Creating Annotation Tools with the Annotation Graph Toolkit

Kazuaki Maeda, Steven Bird, Xiaoyi Ma, and Haejoong Lee

Linguistic Data Consortium, University of Pennsylvania
3615 Market Street, Philadelphia, PA 19104-2608, USA
{maeda, sb, xma, haejoong}@ldc.upenn.edu

Abstract

The Annotation Graph Toolkit is a collection of software supporting the development of annotation tools based on the annotation graph model. The toolkit includes application programming interfaces for manipulating annotation graph data and for importing data from other formats. There are interfaces for the scripting languages Tcl and Python, a database interface, specialized graphical user interfaces for a variety of annotation tasks, and several sample applications. This paper describes all the toolkit components for the benefit of would-be application developers.

1. Introduction

Linguistic databases are widely used in the scientific study of language and in language-technology research and development. Many software tools have been developed to support the creation of annotated linguistic databases, and some of them are documented in (Bird and Harrington, 2001). Bird and Liberman have developed a model for expressing the logical structure of linguistic annotations, and have demonstrated that it can encode a great variety of existing annotation types (Bird and Liberman, 2001). An annotation graph is a directed acyclic graph where edges are labeled with fielded records, and nodes are (optionally) labeled with time offsets. Figure 1 shows an annotation graph for a fragment of the TIMIT corpus (Garofolo et al., 1986).

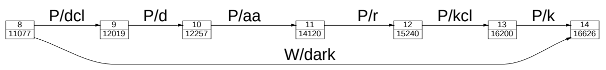


Figure 1: Annotation Graph for a fragment of TIMIT data

Annotation graphs have opened up new possibilities for creation, maintenance and search, and lead to new annotation tools with applicability across the text, audio and video modalities. Annotation graphs are also permitting existing annotation tools – each with large user-bases – to be made fully interoperable.

The Annotation Graph Toolkit (AGTK) is a collection of software supporting the development of annotation tools based on the annotation graph model. AGTK includes application programming interfaces for manipulating annotation graph data and for importing data from other formats, a database interface, wrappers for scripting languages, specialized graphical user interfaces for annotation tasks, and sample applications.

This paper is addressed to developers of linguistic annotation tools. We begin with a high-level overview of the tool architecture (§2), before presenting the most important features of the annotation graph library (§3), the file I/O library (§4), and their scripting language interfaces (§5). In §6 we discuss our approach to tool creation, involving rapid high-level programming in scripting languages, interfaced to stable and optimized C++ libraries. We describe our model of inter-component communication, and our approach to GUI design, in which we create many special-purpose tools that are maximally ergonomic for the task at hand. All software, interface definitions, configuration files and data samples are available under an open source license.

2. Architecture

Existing annotation tools are based on a two level model. The system we discuss in this paper is based on a three level model, in which annotation graphs provide a logical level independent of application and physical levels. This is the three-level model of modern database systems (Abiteboul et al., 1995) applied to linguistic databases in support of data independence, data reuse, and software integration. The application level represents special-purpose tools built on top of the general-purpose infrastructure at the logical level.

The toolkit is comprised of several components, structured according to the architecture shown in Figure 2. This model permits applications to abstract away from file format issues, and deal with annotations purely at the logical level, through the annotation graph API. Annotation tools provide graphical user interface components both for signal visualization and for annotation, and the communication between these components is handled by an extensible event language.

As with other recent architectures for language technologies, e.g. (Allen et al., 2000), the architecture consists of a set of loosely-coupled, heterogenous components that communicate with each other by exchanging messages. This design has three benefits. First, components can be implemented in the most opportune language, and wrappers can easily be added to legacy and third-party components. Second, message traffic can be logged to facilitate error diagnosis and to permit inter-component and human-computer interactions to be replayed and analyzed. Third, message passing permits the transport protocol to be separated from the communication content. The former is enforced by the infrastructure, while the latter is extremely flexible.

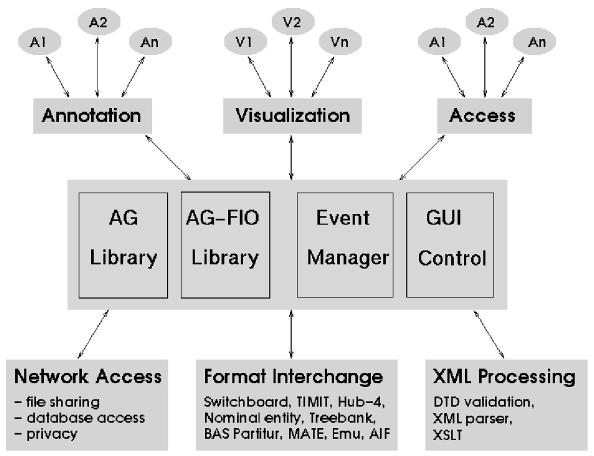


Figure 2: Architecture for Annotation Systems

3. The Annotation Graph Library

The annotation graph library (libag) is implemented in C++,[1] and provides functions for creating, deleting, modifying and searching the following annotation graph objects: *AGSet*, *AG*, *Annotation*, *Anchor*, *Timeline*, *Signal*, *Feature* and *Metadata*. These objects are related to each other according to the object model shown in Figure 3. The various objects will be explained in more detail as we describe the API.

