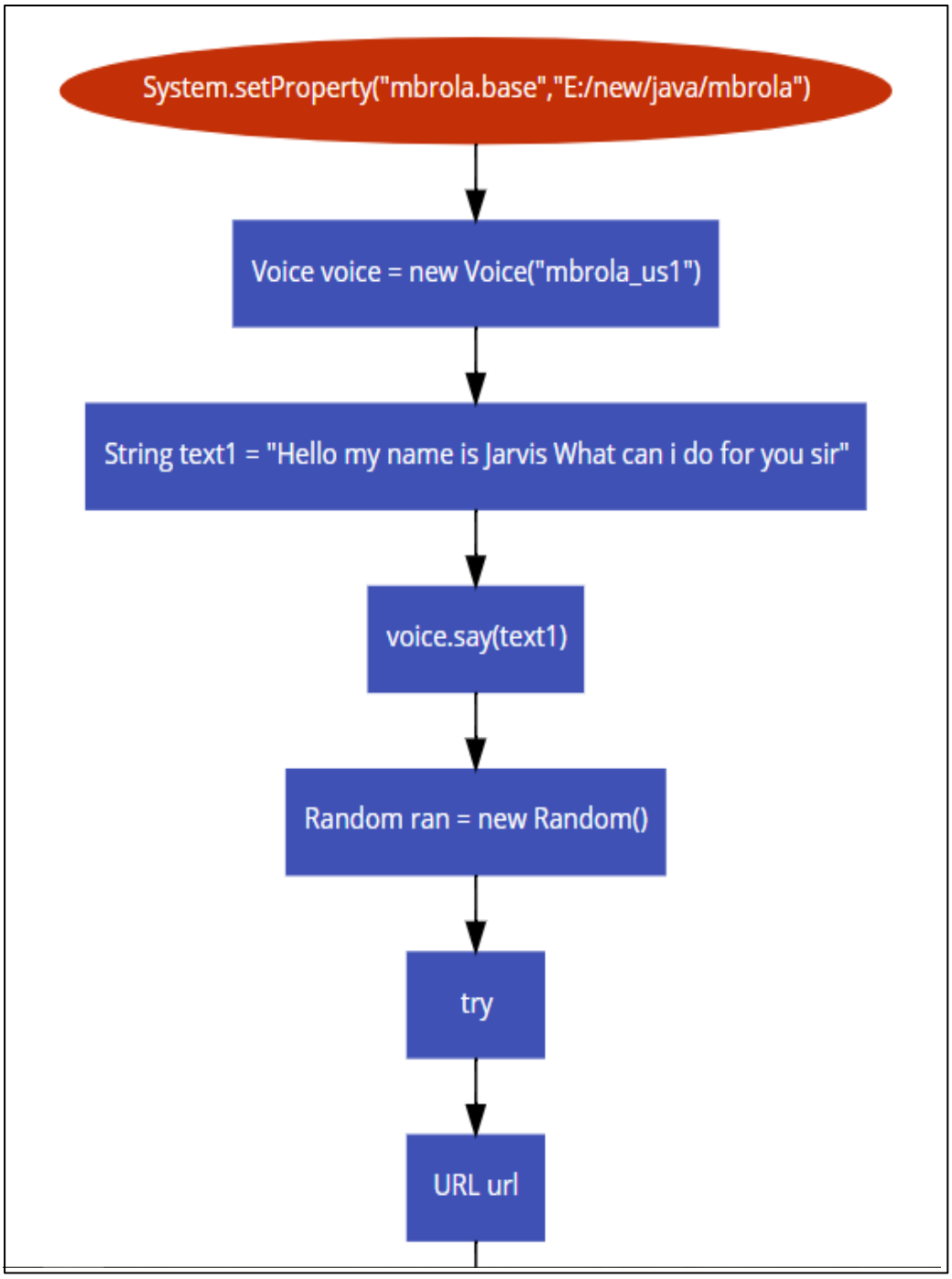
String status = "status eq Windows Firewall"

```
                p = Runtime.getRuntime().exec("cmd /c taskkill /f /fi " +status )
                resultText = result.getBestFinalResultNoFiller()
                System.out.println("You said: " + resultText + "\n")
42. IF resultText.equalsIgnoreCase("start nero")
                 Try Process p
                 p = Runtime.getRuntime().exec("cmd /c start nero")
                resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
43. IF resultText.equalsIgnoreCase("open Access")
                 Try Process p
                 p = Runtime.getRuntime().exec("cmd /c start msaccess")
                 resultText = result.getBestFinalResultNoFiller()
                Print ("You said: " + resultText + "\n")
44. IF resultText.equalsIgnoreCase("close access")
                 Try Process p
  p = Runtime.getRuntime().exec("cmd /c start taskkill /im msaccess.exe /f")
                  resultText = result.getBestFinalResultNoFiller()
                 Print ("You said: " + resultText + "\n")
45. IF resultText.equalsIgnoreCase("speech recognize complete")
                 try
                 System.out.println("Thanks for using !")
                 recognizer.deallocate()
                 System.exit(0)}
46. IF resultText.equalsIgnoreCase("speech recognize start")
                 Try recognizer.notify()
                 System.out.println("Hello again :-) ")
                 System.exit(0)
47. IF resultText.equalsIgnoreCase("stop recognize")
                 try
                 System.out.println("See you later!")
                 System.exit(0)
    Else
        System.out.println("I can't hear what you said.\n")
48. Else
```

```
System.out.println("Cannot start microphone.")
recognizer.deallocate()
System.exit(1)
System.err.println("Problem when loading HelloWorld: " + e)
System.err.println("Problem configuring HelloWorld: " + e)
 e.printStackTrace()
```

