Implementation and Acceptance of NatLink, a Python-Based Macro System for Dragon NaturallySpeaking

Joel Gould
Recently of Dragon Systems
joelg@alum.mit.edu

1 Abstract

This paper describes NatLink, an extension to Dragon NaturallySpeaking that allows customers to develop command and control (speech) macros using the Python programming language. In this paper, I review the original goals of the NatLink project. Then I describe the architecture of the NatLink implementation, focusing on the way Python was integrated with Dragon NaturallySpeaking. Finally, I present a summary of the acceptance of NatLink and Python, based on feedback from users.

2 Introduction

> "*1) NatLink is free.*
> *2) VB sucks... Python rocks!*" – tscheresky

I was first introduced to the Python programming language in the spring of 1999. While studying the limited assortment of Python books available at that time [1], and considering possible projects which I could undertake to properly learn the language, I came up with the idea of using Python as a macro language for Dragon NaturallySpeaking, Dragon Systems' speech recognition product. This document describes the results of my exercise, called NatLink, and presents some observations on how Python was received by users, based on feedback from self-selected NatLink users.

2.1 Dragon NaturallySpeaking

Dragon NaturallySpeaking (NatSpeak) is the world's first and best selling [2] large vocabulary general-purpose continuous speech recognition system. NatSpeak allows users to dictate text by speaking into a headset microphone instead of using the keyboard. NatSpeak also has the ability to recognize spoken commands, and to translate those commands into keystroke sequences or other actions that manipulate the current application or Windows itself. This document was (mostly) dictated by voice using Dragon NaturallySpeaking.

NatSpeak is sold in a variety of different editions, differentiated by feature set and price. The most popular retail version of Dragon NaturallySpeaking is the Preferred Edition that often sells for around $150. The Preferred Edition contains most of the features the average user needs but only offers limited customization. Advanced users are encouraged to purchase the more expensive Professional Edition (at around $700) from a value added reseller. The Professional Edition includes the ability to write command and control macros in a simple scripting language, syntactically similar to Visual Basic (although more limited).

Although sophisticated for a speech recognition product, the built-in macro capabilities in the Professional Edition of Dragon NaturallySpeaking are very limited compared to, for example, the Visual Basic macro system in Microsoft Office applications. Based on my personal experiences, and also feedback from customers, I identified a number of limitations with the existing NatSpeak scripting language and macro capabilities that I wanted to address with my Python based macro system.

First, the macro development process for Dragon NaturallySpeaking is cumbersome, especially when developing multiple macros. For each speech command, a user must navigate an eight-step wizard (dialog) filling in information about the target application, the speech command that will be spoken, and the keystroke sequences or script to be executed.

Although user written macros are saved in human readable form on disk, the macro file is only loaded when Dragon NaturallySpeaking first starts. Therefore, it is possible to create and/or edit macros using a more

conventional programming editor. But, the macro developer has to restart NatSpeak (which might take as long as a few minutes) multiple times during the macro writing and testing process.

Second, the scripting language built-in to Dragon NaturallySpeaking is very limited in its expressive power. Primarily designed for writing simple sequences of operations, the built-in scripting language includes some simple logic and the ability to create local variables. However, there is no persistence of variables across macro invocations, no support for OLE automation, no complex objects (arrays, classes, dictionaries) and no library of functions other than basic built-in string manipulation. [3]

Finally, for historical reasons the types of grammars that users are allowed to specify in the Professional Edition are limited to simple sequences of tokens. The built-in macro system has no support for grammars with optionals, alternatives, repeats or nested rules. Yet, the underlying speech recognizer itself supports these more advanced grammatical structures.

2.2 Project Goals

I had three goals for the development of NatLink:

1. To develop an add-on for Dragon NaturallySpeaking that would allow existing Dragon NaturallySpeaking users to create command and control macros that (a) include the full expressive power of the grammar capabilities of the speech recognition system, (b) use Python as the interpretive scripting language, and (c) overcome the existing limitations of the grammar development process.

2. To develop a Python extension module for controlling Dragon NaturallySpeaking that would allow customers to write Python programs which manipulate the vocabulary, create grammars, perform recognition and process recognition results.

3. To allow Python developers to add speech recognition to their Python programs. Dragon Systems already produces a developer's toolkit that allows Visual Basic and C++ developers to take advantage of the features of Dragon NaturallySpeaking from within the programs. My goal was to piggyback on the NatLink macro system to allow Python programmers access to similar capabilities.

In this paper, I will limit the discussion to the first goal of the project. Details of how NatLink can be used to control Dragon NaturallySpeaking and build speech-enabled Python programs are beyond the scope of this paper.

As a macro system, NatLink has to work with existing versions of Dragon NaturallySpeaking, in such a way that it complemented but did not replace existing functionality. NatLink had to work with NatSpeak APIs (both published and unpublished) so that modifications to Dragon NaturallySpeaking were not required. It was not a goal for the development of NatLink that functionality be restricted only to the Professional Edition.

Additional, the development of NatLink was not to be an exercise in UI design. The goal was to allow grammar developers to use their favorite text editor to write macros that could then be tested by using Dragon NaturallySpeaking itself.

2.3 Project Overview

I implemented NatLink in the spring of 1999. In July of 1999 I posted the system on the Internet [5] in open source form, and advertised in both the speech recognition mailing lists and the Python newsgroups. Details of the implementation of NatLink are described in the next section.

In October of 2000, in preparation for the writing of this paper, I requested feedback from both NatLink users and people who downloaded NatLink but who had not had success with it. A summary of the feedback received is presented in section 4 of this paper.

3 Implementation

NatLink consists of both C++ code (developed using Microsoft Visual C++ 5.0 and 6.0) and Python code (developed using Python version 1.5.2 for Windows). The C++ code is used to interface to Dragon NaturallySpeaking through NatSpeak's native COM interfaces. Python is used to control the loading of grammar files, parsing of grammars and for miscellaneous support classes and functions.

In order to understand the design of NatLink, it is first necessary to review the architecture Dragon NaturallySpeaking and availability of the NatSpeak external interfaces.

3.1 Architecture of Dragon NaturallySpeaking

The architecture of Dragon NaturallySpeaking is based around the design of Microsoft Speech API (SAPI) 4.0. [6] [7] (In 2000, Microsoft announced the availability of version 5.0 of their Speech API; however, SAPI 5.0 is different from SAPI 4.0 and its architectural concepts do not apply to Dragon NaturallySpeaking.) To that end, the functionality of Dragon NaturallySpeaking is accessible to external programs through the objects and methods than make up the Microsoft Speech API.

