Transcribing with Annotation Graphs

Edouard Geoffrois[1], Claude Barras[2], Steven Bird[3], and Zhibiao Wu[3]

[1] DGA/CTA/GIP
16 bis av. Prieur de la Côte d'Or,
94114 Arcueil cedex, France
Edouard.Geoffrois@etca.fr

[2] Spoken Language Processing Group
LIMSI-CNRS, BP 133,
91403 Orsay cedex, France
Claude.Barras@limsi.fr

[3] LDC
3615 Market Street, Suite 200,
Philadelphia, PA, 19104-2608, USA
{sb,wzb}@unagi.cis.upenn.edu

Abstract

Transcriber is a tool for manual annotation of large speech files. It was originally designed for the broadcast news transcription task. The annotation file format was derived from previous formats used for this task, and many related features were hard-coded. In this paper we present a generalization of the tool based on the annotation graph formalism, and on a more modular design. This will allow us to address new tasks, while retaining Transcriber's simple, crisp user-interface which is critical for user acceptance.

1. Introduction

The development and refinement of speech recognition systems requires a large amount of transcribed speech data. Production of these corpora is a highly time-consuming task and requires specialized software tools. In order to support a project involving the automatic transcription and indexing of multilingual broadcast news, DGA developed "Transcriber", a tool for manually segmenting, labeling and transcribing speech signals having extended duration (up to several hours). The first release of the tool was presented at the LREC conference in 1998 (Barras et al., 1998). In order to encourage the production of speech corpora and ease their sharing, it was decided to distribute the tool as free software[1].

Since then, Transcriber has been extended and refined, taking into account user feedback. New features were added, and great care was taken to retain a user-friendly interface and interactive management of long duration signals. Transcriber has been used for the production of over 100 hours of transcribed radio programs and television soundtracks in several languages, and has thus proven suitable for large-scale production of transcribed corpora (Barras et al., 2000).

However, Transcriber was geared towards a single application: the broadcast news transcription task. The internal data structure was derived from the format which is most commonly used for broadcast news transcription (.typ, UTF). This data format prevented the implementation of various minor yet desirable features. For example, markers which are used to indicate the beginning and the end of some annotations (e.g., to delimit a stretch of words in another language) are not logically associated, and cannot be manipulated as a single unit.

Furthermore, many features related to the format and to the application were hard-coded. Even though generality was a concern, the Transcriber development effort has concentrated on efficiency for broadcast news transcription, and the interface was designed around this specific task. As a result, if the data representation needs to be changed, code has to be written again.

[1] http://www.etca.fr/CTA/gip/Projets/Transcriber/

Nevertheless, there has been a demand for generalizing the tool so that it can display and edit a wider range of annotation types. One example is the CHAT format used within the CHILDES research group on language acquisition (MacWhinney, 1995). Other extensions are needed for annotating completely new kinds of data, like multi-channel audio or video. Gracefully handling such a wider variety of formats and functions requires a more general data model.

This need for a broader coverage of annotation types is not unique to Transcriber. Many other annotation projects face the same issues. In response to this, a new general-purpose data model for linguistic annotations, called the "annotation graph" (AG), has been proposed (Bird and Liberman, 1999). In contrast with some other models, annotation graphs are conceptually very simple, and have extremely broad applicability. Recently, DGA decided to migrate Transcriber to the annotation graph model, and to collaborate with LDC on the design and implementation of a new modular architecture. This paper reports these developments.

2. Transcriber features

Transcriber is described more extensively in other articles (Barras et al., 1998; Barras et al., 2000) and in its reference manual (available on the web site and in the tool itself in the online help). This section provides a summary of the main features, illustrating the starting point for our migration to an AG-based Transcriber.

The user interface is shown in Figure 1. It consists mainly in two windows, one for displaying and editing the transcription, and one for the displaying the signal waveform and the segmentation. The annotations include various information: orthographic transcription, speech turns, topic sections, background conditions, and various events. The data format is XML, and a DTD controls the validity of the data. This format used for file input/output is also used directly as the internal data structure. Therefore, no conversion is needed for input/output. But the major drawback is the strong dependency of the code on the file format, so that modifications in the format need to be propagated in the code.