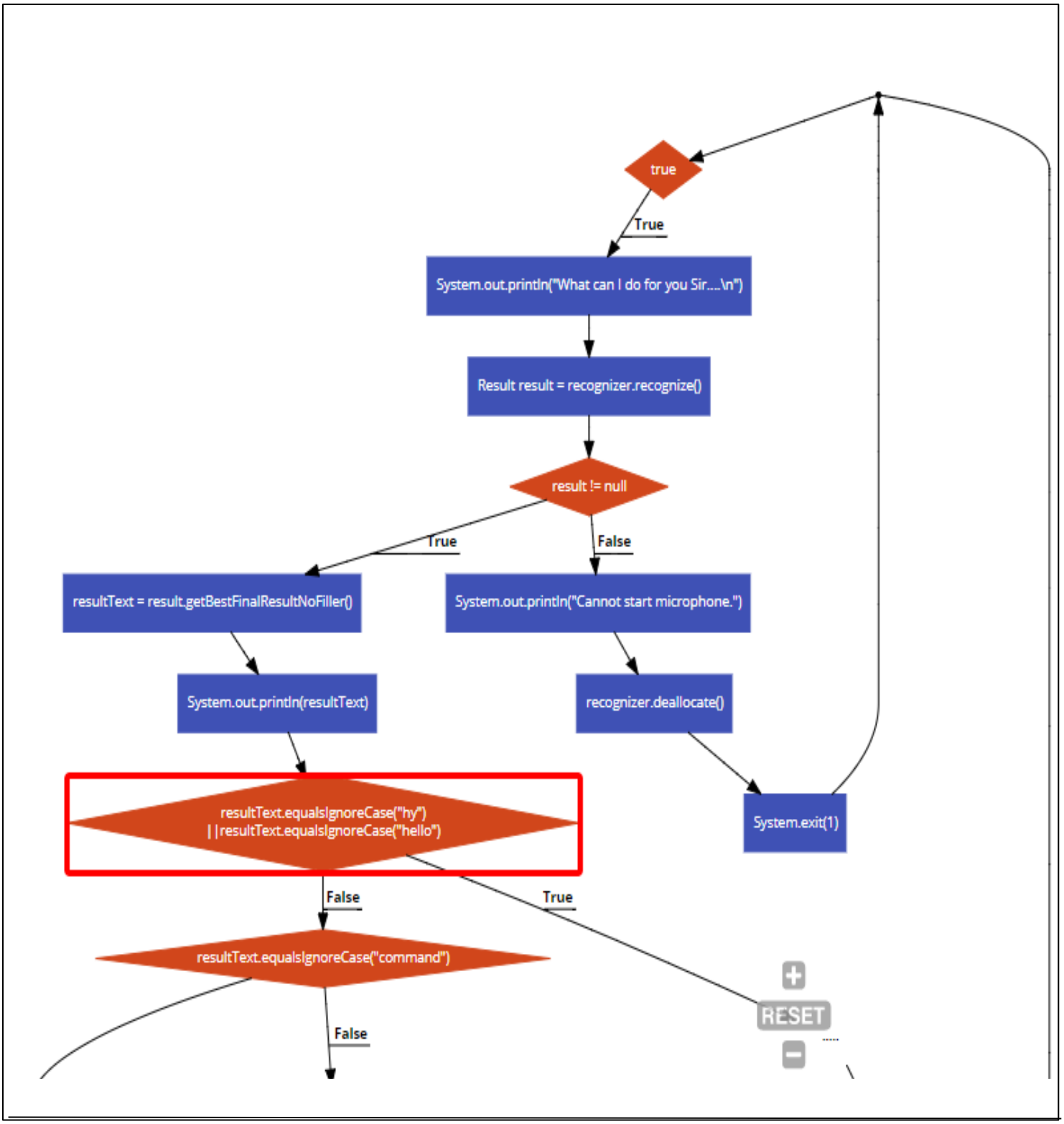
3.2 SYSTEM MODEL

The following system model shows the flow of how the data will be transmitted and processed.

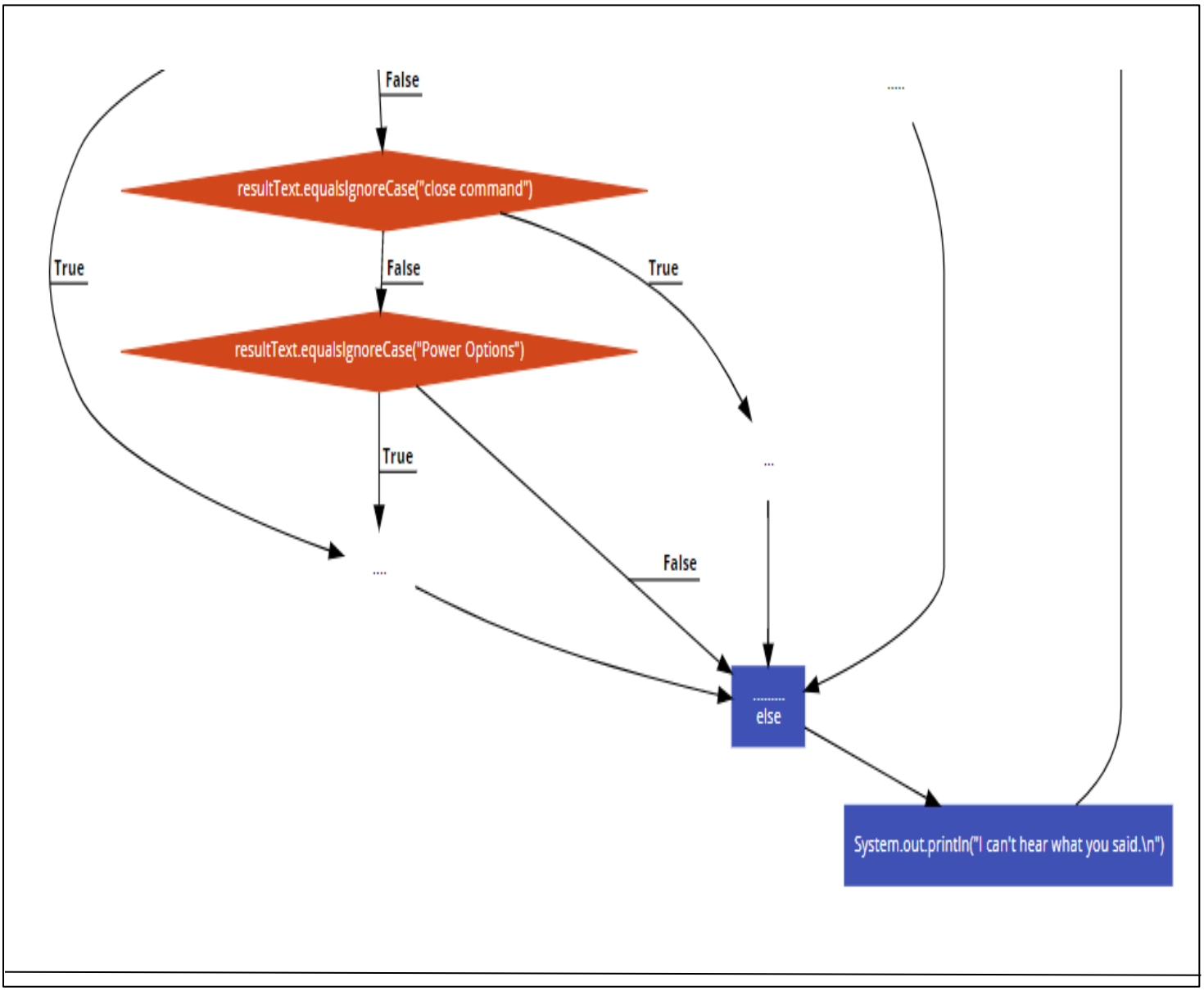


BLOCK 1 FIG 3.1

BLOCK 2 FIG 3.2



BLOCK 3 FIG 3.3



BLOCK 4 FIG 3.4

3.3 ANALYTICAL

Frontend

The function of the FrontEnd is to convert the input signal into the sequence of results or output. As seen in fig. 2, the FrontEnd is made of Data Processors . Data processors is one or more parallel chains of replaceable communicating processing modules.Computation of many different types of parameters is possible in FrontEnd because of multiple chains support.This helps in decoding multiple signals simultaneously using different parameters for e.g. MFCC and PLP, including the parameters derived from video signals.