The Dragon NaturallySpeaking (and SAPI) APIs are all based on COM, the Microsoft Component Object Model. [8] COM is the transport layer used by OLE (Object Linking and Embedding), by ActiveX and by Visual Basic (OLE Automation). However, the Dragon NaturallySpeaking interfaces are all custom COM interfaces, and not accessible to programs (like Visual Basic) through OLE Automation.

Within Microsoft SPAI, the core speech recognition functionality is exposed by a low-level interface that Microsoft refers to as the Direct Speech Recognition (SR) API. Dragon NaturallySpeaking supports the majority of the interfaces that Microsoft specifies in its SAPI design, and then extends the API with some additional, Dragon-specific interfaces.

The SR API provides basic low-level access to the speech recognition process. Using the SR API, clients can control the speech recognition engine including loading speaker profiles, turning the microphone on and off, adding vocabulary words, and creating grammars. To perform recognition, clients use the SR API to create “grammar objects", which encapsulate knowledge about the allowable sequence of words to be recognized. When recognition happens, the speech recognition system informs clients about the result of the recognition by creating "result objects", which contain details about the actual words recognized.

In the Microsoft Speech API architecture there are also two higher level APIs designed to make it easier for applications to use speech recognition. The details of two APIs, called the Voice Command API and the Voice Dictation (VDct) API are beyond the scope of this paper. NatLink does provide some access to the Voice Dictation API but it is not needed when using Python as a macro language for Dragon NaturallySpeaking (this project’s first goal).

In addition, Dragon NaturallySpeaking includes a set of system services only peripherally related to speech recognition. These services are made accessible through another COM-based interface called the Speech Services (SSvc) API. Using the Speech Services API, a client can do things like send keystrokes to a third party application. The Speech Services API also exposes the built-in scripting language, so a client can submit an entire script to be executed through the Speech Services API.

The Dragon NaturallySpeaking Professional Edition's built-in command and control subsystem uses both the VCmd API and the Speech Services API. When the user creates a new voice command through the NatSpeak user interface, a command is registered with the VCmd API and the user’s script (or keystroke sequence) is stored along with the command to be recognized. When the user speaks this command (assuming the command is recognized), the VCmd API returns the stored script, which is then executed by passing the script to the Speech Services API.

The built-in command and control subsystem actually exists in all editions of Dragon NaturallySpeaking. However, only the Professional Edition exposes the user interface that allows users to add their own voice commands. In the less-expensive editions of Dragon NaturallySpeaking, the command set is fixed, and cannot be modified by users.

In addition to the interfaces discussed above, Dragon Systems does have a developer's toolkit available for people who want to interface with Dragon NaturallySpeaking. The developer's toolkit includes the Microsoft SAPI interfaces as well as a wrapper around the Dragon extensions of Microsoft SAPI to make it easy for developers to write simple speech applications in C++ or Visual Basic. [9] For the development of NatLink, I decided not to use the developer's toolkit, but instead I directly interface to the lower-level APIs. This decision was primarily based on the availability of more functionality using the native APIs than was exposed through the toolkit.

3.2 Python Extension Module

The first step in developing the NatLink Python macro system was to develop a Python extension module. This Python extension module, written in C++, provides an interface to the Dragon NaturallySpeaking native APIs from Python scripts.

As described above, the Dragon NaturallySpeaking APIs are based on custom COM interfaces. Because they use custom COM interfaces instead of OLE automation, it was not possible to use Mark Hammond's PythonCOM system [4]. In addition, conventional Python wrapping technology, like SWIG [10] or Py_cpp [11], is designed for libraries that expose C or C++ APIs, not COM interfaces. So that technology could not be used either.

Instead I wrote custom C++ code to export a selected set of objects and functions from the Dragon NaturallySpeaking APIs. The extension module is called natlink.dll and can be used from Python by importing "natlink".

3.2.1 Engine Object

COM is an object-oriented interface. Clients get references to objects implemented in the server, and then call methods on those objects to perform various operations. When using the SR API, many of the global functions (like changing the microphone state) are methods of the "engine object", of which there is exactly one per client.

In the NatLink extension module code, I use C++ class instances to encapsulate the COM objects. The engine object is encapsulated by a C++ class for which there is exactly one instance in the extension DLL. A connection between this class instance and Dragon NaturallySpeaking is created by calling the function natlink.natConnect() from Python. This function causes instances of multiple COM objects to be created in Dragon NaturallySpeaking and references to those COM objects to be stored in the encapsulating C++ class's member variables.

(Like Python, COM uses reference counting to determine when objects can be cleaned up. When you call natlink.natConnect(), reference counts inside of Dragon NaturallySpeaking are incremented insuring that the engine object will remain in memory until the client process is done with it. Python programs that use NatLink have to remember to call natlink.natDisconnect() to release those references and allow Dragon NaturallySpeaking to unload from memory.)

Most of the NatLink Python extension module consists of code that wraps the basic methods of the engine object, the grammar object and the result object from the SR API. Each method of the engine object is exposed to Python by including a C++ function inside the extension module that translates Python arguments into C++ basic types and calls the underlying Dragon NaturallySpeaking methods through COM.

Examples of similar functions can be found in any decent text on extending and embedding Python (see, for example, [1]).

3.2.2 Grammar Objects

In order to perform speech recognition in Dragon NaturallySpeaking, it is necessary to create a grammar object. Grammar objects are created by passing a binary representation of a grammar across the SR API using the GrammarLoad() method of the engine object. This results in the creation of a new COM object, called a grammar object, which includes methods to activate and deactivate the grammar.

In the NatLink Python extension module, grammar objects are encapsulated with the C++ class.. This class is then tied to a Python extension class called natlink.GramObj. The function natlink.GramObj.load() actually

creates the COM grammar object from the primary representation of the grammar passed to natlink.GramObj.load().

To avoid requiring that NatLink users master the SAPI binary grammar representation, the NatLink package includes a grammar compiler, which converts a textual representation of the grammar into the binary format. The grammar compiler is entirely implemented in Python in the module called gramparser.py. The grammar compiler uses a conventional lexical scanner and token parser to convert the textual representation of the grammar into an internal data structure. This data structure is then converted into a binary Python string using the struct module.

Grammars consist of a set of rules. Each rule contains a combination of words connected as sequences, alternatives, optionals or repeats. Rules can also include references to other rules as well as words and lists. Lists, which are not discussed in this paper, are a SAPI construct for dynamically changing the grammar without destroying and then recreating the grammar object.

