Documentation for the NITE XML Toolkit

Document revisions
0.3 - 11 June 09
0.2 - 28 March 07
0.1 - 12 Jan 07

JEAN CARLETTA
STEFAN EVERT
JONATHAN KILGOUR
CRAIG NICOL
DENNIS REIDSMA
JUDY ROBERTSON
HOLGER VOORMANN

Table of Contents

Table of Contents

1 Drafts before v1.0

In October 2006, we decided to move over NXT documentation from being completely web-based to being written in DocBook so that we can generate HTML, JavaDoc, and PDF at will. We are rewriting much much of the documentation at the same time. Versions of the documentation numbered before v1.0 are incomplete, although the outline gives some idea of our intentions for it. In this version, version 0.1, not all of the information has been checked for accuracy yet. The most likely difficulties concern the following areas: corpus resources, ontologies, and object sets; validation; incomplete description of data set concepts. In addition, not all the formatting works, and the query reference manual has not been fully converted over to DocBook, so it is incomplete and hard to read in this version.

2 A Basic Introduction to the NITE XML Toolkit

There are many tools around for annotating language corpora, but they tend to be good for one specific thing and they all use different underlying data formats. This makes it hard to mark up data for a range of annotations - disfluency and dialogue acts and named entities and syntax, say - and then get at the annotations as one coherent, searchable database. It also makes it hard to represent the true structure of the complete set of annotations. These problems are particularly pressing for multimodal research because fewer people have thought about how to combine video annotations for things like gesture with linguistic annotation, but they also apply to audio-only corpora and even textual markup. The open-source NITE XML Toolkit is designed to overcome these problems.

At the heart of NITE there is a data model that expresses how all of the annotations for a corpus relate to each other. NXT does not impose any particular linguistic theory and any particular markup structure. Instead, users define their annotations in a "metadata" file that expresses their contents and how they relate to each other in terms of the graph structure for the corpus annotations overall. The relationships that can be defined in the data model draw annotations together into a set of intersecting trees, but also allow arbitrary links between annotations over the top of this structure, giving a representation that is highly expressive, easier to process than arbitrary graphs, and structured in a way that helps data users. NXT's other core component is a query language designed specifically for working with data conforming to this data model. Together, the data model and query language allow annotations to be treated as one coherent set containing both structural and timing information.

Using the data model and query language, NXT provides:

- a data storage format for data that conforms to the data model
- routines for validating data stored in the format against the data model
- library support for loading data stored in the format; working with it and modifying it; and saving any changes
- a query language implementation
- libraries that make it easier to write GUIs for working with the data by providing data display components that, for instance, synchronize against signals as they play and highlight query results
- annotation tools for some common tasks including video annotation and various kinds of markup over text or transcription (dialogue acts, named entities, coreference, and other things that require the same basic interfaces)
- command line tools for data analysis that, for instance, count matches to a specific query
- command line tools for extracting various kinds of trees and tab-delimited tables from the data for further processing or more detailed analysis

3 Downloading and Using NXT

Platform-independent binary and source distributions of NXT can be downloaded from Sourceforge at http://sourceforge.net/projects/nite/. For most purposes the binary download is appropriate; the source download will be distinguished by `_src` suffix after the version number. For the most up-to-date version of NXT, the SourceForge CVS repository is available. For example

```
cvs -z3 -d:pserver:nite.cvs.sourceforge.net:/cvsroot/nite co nxt
```

would get you a current snapshot of the entire NXT development tree.

3.1 Prerequisites

Before using NXT, make sure you have a recent version of Java installed on your machine: Java 1.4.2_04 is the minimum requirement and Java 1.5 is recommended. Learn about Java on your platform, and download the appropriate version using Sun's Java Pages.

For optimum media performance you may also want to download JMF and the platform-specific performance pack for your OS. NXT comes packaged with a platform-independent version of JMF. Users of MacOS should use the FMJ libraries instead which use *QuickTime* for media playback for improved performance and easier installation. NXT comes packaged with a version of FMJ compiled specifically with *QuickTime* support.

3.2 Getting Started

- Step 1: download and unzip nxt_*version*.zip

- Step 2: Some data and simple example programs are provided to give a feel of NXT. On windows, try double-clicking a `.bat` file; on Mac, try running a `.command` file; on Linux (or Mac) try running a shell script from a terminal e.g. `sh single-sentence.sh`. More details in *Sample Corpora* section below.

- Step 3: Try some sample media files: Download signals.zip (94 Mb) and unzip it into the Data directory in your NXT directory. Now when you try the programs they should run with synced media.

3.2.1 Sample Corpora

Some example NXT data and simple example programs are provided with the NXT download. There are several corpora provided, with at most one observation per corpus, even though in some cases the full corpus can actually consist of several hundred observations. Each corpus is described by a metadata files in the `Data/meta` directory, with the data itself in the `Data/xml` directory. The Java example programs reside in the `samples` directory and are provided as simple examples of the kind of thing you may want to do using the library.

- single-sentence - a very small example corpus marked up for part of speech, syntax, gesture and prosody. Start the appropriate script for your platform; start the *Generic Corpus Display* and rearrange the windows. Even though there is no signal for this corpus, clicking the play button on the NITE Clock will time-highlight the words as the time goes by. Try popping up the seacrh window using the *Search* menu and typing a query like `($g lgest)($w word):$g # $w`. This searches for left-handed gestures that temporally overlap words. You should see the three results highlighted when you click them.

- dagmar - a slightly larger example corpus: a single monologue marked up for syntax and gesture. We provide a sample gesture-type coding interface which shows synchronisation with video (please download signals.zip to see this in action).

- smartkom - a corpus of human compuiter dialogues. We provide several example stylesheet displays for this corpus showing the display object library and synchronization with signal (again, please download signals.zip above to see synchronisation)

- switchboard - a corpus of telephone dialogues. We provide coders for *animacy* and *markables* which are in real-world use.

- maptask-standoff - This is the full multi-rooted tree version of the Map Task corpus. We provide one example program that saves a new version of the corpus with the part-of-speech values as attributes on the `<tu>` (timed unit) tags, moving them from the `"tag"` attribute of `<tw>` tags that dominate the `<tu>` tags.

- monitor - an eye-tracking version of the Map Task corpus.

- ICSI - a corpus of meetings. We provide coders for topic segmentation, extractive summarization etc. The entire meeting corpus consists of more than 75 hours of meeting data richly annotated both manually and automatically.