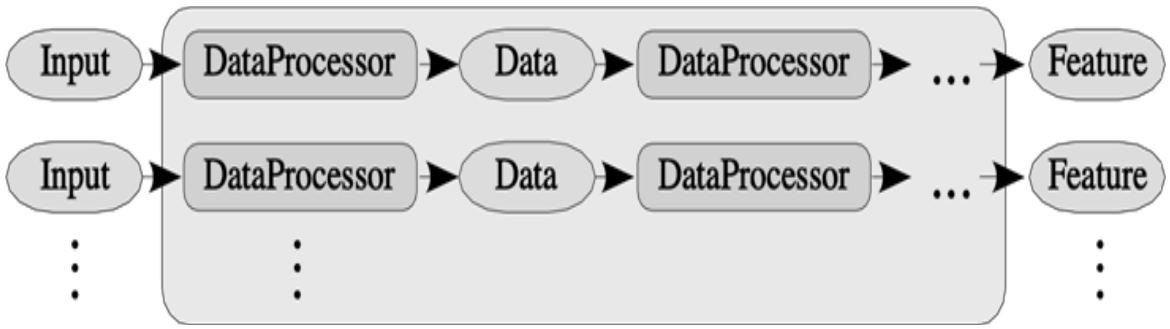


Fig. 3.5. Sphinx-4 FrontEnd.

The frontEnd permits long sequences of chains as every DataProcessor present in the FrontEnd gives input and output and are able to connect to other DataProcessor.End-point detection are the data classification events which are indicated by the markers. DataProcessor input and output are made up of data objects that encapsulate the input data to be processed and the markers.Decoder uses the Features which are basically the last Data object that are composed by the endmost DataProcessor, these Data object are made up of parameterised signals.

Sphinx-4 also has the ability just like AVCSR system to produce parallel sequence of features.But as the sphinx-4 is able to randomly allows many parallel streams it has become unique. The process uses a pull design to communicate between different blocks.In this design the the request is only made by the DataProcessor from the earlier module when there is need which is contrary towards its counterpart of push design which follows conventional method in which the output os send to the succeeding module when the output is generated. The pull design has its own benefits as it helps in buffering which is to look back and forward

in time.This ability helps the decoder to perform different searches including the frame-synchronous Viterbi searches, depth search and A*.common signal techniques is utilised by the Sphinx-4 within the FrontEnd framework.

The implementations has benefits and are main support for some which includes the following: PLP which represents perceptual linear prediction , MFCC represents mel-cepstra frequency coefficient extraction, CMN represents cepstral mean normalisation, LPC represents linear predictive encoding, DCT representing discrete cosine transform, mel frequency filtering, bark frequency warping , Fourier transform, cosine transform via windowing e.g. hamming windows, pre emphasis, reading from a variety of input formats for batch mode operation, reading from the system audio input device for live mode operation.

With the help of ConfigurationManager the user has liability to chain Sphinx-4 Data Processor in any way or manner but also able to implement and incorporate the data processor design.The sphinx-4 is not only limited to low-level structure but its simplicity and pluggable nature applies to high level also for eg the FrontEnd is a pluggable module but also consists of pluggable module itself.

Linguist

Just like all other Sphinx-4, the Linguist is also a pluggable module which permits the module to dynamically configure the software with many other implementations.The linguist has two key functions the first is to produce the SearchGraph that indeed is utilised by the Decoder while doing the search and the other one is to hide the complexities produced during the generation of this graph.

A particular LanguageModel and topological structure of AcousticModel(HMMs for the basic sound units used by the system) represents the language structure which helps the Linguist to generate the SearchGraph.With the help of Dictionary the Linguist maps the words produce in LanguageModel into the sequences of AcousticModel.If provided the Linguist can include Sub-words units with context of random lengths.

Linguist permits different implementations to plugged at runtime and Sphinx-4 allows users to different configurations for recognition requirements and different system.A simple Linguist might be used by a simple numeric digit's recognition application and also keeping the search space entirely in memory.

On contrary if we look at a dictation application which is using a 100k word vocabulary the that might keep entirely a small portion of potential search area inside the memory bur only at a time using a sophisticated Linguist.The Language Model, the dictionary, and also the Acoustic Model are the three components of Linguist itself and are explained with in the following sections.

LanguageModel

The language structure can be represented by a range of pluggable implementations and that world-level language structure is provided by the Language Model of the Linguist.There are two primary categories in which these implementations come under: stochastic N-Gram models and graph driven grammars.In graph driven model the nodes represents single word and arc is used to represent the the probability of a word transition taking place, it is directed word graph.Where as stochastic N-Gram models uses a different approach by utilising the n-1 words to find the probability of the next word.

Aforementioned are some of the implementations that is supported Sphinx-4 in a variety of formats:

• **SimpleWordListGrammar:** it helps to define two things firstly the grammar which is solely based upon listing of words and other is to determine whether the grammar is in loop or not.The grammar are used for isolated word recognition if there is no loop. But if the grammar loops it will provide unigram grammar with same possibilities that is utilised to support trivial connected word recognition.

- **JSGFGrammar**: it uses the JavaTMSpeech API Grammar Format (JSGF) which helps to define like vendor-independent Unicode representation of grammars, platform independent ,a BNF-style.
- **LMGrammar**: This grammar is well utilised by smaller unigram and bigram grammar which does not exceed 1000 words .It generates one grammar node per word and is based on a statistical language model.

• **FSTGrammar**: supports a finite-state transducer (FST) in the ARPA FST grammar format.

• **SimpleNGramModel**: It is efficient and better perform with small language models because it does not try to optimise memory usage , it supports in APRA format for ASCII N-Gram models.

• **LargeTrigramModel:** It works well with large files which can even exceed 100 MB.This is due to its property of optimisation of memory storage.It can also provide support for the CMU-Cambridge Statistical Language Modelling Toolkit generated N-Gram models.

Dictionary

The LanguageModel uses dictionary to find the pronunciations for words which are found within it.The words are broken into sub words with in the AcousticModel.Pronunciations help in breaking of the word.Dictionary also allows a word to go in many categories and then conjointly classify the words.

The CMU is supported by the implementations done by the dictionary, the Sphinx-4 provides it.The size of the active vocabulary is supported by the implementations which are also optimised for the usage patterns.lets consider this example where one implementation can load the complete vocabulary at system initialisation time, whereas another implementation can solely acquire pronunciations on demand.

AcousticModel

The function of AcousticModel is to map the unit of speech provided by the FrontEnd and also HMM will be scored against the incoming speech. The word position and discourse are also taken into considerations by the mapping just like the alternative systems.

Lets take an example - considering the tri-phones the position of the word is given at three places the beginning , middle and the end of the word which represents the position of Triphone because the the context only represents the grammar or phonemes to either left or right of the particular phonemes.Sphinx-4 does not mount the definition and allows the AcousticModels to still be the AcousticModels even after containing allophones and no change in the definition is there.

So first the Linguist brakes the words into sub words which are in sequence and afterwards send it to the AcousticModel in which the HMM Graph is retrieved From the sub-words. Then the LanguageModel works in conjugation with the HMM and produce a SearchGraph.

The Sphinx-4 HMM is a directed graph unlike other speech recognition system who uses set of structures in memory has offered entirely different topologies after the implementation of AcousticModel.In the graph there are nodes and arcs , the nodes represents the state of HMM and the arc represents the transition of states which is likely to happen from one to another in the HMM.