The following figure shows a simple grammar in the textual format supported by NatLink.

```
<start> exported = <ruleOne> | <ruleTwo> ;
<ruleOne> = [ please ] move ( up | down ) ;
<ruleTwo> = the number is ( 1 | 2 | 3 )+ ;
```

Figure 1 - Sample NatLink Grammar

This grammar consists of three rules, “start”, “ruleOne” and “ruleTwo”. Rule names are always included in angle brackets to distinguish them from lists (which are included in braces) and words or phrases (which can optionally be included in quotation marks). The definition of the rule is on the right hand side of the equal sign. The grammar syntax is similar to standard notations. Vertical bars are used for alternatives, parentheses are used for grouping and a plus sign means a repetition of 1 or more times.

The grammar shown above can recognize such utterance as “move up”, “please move down” and “the number is 2 3 2 1”.

The keyword “exported” in the definition of the rule “start” means that the rule “start” can be activated for recognition. The other two rules cannot be directly activated for recognition, but can be used because they are included in the definition of the rule "start".

The following sample code fragment connects to Dragon NaturallySpeaking, loads the sample grammar described above, activates the grammar and turns on the microphone.

```
import natlink

def gotRecognition( words, result_object ):
    print words

textual_grammar = """
    <start> exported = <ruleOne> | <ruleTwo> ;
    <ruleOne> = [ please ] move ( up | down ) ;
    <ruleTwo> = the number is ( 1 | 2 | 3 )+ ; """
binary_grammar = convertGrammar( textual_grammar )

natlink.natConnect()
grammar_object = natlink.GramObj()
grammar_object.load( binary_grammar )
grammar_object.setResultsCallback( gotRecognition )
grammar_object.activate( "start", 0 )
natlink.setMicState( "on" )

natlink.waitForSpeech()
natlink.natDisconnect()
```

Figure 2 - Simple Code to Create and Recognize from a Grammar

The function convertGrammar() does the conversion of the grammar from text to binary form. It can be written using the Python code in gramparser. This function does not really exist in the Natlink package because for most NatLink code, grammar conversion is done in the Python class GrammarBase, described later.

Complete documentation for the textual grammar syntax is included in the gramparser.py source file (part of the NatLink package).

3.2.3 Result Objects

In Dragon NaturallySpeaking, when a grammar is recognized, a callback is made to a client-created COM object called a NotifySink. The intricacies of creating this COM object and passing it into Dragon NaturallySpeaking using the GrammarLoad() call are handled by the C++ code in the NatLink C++ extension module.

For Python macro files, callbacks are handled by allowing the Python clients to specify a reference to a Python function. For grammar objects, the natlink.GramObj() class has a method called setResultsCallback() that takes a reference to a Python function. Then, when a grammar is recognized, the Python function reference passed to setResultsCallback() is called with details of the recognition results.

In Figure 2, above, the penultimate line is used to perform the recognition. This function call, natlink.waitForSpeech(), displays a dialog box on the screen with a single “Cancel” button. The function natlink.waitForSpeech() does not return until the user presses Cancel button. If you actually execute the sample code, you should press the Cancel button after you are done talking to the system. Once control returns to the script, a call is made to natlink.natDisconnect() to release all references to the Dragon NaturallySpeaking COM objects.

When this sample code is executed, if the words "please move up" are spoken into the microphone, the string “[('please', 2), ('move', 2), ('up', 2)]” will be printed.

In Dragon NaturallySpeaking, whenever speech is recognized, a COM object, called a result object, is created and passed back to the client through the NotifySink callback. In NatLink, the Python extension module also has a wrapper C++ class for the result object. This is exposed as a Python class called natlink.ResObj.

For every recognition results callback, a reference to a newly created result object (natlink.ResObj) is passed as the second argument to the Python callback function (the function gotRecognition() in the previous example). The Python code can ignore this object (in which case the Python object and associated COM object will be garbage collected) or use the result object to learn more about the recognition results.

The result object method natlink.ResObj.getResults() can be used to access the recognition results. This function returns a list of tuples, each tuple contains the word (i.e. the string) that was recognized and the number of the rule that contained the word that was recognized. For convenience, the same list returned by natlink.ResObj.getResults(0) is also passed as the first argument to the Python callback function.

In the previous example, the list that was returned when you said “please move up” to Dragon NaturallySpeaking was “[('please', 2), ('move', 2), ('up', 2)]”. As you can see from this example, three words were recognized and each recognized word was from rule number 2. Rules are assigned numbers automatically by the grammar processing code because the names of rules are not passed to Dragon NaturallySpeaking in the binary grammar format (except for rules that are exported). Rule number 2 in this example is the rule named “ruleOne”.

To get around the fact that Dragon NaturallySpeaking (because of the design of Microsoft SAPI) deals in rule numbers instead of rule names, NatLink includes a Python class that simplifies the creation of grammars and handling of grammar results. The Python class is called GrammarBase and it is defined in the file natlinkutils.py included with the NatLink package.

The GrammarBase class encapsulates a NatLink GramObj. The GrammarBase.load() function takes the textual representation of the grammar as input, freeing the user from having to worry about converting the grammar into binary format. In addition, with GrammarBase, handling of results has been improved. Instead of requiring the user to create a result callback function and decode the rule numbers, all the user has to do is subclass GrammarBase and create methods with names derived from the actual rule names.

Working from the previous example, when words are recognized from the rule "ruleOne", a call will be made to the class member function “gotResults_ruleOne”, if such a function exists. Similarly, if words are recognized from the rule “ruleTwo”, than a call will be made to the class member function “gotResults_ruleTwo” if that function exists. The GrammarBase utility class handles the tedious work of translating the rule numbers back into rule names.

The following sample code implements the same example grammar, but using the GrammarBase utility class.

```
import natlink
import natlinkutils

class MyGrammar( natlinkutils.GrammarBase ):

    textual_grammar = “””
        <start> exported = <ruleOne> | <ruleTwo> ;
        <ruleOne> = [ please ] move ( up | down ) ;
        <ruleTwo> = the number is ( 1 | 2 | 3 )+ ; “””

    def initialize(self):
        self.load(self.textual_grammar)
        self.activateAll()

    def gotResults_ruleOne(self, words, fullResults):
        print words

natlink.natConnect()
grammar_object = MyGrammar()
grammar_object.initialize()
natlink.setMicState( “on” )

natlink.waitForSpeech()
natlink.natDisconnect()
```

Figure 3 - Simple Example of Using GrammarBase class

When this sample code is executed, and the words "please move up" are spoken into the microphone, the string “['please', 'move', 'up']” will be printed. If, in the function MyGrammar.gotResults_ruleOne(), the parameter “fullResults” were used instead of “words”, then the following string would be printed: “[('please', ‘ruleOne’), ('move', ‘ruleOne’), ('up', ‘ruleOne’)]”.

Complete documentation for the GrammarBase utility class is included in the natlinkutils.py source file (part of the NatLink package).

