Secrets of VDct: Replacing dictation components in Dragon NaturallySpeaking

Joel Gould
Director of Emerging Technologies
Dragon Systems

Copyright Information

- This presentation was given to the Voice Coder's group on June 25, 2000
- The contents of this presentation are © Copyright 2000 by Joel Gould
- Permission is hereby given to freely distribute this presentation unmodified
- Contact Joel Gould for more information joelg@alum.mit.edu

Introduction

- This presentation explains how to replace VDct, the dictation subsystem in Dragon NaturallySpeaking, with your own.
- Based around NatLink, the Python Macro System for Dragon NaturallySpeaking

Licensing Restrictions

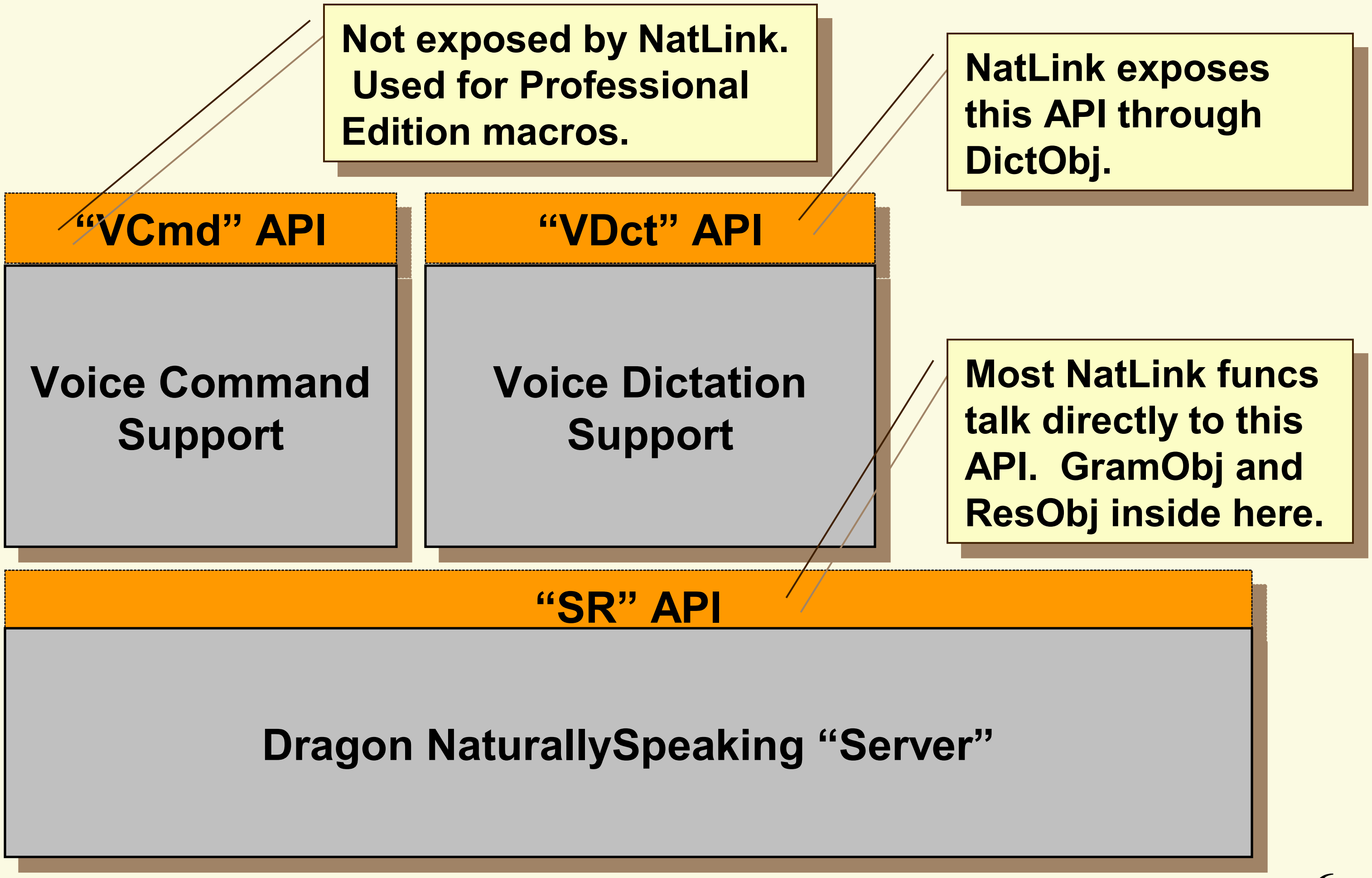
- NatLink requires that you have a legally licensed copy of Dragon NaturallySpeaking