The AcousticModel allows the HMM to have transitions states and the amount out of any state , it even allows to move flexibly that is it can move both forward and backward . The states can vary accordingly from one unit to another in the same AcousticModel.

Every state has the ability to generate the score which is determined by a particular feature. The HMM state itself completes the code required for the value of the scores and that is why the implementations can be concealed from the remaining system and this allows the HMM state to use different probability functions per state.The elements are also shared among the HMM states at each and every level for eg if own element is conjured a state like transition matrices, gaussian mixtures then the mixture weights are shared among the states to a fine degree.

Sphinx-4 has one AcousticModel which has the capability to load and use the models which are generated by the Sphinx-3 trainer .People use the implementations according to their own needs by applying different implementations they will put together the Sphinx-4 as their wants.

3.4 COMPUTATIONAL

Search Graph

Search Graph is the main data structure that is used to decode.As the Linguist is implemented in different ways and all the topologies of the search spaces generated vary greatly , all the reach spaces are called the SearchGraph. As shown by the Figure-3.6

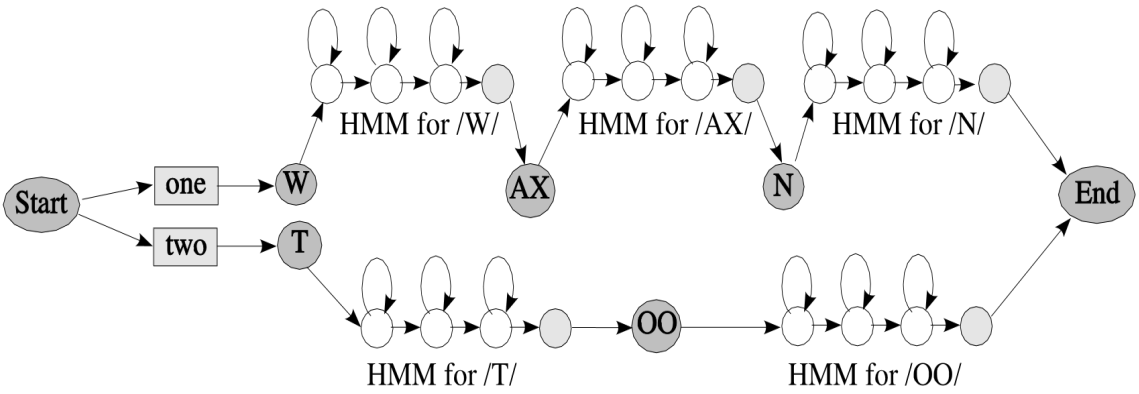


Fig. 3.6. Example Search Graph**.**

Search Graph is composed of two components the search state and search state arcs and all the search state has transition possibilities.Search Graph is a directed graph.The states are composed of three , The LanguageModel which is represented by words in rectangle,Dictionary represented by the sub-words units in dark circles and AcousticModels.All nodes are in two states either emitting or non emitting , the nodes are called the SearchState. The arcs represent the possibility of state transitions and every node has the likelihood of going from one state to another. The Emitting states can be scored against incoming acoustic features while non-emitting states are generally used to represent higher-level linguistic constructs such as words and phonemes that are not directly scored against the incoming features.

All the assumptions and the hard constraints that were put on the Search Graph because of the previous systems were solved because of the Search Graph to permit a wide range of choices to implement . Particularly the Linguist has imposed several restrictions , following are the places where its restrictions don't apply:

- Overall search space topology
- Phonetic context size

- Type of grammar (stochastic or rule based)
- N-Gram language model depth

One of the main aspects of Search Graph is that it has several ways to implement Search State and because of that implementation of them not necessarily be fixed.Every Search State can have different implementation and also the implementation of them can vary even though the Linguist gives a concrete implementation.For e.g. there can be a simple one-to-one mapping of the in-memory search state but if a Linguist represents a large and complex vocabulary but gives a compact representation of the search graph then there will be successors produced by dynamically increasing the compact representation of Search state on demand.

There are other factors also which are affected by the way Search Graph are created like speed, memory footprint, recognition accuracy.But however the Sphinx-4 modularised design permits for the compilation of search state strategy without interrupting or changing other aspects of system.HMM can be produced from the system and depends on the Language model, Vocabulary size and desired memory footprint.

3.5 EXPERIMENTAL

Implementations

Just like the FrontEnd the Sphinx-4 has provided with many implementations to support several different tasks of the linguist.

Recognition tasks are perfectly suited for the Flat Linguist that uses finite-state grammars (FSG), context-free grammars (CFG),small N-Gram language models and finite-state transducers (FST).External grammar formats are converted into the internal formats by the Flat Linguist.Basically the grammar is a word graph in which the nodes are the single word and the arc represents the possibility for the transition , it is a directed graph.The SearchGraph is stored in memory after the Flat Linguist produce the Search Graph from the internal Grammar graph.Still after being this fast it still has problem dealing with the high branching grammar.

The Dynamic Flat Linguist is similar to some extent with the Flat Linguist and that is why is able to perform the tasks by the Flat Linguist but the difference is that it is able to produce the Search Graph on demand and that is why it is able to handle complex Grammar with ease But it also has a drawback because this causes a decrease in the run time performance.

To handle the large N-Gram language models which uses Vocabulary Recognition of large sizes we have another implementation The Lex Tree Linguist.N-Gram has a random order and try N-Gram decoding is supported by the Lex Tree Linguist. The Lex Tree Linguist uses a compact method to define the large vocabulary by organising the words in a Lex Tree.These trees dynamically produce the search trees and this enables it to handle much larger vocabulary and by only using little memory.This implementation supports the Binary Language Models and ASCII. The binary language model is produced by CMU.

Decoder

The main role of the Sphinx-4 Decoder is to create the results or the output by making use of both the FrontEnd and SearchGraph to work in conjugation.The Decoder simplifies the decrypting process as it contains the pluggable SearchManager and other supporting code.The component that makes the decoder interesting and is the main component is the SearchManager.The SearchManager receives the command from the Decoder to recognise a set of feature frames.The SearchManager works at each step and then creates a result object which has all the paths that has reached the non-emitting state.Sphinx-4 also has other capabilities which also produce a lattice and score related to the result which also helps in better performance of the results.But the results objects can be modified between each steps by many applications which allows it to become partner in a recognition process.Just as the Linguist SearchManager has many implementations some of the algorithms are A*,frame-synchronous Viterbi, bi-directional and many more.

Token passing algorithm is the algorithm which is used by all the implementations of the SearchManager.The algorithm was described by Young.A Sphinx-4 token is an object which is related to the search state and has information like language and overall acoustic scores of the path at a given point, a reference to an input Feature frame, a reference to the SearchState and other important information. The SearchState reference allows the SearchManager to relate a token to its state output distribution, context-dependent phonetic unit, pronunciation, word, and grammar state. Every partial hypothesis terminates in an active token.