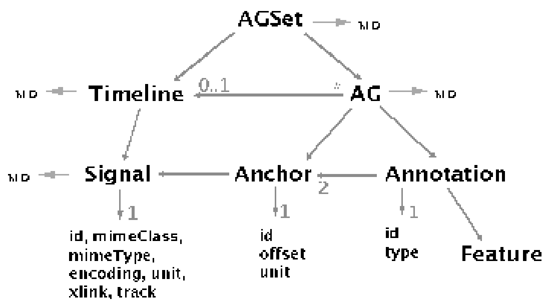


Figure 3: The AG Object Model

The annotation graph library also keeps indexes for the *Annotation*, *Anchor*, *Feature* and *Metadata* types so that searches can be done efficiently.

The API provides access to internal objects – signals, anchors, annotations, etc – through the use of identifier strings. The types of annotation graph identifiers include `AGSetId`, `AGId`, and `AnnotationId`.

3.1. The structure of annotation graph identifiers

All identifiers are represented as strings. The internal structure of an identifier can best be understood in terms of the object hierarchy in Figure 3. Annotation graph identifiers are *fully-qualified*: given an object identifier, all of the ancestor objects can be discovered simply by inspecting the identifier. For instance, an anchor `"Timit:AG1:Anchor2"` belongs to the annotation graph `"Timit:AG1"`, which in turn belongs to the AGSet `"Timit"`.

Internally, the annotation graph library maps these string identifiers to object references. A consequence of this design is that annotation graph objects can be referenced from scripting languages using human-readable names. The use of fully-qualified identifiers also reduces the risk of collision – the accidental re-use of the same identifier in different places – which can have unpredictable consequences.

3.2. Annotation graph API functions

This section explains some typical API functions. The complete IDL definition of the AG-API is available online [`http://www.ldc.upenn.edu/AG/`].

3.2.1. AGSet and AG functions

An AGSet is an object which contains a set of annotation graphs. Typically, an AGSet corresponds to a corpus, but it might also correspond to a user-specified selection from a corpus, or to a selection spanning several different corpora. The first thing to do in working with annotation graphs is to create an AGSet object to hold them.

CreateAGSet. This creates an empty AGSet with a specified AGSetId, and returns the AGSetId:

```
AGSetId CreateAGSet(AGSetId agSetId);
```

Once an AGSet is created, Timelines, Signals, AGs, and then Annotations and Anchors can also be created. Certain functions can then be called on these data types, for example, to test for their existence or to delete them. An AGSet can be deleted by using `DeleteAGSet`. Its existence can be tested by using `ExistsAGSet`.

CreateAG. This creates an AG and returns the AGId; it throws an AGException if the `id` does not contain a valid AGSetId, or if the timeline does not exist:

```
AGSet CreateAG(Id id);
AGSet CreateAG(Id id, TimelineId timelineId);
```

The parameter `id` may be either an AGSetId or an AGId. If it is an AGSetId, an AGId will be assigned to the new AG. However, if it is an AGId, the library will try to use the supplied id. If this id is unavailable, it will assign a new AGId.

The timelineId is the id of the timeline with which the new AG will be associated. An AG can be created without being associated to any timeline.

3.2.2. Timeline and Signal functions

A "timeline" is a collection of synchronized signals, such as separate audio and video recordings of the same event, or a multichannel recording of a conference call. The key defining property of a timeline is that the offsets into its signals are intertranslatable; any offsets into any one of the associated signals can be mapped to an offset into any of the others. Whenever multiple signals or channels are used in an annotation task, they are assigned to a distinct `Signal` object.[2] Multiple synchronized signals are grouped into a single `Timeline` object.

[1]The library has recently been ported to Java.

[2] An exception to this is the situation of a multichannel recording in which the different tracks are not discriminated in the annotation task; here they may be treated as a single `Signal` object.

The functions `CreateTimeline`, `ExistsTimeline` and `DeleteTimeline` will now be explained.

