Sphinx 3 for the Java™ platform Architecture Notes

Overview

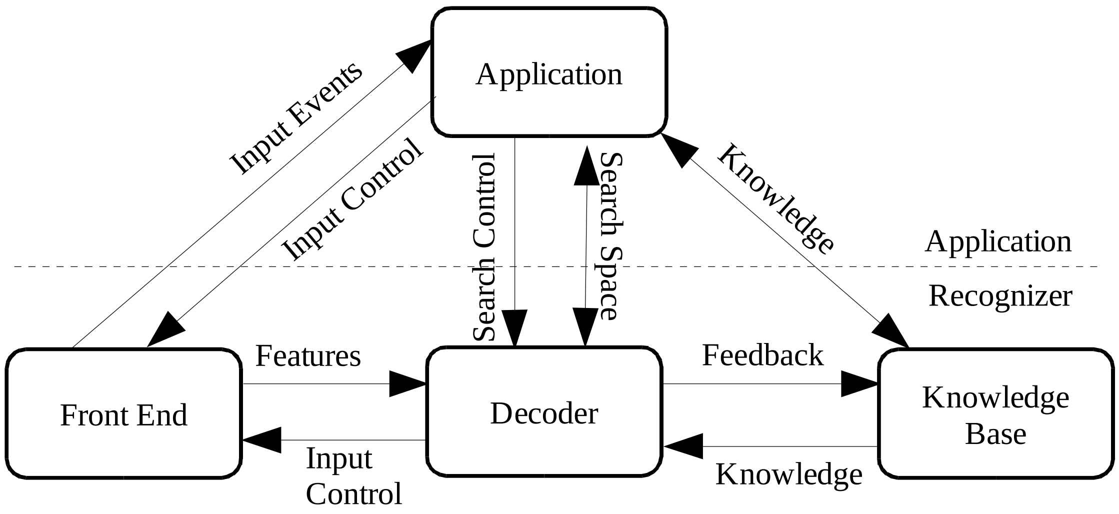
This is a living document that describes the architecture of Sphinx 3 for the Java™ platform (sphinx3j). The architecture was derived as a result of the face−to−face meetings of the sphinx3j team at Sun Microsystems Laboratories on February 21st and 22nd, and April 18th and 19th, 2002. The members of the sphinx3j team present at the meetings are as follows:

Evandro Gouvea,	Carnegie Mellon University
Philip Kwok,	Sun Microsystems Laboratories
Paul Lamere,	Sun Microsystems Laboratories
Bhiksha Raj,	Mitsubishi Electronic Research Laboratories
Rita Singh,	Carnegie Mellon University
Willie Walker,	Sun Microsystems Laboratories (Manager/PI)
Peter Wolf	Mitsubishi Electronic Research Laboratories

Since this is a living document, we will update it as work on sphinx3j progresses [[[WDW: Mental note: rewrite this whole thing... :−)]]].

High Level Architecture

The high level architecture for sphinx3j is relatively straightforward. As shown in the following figure, the architecture consists of the front end, the decoder, a knowledge base, and the application.



The front end is responsible for gathering, annotating, and processing the input data. In addition, the front end extracts features from the input data to be read by the decoder. The annotations provided by the front end include the beginning and ending of a data segment. Operations performed by the front end include preemphasis, noise cancellation, automatic gain control, end pointing, Fourier analysis, Mel spectrum filtering, cepstral extraction, etc.

The knowledge base provides the information the decoder needs to do its job. This information includes the acoustic model and the language model. The knowledge base can also receive feedback from the decoder, permitting the knowledge base to dynamically modify itself based upon successive search results. The modifications can include switching acoustic and/or language models as well as updating parameters such as mean and variance transformations for the acoustic models.

The decoder performs the bulk of the work. It reads features from the front end, couples this with data from the knowledge base and feedback from the application, and performs a search to determine the most likely sequences of words that could be represented by a series of features. The term "search space" is used to describe the most likely sequences of words, and is dynamically updated by the decoder during the decoding process.