At every step an ActiveList is formed by the SearchManager which is consist of art of tokens, but there is no necessary requirement of it.As the technique is common,To support the SearchManagers having ActiveList the Sphinx-4 provides the Sub-framework having a scorer and a pruner and with the help of Pruner it generates an ActiveList from the active tokens in the search trellis with the use of pluggable pruner.There are two types of pruning performed by Pruner relative and absolute beam pruning and applications can configure the Sphinx-4 for their implementations and this implementation has become easy because of the java platform which provides the garbage collection.By removing the terminal token of any path from the ActiveList the pruner can prune easily with the garbage collection.The act of removing the terminal token identifies the token and any unshared tokens for that path as unused, allowing the garbage collector to reclaim the associated memory.

The sub-framework of the SearchManager forms a conjoint connection with the Scorer, the value of the state output density is provided by the module-pluggable state probability estimation and the values are given in demand.First the SearchManager request the score for a particular state at a particular time and then after receiving the feature vector is accessed by the scorer for that given time it starts performing mathematical operations to compute the score.All the data related to state densities is retained by the SearchManager and that is why it does not understand the difference, how the grading is completed, the grading can be completed with continuous, semi-continuous or separate HMMs.Each HMM state is isolated.The scorer provides the feature by which any algorithm which is used to increase the scoring procedure can also be done inside the scorer.The scorer can also be benefited with multiple CPU's if provided.

The Sphinx-4 implementation currently provides pluggable SearchManagers implementations and support parallel decoding, Bushderby and frame synchronous Viterbi:

•**SimpleBreadthFirstSearchManager**: on each frame a pluggable pruner is called and performs the frame synchronous Viterbi search.The results are produced by the search manager which have active paths at the processed last frame .Relative and absolute beams are all managed by the default Pruner.

•**WordPruningBreadthSearchManager**: on each frame a pluggable pruner is called and performs the frame synchronous Viterbi search.WordPruningBreadthSearchManager does not manage only one ActiveList but instead manages set of active lists, Linguist defines the state type for each.The linguist defines the state types in sequence order where as pruning is executed in decomposition.

•**BushderbySearchManager**: As opposed to likelihood it performs the classifications on the basis of free energy and uses the Bushderby algorithm to perform generalised frame-synchronous breadth-first search.

•**ParallelSearchManager**: Unlike AVCSR which uses a coupled HMM search approach, it uses a factored language HMM approach and executes the frame synchronous Viterbi search on multiple feature streams. The basic advantage of the search is that it is far compact and much faster than a full search done over a compound HMM.

3.6 MATHEMATICAL

Discussion

TEST		WER		RT	
	SPHINX 3.3	SPHINX 4	SPHINX 3.3	SPHINX 4 (1 CPU)	SPHINX 4 (2 CPU)
TI46 (11 WORDS)	1.217	0.168	0.14	0.03	0.02
TIDIGITS (11 WORDS)	0.661	0.549	0.16	0.07	0.05
AN4 (79 WORDS)	1.300	1.192	0.38	0.25	0.20
RM1 (1000 WORDS)	2.746	2.739	0.50	0.50	0.40
WSJ5K (5000 WORDS)	7.323	7.174	1.36	1.22	0.96
HUB-4 (64000 WORDS)	18.845	18.878	3.06	4.40	3.80

Table 3.1. Sphinx-4 performance.

Two measurement for comparing the performance of system are WER and RT. Word Error Rate is represented by the WER and is measured in percent. Real time represented by RT is described as the rain between vocalisation duration and the decipher time of the vocalisation. For better performance both must have lower values.The above table's information is measured between different Sphinx on dual hardware 1015 Mega hertz R 3 with 2GB RAM.

The framework for Sphinx-4 is much better and the tasks which were first considered to be hard has now become to easy , for example the implementations like Bushderby SearchManager implementations and the parallel implementations both are created in short amount of time and even after that did not need modification.The basic ability of Sphinx-4 has helped it to use different implementations which varies from general to specific applications regarding formula. Considering the example in which we add or plug a new linguist while the rest of the system remains same, we were able to improve the run time speed for the regression RMI

test by two orders of magnitude.Even many good tasks are able to get support because of modularity of the system for e.g. SearchManager is able to implement efficiently vary vocabulary ranging from little vocabulary tasks such as TIDIGITS and TI46 to large vocabulary tasks such as HUB-4.Considering another example related to Linguist which allows the Sphinx-4 to support different tasks such as applications that use random language models to traditional CFG-based command-and-control applications. Java Platform utilised by the Sphinx-4 has enabled its standard nature.the java platform helps the system to load at runtime which provides an easy support for pluggable framework, some of the advantages of java programming languages are:

• Sphinx-4 will is platform dependent and can run on any platform.

• The coding time is reduced by its rich platform.

• The decoding tasks are distributed into different thread because of the built -in property of multithreading

• Instead of memory leaks developer can focus on the development because of garbage collection

Still after having so many advantages there is the drawback of memory footprint for the java platform.To optimise the memory output some speech engines access the platform directly which in turn helps in optimisation and this direct access is not allowed in the java programming language. people mostly have a doubt related to Java programming language that it is very slow. After the development of Sphinx-4 we have the basic tendency to carefully observe the results compared to its predecessor. That is why we compare both Sphinx-3.3 and Sphinx-4 on basis of RT and WER and table 3.1 shows the performance of both of them.

The results were merely to show the and demonstrate the strength of Sphinx-4's pluggable and modular design as compared to its predecessor.Relatively Sphinx-3.3 was designed for higher and complicated vocabulary with N-Gram language model.That is why it was not able to perform well for the easier tasks like TIDIGITS and TI46 but the property of Sphinx-4 having different implementations for Linguist and SearchManager helps in increasing its performance and was able to perform better.For e.g. look at the table showing the values for WER and RT performance for the easier task of TI46.

Another important and interesting aspect that came in light to us that when considering the RT performance the raw computing speed is not the major criteria. We used the scorer which

divided the task equally among all the accessible CPUs, the rise in the speed was expected but not that much what we aspected , there was no dramatic increase and this helps us to verify that at the actual scoring of the acoustic model states only 30% CPU time is spent and the remaining time is spent at other activities like Pruning and Growing. Garbage collection uses around 2-3% of CPU usage.

3.7 STATISTICAL

Data Set Loading Baseline

Data Set	Bytes	Load Time (seconds)
All Carts	36,496	0.41
Lexicon	1,705,833	6.25
Letter to sound rules	261,318	0.65
Unit Database	7,828,514	13.76
Total	9,832,161	21.07

Table 3.2. Initial Data Set Load Times

Flite is counterpart for FreeTTS as it is based on C.All dat for Flite is static constants and are directly added to the source code .Due to which the data and the code is loaded simultaneously because of which it has zero effective load time. The Flite does not load the datasets from file where as the FreeTTS load it from the data sets that is why we are calculating for how much time will it take to load the data. Table 3.2 shows how much time it took to load a large data sets .

The experiment simply proves that it takes a significant amount of time to load data from lexicon and unit database.

Execution Speed Baseline

After seeing the results our main concern shifted towards the FreeTTS , to make it run faster otherwise improving the performance of others is worthless.But before working on improving the execution speed we first compared it with the Flite to form a baseline.To form the baseline we used a large data sets by inputing a large data set consisting of Alice In The Wonderland, the text contains 4293 words and the time taken was approximately 25 minutes

to speak.Table 3.3 shows the timings when they were ran on SPARC® processor (v9) operating at 296 MHz with 128MB.