3.3 Setting the CLASSPATH

All of the `.bat`, `.command` and `.sh` scripts in the NXT download have to set the Java `CLASSPATH` before running an NXT program. To compile and run your own NXT programs you need to do the same thing. The classpath normally includes all of the `.jar` files in the `lib` directory, plus the `lib` directory itself. Many programs only use a small proportion of those JAR files, but it's as well to include them all. JMF is a special case: you should find NXT plays media if the `CLASSPATH` contains `lib/JMF/lib/jmf.jar`. However, this will be sub-optimal: on Windows JMF is often included with Java, so you will need no jmf.jar on your `CLASSPATH` at all; on other platforms consult 'How to Play Media in NXT' p.7.

3.4 How to Play Media signals in NXT

NXT plays media using JMF (the Java Media Framework). JMF's support for media formats is limited and it depends on the platform you are using. A list of JMF supported formats is at http://java.sun.com/products/java-media/jmf/2.1.1/formats.html. This list is for JMF 2.1.1, which NXT currently ships with.

There are several ways of improving the coverage of JMF on your platform:

- Performance packs from Sun - these improve codec coverage for Windows and Linux, and are available from the JMF download page. In particular, note that MPEG format isn't supported in the cross-platform version of JMF, but it is in the performance packs.

- Fobs4JMF for Windows / Linux / MacOSX is a very useful package providing Java wrappers for the ffmpeg libraries (C libraries used by many media players which have a wide coverage of codecs and formats). Download; information. Make sure you follow the full installation instructions which involve updating the JMFRegistry and amending your `LD_LIBRARY_PATH`.

- MP3 - There's an MP3 plugin available for all platforms from Sun.

Note:

direct playback from DVDs or CDs is not supported by JMF.

NXT comes with a cross-platform distribution of JMF in the `lib` directory, and the `.bat/.sh` scripts that launch the GUI samples have this copy of JMF on the classpath. On a Windows machine, it is better to install JMF centrally on the machine and change the `.bat` script to refer to this installation. This will often get rid of error messages and exceptions (although they don't always affect performance), and allows JMF to find more codecs.

It is a good idea to produce a sample signal and test it in NXT (and any other tools you intend to use) before starting recording proper, since changing the format of a signal can be confusing and time-consuming. There are two tests that are useful. The first is whether you can view the signal at all under any application on your machine, and the second is whether you can view the signal from NXT. The simplest way of testing the latter is to name the signal as required for one of the sample data sets in the NXT download and try the generic display or some other tool that uses the signal. For video, if the former works and not the latter, then you may have the video codec you need, but NXT can't find it - it may be possible to fix the problem by adding the video codec to the JMF Registry. If neither works, the first thing to look at is whether or not you have the video codec you need installed on your machine. Another common problem is that the video is actually OK, but the header written by the video processing tool (if you performed a conversion) isn't what JMF expects. This suggests trying to convert in a different way, although some brave souls have been known to modify the header in a text editor.

Media on the Mac

NXT ships with some startup scripts for the Mac platform (these are the .command files) that attempt to use FMJ to pass control of media playing from JMF to the native codecs used by the Quicktime player.

If the FMJ approach fails, you should still be able to play media on your Mac but you'll need to edit your startup script. Take an existing command file as a template and change the classpath. It should contain lib/JMF/lib (so jmf.properties is picked up); lib/JMF/lib/jmf.jar and lib/fmj/lib/jffmpeg-1.1.0.jar, but none of the other FMJ files. This approach uses JFFMPEG more directly and works on some Mac platforms where the default FMJ approach fails. It may become the default position for NXT in future.

3.5 Programmatic Controls for NXT

This section describes how to control certain behaviours of NXT from the command line.

These switches can be set using Java properties. Environment variables with the same names and values are also read, though properties will override environment variables. Example:

```
java -DNXT_DEBUG=0 -DNXT_QUERY_REWRITE=true CountQueryResults -c mymeta.xml
     -o IS1003d -q '($s summ)($w w):text($w)="project" && $s^$w'
```

This runs the `CountQueryResults` program with query rewriting on in silent mode (i.e. no messages). Setting environment variables with the same names will *no longer work* .

Java Arguments Controlling NXT Behaviour

`NXT_DEBUG=`*`number`*

> The expected value is a number between `0` and `4`. `0`: no messages; `1`: errors only; `2`: important messages; `3`: warnings; `4`: debug information. The arguments `true` and `false` are also accepted to turn messages on or off.

`NXT_QUERY_REWRITE`

> Values accepted: `true` or `false`; defaults to `false`. If the value is `false`, NXT will automatically rewrite queries in an attempt to speed up execution.

`NXT_LAZY_LOAD`

> Values accepted: `true` or `false`; defaults to `true`. If the value is `false`, lazy loading will not be used. That means that data will be loaded en masse rather than as required. This can cause memory problems when too much data is loaded.

`NXT_RESOURCES_ALWAYS_ASK`

> Values accepted: `false` or `false`; defaults to `false`. If the value is `false`, the user will be asked for input at all points where there is more than one resource listed in the resource file for a coding that needs to be loaded. The

user will be asked even if there are already preferred / forced / defaulted resources for the coding. This should only be used by people who really understand the use of resources in NXT.

`NXT_RESOURCES`

A list of strings separated by commas (no spaces). Each string is taken to be the name of a resource in the resources file for the corpus and is passed to forceResourceLoad so that it must be loaded. Messages will appear if the resource names do not appear in the resource file.

`NXT_ANNOTATOR_CODINGS`

A list of strings separated by semi-colons. If any of the strings are coding names in the metadata file, they are used when populating the list of existing annotators for the 'choose annotator' dialog. If no valid coding names are listed, all available annotators are listed.

3.6 Compiling from Source and Running the Test Suites

- Go into the top level `nxt` directory, decide on a build file to use and copy it to the right directory e.g. `cp build_scripts/build.xml .`. Type `ant` to compile (`ant jar` is perhaps the most useful target to use as it doesn't clean all compiled classes and rebuild the javadoc every time). If there are compile errors, copy the error message into an email and send it to Jonathan or another developer (see the SourceForge members page for emails).