CreateTimeline. This creates a new *Timeline* and returns the `TimelineId`:

```
TimelineId   CreateTimeline(Id id);
```

`Id` is `AGSetId` or `TimelineId`. In either case, the *AGSet* to which the *Timeline* belongs must already exist; otherwise an exception will be thrown. For example:

```
/* Create an AGSet with id "Timit" */
AGSetId agSetId = CreateAGSet("Timit");

/* Create a new timeline */
TimelineId timeline1 = CreateTimeline(agSetId);

/* Create another timeline */
TimelineId timeline2 =
             CreateTimeline("Timit:Timeline2");

/* The following causes an exception
since AGSet "CallHome" does not exist */
TimelineId timeline3 =
             CreateTimeline("CallHome");

/* The following also causes an exception */
TimelineId timeline4 =
             CreateTimeline("CallHome:Timeline2");
```

ExistsTimeline. This tests for the existence of the specified *Timeline*, and returns true if it exists and false otherwise:

```
boolean ExistsTimeline(TimelineId timelineId);
```

DeleteTimeline. This deletes the specified *Timeline* if it exists:

```
void DeleteTimeline(TimelineId timelineId);
```

CreateSignal. This creates a new signal and adds it to the timeline.

```
SignalId CreateSignal(Id id, URI uri,
  MimeClass mimeClass, MimeType mimeType,
  Encoding encoding, Unit unit, Track track);
```

The `id` argument might be TimelineId or SignalId. If it is a TimelineId, the library will generate a new SignalId. If it is a SignalId, the library will try the given id first, and if it's taken, generate a new SignalId. If the id given is invalid, it throws an AGException.

The `uri` argument specifies a location where the signal is to be found. Applications may use this information to display and replay a signal. The `mimeClass` and `mimeType` arguments tell an application about the format of the signal, while the `encoding` argument specifies how samples are coded (e.g. mu-law). The `unit` argument specifies the sample rate of the signal; annotation applications may use this information to set the granularity of time coding and time alignment in a user interface. The `track` argument records which track of the signal file contains the signal. In this way, we can create two or more distinct `Signal` objects which reference different tracks of the same signal file.

GetSignals. This returns `SignalIds` of the *Signals* contained in the specified *Timeline*:

```
SignalIds GetSignals(TimelineId timelineId);
```

The `SignalIds` are separated by spaces.

3.2.3. Annotation Functions

CreateAnnotation. This creates a new annotation:

```
AnnotationId CreateAnnotation(Id id,
                   AnchorId start, AnchorId end,
                   AnnotationType annotationType);
```

The `id` argument can be an AGId or an AnnotationId. If it is an AGId, an AnnotationId will be assigned to the new annotation. On the other hand, if it is an AnnotationId, the library will try to use the supplied id. If this id is unavailable, it will assign a new AnnotationId. The other arguments are as follows: *start* is the id of the start anchor; *end* is the id of the end anchor; *annotationType* is the type of the annotation.

`CreateAnnotation` returns the AnnotationId of the new annotation.

ExistsAnnotation. This returns true if the annotation exists:

```
bool ExistsAnnotation(AnnotationId annotationId);
```

DeleteAnnotation. This deletes an annotation:

```
void DeleteAnnotation(AnnotationId annotationId);
```

CopyAnnotation. This copies an existing annotation, with a new identifier assigned to the new annotation:

```
AnnotationId CopyAnnotation(AnnotationId annotationId);
```

SplitAnnotation. This splits an annotation into two, creating a new annotation with the same label data as the original one; returns `Ids` of both annotations.

```
AnnotationIds SplitAnnotation(AnnotationId annotationId);
```

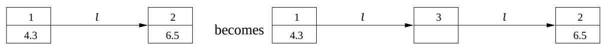


Figure 4: Split an annotation

NSplitAnnotation. This splits an annotation into N annotations, creating N-1 new annotations having the same label data as the original one; returns `ids` of all annotations, including the original one:

```
AnnotationIds NSplitAnnotation(
                AnnotationId annotationId, short N);
```

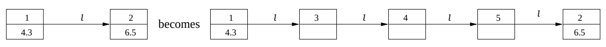


Figure 5: Nsplit an annotation with N = 4

3.2.4. Accessing Label Data

SetFeature. This sets the features of an annotation as well as the features of the metadata associated with *AGSets*, *AGs*, *Timelines* and *Signals*.

```
void SetFeature(Id id, FeatureName featureName,
                       FeatureValue featureValue);
```

The `Id` can be `AnnotationId`, `AGSetId`, `AGId`, `TimelineId` or `SignalId`. This is also true for other *Feature* functions, such as `ExistsFeature`, `DeleteFeature`, `GetFeature`, etc.