Processor	Flite (secs)	FreeTTS (secs)	FreeTTS:Flite (ratio)
Tokenization	0.216	0.393	1.819
Normalization	0.260	0.801	3.081
Phrasing	0.537	0.371	0.691
Segmentation	2.592	1.349	0.520
Pause Identification	0.060	0.083	1.383
Intonation	7.704	10.401	1.350
Post Lexical Analysis	0.838	0.777	0.927
Duration	7.749	6.860	1.350
F0 Contour	10.350	11.119	1.074
Unit Selection	1.058	1.576	1.490
Wave Synthesis	11.206	*47.399*	4.230
Total	**42.570**	**81.129**	**1.906**

Table 3.3. Initial FreeTTS and Flite's processing metrics

The experiment shows us that FreeTTS is approximately two times slower than its counterpart and that our first version for FreeTTS took quite some time for the execution of the data.

Data Set Loading Improvements

After realising the results it was unacceptable that FreeTTS took so much time to synthesise the data.Still considering as penalty that occurred one time we will keep on developing the FreeTTS and will frequently improve the development phase.The development will be this much effective that this will ultimately affect the overall turnaround time.

For the first attempt we took the Flite approach by making the data static and was added directly in the code but we came across an interesting fact and result which shows the limit of pools per class, there can be only 64k pools per class.Moreover the code is restricted too the limit fir it was also 64k bytes.So to overcome this disability we created many classes and then break the data and add in it.But the classes number was much larger, this was also caused because of java programming language as in java when the array is defined the compiler also generate a code to initialise members of array. lets look at the code:

```
private static float[] floatArry =

{
```

1.0f, 2.0f, 3.0f, 4.0f, 5.0f

}

Results in code that looks like this:

newarray float # allocate the array dup

iconst_0 # store the first value fconst_0 fastore dup

iconst_1 # store the second value fconst_1 fastore etc.

Comparing it with its C counterpart in space allocation the FreeTTS is taking three times the space for the initialisation of array, Each element is consuming four bytes to store the value , four bytes for itself and and another four bytes for the index of array.This has created a problem as now the time required to load the class files were more than when it was reading from the data from the file.THe problem was not only because it has three times more data to read but also because a large amount of dates checked by the byte code verifier.The experiments convinceed us to look for the alternative method.

Lazy Tokenization

After completing the previous experiment we start looking for other important causes for the delay and found that the much of the delay in loading the data was because of parsing the ASCII data sets.Now to overcome this we try to delay the parsing of data as much as possible and only doing it if necessary.

Lets look at the entry of the Lexicon Data Sets.

abdicating0 ae1 b d ih k ey1 t ih ng

Each word consists of two parts the first one is the speech and other one is phonemes list.Initially every single word was parsed completely and then the list of phoneme is stored in a list of array and speech part and combined word becomes a key for the Hash Map.So to improve the time the phonemes list which was initially read first was stopped until necessary .Now the list was only read when it is necessary and was saved in one string.Now the results were good and can be seen in table 3.4

	Times in seconds	
Method	Load Time (secs)	Lookup Time (secs)
Tokenize on Load	6.3	1.1
Tokenize on Lookup	2.7	1.1

Table 3.4. Comparison of Tokenization Strategies

This process is called the lazy tokenisation and because of that we were able to save the decrease the load time by half .The table shows the total time spent to find the word Alice. Another benefit was that there is no degradation in the lookup time.

Binary Data

The next Approach we used to reduce the load time was the conversion of ASCII files in their binary representation.Initially it was inefficient to load the data files into ASCII files because of two points ,first it takes a lot more space and it also took a lot of time to convert the ASII into desired format such as the Diphone which is made up of numerical data.

Now only to convert the strings into a particular format it first parse all the strings and there are more than 1.8 million strings in the ASCII format of Diphone data.

Now to overcome this we thought to parse or read the final format and removing the burden of converting it into primitive format first.We knew that Diphone has the largest database so our main concern was that and we put all our efforts on it.we were able to succeed and this gives us the best results so far later on we even tried some new packages for loading the binary data.

File Format	Load Time
ASCII	13.760s
Binary	1.491s
New IO Binary	1.083s

Table 3.5. ASCII vs Binary Load Times

The success of this experiment led us to revisit the lexicon and tried it over there but there was not that much increase in performance, the results were almost similar.We found that in lexicon the java virtual machine was more focused on hash map and in the creation of strings.This was causing the hinderance in binary format of lexicon and that is why we chose not to use binary data in lexicon.

Loading Method	Load Time
Raw Classes and binary files	3.4s
Uncompressed JAR Files	5.1s
Compressed JAR files	7.6s

Table 5 - Comparing Different Methods of Loading FreeTTS

JAR Files

The final step we chose was to put the binary data into JAR files to increase the performance and the results were promising and is shown in the above table.

Data Set Loading Summary

There were other methods also to improve the load time ,foe e.g in lexicon we can use other mapping function than the hash map but the results from binary conversion was really promising we were able to reduce the load time from 21 to 4 seconds.That is why we chose to reduce the execution time.

PERFORMANCE ANALYSIS

Execution Time Improvements

Initially to compute the word Alice it took more that 45 seconds which was really a lot if we compare it with Flite which was able to complete this easy task in just 12 seconds.There are two possibilities where the optimisation can be done:

1. to perform buffer copies more time is being spent.
2. Inner loop calculation time was more.

Eliminating Buffer Copies

Due to an architectural decision, the audio output classes expect data in the form of byte arrays, but the audio wave synthesiser generates data as short arrays. In our first implementation, we wrote the audio wave synthesiser data to an in-memory

ByteArrayOutputStream wrapped in a DataOutputStream, and then finally normalised the data in yet a third byte buffer. By modifying the wave synthesiser to generate byte data directly (a simple modification), we eliminated two buffer copies. This reduced the wave synthesis time for "Alice" by 13 seconds, a significant improvement.

Optimizing the Inner Loops

LPC linear predictive algorithm is used to produce audio sample which is the final stage for the synthesis process.To produce an output sample LPC creates 22 floating-point operations per frame , LPC filters the last 10 sample outputs with a set of filter coefficients associated with the sample frame. Because of which to generate the output for Alice the test cases consists of over 220 million floating point.

In java programming language it is more expensive to access the array that its C counterpart.This is due to its property of checking all the array at the runtime.Conducting experiments we were able to find that the excess time spent in in inner loop was because of the array indexing . Array indexing was taking a significant amount of time.

So to solve this we used data structure linked list instead of array to maintain the output buffer and this solves all the array indexing problems inside the loop. we were able to save the time by 12.5 seconds which was really good as it has reduced time from 34 to 21.5 seconds.

Even the compilers for optimisation like JAVA HotSpotTM also works well with the linked lists.

After optimising other factors we were able to reduce time by 1.1 seconds but this show us that we need to improve other areas .