- Run the test suite(s). The NXT test suite is by no means comprehensive but tests a subset of NXT functionality. To run, you need to have the JUnit jar on your CLASSPATH. Then

  ```
  javac -d . test-suites/nom-test-suite/NXTTestScratch.java
  ```

  Now run the tests:

  ```
  java junit.textui.TestRunner NXTTestScratch
  ```

  Again, any errors should be forwarded to a developer.

- If you are making a real public release, Update the `README` file in the top-level `nxt` directory, choosing a new minor or major release number. Commit this to CVS.

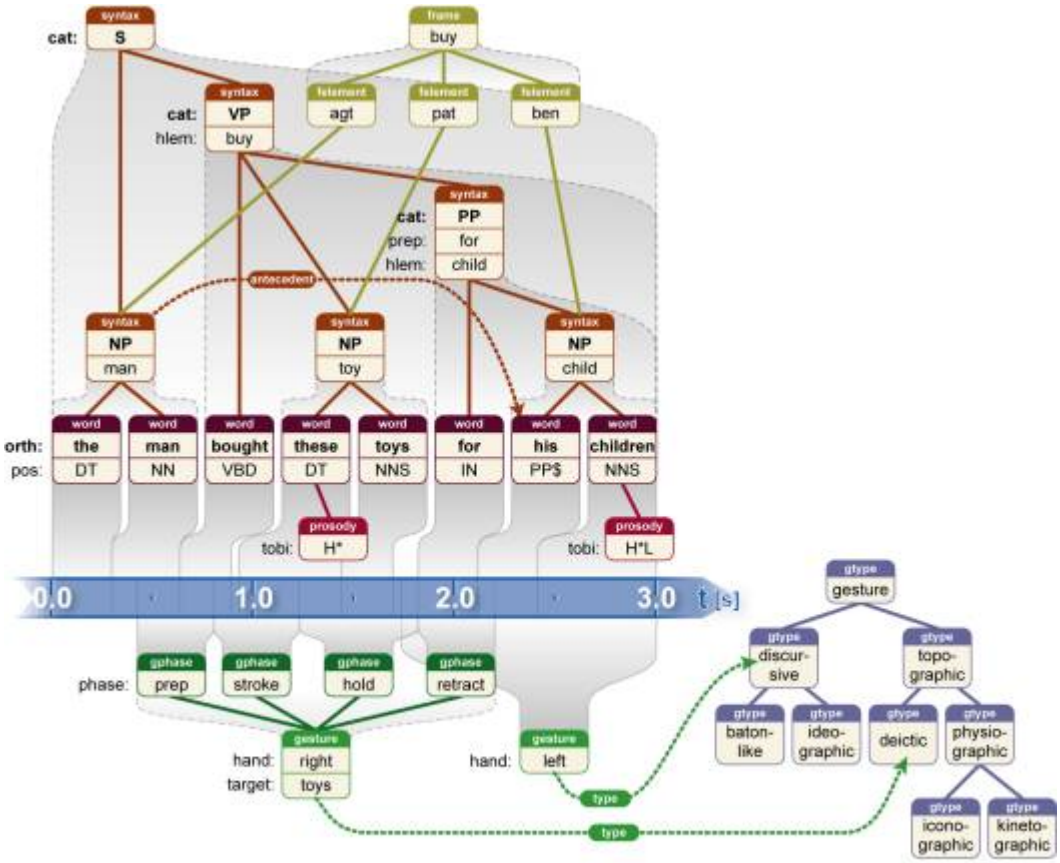
- Now build the release using the `build_scripts/build_release.xml` ant file (use the default target). This compiles everything, makes a zip file of the source, and one of the compiled version for release, and produces the Javadoc. If you're on an Edinburgh machine, copy the Javadoc (in the `apidoc` directory) to `/group/project/webltg/NITE/nxt/apidoc`. Test the shell script examples, and upload the new release to SourceForge.

4 Data

Our approach to data modelling is motivated by the need to have many different kinds of annotations for the same basic language data, for linguistic levels ranging from phonology to pragmatics. There are two reasons why such cross-annotation is prevalent. First, corpora are expensive to collect even without annotating them; projects tend to reuse collected materials where they can. Second, with the advent of statistical methods in language engineering, corpus builders are interested in having the widest possible range of features to train upon. Understanding how the annotations relate is essential to developing better modelling techniques for our systems.

Although how annotations relate to time on signal is important in corpus annotation, it is not the only concern. Some entities that must be modelled are timeless (dictionaries of lexical entries or prosodic tones, universal entities that are targets of referring expressions). Others (sentences, chains of reference) are essentially structures built on top of other annotations (in these cases, the words that make up an orthographic transcription) and may or may not have an implicit timing, but if they do, derive their timings from the annotations on which they are based. Tree structures are common in describing a coherent sets of tags, but where several distinct types of annotation are present on the same material (syntax, discourse structure), the entire set may well not fit into a single tree. This is because different trees can draw on different leaves (gestural units, words) and because even where they share the same leaves, they can draw on them in different and overlapping ways (e.g.,disfluency structure and syntax in relation to words). As well as the data itself being structured, data types may also exhibit structure (for instance, in a typology of gesture that provides more refined distinctions about the meaning of a gesture that can be drawn upon as needed).

The best way to introduce the kind of data NXT can represent is by an example.



The picture, which is artificially constructed to keep it simple, contains a spoken sentence that has been coded with fairly standard linguistic information, shown above the representation of the timeline, and gestural information, shown below it. The lowest full layer of linguistic information is an orthographic transcription consisting of words marked with part-of-speech tags (in this set, the tag `PP$` stands for “personal pronoun”). Some words have some limited prosodic information associated with them in the form of pitch accents, designated by their TOBI codes. Building upon the words is a syntactic

structure — in this formalism, a tree — with a category giving the type of syntactic constituent (sentence, noun phrase, verb phrase, and so on) and the lemma, or root form, of the word that is that constituent's head. Prepositional phrases, or PPs, additionally specify the preposition type. The syntactic constituents are not directly aligned to signal, but they inherit timing information from the words below them. The very same syntactic constituents slot into a semantic structure that describes the meaning of the utterance in terms of a semantic frame (in this case, a buying event) and the elements that fill the roles in the frame (the agent, patient, and beneficiary). The last piece of linguistic information, a link between the syntactic constituent "the man" and the personal pronoun "his", shows that the former is the antecedent of the latter in a coreference relationship.