3.2.4 Speech Services

The engine object, grammar object and results object are all features of the Direct Speech Recognition API in Dragon NaturallySpeaking. The NatLink extension module also exports some of the functions available in the Speech Services API.

The most useful function from the Speech Services API included in the NatLink extension module is called natlink.playString(). It takes a string, which represents a series of keystrokes, and generates those keystrokes as if they were typed at the keyboard. For example, calling natlink.playString(‘hello’) causes the string ‘hello’ to be typed into the currently active application.

The function natlink.playString() uses the standard Dragon NaturallySpeaking syntax for special keystrokes. Non-printible keystrokes are represented by the key name in braces. For example, “{enter}”, “{pgup}” and “{ctrl+c}”.

In NatLink, generating keystrokes is synchronous; in other words, natlink.playString() does not return until after all the keys in its parameter string have been send to the active application. This turned out to be difficult to implement because the corresponding COM function in the Speech Services API is not synchronous. The COM function returns immediately after the keystrokes have been queued up and a callback (to a client specified COM NotifySink) is used to tell the client when all the keystrokes have been generated. This complexity is hidden inside the NatLink extension module so the Python user does not have to be bothered.

Another interesting function from the Speech Services API that has been included in the NatLink extension module is called natlink.execScript(). This function takes a string that contains one or more statements in NatSpeak's built-in scripting language, and executes the script. Natlink.execScript() returns synchronously after the script is finished.

The purpose of exposing an interface to NatSpeak's built-in scripting language is to ensure that the NatLink macro system is a superset of the command and control capabilities available in the Dragon NaturallySpeaking Professional Edition. If any functions in the built-in scripting language are not available in either Python or the NatLink extension module, then the natlink.execScript() function can be used as a fallback.

(The one place where I have found natlink.execScript() useful is to execute the NatSpeak scripting language command "AppBringUp", which launches or activates a named application.)

NatLink also exposes the ability to control the mouse pointer through the function natlink.playEvents(). This function supports simulating button clicks, mouse moves and mouse drags.

3.2.5 Extension Module Summary

In this section of the paper, I have discussed how NatLink includes a Python extension module that encapsulates some of the functions available in the Dragon NaturallySpeaking API. Not every function in NatSpeak has been exposed in Python. Instead, I focused on those capabilities that I felt would be most useful to macro writers.

The NatLink extension module also includes a couple of non-NatSpeak functions that I have found useful in writing Python macros. Once such function is natlink.getClipboard(), which returns a string containing the current contents of the Windows clipboard. This function is useful for transferring strings into Python for manipulation.

As an example, the following snippet of Python code reverses the characters that are currently selected on the screen. This code works because almost all Windows applications use the same keyboard shortcuts.

```
natlink.playString( '{ctrl+c}' )  # copy to clipboard
text = natlink.getClipboard()     # get the text into Python string
text.reverse()                    # reverse the text
natlink.playString( text )        # type the text back, replaces selection
```

Figure 4 - Python Code to Reverse Text on Screen

In the next chapter, I describe how this extension module was turned into a macro system integrated into Dragon NaturallySpeaking.

3.3 Python Macro System

With the components described so far, NatLink is almost good enough to be used as a macro system for Dragon NaturallySpeaking. As shown earlier (Figure 3, for example), it is possible to develop a Python file that loads a grammar, enters a message loop, and when the grammar is recognized, processes the speech recognition results.

This approach, however, has two significant drawbacks. First, it is necessary to start running the script. This means that after Dragon NaturallySpeaking is started, the Python interpreter needs to be started and then the script needs to be executed from the Python command line. Second, the Python process needs to be in a Windows message loop, which probably means that there is a dialog box on the screen whenever the Python script is running. Finally, the user has to remember to close that dialog box before shutting down NatSpeak.

3.3.1 Creating a Compatibility Module

Fortunately, Dragon NaturallySpeaking has an architectural feature that makes this unnecessary. From the beginning, NatSpeak has included the concept of "compatibility modules", which are bodies of code that can be

dynamically added to Dragon NaturallySpeaking to support speech enabling of third party applications. NaturalWord, the support for Microsoft Word and WordPerfect, is one example of a compatibility module. The support for Lotus Notes introduced in Dragon NaturallySpeaking is also implemented using a compatibility module.

A NatSpeak compatibility module is implemented as a COM server that exposes a well-defined (Dragon-specific) COM interface. To add a compatibility module to Dragon NaturallySpeaking, one would create an entry in the NatSpeak section of the Windows registry that references the compatibility module to be loaded. (When using COM, references to modules are implemented by using GUIDs, which are large numbers guaranteed to be unique for each COM server and interface.)

When Dragon NaturallySpeaking starts running, it checks the Windows registry for references to any compatibility modules. Each compatibility module listed in the Windows registry is then loaded using the standard COM techniques for connecting to a server.

What I did was turn the NatLink Python extension module into a compatibility module by turning it into a COM server. This means that there are two ways to load the NatLink Python extension module into memory. First, if your Python program includes a line “import natlink”, then Python dynamically loads the library (natlink.dll), and calls the exported DLL function named “initnatlink” to initialize the extension module.

The second way to load the NatLink extension module is as a compatibility module. If a reference to the NatLink Python extension module is added to the Dragon NaturallySpeaking section of the Windows registry, then when Dragon NaturallySpeaking starts it will use COM to load the NatLink Python extension module into memory. Then NatSpeak, through COM, will call the interface function IDgnAppSupport::Register() to initialize the module. The effect is the same; the code gets loaded into memory.

The following code is extracted from the NatLink source code. It is the function (without the error handling) that is automatically called by Dragon NaturallySpeaking when NatLink is loaded as a compatibility module.

```
STDMETHODIMP CDgnAppSupport::Register( IServiceProvider * pIDgnSite )
{
   // load and initialize the Python system
   Py_Initialize();

   // load the natlink module into Python and return a pointer to the
   // shared CDragonCode object
   m_pDragCode = initModule();
   m_pDragCode->setAppClass( this );

   // simulate calling natlink.natConnect() except share the site object
   m_pDragCode->natConnect( pIDgnSite );

   // now load the Python code which does most of the work
   m_pDragCode->setDuringInit( TRUE );
   m_pNatLinkMain = PyImport_ImportModule( "natlinkmain" );
   m_pDragCode->setDuringInit( FALSE );

   return S_OK;
}
```

Figure 5 - Implementation of IDgnAppSupport::Register()

As you can see, when NatLink is loaded from Dragon NaturallySpeaking, it launches the Python interpreter in embedded mode. Then a call is made to initModule(), which is the same function that is executed when NatLink is imported into a running Python process.