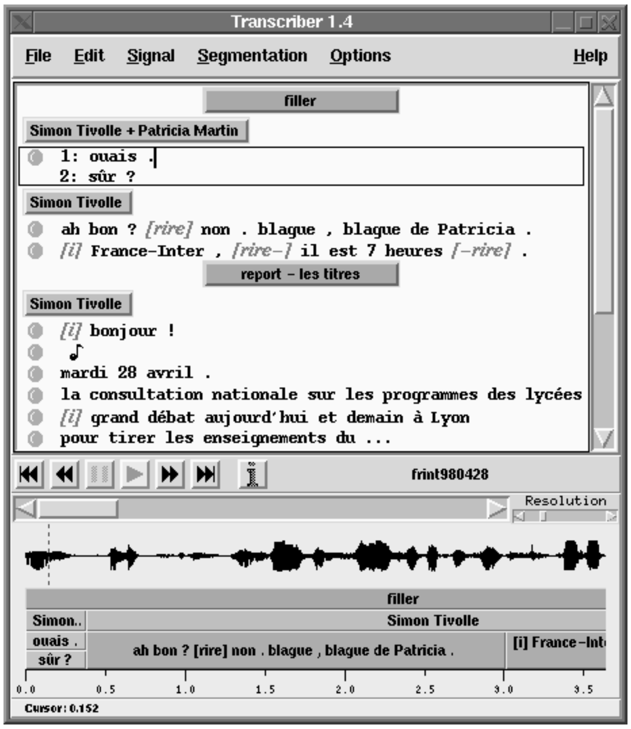


Figure 1: Screenshot of Transcriber user-interface.

3. Using annotation graphs

The annotation graph model provides a general-purpose abstraction layer between physical annotation formats and graphical user interfaces. As a consequence, the connections between this logical model and various physical and graphical representations can be fully modularized. New annotation formats and new user-interfaces to an annotation task can thus be implemented as pluggable components.

The annotation graph data model is composed of two low-level structures – nodes and arcs – and two high-level structures – graphs and subgraphs. A graph object is a collection of zero or more arcs, each specifying an identifier, a type, and some content consisting of domain-specific attributes and values. An arc also references a start and end node, and each node provides an optional temporal offset. This temporal offset may be qualified with a "timeline", which is a symbolic name for a collection of signal files which are temporally co-extensive and whose times can be meaningfully compared. Node and arc identifiers may also be qualified with a user-specific namespace, to avoid collisions when multiple independent annotations are combined.

A recent collaboration between NIST, LDC and MITRE has produced an applications programming interface to annotation graphs, and implementations in Java and C++ are in development. A prototype in Perl/tk has also been produced and is available online[2]. An even more general model has been proposed recently[3] (Bird et al., 2000b). A query language is also being developed (Bird et al., 2000a).

[2] http://www.ldc.upenn.edu/annotation/AG/prototype/

[3] http://www.nist.gov/speech/atlas/

3.1. Interpreting Transcriber features in terms of annotation graphs

Transcriber can be generalized by incorporating annotation graphs as the internal data model. Many aspects of this process are quite natural and obvious. It is interesting to note that the duality of the Transcriber interface closely resembles the duality of the graph structure: the text pane represents a node-based view where one can edit the content of the arcs, while its signal pane represents an arc-based view where one can modify the time of the anchored nodes. The display of the transcription in the text editor and of the segmentation under the signal can be expressed as general transformations of the annotation graphs. Note that the constraint that all arcs of a given type (orthographic transcription with markers, speaker turns or sections) are a partition of the recording has to be imposed by the interface, not by the data structure.

In order to prepare for the transition to AG, Transcriber features were systematically enumerated and classified according to how the transition to annotation graphs affects them.

Some features are completely independent of the annotation format. These incude signal management (scrolling and zooming, selecting a portion of signal, moving the cursor to a given position, continuous playback), interface display options (fonts, colors, toggling second signal view or button bar display), and other general options (choice of language, defaults, shortcuts). There are also features which, though they interact with the AG structure, should remain outside of it. This is the case of speaker and topic databases, which were already clearly separated in the original format.