Meanwhile, the gesture coding shows two timed gestures and their relationship to a static gesture ontology. In the ontology, one type is below another if the former is a subtype of the latter. The first gesture, with the right hand, is a deictic, or pointing, gesture where the target of the pointing is some toys. This gesture is divided into the usual phases of preparation, stroke, hold, and retraction. The second gesture, made with the left hand, is discursive, but the coder has chosen not to qualify this type further. Gesture types could be represented on the gestures directly in the same way as parts of speech are represented for words. However, linking into an ontology has the advantage of making clear the hierarchical nature of the gesture tag set.

All of these kinds of information are used frequently within their individual research communities. No previous software allows them to be integrated in a way that expresses fully how they are related and makes the relationships easy to access. And yet this integration is exactly what is required in order to understand this communicative act fully. No one really believes that linguistic phenomena are independent; as the example demonstrates, deictic speech can only be decoded using the accompanying gesture. Meanwhile, many linguistic phenomena are correlated. Speaker pauses and hearer backchannel continuers tend to occur at major syntactic boundaries, an argument builds up using rhetorical relations that together span a text, postural shifts often signal a desire to take a speaking turn, and so on. The NITE XML Toolkit supports representing the full temporal and structural relationships among different annotations both as a way of keeping all of the annotations together and to allow these relationships to be explored, since understanding them should help our research.

Although the example shows a particular data structure that necessarily makes choices about for instance, how to represent coreferential relationships and what gestures to include in a taxonomy, NXT deliberately does not prescribe any particular arrangement. Instead, it is designed to be theory-neutral. NXT allows users to define their annotations and how they relate to each other, within constraints imposed by its internal data representation, the NITE Object Model. Notice that in the example, although the overall graph is not a tree, it contains trees as prominent components. The NITE Object Model treats annotations as nodes in a set of intersecting trees. Each node in the model must have at most a single set of children, but might have several parents, defining its placement in different trees. Each tree has an ordering for the nodes that it contains, but there is no order for the set of annotations overall. In addition to the intersecting tree structure, each node can have out-of-tree links, called "pointers", to other nodes. In the NITE Object Model, pointers can introduce cycles into the data structure, but parent-child relationships cannot. This makes it technically possible to represent any graph structure in the model, but at a high processing cost for operations involving pointers.

4.1 The NITE Object Model

The NITE Object Model consists of a general graph structure, and then some properties imposed on top of that graph structure that make using that structure more computationally tractable whilst still expressing the sorts of relationships that are prevalent among annotations.

The NITE Object Model is a graph where the nodes are required to have a simple type and may additionally have attribute-value pairs elaborating on the simple type, timings, children that the node structurally dominates, textual content, pointers relating the node to other nodes, and external pointers relating the node to external data not represented in the NITE Object Model. Any individual node may have either children or textual content, but not both.

The simple type is a string.

An attribute is identified by a simple label string and takes a value that conforms to one of three types: a string, a number, or an enumeration. The simple type of the element determines what attributes it can contain. For any element, the simple type plus the attribute-value pairs defined for the element represent its full type.

Timing information can be present, and is represented by reserved start and end attributes containing numbers that represent offsets from the start of the synchronized signals.

The children are represented by an (ordered) list of other nodes.

The textual content is a string. For nodes that have children instead of textual content, some NXT-based tools use an informal convention that the textual content of the node is equivalent to textual content of its descendants, concatenated in order and whitespace-separated.

The pointers are represented by a list of role and filler pairs. A role is a simple label string that has an expected arity, or number of nodes, expected to fill the role: one, or one-or-more. A role is filled by a set of nodes with the expected arity. We sometimes use the term features for these pointers.

The external pointers are also represented by a list of role and filler pairs. A role is again a simple label string with an expected arity of one or one-or-more. The role of an external pointer is filled by a string that specifies a datum in the external reference format, with the details of how the referencing works left to the application program. This can be useful, for instance, in tailored applications that need to cooperate with existing tools that display data in the other format.

The object model also imposes some properties on the parent-child relationships within this general graph structure. Firstly, the parent-child relationships in this graph must be acyclic, so that its transitive closure can be interpreted as a dominance relation. Secondly, there must not be more than one path between any two elements. Because of these constraints, the parent-child graph (which, unlike a tree, allows children to have multiple parents) decomposes into a collection of intersecting tree-like structures, called hierarchies. Each hierarchy has its own structural ordering (similar to an ordered tree), but these orderings must be consistent where hierarchies intersect.

If an element has timing information, the element's start time must be less than or equal to its end time. In addition, if elements in a dominance relation both have timing information, the time interval associated with the ancestor must include that of the descendant. The times of elements need not be consistent with any of the structural orderings. Timing information can thus be used to define an additional partial ordering on the graph, which is not restricted to a single hierarchy.

In the object model, there are no structural or timing constraints imposed on nodes based on the pointers between them. The pointers merely provide additional, arbitrary graph structure over the top of the intersecting hierarchy model.

4.2 The NITE Data Set Model

Our object model is simply an abstract graph structure with a number of properties enforced on it that govern orderings. However, it can be difficult for data set designers to think of their data in terms this abstract, rather than the more usual concepts such as corpus, signal, and annotation. For this reason, we provide a data set model in these familiar terms that can easily be expressed using our object model and from whose structure the essential properties we require regarding orderings and acyclicity fall out. Data set designers use this level of the model to describe their designs, and by providing metadata that expresses the design formally, make it possible to validate the overall structure of any specific data set against their intended design.

Here we describe the main entities and relationships that occur in our data set model.

Data Set Model Concepts

`Observation`

> An observation is the data collected for one interaction — one dialogue or small group discussion, for example.

`Corpus`

> A corpus is a set of observations that have the same basic structure and together are designed to address some research need. For each simple data type, metadata for the corpus determines what attribute-value pairs can be used to refine the type, whether or not elements of that type have timing information and/or children, and what features can be present for them.

`Agent and Interaction`

> An agent is one interactant in an observation. Agents can be human or artificial. We provide the concept of agent so that signals and annotations can be identified as recording or describing the behaviour of a single agent or of the interacting group as a whole. As an example, individual agents speak, but it takes two of them to have a handshake, and possibly the entire set to perform a quadrille. Any signal or annotation involving more than an individual agent counts as belonging to the interaction even if it involves a subset of the agents present.