After that, the compatibility module initialization code calls natlink.natConnect(), the same call which would be made if NatLink was used to connect Dragon NaturallySpeaking from the Python interpreter. Finally, when NatLink is loaded from Dragon NaturallySpeaking, an additional Python module named "natlinkmain” is automatically loaded. This is equivalent to running the following script from Python.

```
import natlink
natlink.natConnect()
```

```
import natlinkmain
```

Figure 6 - Equivalent Python Code Run When NatLink is Loaded

Because starting NatLink as a compatibility module also launches the Python interpreter, it is not necessary for the user to separately start Python and load their macro file. In addition, the code within Dragon NaturallySpeaking that manages compatibility modules will also cleanly unload the compatibility module when Dragon NaturallySpeaking terminates by calling a separate function called IDgnAppSupport::Unregister(). In NatLink, this cleanup code, which is shown below, includes the necessary call to natlink.natDisconnect() and also decrements the reference counts to the previously imported natlinkmain module, to allow Python to cleanup.

The compatibility module approach support solves the problem of getting Python started, but that is not enough to get the user's macro files loaded. That additional step is handled in Python in the file natlinkmain.py, which gets loaded when the NatLink compatibility module is started.

3.3.2 Macro System Controller

The support for loading and unloading user-defined macro files is written in Python in a module called natlinkmain, or the macro system controller. The goal is to allow users to create Python files that contain both the grammar specifications and the code to execute when those grammars are recognized. Although I toyed with the idea of inventing a file format that would logically contain syntax for the grammars, and a way of specifying the code to be executed when the grammars are recognized, I decided instead to stick with pure Python files.

The macro system controller will simply load user specified macro files by importing them as it would any other Python script. This leaves the user free to write any Python code he wants in the macro file; although, the usual scenario is for the user to define a grammar and the code to be executed when the grammar is recognized.

When I designed NatLink, I decided on a simple mechanism for locating and loading macro files. User macro files would be standard Python scripts, except their name would start with a leading underscore to identify them as macro files to be loaded when the Python macro system starts. The macro system controller will, therefore, look for all Python modules whose names begin with an underscore, and which are located the same directory as the natlinkmain script itself.

natlinkmain is imported as part of the initialization of the NatLink module when NatLink is loaded as a compatibility module. When natlinkmain is loaded, it firsts get the name of the directory currently containing natlinkmain.py, so that the directory can be searched for macro files. Then natlinkmain scans its directory for any appropriately named Python files (_xxx.py).

Once a macro file is loaded, its name is added to a list of loaded macro files. This list is maintained so it is possible to cleanup any references held by the objects defined in those macro files, should the macro file be changed or deleted. (By convention, the macro system controller calls the function named “unload” in any macro file previously loaded, when that macro file is unloaded. Users do not have to define an unload function, but if any objects are created when macro file is loaded, then an unload function is necessary to release the references.)

Individual macro files are loaded simply by importing them. The process of importing the file has the side effect of executing any code in that macro file. Users are not required to write any "initialize" function, since initialization is assumed by executing any global statements when the macro file is imported.

The macro system controller has been designed to try to reload all of the users Python macro files at various convenient times. This includes when (1) the user starts speaking, (2) the user turns on or off the microphone, or (3) the user changes the currently active speech profile. However, before reloading any macro file, a check is made to see if that file has been changed (by comparing file date). Because a macro file is not reloaded unless it is changed, the overall effect is that any user macro file that is modified, even while Dragon NaturallySpeaking is running, will be reloaded at the next convenient time.

The macro system controller has also been designed to additional look for macro files in a speech profile specific directory. This allows a single installation of Dragon NaturallySpeaking to have different sets of Python macro files based on the active speech profile. Additionally, the macro system controller has been designed to load

some Python macro files only when a particular application is started. However, the details of these particular enhancements are beyond the scope of this paper.

At this point, you may be wondering where the Windows message loop for the Python macro system is to be found. User macro files cannot have a Windows message loop (no call to natlink.wairForSpeech) or the macro system controller would stop execution every time it loaded a user's macro file.

However, there is also no Windows message loop in the macro system controller code itself (natlinkmain.py). When natlinkmain is loaded from the initialization of the NatLink compatibility module, it loads all of the user macro files and then returns control to the compatibility module initialization routine. The compatibility module initialization code then returns control back to Dragon NaturallySpeaking itself.

Instead, the Windows message loop is inside Dragon NaturallySpeaking. The code in NatSpeak that load compatibility modules will enter a Windows message loop once all of the specified compatibility modules have been loaded. Individual compatibility modules do not have to enter a Windows message loop, and they should not enter a Windows message loop since this might impact the initialization of other compatibility modules (or user macro files).

3.3.3 Getting Output

When NatLink is used as a compatibility module instead of as a Python extension module, the Python interpreter is embedded in NatLink instead of the other way around. In this scenario, there is no Python interpreter input loop running, and there is no Python interpreter user interface for displaying output.

To make debugging simpler, the macro system controller redirects standard output, changing writes to stdout and stderr into function calls to natlink.displayText(). The function displayText() is written in C++ inside of the NatLink extension module. This function displays a dialog box on the screen that contains the text passed to it. Thus, if any Python code running under the NatLink compatibility system generates either stdout or stderr output, that output will be redirected to a dialog box, which will pop up on the screen so the user can read the text.

Here is a very simple macro file that will generate output when it is run.

```
print 'This was printed from a user macro file.'
raise ValueError
```

Figure 7 - Macro File That Generates Output

When this code is included in a file called _error.py, and then the Dragon NaturallySpeaking microphone is toggled to force the macro system controller to load this new macro file, the following dialog box appears on the screen.

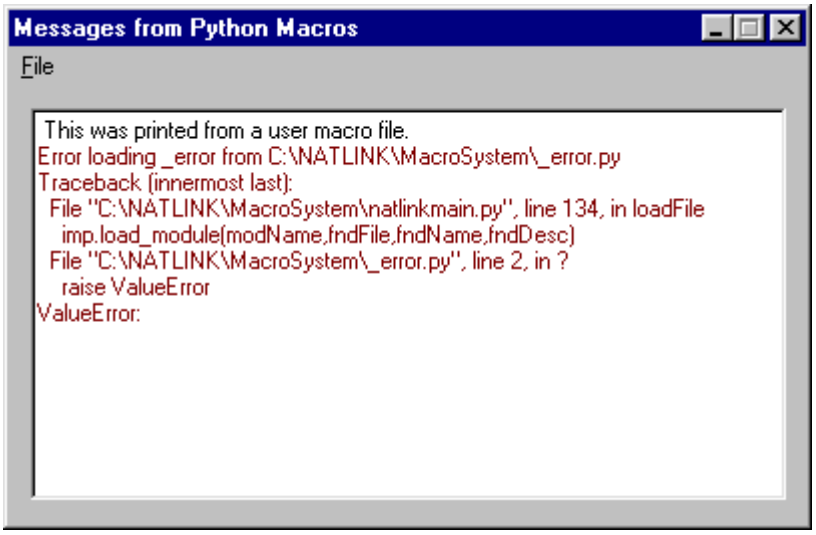

Figure 8 - Sample Output from Python Macro System

This technique of redirecting output makes it possible to both catch exceptions raised from user specified macro files and also to allow users the opportunity of debugging the macro files using "print" statements. Simply clicking the mouse on the “X” in the upper right hand corner can close the message window. It will automatically reappear if, and only if, there is additional output from a Python macro script.