- To use NatLink you must also agree to the license agreement for the NatSpeak toolkit
  - Soon Natlink will require the NatSpeak toolkit
  - The NatSpeak toolkit is a free download from http://www.dragonsys.com

What is SAPI?

- Speech Application Programming Interface
- Designed by Microsoft as a uniform way of supporting speech recognition in Windows
- NatSpeak is architected to mirror SAPI 4.0
  - Implements SAPI SR, VDct and VCmd APIs
  - Although NatSpeak contains no Microsoft code
  - Includes numerous Dragon-specific extensions

SAPI Architecture



Not exposed by NatLink. Used for Professional Edition macros.

NatLink exposes this API through DictObj.

"VCmd" API

Voice Command Support

"VDct" API

Voice Dictation Support

Most NatLink funcs talk directly to this API. GramObj and ResObj inside here.

"SR" API

Dragon NaturallySpeaking "Server"

Overview of Server Objects

- Clients create grammar objects
  - Command (CFG) grammars, like NatLink macros
  - Dictation grammars, which return text words
  - Selection grammars, for “Select XYZ”
- Client registers a callback function for when that grammar is recognized
- At end of recognition, server creates result object
  - Passes result object back to recognized grammar
  - Result object can be queried for choice list

Natlink Interface to Server 1

- GramObj exposes grammar objects in Python
  - GramObj.load() creates grammar from binary
  - Same function creates all 3 grammar types
- GramObj.setResultsCallback() to register a callback when grammar is recognized
- GrammarBase is a wrapper around GramObj
  - DictGramBase for dictation grammars
  - SelectGramBase for selection grammars
- Using the grammar base classes is optional
  - Have code to build binary form (which can be copied)
  - Turns callbacks into calls of member functions

NatLink Interface to Server 2

- ResObj exposes result objects in Python
- Reference to ResObj is passed to callback function (GramObj.setResultCallback)
- ResObj.getWords(N) returns recognized words for Nth choice
- ResObj.correction() is used to train recognizer after correction
- ResObj.getWave() returns wave for playback

VDct Overview

- VDct implements formatting and correction
- Based on concept of "Hidden Edit Control"
  - VDct contains a copy of the user's document
  - If user types, changes made to user's document are copied into VDct's copy of text
  - If user dictates, VDct inserts dictated text into its copy and then tells user's document about the changes
- DictObj exposes VDct object in Python
  - See windict.py (sample code) and natlink.txt (doc)

VDct: Example of Typing

- User types
- Edit control updates its text
- Text changes copied to VDct's copy of text
  - DictObj.setText()
- VDct updates Select XYZ grammar

VDct: Example of Dictating

- User dictates a phrase
- VDct gets grammar callback with result
- VDct formats text and inserts it in its copy
- VDct calls back to edit control
  - DictObj.setChangeCallback()
  - Passes back information about the text change
- Edit control updates its contents

How to Replace VDct

- Design a module which talks directly to NatSpeak Server (using NatLink or in C++ directly)
- Implement desired subset of VDct components
- Interface to application can be anything
  - I recommend using the hidden edit control model and mimicking the same VDct data flow
- No need to modify NatSpeak, your applications simply use your replacement VDct
  - NatSpeak editor, Microsoft Word, etc. will continue to use built-in version of VDct

List of VDct Components 1

- Dictation Grammar
- Basic formatting
  - Spacing, capitalization, etc. from punctuation
- Advanced formatting
  - Dates, times, numbers, currency, phone numbers, etc.
- Dictation context
- Selection grammar
- “Scratch That” command

List of VDct Components 2

- Correction commands
  - “Correct That”, “Spell That”, etc.
- Choice list for correction
- Spelling grammar during correction
- Adaptation after correction
- “Resume With” command
- Playback of recorded speech

Implementing VDct Components

. . .