`Signal`

> A signal is the output from one sensor used to record an observation: for example, an audio or video file or blood pressure data. An observation may be recorded using more than one signal, but these are assumed to be synchronized, so that timestamps refer to the same time on all of them. This can be achieved through pre-editing. Individual

signals can capture either one agent (for instance, a lapel microphone or a close-up camera) or the interaction among the agents (for instance, a far-field microphone or overhead camera).

`Layer`

A layer is a set of nodes that together span an observation in some way, containing all of the annotations for a particular agent or for the interaction as a whole that are either of the same type or drawn from a set of related types. Which data types belong together in a layer is defined by the corpus metadata. For instance, the TEI defines a set of tags for representing words, silences, noises, and a few other phenomena, which together span a text and make up the orthographic transcription. In this treatment, these tags would form a layer in our data set model.

`Time-aligned layer`

A time-aligned layer is a layer that contains nodes timed directly against signal.

`Structural layer`

A structural layer is a layer where the nodes have children. The children of a structural layer are constrained to be drawn from a single layer, which, in order to allow recursive structures, can be itself. Ordinarily nodes in this layer will inherit timing information from their children if their children are timed, but this inheritance can be blocked.

`Featural layer`

A featural layer is a layer where the nodes point to other nodes, but do not contain children or timing information. A featural layer draws together other nodes into clusters that represent phenomena that do not adhere to our timing relationships. For instance, a featural layer might contain annotations that pair deictic gestures with deictic pronouns. Since deictic pronouns and their accompanying gestures can lag each other by arbitrary amounts, there is no sense in which the deictic pair spans from the start of one to the end of the other.

`External reference layer`

External reference layers give a mechanism for pointing from an NXT data graph into some data external to NXT that is not in NXT's data format. In an external reference layer, the nodes point both to other NXT nodes and specify a reference to external data. For instance, an external reference layer might contain annotations that pair transcribed words with references in an ontology represented in OWL.

`Coding`

A coding is a sequence of one or more layers that describe an observation, all either for the same agent or for the interaction as a whole, where each layer’s children are taken from the next layer in the sequence, ending either in a layer with no children or in a layer whose children are in the top layer of another coding. Codings defined in this way consist of tree structures, and the relations among codings allow the intersecting hierarchies of the NITE Object Model. Since most coherent annotations applied to linguistic data fit into tree structures, for many corpora, the codings will correspond to what can be thought of loosely as types of annotation.

`Corpus Resource`

A corpus resource is a sequence of one of more layers that provide reference data to which coding nodes can point. A corpus resource might be used, for instance, to represent the objects in the universe to which references refer, the lexical entries that correspond to spoken word tokens, or an ontology of types for a linguistic phenomena that provides more information than the basic strings given in a node's simple type. The nodes in a corpus resource will not have timing information. For backwards compatibility, NXT corpora may describe individual corpus resources as object sets or ontologies, where object sets are expected to form flat lists and ontologies may have tree structure.

`Code`

A code is an individual data item, corresponding to one node in the NITE Object Model. The metadata declaration for codes of a specific type defines the attribute-value pairs that are allowed for that type. If the code relates to other codes using pointers, the declaration specifies by role in which layer the target of the pointer must be found. Further restrictions on the types allowed as children for any given code arise from the layer in which the code is placed.

Together, these definitions preserve the ordering properties that we desire; intuitively, time-aligned and structural layers are ordered, and timings can percolate up structural layers from a time-aligned layer at the base. The layer structure within a coding prohibits cycles.

The structure of any particular data set is declared in these terms in the metadata file for the corpus and imposed by it; for instance, if you validate a corpus against the metadata, any nodes that violate the layering constraints declared in the metadata will be flagged. However, technically speaking, the NITE Object Model itself is perfectly happy to load and work with data that violates the layer model as long as the data graph itself contains no cycles. A number of previous corpora have violated the layering model deliberately in order to avoid what the designers see as too rigid constraints (see 'Skipping Layers' p.32). We don't recommend this because violations can have unintended consequences unless the designers understand how NXT's loading, validation, and serialization work, and may not continue to have the same effects as NXT development continues.

4.3 Data Storage

NXT corpora are serialized, or saved to disk, into many different XML files in which the structure of the data graph in the NITE Object Model is expressed partly using the XML structure of the individual files and partly using `links` between files.

4.3.1 Namespacing

NXT is designed to allow data to be divided into namespaces, so that different sites can contribute different annotations, possibly even for the same phenomena, without worrying about naming conflicts. Any code or attribute in a NITE Object Model can have a name that is in a namespace, by using an XML namespace for the XML element or attribute corresponding to the NOM code or attribute. However, see NXT bug 1633983 for a description of a bug in how the query parser handles namespaced data.

The `nite:` namespace, "http://nite.sourceforge.net/", is intended for elements and attributes that have a special meaning for NXT processing, and is used for them by default. This covers, for instance, the ids used to realize out-of-file links and the start and end times that relate data nodes to signal. Although use of the `nite:` namespace makes for a clearer data set design, nothing in the implementation relies on it; the names of all the elements and attributes intended for the `nite:` namespace can be changed by declaring an alternative in the metadata file, and they do not have to be in the `nite:` namespace.

In addition the `nite:` namespace, corpora that choose XLink style links (see) make use of the `xlink:` namespace, `http://www.w3.org/1999/xlink`.

As for any other XML, NXT files that make use a namespace must declare the namespace. This includes the metadata file. One way of doing this is to add the declaration as an attribute on the document (root) element of the file. For instance, assuming the default choices of `nite:root` for the root element name and `nite:id` for identifiers, the declaration for the `nite:` namespace might look like this:

```
<nite:root nite:id="stream1" xmlns:nite="http://nite.sourceforge.net/">
...
</nite:root>
```

For more information about namespacing, see .

4.3.2 Coding Files

The data set model places codes strictly in layers, with codings made up of layers that draw children from each other in strict sequence. This is so that within-coding parent-child relationships can be represented using the structure of the XML storage file itself. For a single observation, each coding is stored as a single XML file. The top of each XML tree is a `root` element that does not correspond to any data node, but merely serves as a container for XML elements that correspond to the nodes in the top layer of the coding being represented. Then the within-coding children of a node will be represented as XML children of that element, and so on. Within a layer, represented as all elements of a particular depth from the XML file root (or set of depths, in the case of recursive layers), the structural order of the nodes will be the same as the order of elements in the XML file.