GetAnchorSet. This returns all the *Anchors* in a given *AG*.

```
AnchorIds GetAnchorSet(AGId agId)
```

GetAnchorSetByOffset. This returns all anchors with its offset in between offset-epsilon and offset+epsilon, inclusive. The default value for epsilon is 0.

```
AnchorIds GetAnchorSetByOffset(AGId agId,
          Offset offset, float epsilon=0);
```

GetAnchorSetNearestOffset. This returns all anchors at the nearest offset to the given offset:

```
AnchorIds GetAnchorSetNearestOffset(
          AGId agId, Offset offset);
```

3.2.5. Accessing Annotations

GetIncomingAnnotationSet. This returns the incoming annotations of the specified anchor. The incoming annotations of anchor `a` are the annotations which end with anchor `a`:

```
AnnotationIds GetIncomingAnnotationSet(
                AnchorId anchorId);
```

GetOutgoingAnnotationSet. This returns the outgoing annotations of the specified anchor. The outgoing annotations of anchor `a` are the annotations which start with anchor `a`:

```
AnnotationIds GetOutgoingAnnotationSet(
                AnchorId anchorId);
```

For example, in the annotation graph shown in Figure 6, the incoming annotations of anchor 2 are a,b,c,d,e, and the outgoing annotations of anchor 2 are f,g,h.

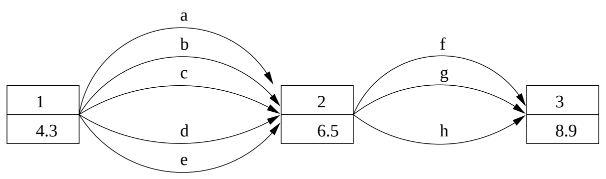


Figure 6: Incoming and outgoing annotations

GetAnnotationSetByOffset. This returns all annotations that overlap a particular offset.

```
AnnotationIds GetAnnotationSetByOffset(
                AGId agId, Offset offset);
```

GetAnnotationSeqByOffset. This returns all annotations sorted by start anchor offsets as the first sorting key, end anchor offsets as the second, and `AnnotationIds` as the third.

```
AnnotationIds GetAnnotationSeqByOffset(AGId agId);

AnnotationIds GetAnnotationSeqByOffset(
                AGId agId, Offset begin);
```

In this case, `GetAnnotationSeqByOffset` returns all the annotations with their start anchor offset greater than or equal to a specified offset, sorted by start anchor offsets as the first sorting key, end anchor offsets as the second, and `AnnotationIds` as the third.

```
AnnotationIds GetAnnotationSeqByOffset(
                AGId agId, Offset begin, Offset end);
```

In this case, `GetAnnotationSeqByOffset` returns returns all the annotations with the start anchor offset in between the specified offsets, sorted by start anchor offsets as the first sorting key, end anchor offsets as the second, and `AnnotationIds` as the third.

3.2.6. XML save function

toXML. This returns a string in the ATLAS Level 0 XML format of the specified *AGSet* or *AG*:

```
string toXML(Id id);
```

The annotation graph library also provides functions to access a database server, for the persistent storage of annotations which may be shared by multiple annotators, as described by Ma et al. (2002).

4. The File I/O Library

The I/O library is also implemented in C++. This section describes how the I/O classes are used to read native format files into annotation graphs and write them back.

Table 1 summarizes the formats which are currently supported by the I/O library.

There is one abstract class named `agfio` which declares the interface for the `load` and `store` methods, which are virtual functions. By inheriting the `agfio` class and implementing the `load` and `store` methods, each format will be a class with `load` and `store` methods. Loading or storing is done by creating an instance of a format class and calling the `load` or `store` method with the proper arguments.

As shown in Table 1, currently only AIF, LCF and TF formats can be stored.

5. Scripting Language Access to the APIs

The toolkit provides interfaces to the annotation graph libraries for the scripting languages Tcl and Python. To avoid having to manage object references across the Tcl/C and Python/C language interfaces and in the event language, all communication is via strings. These strings hold object identifiers, feature names and feature values. This section describes how to access the APIs from Tcl and Python.

For a Tcl program to access annotation graph functions, it must contain either of the following declarations:

```
package require ag
```

or

```
load ag_tcl.so
source ag.tcl
```

The following Tcl statement creates an annotation.

Format name	Supported I/O	Target corpus or format
AIF	input/output	ATLAS Interchange Format, Level 0 [http://www.ldc.upenn.edu/AG/doc/xml/ag.dtd]
BAS	input	BAS Partitur format [http://www.phonetik.uni-muenchen.de/Bas/BasFormatseng.html]
BU	input	Boston University Radio Speech Corpus
LCF	input/output	LDC Callhome Format
SwitchBoard	input	Switchboard
TF	input/output	Table Format
TIMIT	input	TIMIT Corpus [http://www.ldc.upenn.edu/lol/docs/TIMIT.html]
TreeBank	input/output	Penn Treebank [http://www.cis.upenn.edu/˜treebank/home.html]
xlabel	input	xlabel format

Table 1: Supported file formats

```
AG_CreateAnnotation $agId $a1 $a2 $ann_type
```

Note that the function name has the prefix `AG_`. Newer versions of AGTK support the Tcl namespace as follows.

```
AG::CreateAnnotation $agId $a1 $a2 $ann_type
```

The argument `$agId` is an AG identifier, and the arguments `$a1` and `$a2` are Anchor identifiers. The argument `$ann_type` is the type of the annotation (e.g. “word”).