3.3.4 Sample Macro File

As mentioned before, it is possible to include almost any code in a user macro file. However, there is a canonical format for macro files that contain grammars and code to handle recognition results. This code is based on using the GrammarBase class presented earlier. This class is a wrapper around the underlying NatLink grammar object (natlink.GramObj) thet makes it more convenient for users to write simple macro files.

The following code is taken from _sample1.py, which is included in the NatLink package, and represents the simplest example of a canonical macro file.

```
import natlink
from natlinkutils import *

class ThisGrammar(GrammarBase):

    gramSpec = """
        <start> exported = demo sample one;
    """

    def initialize(self):
        self.load(self.gramSpec)
        self.activateAll()

    def gotResults_start(self,words,fullResults):
        natlink.playString('Heard macro "sample one"{enter}')

thisGrammar = ThisGrammar()
thisGrammar.initialize()

def unload():
    global thisGrammar
    if thisGrammar: thisGrammar.unload()
    thisGrammar = None
```

Figure 9 - Simple Example of User Macro File

When this macro file is loaded into Python by the macro system controller, an instance of the ThisGrammar class is created. Then, a sample grammar is loaded into Dragon NaturallySpeaking and activated for recognition.

Later, if the user says “demo sample one”, a callback will be made into the gotResults_start() function defined in this file. This will result in sending the string 'Heard macro "sample one"', followed by a character return, to the currently active application window.

Notice that the side effect of loading this file is the creation of an instance of the ThisGrammar class. This instance will remain existence until the unload() function is called; a call that will be made from the macro system controller automatically when this file is unloaded or reloaded.

Multiple user macros can be defined and loaded at the same time. They all operate independently of each other, and there is no interaction between user macro files (unless they define the same exact grammar, in which case it is difficult to predict which macro file will receive control when that grammar recognized).

The macro system controller handles all of the intricacies of loading and unloading the user macro files. If a new user macro file is created, the macro system controller will automatically load it the next time a qualifying event occurs (like the start of a new utterance or toggling the microphone state). If a user macro file is deleted, then the macro system controller automatically unloads it at the next qualifying event.

Within a user’s macro file, the user is free to include whenever Python code they would like. The simple example shown above only sends keystrokes to the current application. But there is no reason that more complex Python code could not be used, including code which accesses other applications through, for example, OLE automation interfaces.

3.3.5 Summary of Python Macro System

In this chapter, I have shown how the NatLink extension module is also configured as a Dragon NaturallySpeaking compatibility module. This allows Dragon NaturallySpeaking to automatically load the NatLink code whenever NatSpeak starts. When used as a compatibility module, NatLink runs a macro system controller, written in Python, that coordinates the loading and unloading of user defined macro files (each also written in Python).

With the addition of the compatibility module infrastructure and macro system controller, I have met the number one project goal: To develop an add-on for Dragon NaturallySpeaking that would allow existing Dragon NaturallySpeaking users to develop command and control macros using Python as the interpretive scripting language.

In addition, by using the Direct Speech Recognition API in NatLink, instead of the more limited Voice Command API (that is used by Dragon NaturallySpeaking’s built-in macro system), I am able to include the full expressive power of the grammar capabilities of the speech recognition system.

Finally, I believe that I overcome the existing limitations of the grammar development process by allowing users to create and edit macro files using any third party editor, and have those files automatically be reloaded by the macro system controller whenever changes are detected.

3.4 Putting it All Together

From the beginning, my intention in the development of NatLink was to make the system publicly available to customers of Dragon NaturallySpeaking. Many of the features that I included in NatLink were designed specifically based on feedback from users of Dragon NaturallySpeaking with respect to aspects of the product that they find useful, or wanted added.

3.4.1 The Mouse Movement Experiment

The first critical test of NatLink was when I attempted to add mouse and cursor movement to Dragon NaturallySpeaking.

The ability to move the mouse by voice was a feature that was first introduced in DragonDictate for Windows (DDWin) in 1994. [12] This is a critical feature of a speech recognition system if it is to be used by people with limited or no use of their hands.

With DDWin, a user could control mouse with command sequences like “mouse up”, “faster”, “left”, “stop”. When the user first set "mouse up", the mouse cursor would start moving slowly towards the top of the screen. When the user said "faster", the mouse cursor would increase its speed. The command "left" would change the direction of the moving mouse cursor, sending it towards the left side of the screen. When the user felt that the mouse cursor was positioned properly, he could say "stop" and then, "button click" to simulate a button press at its current location.

Dragon NaturallySpeaking does not include this style of mouse movement. Instead, the only mechanism for moving the mouse included in NatSpeak is the MouseGrid [13] (also available in DDWin as an option). My first big test for the NatLink macro system was to see whether I could create a Python macro that would behave like the mouse movement feature of DragonDictate for Windows.

Implementing the grammars was easy. The grammars for controlling the mouse are simple phrases, but they do require that the system be put in a modal state. In other words, the phrase "mouse up" should be active all of the time, but once the mouse has started moving the speech recognition system should be limited to only those commands that directly control the mouse.

It turns out the Dragon NaturallySpeaking does support the concept of limiting recognition to only one active grammar, disabling all other grammars loaded into the system. This feature is used in NatSpeak when you issue the command "go to sleep". That command disables all grammars in Dragon NaturallySpeaking except for the special command "wake up", effectively putting the system to sleep.

The ability to limit recognition to a single grammar is not exposed in the scripting language of the Professional Edition of Dragon NaturallySpeaking. But it is accessible through the Direct Speech Recognition API, so I was easily able to add it to NatLink. A method on the natlink.GramObj class called setExclusive() can be used to tell Dragon NaturallySpeaking that that particular grammar should be activated exclusive of all other grammars in the system.

The next problem was a little trickier. NatLink already includes the ability to position the mouse using the natlink.playEvents() function. However, for mouse movement I needed to be able to move the mouse continually across the screen. Using a loop was out of the question since including a loop in the Python macro file would prevent additional recognition results from being handled.