Dictation Grammar

- Create an instance of DictGramBase
  - Wrapper around GramObj, defined in natlinkutils.py
- Define gotResultsObject()
  - Called when recognition occurs
  - Passed recognized words and ResObj
- Activate the grammar whenever the target application has the focus
  - Use beginCallback() to test for active window
  - Call activate() with window handle
    - do not make your dictation grammar global, it will conflict with NaturalText)

Dictation Sample Code

```
class MyGrammar(DictGramBase):

    def __init__(self):
        DictGramBase.__init__(self)
        self.load()
        self.state = None
        self.isActive = 0

    def gotBegin(self, moduleInfo):
        print 'Start of recognition...'
        if not self.isActive:
            self.activate(moduleInfo[2])
            self.isActive = 1

    def gotResults(self, words):
        print 'Heard: <%s>' % string.join(words)
        output, self.state = nsformat.formatWords(words, self.state)
        print 'Formatted: <%s>' % output
```

Use DictGramBase for dictation grammars

Just call load(), there is no text form of the grammar

Activate like a command grammar except there is no rule name

gotResults() is called with the list of recognized words; gotResultObject() also works

Recognition Hyphothesis

- While speaking, current best guess at the recognized text is available (“hyphothesis”)
- Define a hypothesisCallback
  - Will be passed a list of words
- Format the words and display during recog
  - Either in the application window itself
  - Or in a pop-up window like with NatSpeak
  - Do not call back into NatLink from hyphothesis callback (wordInfo is not available)
- Seeing hypothesis displayed makes recognizer seem more responsive

Basic Formatting

- Every word has an associated 32-bit wordInfo
- Most of those bits control basic formatting
- To format text, use a state machine
  - State is current capitalization/spacing state
  - Input is 32-bit wordInfo value for each word
    - NatSpeak never tests the spelling of the word
  - Output is modified state, formatted text
- Bits are defined in natlinkutils.py
- Use VocEdit to look at flags for existing words

Formatting State Machine

- Now distributing a new file: nsformat.py
  - Will be part of next NatLink release
- nsformat.py contains a simplified formatting state machine for NatSpeak
- Handles capitalization and spacing for normal text
- To use:
  - output,state = formatWords(words,state)
  - Use an initial state of None for empty document
  - Or call formatWord for every word so you can record the formatting state after every word

Formatting States

- Remember the formatting state after every word
- If the insertion point is moved, you can use the formatting state for that position in the document
- If necessary, compute the formatting state by looking backwards
  - After normal word:
    formatting state = 0
  - Start of document:
    flag_no_space_next, flag_active_cap_next
  - After period:
    flag_two_spaces_next, flag_active_cap_next

Other Word Flags

- Bit 0 – set for all user added words
  - This causes word to be marked in Voc Editor
- Bit 3 – set to prevent deletion of word
  - Turn this off to allow word to be deleted
  - Do not delete too many words marked as do-not-delete
- Bit 29 – set if word added from Voc Builder
  - Causes word to be added with a lower LM score
  - Use this flag when adding hundreds of words to avoid screwing up the language model

Advanced Formatting

- NatSpeak’s VDct uses a chart parser to format dates, time, numbers, currency, etc.
  - one hundred dollars and two cents ⇒ $100.02
- It is driven from a set of rewrite rules
  - If indicated sequence of tokens is seen in hidden edit,
  - Compute a block of replacement text
- If you want advanced formatting in your own VDct, you will have to:
  - (1) Code a simple chart parser
  - (2) Develop your own set of rewrite rules