Utterance Structure Modifications

By conducting experiments we were able to find yet another area of development which is the Utterance structure.we discovered that a lot of time was used to traverse the utterance structure. They use a query text for referring the results of previous utterance structure.They usually refers to the item ,before or after a given item. The queries are mostly rational.

All the utterance items are stored by java.util.List, and that is why all the search for the items are linear searches.We decide to use link list in this as well and were able to save time by 4 seconds.

Algorithm Improvement

Till this point we were mainly focusing on improving the code , to make it run fast , to improve the performance but we still followed the Flite algorithms.But even still we were able to bring the most out of it just by using the necessary algorithms related to Flite even for High performance data structures such as hash map.All the improvements were successful and was able to match the processing time of Flite.

We initially tried to improve the performance following a non algorithmic procedure and can bring more out of it but we decided to use algorithmic changes that will make a significant improvement.

By analysing the data we took our attention towards the intonation processor.We were able to find that a query text was used throughout FreeTTS and every time it was parsed on every use, we were able to find this by simple process of elimination.So to overcome this we preprocessed the data first and then send the output directly into subsequent queries.The results came out to be drastic by simply this we were able to save almost 12.5 seconds of total time .Moreover the results were almost similar if we preprocessed the query or the query is processed at load time.By improving this we were able to improve the performance of other processors as well such as utterance as the same query was used many time by it.

Java programming language grants the permission to make changes to the algorithm faster , easier and much better.we tried doing the same changes to its counterpart but were stopped because of the hindrances caused by not being a Java programming language.

A New Java 2 Platform Release

We were trying to improve the performance by all means now a new version of java came Java 2 Platform, standard Edition (J2SETM) version 1.4.we tried our FreeTTS to check the performance improvement and were shocked to find that its performance just increased simply by the up gradation of the version.The time to synthesise the word Alice now dropped from 21 seconds to 14 seconds while the total time comes down to 24 seconds from 34 seconds.

One notable improvement within the J2SE 1.4 upgrade was improved vary check elimination. The Java HotSpot compiler will notice certain array access idioms, notably for loop accesses, and eliminate the vary checking on the array index if it determines the index continuously falls within range of the array. some of the aforementioned optimizations that we performed on FreeTTS to eliminate array accesses to avoid index vary checking became unnecessary because of these Java HotSpot compiler enhancements.

Improving Performance with the Java Hotspot Virtual Machine

The Java HotSpot virtual machine is Sun Microsystems' virtual machine for the Java platform. The J2SE 1.4 release provides two flavours of the Java HotSpot virtual machine: a consumer compiler that provides for quicker program start times, and a server compiler that maximises program speed however with a extended program start time and a bigger memory footprint. The server compiler performs a large range of optimisations including aggressive inlining of virtual ways, loop unrolling, dead code elimination, common sub-expression elimination, and array range check elimination.

Speech applications are usually constructed as client/server applications with the recognition and synthesis engines running as separate servers presumably on separate machines. This architecture can improve the measurability, flexibility and reliability of a system. With this in mind, we developed a client/server version of FreeTTS that permits the synthesis engine to receive synthesis requests via a socket connection, synthesise the wave data and return it to the client via the socket. when used in contexts like this where startup time is less necessary

than overall TTS performance, FreeTTS can be run using the server compiler with a major performance boost.

1-CPU 296 MHz SPARC® processor (v9) w/ 128Mb			
	Load Time	Run Time	Total
Flite	0s	44.3s	44.3s
FreeTTS -client	3.4s	24.4s	27.8s
FreeTTS -server	5.9s	32.7s-41.7s	38.6s-47.6s

Table 3.7. shows a comparison of Flite and FreeTTS running with the client and the server compiler.

We were perplexed by these results. first of all, the timings with the server compiler were inconsistent, starting from 38 to 47 seconds. Secondly, the performance of the server compiler was worse than the client compiler. Some investigation showed that the Java HotSpot server compiler wants a longer amount of time to spot and compile the hot spots. we ran the test once more replacing the “Alice” input text with the text of Jules Verne's Journey to the center of the planet, that is about twenty times as long. Table 7 shows the results of this check.

Giving the server compiler a extended amount of time to optimize the hot spots proved beneficial: FreeTTS using the server compiler is about 30 minutes quicker than running it using the client compiler.

1-CPU 296 MHz SPARC® processor (v9) w/ 128Mb			
	Load Time	Run Time	Total
Flite	0s	44.3s	44.3s
FreeTTS -client	3.4s	24.4s	27.8s
FreeTTS -server	5.9s	32.7s-41.7s	38.6s-47.6s

Table 3.8. Comparison of Client and Server Compiler

Multiple CPU Improvements

The U.S. created the FreeTTS as multithread application because of its predecessor Flite which was a single thread application and was not able to utilise the multiCPU system. But FreeTTS being a multithreaded system was overcome this disability as it creates different threads for different process and then send these threads among different CPUs to obtain performance boost .

Table 3.9 shows the results of process the “Journey” text on a 2-CPU 360 MHz SPARC® processor (v9) with 512 Mb of memory.

2-CPU 360 MHz SPARC® processor (v9) w/512 Mb

2-CPU 360 MHz SPARC® processor (v9) w/512 Mb			
	Load Time	Run Time	Total
Flite 2-CPU using 1 CPU	0s	803.2s	803.2s
Flite 2-CPU using 2 CPUs	0s	800.4s	800.4s
FreeTTS -server 2-CPU using 1 CPU	5.2s	282.9s	288.1s
FreeTTS -server 2-CPU using 2 CPUs	3.1s	193.1s	196.2s

Table 3.9. Single vs Multiple CPU Performance Comparison

This table shows that, as expected, FreeTTS shows a 33% improvement in runtime when running on a 2-CPU system. Whereas Flite achieves nearly identical run-times when running on a single or multi-CPU system.

Time-to-First-Sample Performance Tuning

The most important Benchmark in FreeTTS synthesis is the time-to-first-Sample. It is the time the synthesiser gets the text and then synthesise and produce the first audio sample.

This is really important to decrease the time so out FreeTTS has two approaches to counter this , The first one to separate the wave outputs and wave synthesis into different threads.So what it does is , it allows vocalisation generation to come at the same time with the audio output and wave synthesis.This helps the samples to execute fast and the output is generated as audio fast and also allowing the other portioned vocalisation to overlap and reduce the time.Moreover the as the wave synthesis is in different thread the JVM will send it to different and its own CPU, and thus giving extra boost and reducing the time-to-first-sample.

The other method is similar to Flite in which as soon as the output is generated it is send to sound system to play.so first the whole audio is generated and then the output is send to the player.

	Flite		FreeTTS -client		FreeTTS -server	
Input Size (Words)	1-CPU Time	2-CPU Time	1-CPU Time	2-CPU Time	1-CPU Time	2-CPU Time
1	13ms	11ms	5ms	4ms	5ms	3ms
2	22ms	18ms	8ms	6ms	11ms	6ms
5	40ms	34ms	18ms	14ms	27ms	14ms
10	79ms	68ms	38ms	30ms	50ms	26ms
100	1034ms	813ms	864ms	690ms	631ms	485ms

Table 3.10. Time to First Sample

Table 9 compares the time-to-first-sample for Flite and FreeTTS while running on single CPU 296 MHz (1-CPU) system and a dual CPU 360 MHz system (2-CPU).