To solve this problem, I added an additional callback capability to the NatLink Python extension module. This callback, called natlink,setTimerCallback(), allows you to specify a reference to a Python function that is an called every N milliseconds (where N is specified as a parameter to natlink.setTimerCallback). This callback acts similar to recognition results callbacks. Every N milliseconds, my referenced Python routine would be executed allowing me to move the mouse cursor a small increment on the screen.

With these two changes, I was able to put together a Python file that mimics the mouse movement behavior of DragonDictate for Windows. This file, called_mouse.py, is now included as a standard part of the NatLink package.

I should mention that the mouse movement capabilities resulted in the only outside code contribution to NatLink. Jonathan Epstein submitted an enhancement to the NatLink Python extension module that allows a Python macro file to display an icon in the Windows tray (usually on the bottom right hand corner of the screen). Jonathan added this functionality so that an animated icon could be displayed while the mouse or caret was moving, to remind the user that the system was in a limited recognition state. This user submitted enhancement was added to the official NatLink code base in April of 2000.

3.4.2 Going Public

As I mentioned earlier, I initially releases the NatLink package to the world as an open source, open license download from my personal Dragon NaturallySpeaking website [5] along with a copyright and installation instructions.

The original release of NatLink included approximately six sample macro files that demonstrated the basics of creating a macro file. The six sample files varied in difficulty from _globals.py, which demonstrated some simple grammars to turn the microphone off by voice, and also to simulate the effect of the built-in "sleeping" feature, to _mouse.py, described earlier. (Later releases included more sample macro files.) Basic documentation was also included, both in the source code and in a reference file listing all the available functions and classes in the natlink extension module.

Because enabling the Python macro system requires creating entries in the Windows registry, I added an additional C++ program, called EnableNL, to the package. EnableNL is designed to search through the Windows registry for installations of Dragon NaturallySpeaking, and offer the user the opportunity to selectively enabled or disable the Python macro system for each installation. EnableNL makes the necessary Windows registry entries to enable NatLink as a NatSpeak compatibility module, and also adds the NatLink directories to the Python path.

As of the writing of this paper, the NatLink system has been available for over a year and a half. I have had six official releases, as I occasionally enhance the system with new functionality. I do use it myself, both at home and at work, and e-mails that I have received lead me to believe that a number of Dragon NaturallySpeaking customers also use NatLink.

4 Results

As part of the research for writing this paper, I attempted to collect feedback from people who have tried to use NatLink, whether or not they were successful. The goal for collecting this information was to understand how well NatLink was received both among Dragon NaturallySpeaking users and among Python programmers.

Based on advertising on my own website, as well as posting messages in the major mailing lists for users of speech recognition systems [14] [15] [16], I received feedback from 36 people who either currently use NatLink, or have tried to use NatLink in the past. It is important to understand that this is a self-selecting group that may not represent a statistically significant population by any measure. But the feedback from this small set of NatLink customers offers some insights into the success of the NatLink project, and the acceptance of Python as a general-purpose macro language.

4.1 Who Uses NatLink and Why

Of the people that responded to my survey, almost everyone was already a user of Dragon NaturallySpeaking. One person was not a Dragon NaturallySpeaking user, but is using NatLink to develop a programming by voice package. One person uses a competitive speech recognition product, but is considering switching to Dragon NaturallySpeaking because of NatLink.

No one who either responded to my survey or had sent me e-mail in the past about NatLink was a Python programmer who wanted to add speech recognition to his or her Python application. Some of this may have to do with my method of advertising. Most people learn about NatLink from one of the voice users group, or from visiting my Dragon NaturallySpeaking website. I have announced NatLink in the past on the Python newsgroup; but, I have never received any feedback or indication that there was any interest from the existing Python community. (Maybe that will change based on this paper.)

Of the people who gave a reason for using NatLink, the most often cited reason was to avoid having to purchase the Professional Edition (mentioned by seven of the respondents). The Professional Edition can cost as much as $500 more than the Preferred Edition, and for most people its only advantage is the ability for users to write their own voice macros using Dragon NaturallySpeaking's built-in scripting language. Clearly, the fact that NatLink worked with all versions of Dragon NaturallySpeaking made NatLink attractive as an alternative to spending the extra money for the Professional Edition. (Although I should mention that in one case, the respondent did not cite the cost of the Professional Edition, but the fact that the Professional Edition was not available in Dutch.)

As it happens, it would have been very hard to make NatLink only work with the Professional Edition of Dragon NaturallySpeaking. The programming interfaces that NatLink takes advantage of are available in all versions of Dragon NaturallySpeaking. To limit NatLink would mean adding specific code that would detect the version of Dragon NaturallySpeaking, and then disable functionality when an edition other than the Professional was detected. Although I could have added such code, it would have been very easy to circumvent given that the entire NatLink project is open source.

It is also interesting to note that when I originally published NatLink, one of the Dragon Systems' value-added resellers complained that I was undercutting his business by providing a means for customers to avoid purchasing the (for the reseller) more profitable Professional Edition. At the time, I responded that I did not expect a large impact on Professional Edition sales, given the complexity of writing NatLink macros. And that there was nothing in NatLink that could not be duplicated by an enterprising developer using the publicly available developer tools for Dragon NaturallySpeaking.

That said, at least three of the NatLink users who responded to my survey were value-added resellers of Dragon NaturallySpeaking. And in one case, he sees the ability to make additional revenue by producing macro packages for sale based on NatLink.

The next most often cited reason for people who want to use NatLink was its ability to write more complex macros then was possible using the Dragon NaturallySpeaking Professional Edition scripting language. This is encouraging, since it was for this reason that I originally undertook the development of NatLink.

A number of respondents to my survey do not actually use NatLink. At least seven people indicated that they had downloaded NatLink and tried it, but do not currently use it. Some complained that they just did not have enough time to study the system.

But more often than not, the people who gave up on NatLink just found it too difficult to figure out. I can't say that I am very surprised by this fact. Python is a programming language, and NatLink is not very polished. There are no graphical wizards to guide users through the creation of macros, and very little help when your Python macro generates an error message that you do not understand.

On the other hand, it was never my intent to create an easy to use system. My goal was to develop something that was more powerful than the (possibly easier to use) macro system that is included with Dragon NaturallySpeaking Professional Edition.

4.2 What People Do with NatLink

Almost everyone who sent me feedback uses NatLink primarily to write Python macros for Dragon NaturallySpeaking. Enabling users to develop Python macros was, of course, the first goal of this project. So it is encouraging to realize that I have achieved that objective.