Other features depend on the annotation format, and must be re-implemented. Some have a natural implementation in the new representation. For example, moving a boundary naturally applies to a timed node; Events are linked with the orthographic transcription using shared nodes; Selecting a segment (or several) applies to the corresponding arcs; The consistency between cursors in the text editor and in the signal waveform can be easily checked, since the bounding time interval is defined for any arc (Bird et al., 2000a); Simultaneous speech from two speakers can be represented in various ways, and the original implementation where words are labelled as associated to one of the two speakers can be seen as a particular case of the equivalence class mechanism of the AG; The information associated to an episode (recording date and source, primary language...), originally stored separately, can be associated to an arc spanning the whole recording.

In some cases, this natural generalization requires certain types of arc to be distinguished. Indeed, some features apply to text only (spell checking, glossary, finding a word), some features apply to text and events (cut/copy/paste as in the current implementation). some to segments corresponding to text and event arcs (highlight segment, go to next/previous segment, pause at segment boundary during playback, insert/delete breakpoint), and some others to turn or topic arcs (next/previous turn/topic, find speaker/topic).

For certain other features, the change of representation leads to consider them under a new angle. This is the case of the creation and editing of speech turns, events and comments. For example, inserting a turn was originally viewed as a change of boundary type, but can now be viewed as splitting a speaker arc.

The interpretation of a transcription as an AG also opens up new possibilities. Features associated to a particular type of arc could be extended to other types (e.g., cut/copy/paste on any type of arc, go to next acoustic backgroung change, pause at all timed nodes during payback). It is also possible to display only certain types of arcs, for example to see only the speaker turns or the topical structure of the annotations. To summarize, in many cases the AG model makes it easier to improve existing features or implement new ones.

3.2. Implementation issues

Using AG as the internal data model should make the code independent of the file format, and task-dependent issues should be removed or located in separate, task-configurable modules. However, these generalization steps must not compromise the efficiency of the tool, as perceived by the human annotator.

As was previously described, two views of the transcription are available: a text editor for creating or modifying the content of the transcription, and a temporal view for controlling the synchronization between the signal and the annotation, with several segmentation tiers. Modifications in the editor are immediately viewed in the segmentation. This is currently done with ad-hoc coding, and should be implemented in a more generic fashion using AG, while retaining a high degree of efficiency.

Incorporating AG in Transcriber involves fundamental redesign of the tool. In this context, we should consider other limitations of the tool and how they can be improved. For example, we will address unlimited undo, incremental save, and version control. These functions, much like segmental display update, can take advantage of tracing modifications of the annotations.

3.2.1. Tracing AG modifications

All operations on an AG split up into elementary operations of creating, modifying and destroying arcs and nodes. Given the state of the graph at time t_0, the elementary operations applied between t_0 and t_1 can be automatically stored along with the graph in an additional data structure. This allows efficient updates of the interface taking into account only the effective changes since a previous update. It also makes it possible to return to the t_0 state. Several traces can be maintained for different purposes. Note that the value given for the creation or modification of an arc does not need to be stored in the traces, since they are already available in the (updated) graph itself. On the other hand, the initial value of a modified attribute or the whole content of a destroyed arc is necessary for the undo feature.

3.2.2. Incremental save

The first application we describe is incremental save. Tracing makes it easy to dump any graph modifications that occurred since a previous dump in a concise manner. Figure 2 gives an example of a possible incremental output (we do not commit to this format for the final implementation).

```
<IncrementalGraph author="JB" date="04/01/00, 16:00">
<DestroyNode id="nd18"/>
<DestroyArc id="ar15"/>
<NewNode id="nd20" offset="10.3"/>
<NewArc id="ar17" start="nd19" end="nd20"/>
<ModifyNode id="nd19" offset="7.2"/>
<ModifyArc id="ar16" type="word" value="absolutely"/>
</IncrementalGraph>
```

Figure 2: Sample of an incremental file output.

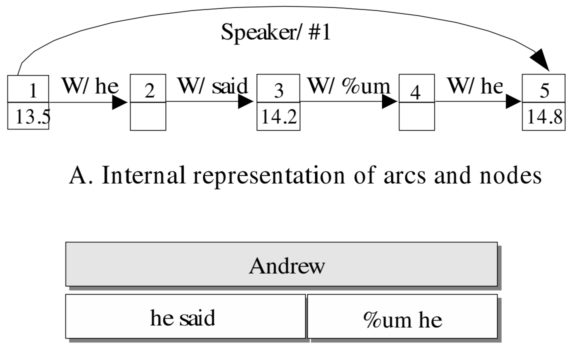
For modified arcs and nodes, only the actually modified attributes need to be dumped along with the arc or node identifier, which makes the output more compact. This scheme also provides a very good tracing of the transcription history, since it makes it possible to know precisely by who and when each attribute of the graph was modified. One can, when reading the cumulative file of incremental saves, stop reading at a given date, thus coming back to a previous version. One can also work on an original transcription without modifying it and store any other modifications in a separate file. If several persons work on the same transcription and apply modifications to different arcs, merging their results is trivial. Of course, these new possibilities do not prevent us from providing import and export of the complete annotations in other formats.