As Flite is designed in C Programming language the improvements to improve the performance was getting a lot difficult.

Memory Footprint Analysis

The main and the basic goal regarding the development of FreeTTS was to find out the performance of the system developed and written in Java Programming language.Our main efforts in that were regarding the time to first sample and the overall processing time but we didn't try to reduce the memory footprint of it .Right now Flite has muss less memory Footprint.

Helpful Features of the Java platform

Aforementioned are some of the features of JAVA which has led to the successful execution of our project :

Object Oriented Language -The pluggable and configurable requirements of the system was easily handled because of object oriented programming .

Dynamic Loading of Code -The new voices we were able to add easily in the system because offering a JAR file and executing it separately all thanks for the dynamic nature provided by the JAVA platform.

Multi-threaded Language -The first sound time has decreased and thus we were able to compute fast and get good results because of adding multithreading to FreeTTS.

Garbage Collection -this has just make the things really easy regarding the memory optimisation which could be really difficult with other features.

Vast API-The Java platform has many large API's has made the work for developing the FreeTTS easier and simple.We were able to give give extra boost because of the J2SE high quality implementations of data structures and algorithms.Also gaining extra features like the Java Sound API, regular expressions API ,the collections API are appreciated.

Portability - Unlike Flite where it should be ported or built specifically for every platform we were able to use the FreeTTS jar files on different platforms without any modification like Windows, Linux, Solaris 8 operating environment.

Documentation - We were able to use the source code to create a good-quality API documentation for the FreeTTS by writing FreeTTS with the Javadoc documentation standards.

FRONT-END ANALYSIS

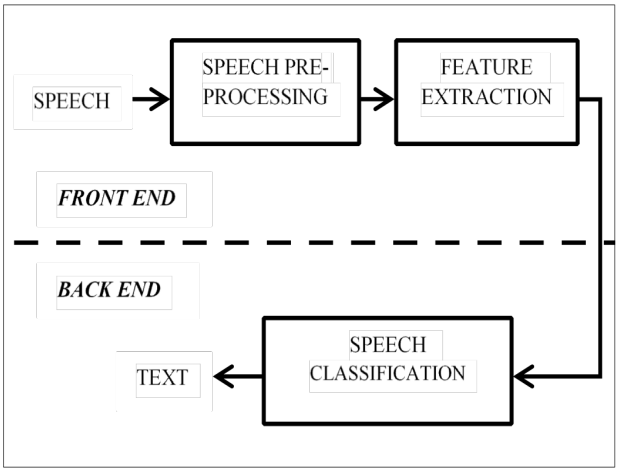


Fig 4.1. Sphinx 4 Front End Components

There are two components of SR system's FrontEnd the first one is Speech Preprocessing and the other being the Feature Extraction Block.Many issues are present like the noise and amplitude difference can have hard influence on word, while even the slightest variations in the time can cause a large difference among different words.These issues are solved by the Signal preprocessing part which uses the Framing,Windowing,Noise Filtering,End Point Detection,Echo Cancelling e.t.c.

CHAPTER-4

CONCLUSIONS AND FUTURE WORKS

In the beginning of our study of the routine characteristics of a speech synthesis engine programmed in the Java programming language, we expected that it would positively be able to run almost as fast as the native-C counterpart. By using few straightforward optimisations and relying on the aggressive optimisations performed by the Java HotSpot compiler, we found that FreeTTS runs two to four times faster than its native-C counterpart, Flite.

Evidently, it is possible for us to utilize some of these optimisations back into Flite with the possible result of enhancing Flite's performance to levels alike to FreeTTS. The absence of Java platform features for example garbage collection and high-performance collection utilities, though performing these optimisations in Flite makes much more time consuming from a programming point of view. Sphinx-4 now provides just one application of the AcousticModel, which includes Sphinx-3.3 models created by the SphinxTrain acoustic model trainer. HMMs with a fixed number of states, fixed topology, and fixed unit contexts is produced by The SphinxTrain trainer. Moreover, the constraint tying [5] between the SphinxTrain HMMs and their related probability density functions is extremely rough. Since, the Sphinx-4 framework does not have any of these limitations, it is capable of handling HMMs with an random topology over an arbitrary number of states and variable length left and right unit contexts. Furthermore, the Sphinx-4 acoustic model design allows for very fine parameter tying. We expect that taking advantage of these capabilities will significantly rise both the speed and accuracy of the decoder.

So far, we have generated a design for a Sphinx-4 acoustic model trainer that can produce acoustic models with some of these favourable characteristics [31]. As with the Sphinx-4 framework, the Sphinx-4 acoustic model trainer has been designed to be a modular, pluggable system. Such an undertaking, however, represents an important effort. As an interim step, another area for experimentation is to create FrontEnd and AcousticModel implementations that support the models generated by the HTK system. The architectural modifications that would be needed to support segment-based recognition frameworks such as the MIT SUMMIT speech recogniser were also considered. [32]. A cursory analysis indicates the modifications to the Sphinx-4 architecture would be negligible, and would provide a platform to do significant comparisons between segmental and fixed-frame-size systems. Finally, the SearchManager delivers fertile ground for implementing a number of search approaches, including A*, fast-match, bi-directional, and multiple pass algorithms.

CHAPTER-5
REFERENCES

[1] S. Young, “The HTK hidden Markov model toolkit: Design and philosophy,” Cambridge University Engineering Department, UK, Tech. Rep. CUED/FINFENG/TR152, Sept. 1994.

[2] N. Deshmukh, A. Ganapathiraju, J. Hamaker, J. Picone, and M. Ordowski, “A public domain speech-to-text system,” in Proceedings of the 6th European Conference on Speech Communication and Technology, vol. 5, Budapest, Hungary, Sept. 1999, pp. 2127–2130.

[3] X. X. Li, Y. Zhao, X. Pi, L. H. Liang, and A. V. Nefian, “Audio-visual continuous speech recognition using a coupled hidden Markov model,” in Proceedings of the 7th International Conference on Spoken Language Processing, Denver, CO, Sept. 2002, pp. 213–216.

[4] K. F. Lee, H. W. Hon, and R. Reddy, “An overview of the SPHINX speech recognition system,” IEEE Transactions on Acoustics, Speech and Signal Processing, vol. 38, no. 1, pp. 35–45, Jan. 1990.

[5] X. Huang, F. Alleva, H. W. Hon, M. Y. Hwang, and R. Rosenfeld, “The SPHINX-II speech recognition system: an overview,” Computer Speech and Language, vol. 7, no. 2, pp. 137–148, 1993.

[6] M. K. Ravishankar, “Efficient algorithms for speech recognition,” PhD Thesis (CMU Technical Report CS-96-143), Carnegie Mellon University, Pittsburgh, PA, 1996.

[7] P. Lamere, P. Kwok, W. Walker, E. Gouvea, R. Singh, B. Raj, and P. Wolf, “Design of the CMU Sphinx-4 decoder,” in Proceedings of the 8th European Conference on Speech Communication and Technology, Geneve, Switzerland, Sept. 2003, pp. 1181–1184.