Similarly, for a Python program, we need to import the AG module first:

```
import ag
```

The following code fragment creates an AGSet, a timeline, an AG, two anchors and an annotation.

```
agSetId = ag.CreateAGSet('Test')
timelineId = ag.CreateTimeline(agSetId)
agId = ag.CreateAG(agSetId, timelineId)
anc1 = ag.CreateAnchor(agId)
anc2 = ag.CreateAnchor(agId)
ann1 = ag.CreateAnnotation(agId, anc1, anc2, "Word")
```

The following fragment specifies features for a particular annotation, and prints the AIF representation to standard output.

```
ag.SetFeature(ann1, "English", "cat")
ag.SetFeature(ann1, "Japanese", "neko")
print ag.toXML(agId),
```

6. Building Annotation Tools with Tcl/Tk and Python

Most large-scale annotation projects must deal with multiple tools and formats. If all required tools were developed by a single project, and are used exactly as the original developers intended, there is usually no problem with interoperability. However, in 99% of the remaining cases, format translation and tool interoperability is a critical issue.

If annotation tools are created around a common architecture and a shared data model, these issues do not arise. New file formats are supported by writing a converter between the the format and the general-purpose data model. The sharing of components among different annotation tools is straightforward, and new applications can be developed quickly using existing components. New functionalities added to a component can be used by all existing tools which include that component.

The annotation graph library and the file I/O library provide the means to create, manipulate, read and write annotation graph data. In this section, we explain our approach to tool creation using these libraries as the core of the tool architecture.

All of our annotation tools are created using scripting languages having easy-to-use GUI libraries. This permits a rapid development cycle and easily customizable user interfaces. Tools are relatively small since much of the work is done by AGTK, and this small size and common data model mean that tool components are readily repurposable. We deliberately avoid the temptation to create general purpose annotation tools since each annotation task is idiosyncratic. Annotators work best when they use an interface which is maximally ergonomic for the peculiarities of the task, as compared with tools that include much irrelevant functionality and have an interface that is balanced for a wide variety of tasks.

This section describes the steps in creating a new special-purpose tool based on the general-purpose components provided by AGTK. It includes a discussion of inter-component communication, and of an example tool.

6.1. Building tools with Tcl/Tk

The Tk toolkit provides a graphical user-interface for Unix, Windows and Macintosh platforms. Some graphical user-interface components, called widgets, are provided with the Tcl/Tk standard distribution. For example the following fragment of Tcl/Tk code creates and displays a text widget:

```
set t [text .t]
pack $t
```

Other GUI components, such as buttons, menubars and canvases, come standard with Tk. In addition, open-source GUI components and extensions are developed and distributed by various developers around the world. Provides pointers to such software are provided at [http://dev.scriptics.com].

6.2. Building tools with Python

Python has become very popular as a scripting language. Python provides an object-oriented programming framework, and is easy to learn. A GUI package called Tkinter (Grayson, 2000) based on the Tk toolkit is included in the standard Python distribution. Tkinter provides class definitions for the standard widgets included in Tk. In addition, it is relatively easy to write a Tkinter class for a third-party Tk widget that is not directly supported by Tkinter.

The following segment of Python code creates and displays a Tk text widget.

```
from Tkinter import *
root = Tk()
t = Text(root)
t.pack
root.mainloop()
```

6.3. The inter-component communication model

An annotation tool built with the Annotation Graph Toolkit will consist of several major components, including: (i) a main program (script); (ii) an annotation/transcription component in which the user would enter annotations and transcriptions, and (iii) a signal display component giving access to recorded digital signals, such as speech waveforms. Typically, annotation/transcription components and waveform display components are reused by different specialized annotation tools. To create a new annotation tool, the developer writes a main program using the right selection of widgets and provides callback functions to handle widget events.

Events are passed around among components so that necessary tasks can be performed within each component. Consider the following example. Suppose that the user already has an annotation assigned to a specific region in the signal. He/she now wants to assign new start and end offsets for the signal to the annotation. Suppose the keyboard input *Control-g*, in the waveform component is assigned to such a task. When the user hits *Control-g* while there is a newly highlighted region in the waveform, this information (*event*) needs to be passed to the main program, and then to the annotation/transcription component and the annotation graph library. This propagation of event information is illustrated in Figure 7.