3.2.3. Unlimited undo

Allowing an unlimited level of undo, like in many modern office applications, is a desirable feature. This implies to manage a stack of AG modifications: from t_0 to t_1, from t_1 to t_2, ..., and from t_n to current time. Given the fact that user actions can involve several elementary modifications of the graph, it is up to the application to choose the times corresponding to the end of each undoable user action.

3.2.4. Update of segmental display

One or several segmentation tiers are shown under the signal, each tier containing a specific kind of arc. Each segment of a tier consists of contiguous arcs, for which the outermost nodes are anchored (cf. Figure 3). When modifying the graph (e.g., changing the attribute of an arc, anchoring a node, creating a new arc), the display of the segmentation tiers needs to be updated. This should be done without pro-



A. Internal representation of arcs and nodes

B. Display of related segmentation tiers

Figure 3: Display of an annotation graph as segmentation tiers.

cessing the whole graph, and untouched segments should be kept. For this, the trace of the AG modifications since the last segmental display update is used. For each created arc, its membership of any existing tier must be checked; in the most general case, a user-defined boolean callback function performs the test. The character string displayed in the segment box is often the concatenation of some attribute value for the segment arcs. But in some cases the string cannot be deduced trivially from the attribute values (e.g., for displaying the name of a speaker instead of its id) and a user-defined callback function, if defined, will be used. Note that the display of the character string in the segment may be truncated, depending on the width of the segment on the screen at the current resolution. The segment color follows the same scheme, with a default color for each tier and a user-defined function for the actual color of the segment.

3.2.5. Management of the text editor

In the text editor, one should be able to see and modify the content of the arcs. A strict ordering of the arcs needs to be defined, so that each arc can sequentially displayed in the text pane. The natural arc order is driven by node order, but a hierarchy of arc types has to be given when several arcs share the same node. Also, unanchored nodes require decisions which may depend on the application.

It will be possible to display only a subset of the graph (e.g., only sections and turns without the transcription, or only the transcription for a specific speaker or a specific channel). Display of the arc content is application specific; depending on the kind of arc, it may be icons, buttons, text... Edition of the arc can be done directly in the text editor, or in a pop-up menu or window when clicking on some part of the display. All this will be controlled in the standard Tk text widget by user-defined callbacks. When using a generic XML editor instead (as discussed in 4.4.), the display will be rather controlled by stylesheets, as defined by CSS and XSL standards.

Most often, the modifications in the text editor will be simply propagated to the graph. The opposite case (update of the text editor according to possibly external modifications of the graph) is more complex and will involve specific optimizations similar to the ones described in 3.2.4..

4. Modular design

Previous discussions about Transcriber and AG demonstrated the need to have a modular, flexible design for Transcriber. We propose a new modular architecture for Transcriber. This architecture will allow us to separate Transcriber into several independent components. It will then be possible for developers to reuse those components or replace one of them without the need to change other components.

4.1. Design goals

We have the following design goals in mind.

Object oriented design: The system needs to be structured into several components based on its major functionalities. Each component should be self-sufficient in fulfilling its functionality. In doing so, the components will be maximally reusable.

Simple interfaces between components: The interface among components should be simple and clear. The interface should be abstract enough so that it won't be wedded to any specific annotation task.

Easy integration with external software modules: Given that many annotation tools exist already, it is important that our system is able to communicate with them, so that not only the file formats can be interchanged, but also the functional components can be interchanged to a certain degree.

Using XML as the interchange bridge: In order to make configuration easy, we will use XML as the bridge to do the communication among components, as well as file format interchange. We will also explore ways to use XML to define a graphical interface so that the GUI is configurable at run time.