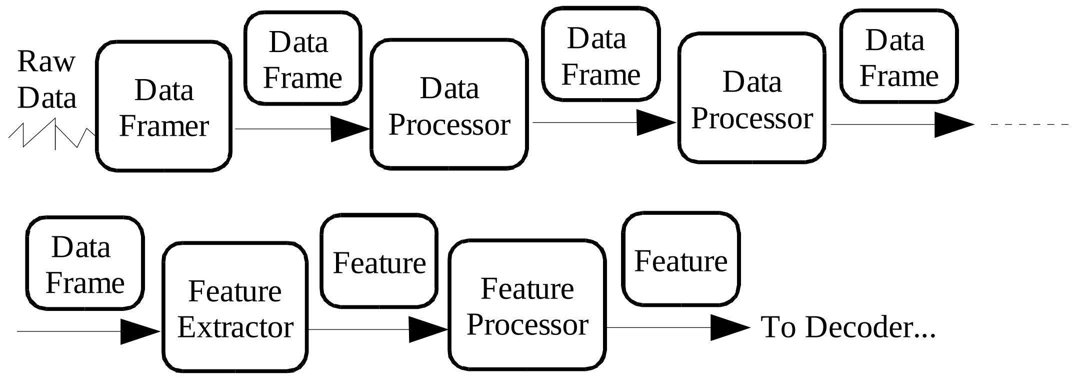
Unlike many speech architectures, the sphinx3j architecture allows the application to control various features of the speech engine, permitting more sophisticated speech application development. As depicted in the previous figure, the application can receive events from the front end and can also provide some level of control over the front end. The type of control can be as simple as turning the audio input on or off, but may also include more sophisticated operations.

During the decoding process, the application may also receive events from the decoder while the decoder is working on a search. These events allow the application to monitor the decoding progress, but also allow the application to affect the decoding process before the decoding completes. Furthermore, the application can also update the knowledge base at any time.

The following sections describe each piece of the high level architecture in more detail.

Front End

The front end can be broken in several simple pieces as depicted in the following illustration [[[WDW − needs to be updated to reflect the current implementation]]]:



Examining the illustration left to right, top to bottom, the front end first reads in raw data via the Data Framer. This data can arrive by a stream, a file, or any other means. In addition, while the data is typically audio, there can be a number of simultaneous data sources, such as both video and audio. The Data Framer packetizes the data into annotated Data Frames. These Data Frames contain information about the data packets, including information such as if the data is the beginning or end of a segment.

The front end passes the Data Frames to a series of Data Processors. The Data Processors perform successive modifications to the Data Frames, such as automatic gain control, noise cancellation, down/up sampling, and preemphasis.

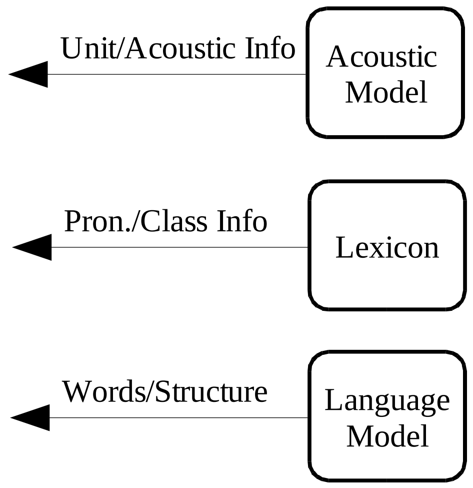
Once the preprocessing of a Data Frame is complete, the front end passes the Data Frame to a Feature Extractor. The Feature Extractor extracts the feature(s) necessary for the decoder to do its work. For audio, this typically involves obtaining cepstral and delta cepstral information, but can be anything the decoder accepts. Furthermore, the resulting Feature is not necessarily restricted to one data type. For example, the Feature may contain information for both audio and video.

The front end then passes Feature frames to to a series of Feature Processors. The Feature Processors may perform a number of operations including end pointing, noise cancellation, and cepstral mean calculation.

When the front end has completed processing a Feature, it passes it to the decoder, which is described later on this document.