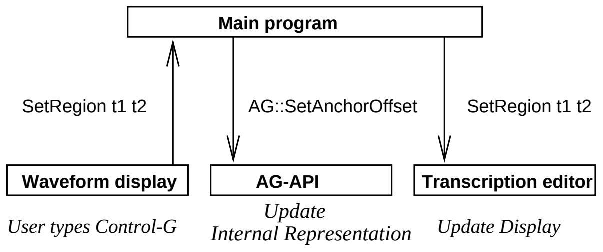


Figure 7: An example of inter-component message passing

The event *SetRegion* is generated in the waveform component and passed to the main program with the two parameters, the start time and the end time. Then, the main program sends *SetRegion* to the transcription component so that the start and end offsets can be updated in the transcription component. Also, the main program uses the annotation graph function *SetAnchorOffset* to update the internal representation of the annotation graph data. Table 2 shows a list of typical events passed around in this manner.

These events and their necessary arguments are passed as associative arrays (e.g., Tcl arrays and Python dictionaries). These arrays are passed to event handlers defined in the recipients.

Event name	Typical parameters
CreateAnnotation	start time, end time
DeleteAnnotation	annotation identifier
SetFeature	feature, value
SetRegion	start time, end time
GetRegion	start time, end time
SetCurrentAnnotation	annotation identifier
Play	start time, end time
Stop	

Table 2: A list of common events

6.4. GUI design

A critical aspect of designing a successful annotation tool is to make it highly ergonomic for the particular annotation task. Writing a tool in a scripting language makes it easy to experiment and change various aspects of a user interface, and facilitates “power users” who can tweak the code to help streamline their work.

Where possible, our tools use simple keybindings that work in multiple contexts, making it easy for users to focus on the annotation task instead of having to hunt for obscure key combinations. In the MultiTrans annotation tool, a tool based on AGTK, the *Return* key is active in multiple contexts. If a new region in the waveform display is chosen (i.e., highlighted), pressing the *Return* key will create and insert a new annotation into the transcription display. If an existing annotation in the transcription display is chosen, its corresponding region in the waveform is highlighted. A single mouse-button click in the highlighted region will set a point. If the *Return* key is pressed then, it will split the annotation and the region. Figure 8 and Figure 9 show before and after split, respectively.

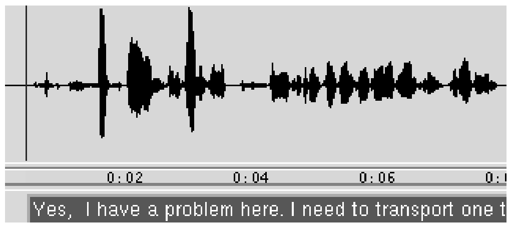


Figure 8: Before Split

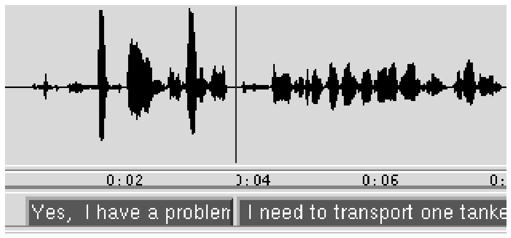


Figure 9: After Split

If the *Return* key is pressed during a playback, it will set an anchor point. When the playback of speech is begun, the user may press the *Return* key to insert an anchor in the current channel. When *Return* is pressed a small black bar will appear below the waveform (Figure 10). This designates the current starting position of the annotation. When *Return* is pressed a second time the end anchor for the annotation is inserted and the annotation is created.

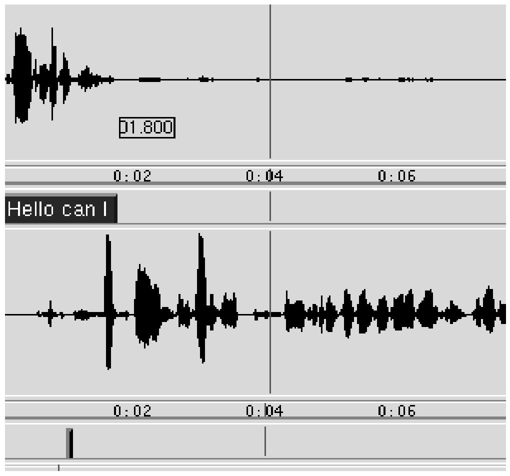


Figure 10: Setting Anchor Point during Playback

All these different actions have a common theme, namely to “create” something. This heavy re-use of a keybinding requires a little more effort in the coding process, but we have found that it makes the annotation process faster and easier. Using scripting languages, it is very easy to experiment with the user interface in this way.