4.3.3 Links

The structure of a coding file suffices for storing information about parent-child relationships within a single coding. However, nodes at the top and bottom layers of a coding may have parent-child relationships with nodes outside the cod-

ing, and any node in the coding may be related by pointer to nodes that are either inside or outside the coding. In addition, nodes in external reference layers may be related to external data stored in files that are not in XML format. These relationships are expressed in the XML using `links`. NXT relies on having two reserved XML element types for representing links, one for children and one for pointers, including external reference pointers. The XML element types are, by default, `nite:child` and `nite:pointer`, but they can be changed in the section of the metadata file that declares reserved elements (see 'Reserved Elements and Attributes' p.17). If a NOM node has an out-of-coding child or a pointer, then the XML element that corresponds to it will represent this relationship by containing an XML child of the appropriate type.

Links can be represented in either LTXML1 style or using XLink, but the choice of link style must be uniform throughout a corpus. The choice is specified on the `<corpus>` declaration within the metadata file (see 'Top-level corpus description' p.16). NXT can be used to convert a corpus from LTXML1 link style to XLink style (see).

With either link style, the document (XML file) for a reference must be specified without any path information, as if it were a relative URI to the current directory. NXT uses information from the metadata file about where to find files to decode the links. This creates the limitation for external XML processes that use the links that either all the XML files must be in one directory or the external process must do its own path resolution.

Also, with either link style, NXT can read and write ranges in order to cut down storage space. A range can be used to represent a complete sequence of elements drawn in order from the data, and references the sequence by naming the first and last element. Ranges must be well-formed, meaning that they must begin and end with elements from the same layer, and in a recursive layer, from elements at the same level in the layer. To make it easier to use external XML processing that can't handle ranges, NXT can be used to convert a corpus that uses ranges into one that instead references every element individually (see).

4.3.3.1 LTXML1 Style Links

In LTXML1 style, pointers and children will have an (un-name-spaced) `href` attribute that specifies the target element.

The following is an example in the LTXML1 style of an NXT pointer link (using the default `nite:pointer` element) that refers to one element.

```
<nite:pointer role="foo" href="q4nc4.g.timed-units.xml#id('word_1')"/>
```

The following is an example in the XLink style of an NXT child link (using the default `nite:child` element) that refers to a range of elements.

```
<nite:child href="q4nc4.g.timed-units.xml#id('word_1')..id('word_5')"/>
```

4.3.3.2 XLink Style Links

When using the XLink style, the target element should conform to the XLink standard for describing links between resources , and use the XPointer framework with the XPointer xpointer() Scheme to specify the URI. NXT only implements a small subset of these standards, and so requires a very particular link syntax. NXT elements that express links in this style must include the `xlink:type` attribute with the value `simple`, and specify the URI in the `xlink:href` attribute. The xpointer reference must either refer to a single element within a document by id or by picking out the range of nodes between two nodes using the `range-to` function with the endpoints specified by `id`.

The following is an example in the XLink style of an NXT pointer link (using the default `nite:pointer` element) that refers to one element.

```
<nite:pointer role="foo" xlink:href="o1.words.xml#xpointer(id('w_1'))"
      xlink:type="simple"/>
```

The following is an example in the XLink style of an NXT child link (using the default `nite:child` element) that refers to a range of elements.

```
<nite:child xlink:href="o1.words.xml#xpointer(id('w_1')/range-to(id('w_5')))"
      xlink:type="simple"/>
```

4.3.4 Data And Signal File Naming

The actual names used for data and signal files in an NXT corpus depend on the codings and signals defined for it. Rather than containing a complete catalog mapping individual codings and signals to individual files on disk, NXT assumes consistent naming across a corpus. It constructs the names for a file from pieces of information in the data set model (see

'NITE Data Set Model' p.12). Both these pieces of filenames and the paths to directories containing the various kinds of files are specified in the metadata file (see 'Metadata' p.16).

4.3.4.1 Signal Files

For signal files recording interaction, the name of the file is found by concatenating the *name* of the `observation`, the *name* of the `signal`, and the `extension` declared for the signal, using dots as separators. For instance, the `overhead` video, an AVI with extension `avi`, for observation `o1` would be stored in file `o1.overhead.avi`.

For signal files recording individuals, the filename will additionally have the `agent` *name* after the `signal` *name*. For instance, the `closeup` video, an AVI with extension `avi`, for agent `A` in observation `o1` would be stored in file `o1.A.closeup.avi`.

4.3.4.2 Coding Files

For coding files representing interaction behaviour, the name of the XML file is found by concatenating the *name* of the `observation`, the *name* of the `coding`, and the extension `xml`, using dots as separators. For instance, the `games` coding for observation `o1` would be stored in file `o1.games.xml`.

For coding files representing agent behaviour, the name of the XML file will additionally have the `agent` *name* after the `observation` *name*. For instance, the `words` coding for agent `giver` in observation `o1` would be stored in file `o1.giver.moves.xml`.

4.3.4.3 Corpus Resources and Ontologies

Corpus resources and ontologies are for the entire corpus, not one per observation. The name of the XML file is found by concatenating the `name` of the corpus resource or onotology and the extension `xml`.

4.4 Metadata

Because NXT does not prescribe any particular data representation, in order to load a corpus it requires `metadata` declaring what observations, codings, signals, layers, codes, and so on come with a particular corpus, and where to find them on disk. This metadata is expressed in a single file, which is also in XML format. Each of the example data set extracts comes with example metadata that can be used as a model. The DTD and schema for NXT metadata can be found in the `lib/dtd` and `lib/schema` directories of the NXT distribution, respectively.

4.4.1 Preliminaries

4.4.1.1 Attribute definitions

A number of sections of the metadata (`<code>`, `<ontology>`, `<object-set>`) rely on the same basic mechanism for defining attributes. Attributes can have three different types: string, meaning free text; number, where any kind of numeric value is permitted; or enumerated, where only values listed in the enclosed `value` elements are permitted. They are defined using an `<attribute>` tag where the `name` attribute gives the name of the attribute and the `value-type` attribute, the type. For enumerated attributes, the attribute declaration must also include the enumerated values within `<value>` tags. For instance,

```
<attribute name="affiliation" value-type="string"/>
```

defined an attribute named "affiliation" that can have any string value, whereas

```
<attribute name="gender" value-type="enumerated">
        <value>male</value>
        <value>female</value>
</attribute>
```

defines an attribute named "gender" that can have two possible values, "male" and "female".