4.2. Architecture overview

The system is designed based on the "Component" concept as used in Java Beans and Microsoft COM objects. A component is an object which fulfills a certain task, and provides interfaces so that other components can communicate with it or control its behavior. The data is passed around different components through XML. The format of the data has to conform with a DTD for inter-component messages, which all components must agree on. Currently, the system is divided into three highly abstracted components. One is the Transcriber engine. This component will take care of internal data representation, management of other components, time alignment and coordinate communications among them. The second one is the Transcription component. It will display and accept editing commands on the transcription. The third component is the media component. It will take care of media operations such as media file open and close, play or stop playing, etc.

A complete running system will have one Transcriber engine, but may have more than one instance of transcription components and media components. In this way, a video player and an audio player can be run at the same time. Also the same transcription can be displayed in different transcription component instances for the convenience of editing and viewing.

4.3. The Transcriber Engine

The central piece of the system is the Transcriber Engine. This component will coordinate the other components and pass events around. New instances of transcription components or media components which support Transcriber interfaces can register with the component engine. As soon as an instance is registered, it can communicate with the engine to report its current status and send annotation request to the engine. In this way, the system permits components to operate in a plug and play fashion. With a suitable wrapper, existing systems (e.g., text editors, waveform display tools) can be easily incorporated into the system. The engine will maintain a list of instances and manage the communication among them. For example, when a region of signal is selected, the engine will calculate the correct size and position for all transcription instances and issue an alignment command to each of them.

4.4. The transcription component

This component presents the transcript to the user. The annotation contents are defined with an XML DTD. Based on this DTD, the transcription component should be able to configure the GUI when the DTD is changed. When focus is changed in another component, this component should be able to change its focus too. As long as a component supports the Transcriber interface, the transcription component can actually be in quite different forms. The same transcription can be displayed in different views depending on the task, or the domain, or the user. Some components can be a text editor, some can be a horizontal viewer, or a generic XML editor. Transcriber developers can either build a component from scratch or write wrappers around existing editors such as Emacs or Tk Text Widget to support Transcriber interface.

4.5. The media component

This component takes care of the media files. It will be responsible for opening and closing files, displaying media, playing audio or video in a certain region or speed, and changing resolution and other parameters. It will respond to requests from the Transcriber engine to do certain tasks such as align the display to the focus point, show the current focus, play the current region, zoom in, zoom out the display, etc.

5. Conclusion

We have presented the current developments around the Transcriber speech annotation tool. They are based on the annotation graph model and on a highly modular design. The main challenge is to generalize the tool while keeping backward compatibility and without degrading its efficiency. Transcriber's simple, crisp user-interface has been a critical component of its success, and we are careful to retain this property.

Using the annotation graph formalism allows many hard-coded functions to be replaced by instances of a more general, parameterized mechanism. The user-interface can be more easily customized to handle new annotation tasks, and the underlying generic data model simplifies the interoperability with other tools. The resulting annotation tool will be extremely flexible and general, and will be openly distributed to the community.

6. References

Claude Barras, Edouard Geoffrois, Zibiao Wu, and Mark Liberman. 1998. Transcriber: a free tool for segmenting, labeling and transcribing speech. In *International Conference on Language Resources and Evaluation (LREC)*, pages 1373–1376.

Claude Barras, Edouard Geoffrois, Zibiao Wu, and Mark Liberman. 2000. Transcriber: development and use of a tool for assisting speech corpora production. *Speech Communication*. to appear.

Steven Bird and Mark Liberman. 1999. A formal framework for linguistic annotation. Technical Report MS-CIS-99-01, Department of Computer and Information Science, University of Pennsylvania. [xxx.lanl.gov/abs/cs.CL/9903003], expanded from version presented at ICSLP-98, Sydney, revised version to appear in *Speech Communication*.

Steven Bird, Peter Buneman, and Wang-Chiew Tan. 2000a. Towards a query language for annotation graphs. In *Proceedings of the Second International Conference on Language Resources and Evaluation*.

Steven Bird, David Day, John Garofolo, John Henderson, Chris Laprun, and Mark Liberman. 2000b. Atlas: A flexible and extensible architecture for linguistic annotation. In *Proceedings of the Second International Conference on Language Resources and Evaluation*.

Brian MacWhinney. 1995. *The CHILDES Project: Tools for Analyzing Talk*. Mahwah, NJ: Lawrence Erlbaum., second edition. [childes.psy.cmu.edu/].