6.5. Audio and video display using third-party software: WaveSurfer and QuickTime

WaveSurfer (Sjölander and Beskow, 2000) was developed by Kåre Sjölander and Jonas Beskow of KTH as a tool for displaying and manipulating sound files. WaveSurfer uses Snack (Sjölander, 2000) as its signal processing module. Both packages are distributed under an open source license. WaveSurfer is written in Tcl/Tk and its widget, called *wsurf*, can be embedded in an application written in Tcl/Tk. A Python interface has also been developed. This makes WaveSurfer an excellent component to use with AGTK.

QuickTime Tcl is another component that can be used with AGTK. QuickTime Tcl requires the QuickTime player created by Apple Corporation. Currently, no UNIX version of QuickTime player is provided by Apple.

6.6. Embedding a third-party Tk widget in Python

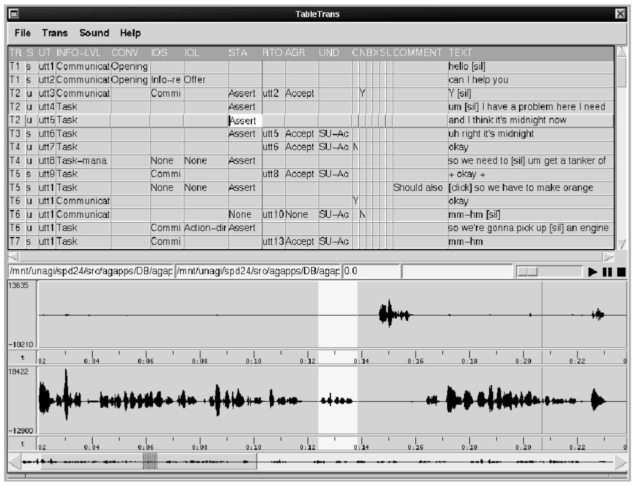
Tkinter provides class definitions only for the original Tk widgets. However, it is relatively easy to write an extension of Tkinter for a third-party Tk widget. For example, AGTK includes a Python/Tkinter class definition for the Wsurf widget. The class *Wsurf* covers the Wsurf API. The class *agWsurf*, which is a subclass of *Wsurf*, provides methods specific to the tasks required by the tools we create.

6.7. A Tk table widget

AGTK comes with a table annotation component called agTable (or, ag-table as originally called in the Tcl version). The agTable component is based on TkTable written by Jeffrey Hobbs, et. al. [`http://sourceforge.net/projects/tktable/`].

6.8. Putting it all together: the case of TableTrans

Now that we have seen the major components of AGTK, we can examine how to build an annotation tool using these components. As an example, we will examine an annotation tool called TableTrans (Figure 11). TableTrans is a spreadsheet-style annotation tool that uses the components we have described in this paper.



TR	S	UT	INFO-LVL	CONV	IOS	IOL	STA	RTO	AGR	UND	C	N	B	X	S	L	COMMENT	TEXT
T1	s	utt1	Communicat	Opening														hello [sil]
T1	s	utt2	Communicat	Opening	Info–re	Offer												can I help you
T2	u	utt3	Communicat		Commi		Assert	utt2	Accept			Y						Y [sil]
T2	u	utt4	Task				Assert											um [sil] I have a problem here I need
T2	u	utt5	Task				Assert											and I think it's midnight now
T3	s	utt6	Task				Assert	utt5	Accept	SU–Ac								uh right it's midnight
T4	u	utt7	Task					utt6	Accept	SU–Ac	N							okay
T4	u	utt8	Task–mana		None	None	Assert											so we need to [sil] um get a tanker of
T5	s	utt9	Task		Commi			utt8	Accept	SU–Ac								+ okay +
T5	s	utt1	Task		None	None	Assert										Should also	[click] so we have to make orange
T6	u	utt1	Communicat								Y							okay
T6	u	utt1	Communicat				None	utt10	None	SU–Ac		N						mm–hm [sil]
T6	u	utt1	Task		Commi	Action–dir	Assert											so we're gonna pick up [sil] an engine
T7	s	utt1	Task		Commi			utt13	Accept	SU–Ac								mm–hm

/mnt/unagi/spd24/src/agapps/DB/agap /mnt/unagi/spd24/src/agapps/DB/agap 0.0

Figure 11: TableTrans

TableTrans consists of three major components: (i) the main script, *agTableTrans*, (ii) the table component, *agTable*, and (iii) the waveform display component, *agWsurf*. Here we will look at two common operations: pressing the *Return* key in the table to insert a new annotation, and pressing the *Control-d* key combination in the table to delete the highlighted annotation.