Knowledge Base

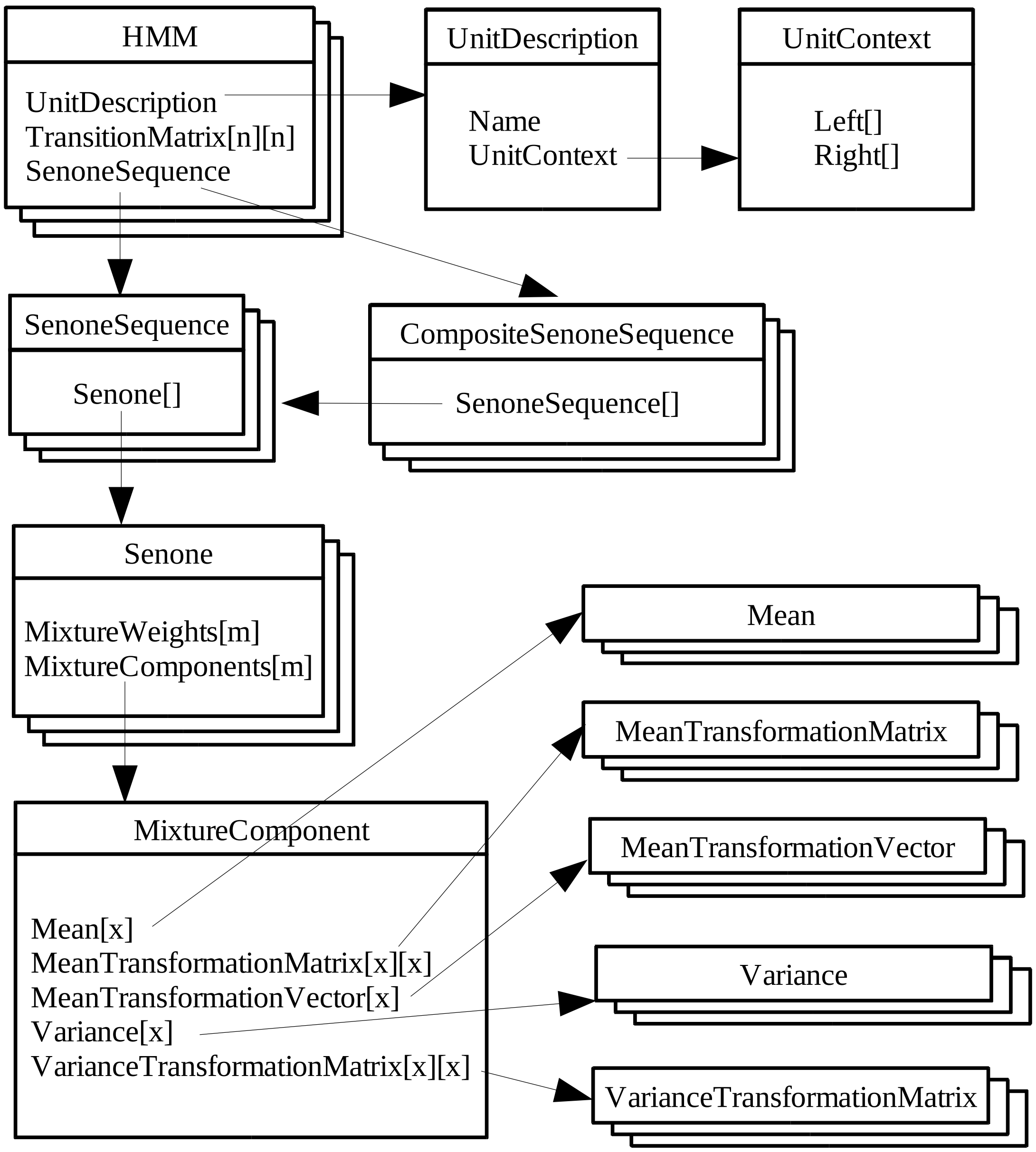
The knowledge base provides the information the decoder needs to do its job. Typically, the knowledge base consists of the acoustic model, the language model, and the lexicon.



The acoustic model provides the knowledge for converting frame sequences into unit hypotheses, the lexicon provides the pronunciation (unit sequence) and part−of−speech classification for words, and the language model provides the knowledge for converting unit sequences into word and word sequence hypotheses.

Acoustic Model

In sphinx3j, the acoustic model consists of a set of left−to−right Hidden Markov Models (HMMs), with one HMM per unit. The units typically represent phones in a triphone context. The following diagram illustrates the definition of the HMMs of the acoustic models in sphinx3j:



In the drawing, any object shown as a "stack of cards" represents a shared pool of object instances. For example, there is a shared pool of Senones that are referred to by SenoneSequences. The set of shared pools allows the sphinx3j HMMs to support concepts known as "state tying" and "parameter tying." With state and parameter tying, the HMMs can share a large variety of features. There are at least two reasons for doing tying: the primary reason is to get sufficiently trained models, and the second reason is to help reduce the number of calculations during the search.

Each state of the HMMs in sphinx3j are called "Senones." The Senones are based on probability density functions (pdfs). As shown in the illustration, the pdfs are continuous Gaussian Mixtures. The exact type of pdf, however, does not have to be a Gaussian Mixture. Instead, the pdf merely needs to be able to take a Feature from the front end and return a score. As illustrated by the MixtureComponent, each GaussianMixture obtains its parameters from several shared pools (e.g., the GaussianMixture parameters are tied).

Several HMMs can share the same senone sequence. As a result, the HMMs point to a SenoneSequence that comes from a shared pool of SenoneSequences. In some HMMs [[[WDW − give example]]], the SenoneSequence of an HMM is a special CompositeSenoneSequence as illustrated.

Lexicon

The lexicon serves two purposes: first, it provides unit sequences for words; second, it provides optional classification of words (e.g., noun, verb, etc.). The meaning of a unit varies depending upon the recognition task. For example, for isolated word recognition, the unit could be a whole word. For large vocabulary continuous recognition, the unit could be a triphone.