4.4.2 Top-level corpus description

The root element of a metadata file is `corpus` and here's an example of what it looks like:

```
<corpus description="Map Task Corpus" id="maptask"
        links="ltxml1" type="standoff" resource_file="resource.xml">
        ...
</corpus>
```

The important attributes of the `corpus` element are `links` and `type`. The `type` attribute should have the value `="standoff"`. The previous use of `simple` corpora is deprecated. The `links` attribute defines the syntax of the standoff links between the files. It can be one of: `="ltxml1"` or `xpointer`. See 'Links' p.14 for an explanation of these two link styles. The `resource` attribute is optional: if it is present it specifies a resource file which will be parsed by NXT (from versions 1.4.1 onwards). Resource files provide a more flexible way to manage a large NXT corpus, particularly where many annotators and automatic processes could provide competing annotations for the same things. See 'Resource Files' p.26 for an explanation of resource files and their format.

4.4.3 Reserved Elements and Attributes (optional)

There are a number of elements and attributes that are special, or `reserved`, in NXT because they do not (or do not just) hold data but are used in NXT processing, for instance, in order to identify the start and end times of some timed annotation. The `<reserved-elements>` and `<reserved-attributes>` sections of the metadata-file can be used to override their default values. They contain as children declarations for each element or attribute separately; each child declaration uses a different element name (see table), with the `name` attribute specifying the name to use for that element or attribute in the NXT corpus being described. If the element in the metadata for declaring the name to use for a particular element or attribute is missing, then NXT will use the default value for that attribute. The `<reserved-attributes>` and `<reserved-attributes>` sections of the metadata-file can be omitted entirely if no declarations are required for the corpus being described.

Caution:

At present, off-line data validation can not fully handle alternative names (see 'Data validation' p.28).

4.4.3.1 Reserved Attributes (optional)

The following table shows each of the reserved attributes along with the name of the element in the metadata file used to declare its name and the default value.

Reserved Attributes

attribute	metadata tag name	default value
Root / stream element name	stream	nite:root
Element identifier	identifier	nite:id
Element start time	starttime	nite:start
Element end time	endtime	nite:end
Agent	agentname	agent
Observation	observationname	-
Comment	commentname	comment
Key Stroke	keystroke	keystroke
Resource	resourcename	-

For instance, a metadata declaration that changes just the names of the id, start, and end time attributes might look like this:

```
<reserved-attributes>
        <identifier name="identifier"/>
        <starttime name="starttime"/>
        <endtime name="endtime"/>
</reserved-attributes>
```

They are used in NXT processing as follows.

Reserved Attribute Meanings

`Stream`

The `stream` attribute occurs at the root elements of all XML data files in a corpus apart from ontology files and corpus resources, and gives a unique identifier for the XML document in the file. This attribute does not form part of the data represented in the NITE Object Model, but is required for serialization.

`Identifier`

Identifiers are required on all elements in an NXT corpus, and are used by NXT to resolve out-of-file links when loading and to maintain the correspondence between the data and the display in GUI tools.

`Start and End Times`

Start and end times may appear on time-aligned elements. They give the offset from the beginning of a signal (or set of synchronized signals) in seconds.

`Agent and Observation`

These reserved attributes describe not attributes that should occur in the XML data files, but attributes that can be added automatically as data is loaded, for access in the NITE Object Model. Normally, corpora do not explictly represent the name of the observation which an annotation describes, or the name of the agent if it is an annotation for an individual, since this information is represented by where in the set of XML files the data is stored. It would take a great deal of space to stick this information on every data element, but it is useful to have it in the NOM, for instance, so that queries can filter results based on it. The `agentname` and `observationname` declarations specify the names to use for these attributes in the NOM. The attributes will be added at load time to every element that doesn't already have an attribute with the same name.

`Comment`

The `comment` attribute gives space to store an arbitrary string with any data element in the corpus. It is typically used for temporary information to do with data management or to represent the cause of uncertainty about an annotation.

`Keystroke`

Any element can have an associated `keystroke`. This is normally used to represent keyboard shortcuts for elements in an ontology, though it can be used for other purposes. The value is simply a string, and what application programs do with the string (if anything) is up to them.

`Resource`

If the metadata refers to a `resource` file, this attribute contains the ID of the resource this element is associated with, if any.

4.4.3.2 Reserved Elements (optional)

The following table shows each of the reserved elements along with the name of the element in the metadata file used to declare its name and the default value.

Reserved Elements

element	metadata tag name	default value
Pointer	pointername	nite:pointer
Child	child	nite:child
Stream element	stream	nite:root

For instance, a metadata declaration covering just the names of pointer elements might look like this:

```
<reserved-elements>
        <pointername name="mypointer"/>
</reserved-elements>
```

4.4.3.3 Example

Changing the reserved element and attribute names from the default affects the representation that NXT expects within the individual XML data files. For instance, suppose we include the following in the metadata file:

```
<reserved-attributes>
        <stream name="stream"/>
        <identifier name="identifier"/>
        <starttime name="starttime"/>
        <endtime name="endtime"/>
        <agentname name="who"/>
        <observationname name="obs"/>
        <commentname name="mycomment"/>
        <keystroke name="mykey"/>
</reserved-attributes>
<reserved-elements>
        <pointername name="mypointer"/>
        <child name="mynamespace:child"/>
        <stream name="stream"/>
</reserved-elements>
```

Further suppose that the metadata file specifies the use of ltxml1-style links and goes on to define a coding file containing one time-aligned layer, with one code, `<word`, that declares no further attributes but can contain syllables as children, and can point to syntactic constituents using the "antecedent" role. Then the full XML file representing those words might look like this:

```
<stream>
        <word identifier="word_1" starttime="1.3" endtime="1.5"
                <mypointer role="antecedent" href="obs1.syntax.xml#ante_2"/>
                <mynamespace:child href="obs1.syllables.xml#syllable_1"/>
        </word>
...
</stream>
```

4.4.4 CVS Details (optional, beta)

Many projects keep their annotations in a CVS repository. Concurrent version control (see) allows different people to edit the same document collaboratively, maintaining information about who has done what to what file when. One NXT contributor is adding functionality to allow NXT GUIs to work directly from a CVS repository rather than requiring the annotator to check out data from CVS and then commit changes as additional steps. The `cvsinfo` section of the metadata declares where to find the CVS repository for these purposes. It has three attributes, all of which are required: `protocol` (one of `pserver, ext, local, sspi`); `server` (the machine name on which the CVS server is hosted); and `module` (the top level directory within the CVS repository where the corpus is found). For instance,

```
<cvsinfo protocol="pserver" server="cvs.inf.ed.ac.uk" module="/disk/cvs/ami"/>
```