Only three respondents indicated that they use NatLink to control Dragon NaturallySpeaking itself (my second goal for the project). No one claimed to use NatLink to speech enable Python applications (my third goal for the project), except for the occasional person who experimented with the sample applications.

The most often cited use for NatLink was to write macros to help enable programming by voice. (Although much of that was intended use, not actual use.) At least eight respondents specifically mention programming by voice as a goal for their use of NatLink. Five other respondents mentioned command and control of Emacs, which is probably for programming purposes.

I believe that the NatLink package appeals to programmers specifically, perhaps because of its power and flexibility, and perhaps because it uses an actual programming language. Otherwise, one might draw the conclusion that a significant fraction of the users Dragon NaturallySpeaking purchasing primarily for programming by voice; but no marketing survey information has indicated that this is the case.

Based on reading between the lines of the respondents e-mails, and looking at the sample macros which people sent me, I have to conclude that at least one-third of the current users of NatLink are only modifying the sample

programs and not developing anything more sophisticated. Another third of the respondents are writing moderately sophisticated macros that involve more than a couple of single line modifications to an existing sample program.

Of the people that only write simple macros, the most often cited reason had to do with the complexity of learning to do more, especially in the face of only limited documentation. People learn by copying and modifying existing examples (that's how I originally learned Python). Dumping a bundle of documentation on people's desks does not seem to work as well as giving them good examples of the types of things they may be interested in doing.

I will be the first to admit that there are not enough examples of good macros in the current NatLink package. Most of the macros that I included, were included as learning exercises rather than as examples of complete, useful grammars. This lack of more detailed examples may be limiting people's ability to effectively use NatLink for sophisticated tasks.

On the other hand, some of the NatLink users have done extremely sophisticated systems based around NatLink. One user, for example, uses NatLink to manipulate databases stored in Microsoft Word files through OLE Automation. The sample code he sent to me indicates that he is doing OLE Automation calls into Microsoft Word from Python using Mark Hammond's OLE Automation support for Python to extract information stored in structured Word documents.

One user has even gone so far as to invent his own grammar format, and uses NatLink as an infrastructure for parsing his grammar files, and loading those grammars into Dragon NaturallySpeaking.

4.3 How Python was Received

Of particular interest to the Python community may be my conclusions on how well Python was received by the users of NatLink.

None of the NatLink users who responded to my request for information indicated that they knew Python before they picked up NatLink. This means that all 36 respondents, and probably all of the other people who use NatLink, first had to learn Python.

The reaction to Python among NatLink users was mixed, but mostly positive. Most people who offered an opinion claimed that they liked Python, and a significant fraction of people indicated that they have now started to use Python for other activities (besides NatLink macros). A couple of people did not like Python for various reasons; one cited Python's lack of compile time type checking, and another complained about Python's poor integration with Windows.

The lack of good documentation was an often-cited issue. In this case I think that people were complaining both about the limited documentation for Python, and the even more limited documentation for NatLink.

But in general, I think the results of the NatLink project were very positive for the Python community. A significant fraction of the NatLink users were able to pick up enough Python to become productive writing NatLink macros. And NatLink seems to have been a successful missionary for the Python religion.

4.4 Conclusion

When I started the NatLink project, I set out three goals that I wanted to achieve with this development effort.

1. To develop an add-on for Dragon NaturallySpeaking that would allow existing Dragon NaturallySpeaking users to create command and control macros that (a) include the full expressive power of the grammar capabilities of the speech recognition system, (b) use Python as the interpretive scripting language, and (c) overcome the existing limitations of the grammar development process.

2. To develop a Python extension module for controlling Dragon NaturallySpeaking that would allow customers to write Python programs which manipulate the vocabulary, create grammars, perform recognition and process recognition results.

3. To allow Python developers to add speech recognition to their Python programs.

This paper discusses how I met the first goal for this project. I have both presented an overview of the implementation of NatLink and included an analysis of the acceptance of NatLink among the users of Dragon NaturallySpeaking based on feedback. This project is especially interesting to the Python community because it sheds some light on the acceptance of Python as a general-purpose macro language, outside the computer science field.

5 References

[1] M. Lutz. Programming Python. O'Reilly & Associates, 1996.

[2] Based on US retail sales revenue as reported by PC Data (http://www.pcdata.com) for the three years 1997, 1998 and 1999.

[3] "Scripting Language Reference". Online help, Dragon NaturallySpeaking Professional Edition 5.0. Lernout & Hauspie Speech Products N.V. (d/b/a Dragon Systems), 2000.

[4] Mark Hammond, Andy Robinson. Python Programming on Win32. O'Reilly & Associates, 1996. Also http://starship.Python.net/crew/mhammond/.

[5] Joel Gould. "Python Macro System". http://www.synapseadaptive.com/joel/PythonMacroSystem.html.

[6] "SAPI Basics". Microsoft Speech API 4.0 documentation, Microsoft Developer Network. Microsoft, 1999.

[7] Mike Rozak. *An Overview of the Microsoft Speech API*. http://msdn.microsoft.com/library/backgrnd/html/overviewapi.htm.

[8] "About Microsoft COM". Microsoft Developer Network Online. http://www.microsoft.com/com/about.asp.

[9] "Dragon Systems Developer Den". Dragon Systems web site. http://developer.dragonsys.com/.

[10] "SWIG, Simplified Wrapper and Interface Generator". http://swig.sourceforge.net/index.html.

[11] David Abrahams. "Py_cpp system for interfacing C++ code with Python". http://people.ne.mediaone.net/abrahams/downloads/py_cpp.html.

[12] DragonDictate for Windows is no longer offered by Dragon Systems. It is, however, still available from some resellers. See, for example, "DragonDictate". Synapse Adaptive web site. http://www.synapseadaptive.com/NaturallySpeaking/dictate.html.

[13] Patri Pugleise and Joel Gould. "Voice Controlled Cursor Movement". US Patent 5,818,423. http://www.delphion.com/details?pn=US05818423__.

[14] "VoiceCoder" mailing list. http://www.egroups.com/group/VoiceCoder.

[15] "VoiceGroup" mailing list. http://www.egroups.com/group/VoiceGroup.

[16] "Natpython – Dragon NaturallySpeaking Python users list" mailing list. http://harvee.billerica.ma.us/mailman/listinfo/natpython.

[17] "VoiceCode Programming by Voice Toolbox". http://ii2.ai.iit.nrc.ca/VoiceCode/.

[18] Joel Gould. "NatLink: A Python Macro System for Dragon NaturallySpeaking". http://www.synapseadaptive.com/joel/NatLinkTalk.ppt.

[19] "Secrets of VDct: Replacing dictation components in Dragon NaturallySpeaking". http://www.synapseadaptive.com/joel/VoiceCoders.ppt.