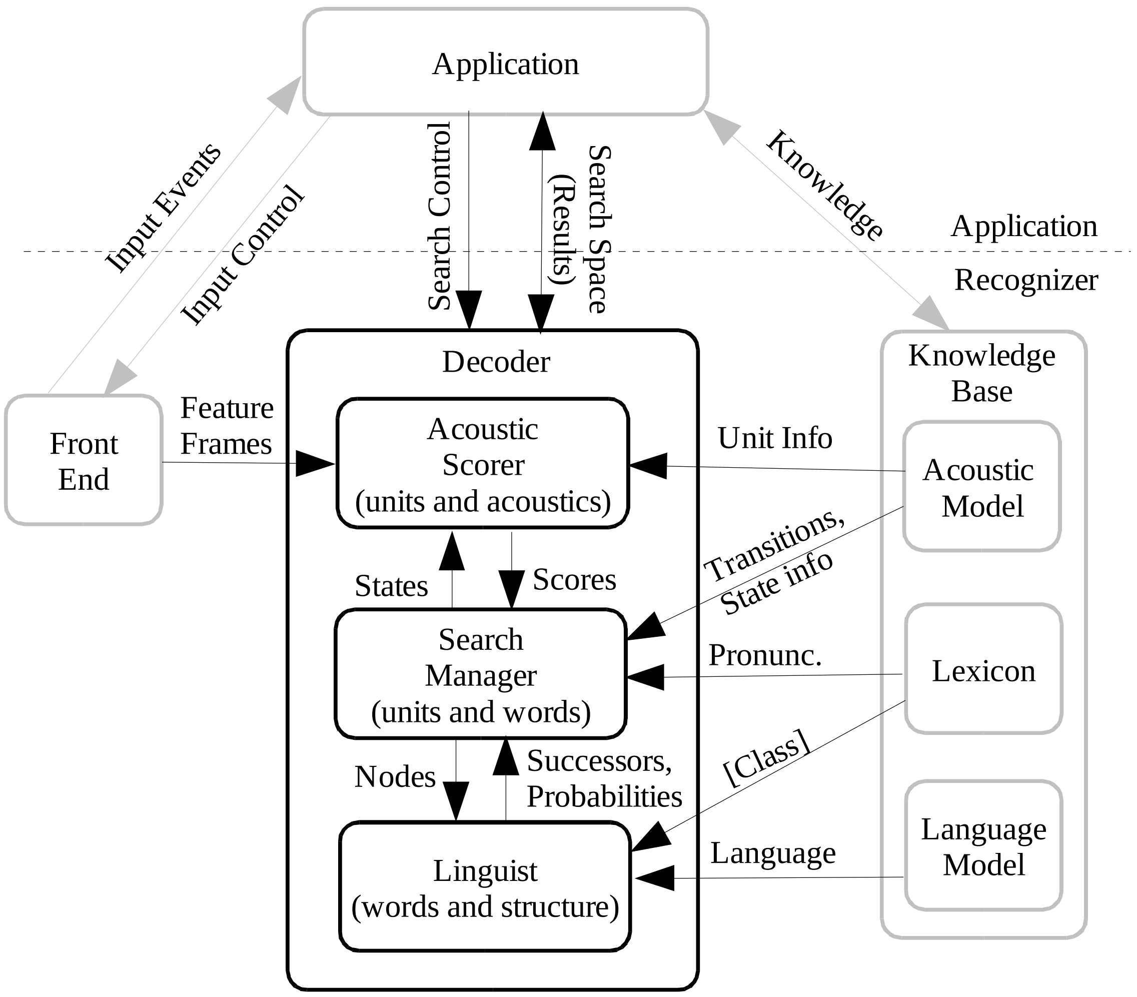
Language Model

We’re still creating the language model portion of sphinx3j, but at a minimum, sphinx3j will support three type of language models. The simplest type of language model is no language model at all, and is used in the case of isolated word recognition. The next common language model type is a context free grammar that can used in speech applications based on command and control. The final language model type is based on n−gram grammars, and is typically used for free form speech.

Since we’re still working on the format of the language model, this section is expected to change over time.

Decoder

We are still working on the architecture for the decoder. The current decoder in Sphinx 3 combines a LexTree with an n−Gram language model. We feel this is too specific to include in sphinx3j, and are working on a flexible model that permits the Sphinx 3 model as well as other types of searches. At a minimum the other types of searches we are looking at include support for isolated word recognition, context free grammars, and n−gram language models.



The basic idea behind this model is to allow for more flexibility when it comes to the search strategy. [[[WDW: For now, refer to search.txt.]]]