Dictation Context

- Recognition is more accurate if you tell recognizer the words just before cursor
- Call DictGramBase.setContext() at recog start
  - Pass in text just before insertion point
  - Include at least two words if possible
  - Words after insertion point are not used
- Not needed if cursor is not changed after dictation
  - NatSpeak automatically remembers the last result as the context for the next recognition

Selection Grammar

- NatSpeak has special grammar type to implement "Select XYZ"
- Create an instance of SelectGramBase
  - Wrapper around gramObj, defined in natlinkutils.py
- When creating the grammar, pass in a list of verbs
  - NatSpeak uses "Select", "Correct", "Insert After", …
- At recog start, make sure grammar contains a copy of the text currently on the screen
  - SelectGramBase.setSelectText()
  - NatSpeak automatically parses text into words

Getting Selection Results

- Selection grammar gotResultsObject() gets called when user says “Select XYZ”
  - Results include the verb (select or correct)
  - Results also include the range of text selected
- NatSpeak automatically handles “Select XYZ through ABC”
- NatSpeak does not always find closest text
  - Search through choice list to find alternatives
  - Pick the alternative which is closest to cursor

Selection Sample Code 1

```
class MyGrammar(SelectGramBase):

    def __init__(self):
        DictGramBase.__init__(self)
        self.load( ['Select', 'Correct'] )
        self.setSelectText(textBuffer)
        self.isActive = 0

    def gotBegin(self, moduleInfo):
        print 'Start of recognition...'
        if not self.isActive:
            self.activate(moduleInfo[2])
            self.isActive = 1

    def gotResults(self, words, startPos, endPos):
        # Print the results of the Select recognition
        print 'Heard: <%s>' % string.join(words)
        output = textBuffer
        output = ( output[:startPos] + '<' +
            output[startPos:endPos] + '>' + output[endPos:] )
        print 'Top choice =',output
```

Use SelectGramBase for selection grammars

Call load() and pass in a list of verbs

Tell the selection grammar the block of text to select within

gotResults() returns the range of one possible selection

Selection Sample Code 2

- You need to search the choice list for all blocks of text which match the selection

```
def gotResultsObject(self,recogType,resObj):
    self.ranges = []
    try:
        bestScore = resObj.getWordInfo(0)[0][2]
        for i in range(100):
            wordInfo = resObj.getWordInfo(i)
            if wordInfo[0][2] != bestScore:
                return
            self.ranges.append(resObj.getSelectInfo(self.gramObj, i))
    except natlink.OutOfRange:
        return
```

Score is 3rd element of wordInfo tuple for first word in result

Look up the selection range for every entry in the choice list with the same score

Dictation Commands

- You can create command grammars inside of your VDct for correction and formatting
- Create an instance of GrammarBase
- Pass a set of rules to GrammarBase.load()
- Command processing is the same as when you use NatLink as a macro system
- Use command grammars for:
  - Scratch That, Correct That, Spell That, …

Undo, Redo, Scratch That

- Implement your own undo/redo stack
  - Algorithms are very easy and well understood
- "Scratch That" is like an undo
  - But does nothing if last change was not speech
  - Multiple Scratch That's do multiple undos
  - But, undo should undo Scratch That
- You are free to define your own behavior

Correction Commands

- You will have to implement your own correction commands and mechanism
- Use command grammar for correction cmds
  - <cmd1> = correct that
  - <cmd2> = spell that [<dgnletters>]
- Create your own user interface for correction
- Remember that you know what text is selected

Creating a Choice List

- ResObj can be queried for choice list
  - ResObj.getWords(N) for Nth choice
- If you are correcting only part of an utterance, you have to extract choices from list:
  - ResObj.getWordInfo() returns word times
  - Look up word start and end time for word/phrase being corrected
  - Search through other choices to find word/phrases which similar start and end times