The keybindings for *Return* and *Control-d* are already defined in the agTable component. The operations above pass the events *CreateAnnotation* and *DeleteAnnotation* to the main script, respectively. A callback function (*agTableEvent*) is defined in the main script. When the *Return* key is pressed, the table component sends the event *CreateAnnotation* to the main script. The callback function in the main script then performs the following tasks:

First, using the start and end offsets of the current region stored in the main script, it calls the annotation graph function *CreateAnchor* twice to create the start anchor and the end anchor:

```
a1 = ag.CreateAnchor(self._AGName)
ag.SetAnchorOffset(a1, self._currentStartPosition)
a2 = ag.CreateAnchor(self._AGName)
ag.SetAnchorOffset(a2, self._currentEndPosition)
```

Second, it calls the annotation graph function *CreateAnnotation* using the start and end anchors:

```
id = ag.CreateAnnotation(self._AGName,
                                    a1, a2, annotationType)
```

This function returns the annotation identifier. Finally, the program returns the new identifier, along with the start and end offsets, to the table component. The table component inserts a new row using the annotation identifier and the start and end offsets.

Similarly when the *Control-d* key combination is pressed, the table component sends the event *DeleteAnnotation* to the main script together with the annotation identifier of the highlighted row. The callback function in the main script performs the following command, where dictionary *event* contains the event message.

```
ag.DeleteAnnotation(event['AnnotationId'])
```

Then, the table component can remove the highlighted row from its table. Figure 12 shows the components of TableTrans and their relationships.

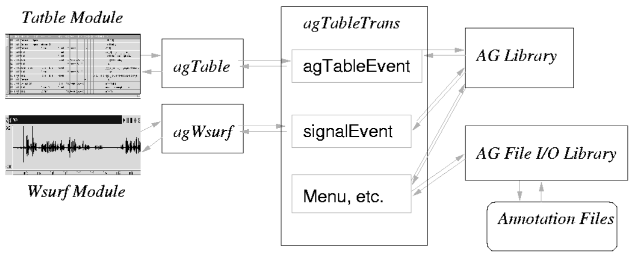


Figure 12: Components in TableTrans

7. Conclusion

This paper has described a new toolkit, AGTK, which supports rapid development of linguistic annotation software. AGTK is being used for annotation projects at the Linguistic Data Consortium [www.ldc.upenn.edu]. The toolkit is available for download at [http://www.sourceforge.net/projects/agtk/]. For news and updates, please visit [http://www.ldc.upenn.edu/AG/].

Existing third-party tools, namely Emu and Transcriber (Cassidy and Harrington, 2001; Barras et al., 2001), are being migrated to AGTK. They will share the same internal data model and relational storage model, while keeping their distinctive user interfaces and file formats. Once these ports have been completed, we will have a shared library of user interfaces to complement the AG and file I/O libraries. We hope that these shared libraries will continue to grow as members of the wider community contribute I/O and GUI components to AGTK.

In future work, we hope to develop more applications for annotation in other areas, possibly including: sociolinguistics, conversational analysis, sign and gesture, discourse and dialogue. On the technical side, we hope to add interfaces to video widgets on all platforms, and to support data entry for extended-Roman and non-Roman scripts.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Nos. 9978056, 9980009 (Talkbank). The authors are grateful to Claude Barras, Steve Cassidy, Mark Liberman and many others for discussions on the material presented here.

8. References

S. Abiteboul, R. Hull, and V. Vianu. 1995. *Foundations of Databases*. Addison Wesley.

J. Allen, D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, and A. Stent. 2000. An architecture for a generic dialogue shell. *Natural Language Engineering*, 6:213–228.

C. Barras, E. Geoffrois, Z. Wu, and M. Liberman. 2001. Transcriber: development and use of a tool for assisting speech corpora production. *Speech Communication*, 33:5–22.

S. Bird and J. Harrington, editors. 2001. *Speech Communication: Special Issue on Speech Annotation and Corpus Tools*, volume 33. Elsevier.

S. Bird and M. Liberman. 2001. A formal framework for linguistic annotation. *Speech Communication*, 33:23–60.

S. Cassidy and J. Harrington. 2001. Multi-level annotation of speech: An overview of the emu speech database management system. *Speech Communication*, 33:61–77.

J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, and N. L. Dahlgren. 1986. *The DARPA TIMIT Acoustic-Phonetic Continuous Speech Corpus CDROM*. NIST. http://www.ldc.upenn.edu/Catalog/LDC93S1.html.

J. E. Grayson. 2000. *Python and Tkinter Programming*. Manning.

X. Ma, H. Lee, S. Bird, and K. Maeda. 2002. Models and tools for collaborative annotation. In *Proceedings of the Third International Conference on Language Resources and Evaluation*.

K. Sjölander and J. Beskow. 2000. WaveSurfer – an open source speech tool. In *Proceedings of the 6th International Conference on Spoken Language Processing*. http://www.speech.kth.se/wavesurfer/.

K. Sjölander. 2000. The Snack sound toolkit. http://www.speech.kth.se/snack/.