Backup Dictionary

- Once the user start typing, you will have to get words from a word list
- You can not use Dragon's word list
  - The iterator function has not been exposed
- Find a list of words from somewhere else
  - Build a dictionary which can be queried by prefix
  - You do not need prons, once you have a word list NatSpeak can look up the prons in its own dictionary

Adapting after Correction

- After a real correction, perform adaptation
  - Compute the words which match the whole utterance (only a part may have been corrected)
  - Call ResObj.correction()
- Recognizer may reject is correction is too different from utterance to use for training
  - No further action is required in either case

“Resume With” Command

- “Resume With <word> <more text>”
  - Where <word> was dictated recently
  - Replaces everything after <word> with <more text>
- If you want this command, you will have to implement it with a command grammar
  - <rule> = resume with {words} <dgndictation>
  - Set list “words” with last N words dictated
- When grammar is recognized, modify the text

Using Playback

- You can get the wave for any result
  - resObj.getWave()
- Wave is 11.025Khz, 16 bit, mono
- Playback using Windows multimedia API
  - You will have to find or write your own code for this
- To play part of an utterance
  - Index into the wave using the word starts
  - From resObj.getWordInfo()

Implementation Hints

. . .

Keeping Track of Results

- For many of the VDct algorithms you need to know what result object corresponds to a block of text on the screen
  - For example: correction and playback
- Remember the result objects passed to gotResultsObject() for the dictation grammar
- Keep a link between the copy of the user’s text and the result object for that dictated text

Handling Text Modifications

- What happens if user types or overspeaks a portion of an utterance?
- NatSpeak version 1 and 2 simply discarded the result object for the modified text
  - This prevented adaptation and playback
- Modern NatSpeaks try to keep track of sections of result objects
  - But this extreme is probably not necessary

Keeping Text Synchronized

- Keep the real text and VDct's copy of the text synchronized at all times
- It is best to update VDct as soon as user changes the edit control (i.e. by typing)
  - This allows VDct to update Select grammar
  - Makes it easier to keep text and results aligned
- For correctness, it is enough to update the contents of the hidden edit control at recognition start
- It is also best to lock out user input in the middle of recognition
  - To avoid user changes at the same time as dictation

Recognition Start Bookkeeping

- gotBeginCallback() is called at start of every recognition
- Recognizer will pause until you return from func.
- During callback, do bookkeeping:
  - Make sure text is synchronized with application
  - Get the location of the insertion point from the app.
  - Activate or deactivate grammars
  - Update select grammar from text
  - Update dictation context
  - Update “Resume With” word list

Mixing Commands in Dictation

- Command (and select) grammars are only recognized when surrounded by pauses
- It is possible to implement pause-less commands when you rewrite VDct
- Write your commands in some CFG format
- Scan every dictation result for a sequence of words which matches CFG
  - For example, with a chart parser
- Remove those words from the text to be inserted and execute the command action

Managing Words

- addWord() adds a word to dictation state
  - You do not need to specify the pron, NatSpeak will either lookup the pron or guess it
- Be sure to set the word flags
  - dgnwordflag_useradded for all new words
  - dgnwordflag_topicadded if adding lots of words
  - Other formatting flags as appropiate
- NatSpeak's VDct automatically adds any words which are in the user's document if they are also in the backup dictionary
  - Use getWordInfo() to see if the word is in backup dict

Who Calls Whom

- To use NatLink (or NatSpeak), you must be in a Windows message loop for receive callbacks
- You can:
  - Write a NatLink grammar file which will be loaded automatically; in this case the message loop is inside NatSpeak itself
  - Be run from a Win32 GUI which includes a message loop (like DoModal() in winspch.py)
  - Or, include a call to natlink.waitForSpeech() which enters a message box modal loop (like dictsamp.py)

Summary

- VDct is designed for dictating English text
- Its behavior makes it hard to use for programming
- But most VDct functionality can be written outside of NatSpeak, using the Server API
- By replacing VDct, you can change:
  - Formatting, correction mechanism, correction commands, selection behavior, etc.
- NatLink wraps enough of Server API to make it possible to rewrite VDct in Python

All Done

“Microphone Off”