4.4.5 Independent Variables on Observations

4.4.6 Agents (optional)

A corpus is a set of observations of language behaviour that are all of the same basic genre, all of which conform to the same declared data format. For each corpus, there will be a set number of agents, or individuals (whether human or artificial) whose behaviour is being observed. The `agents` section of the metadata contains `agent` declarations for each one. Each `agent` declaration must contain a `name` attribute, which can be any string that does not contain whitespace. The agent name will be used to name data and signal files. If the reserved attribute `agentname` is declared (see 'Reserved Attributes' p.17, it will also be available as an attribute on all agent annotations during queries. Agent declarations may

also contain a `description`, which can be any string, and is intended to be a short, human-readable description. Its use is application-specific.

Note that this strategy for naming agents gives uniformity throughout the corpus; every observation in a corpus will use the same agent names. That is, the name of an agent expresses its role in the language interaction being observed. Typical agent names are e.g. `system` and `user` for human-computer dialogue. For corpora without discernible roles, the labelling is often arbitrary, using letters to designate individuals, or based on seating. NXT deliberately places personal information about the individuals that fill the agent roles for specific observations in corpus resources, not the metadata, so that it is accessible from the query language.

Usually, it is obvious how many agents to use for a corpus; e.g., two for dialogue, five for five-person discussion. However, there are a few non-obvious cases.

For corpora of discussions where the size varies but everything else is the same, in order to treat all observations in the same metadata file, you must declare the largest number of agents required by any observation so that one metadata file can be used throughout. In this case, it will be impossible to tell whether an agent speaks no words because they were absent or because they were silent without encoding this information in a corpus resource.

For monologue and written text, it is possible either to declare the corpus as having one agent or as having no agents, treating every annotation as a property of the interaction. The only difference is in how the data files will be named.

4.4.6.1 Example

This example, used by the Map Task Corpus, declares two agents, the route giver and the route follower.

```
<agents>
        <agent name="g" description="giver"/>
        <agent name="f" description="follower"/>
</agents>
```

4.4.7 Signals (optional)

Most (but not all) corpora come with either a single signal for each observation, or a set of signals that together record the observation from different angles. In a corpus, the usual aim is to capture all observations with the same recording set-up, resulting in the same set of recordings for all of them. This section of the metadata declares what signals to expect for the observations in the corpus and where they reside on disk. There is an explanation of how filenames are concatenated from parts in ' Data And Signal File Naming' p.15. Filenames are case-sensitive.

At the top level of this section, the `signals` declaration uses the `path` attribute to specify the initial part of the path to the directory containing signals. If the path is relative, it is calculated relative to the location of the metadata file, not to the directory from which the java application is started. The default path is the current directory. The `signals` declaration can also declare a `pathmodifier`, which concatenates additional material to the end of the declared path. For any given observation, the additional material will be the result of replacing any instances of the string `observation` with the name of that observation. This allows the signals for a corpus to be broken down into subdirectories for each observation separately.

Below the `signals` tag, the metadata is divided into two sections, `agent-signals` and `interaction-signals`, for agent and interaction signals, respectively.

Within these sections, each type of signal for a corpus has its own `signal` declaration, whch takes an `extension` giving the file extension; `name`, which is used as part of the filename and should not contain whitespace; `format`, which is a human-readable description of the file format, and `type`, which should be `="audio"` or `video`; and again a `pathmodifier`, treated in the same way as the pathmodifier on the `signals` declaration, and appended to the path after it. Of these attributes, `extension` and `name` are required, and the rest are optional.

Occasionally the recording setup will not be entirely uniform over a corpus, with individual signals missing or individual observations having one signal or another from the setup, but not both. In these cases, you must over-declare the set of signals as if the corpus were uniform and treat these signals as missing. The main ramification of this in software is that GUIs will give users the choice of playing signals that turn out not to be available unless they check for existence first.

4.4.7.1 Example

Assume that there is an observation named `o1` and agents `g` and `f`. Then this declaration:

```
<signals path="../signals/">
    <agent-signals>
       <signal extension="au" format="mono au"
               name="audio" type="audio"/>
    </agent-signals>
    <interaction-signals>
       <signal extension="avi" format="stereo avi"
              name="interaction-video" type="video"/>
    </interaction-signals>
</signals>
```

will cause NXT to expect to find the following media files at the following paths:

```
../signals/o1.g.audio.au
../signals/o1.f.audio.au
../signals/o1.interaction-video.avi
```

If we were to add the pathmodifier `observation` to the `signals` tag, NXT would look for the signals at, e.g., `../signals/o1/o1.interaction-video.avi`. If we then also added the pathmodifier `video` for the interaction-video, leaving the other signal with no additional pathmodifier, i.e. declaring as

```
<signals path="../signals/" pathmodifier="observation">
    <agent-signals>
        <signal extension="au" format="mono au"
              name="audio" type="audio"/>
    </agent-signals>
    <interaction-signals>
        <signal extension="avi" format="stereo avi"
              name="interaction-video" type="video"
              pathmodifier="video"/>
    </interaction-signals>
</signals>
```

NXT would look for the signals as follows.

```
../signals/o1/o1.g.audio.au
../signals/o1/o1.f.audio.au
../signals/o1/video/o1.interaction-video.avi
```

4.4.8 Corpus Resources (optional)

A corpus resource is a set of elements that are globally relevant in some way to an entire corpus. They are not as strictly specified as ontologies or object sets (below). They will probably eventually replace the use of those things. Typically these will be files that come from the original application and can be used almost without alteration. You may specify the exact hierarchical breakdown of such a file, but typically there will just be one recursive layer (pointing to itself) that specifies all the codes permissible. Here is an example where the resource describes participants in a meeting corpus:

```
<corpus-resources path=".">
        <corpus-resource-file name="speakers" description="meeting speakers">
        <structural-layer name="speaker-layer"
                recursive-draws-children-from="speaker-layer">
        <code name="speaker">
                <attribute name="id" value-type="string"/>
                <attribute name="gender" value-type="enumerated">
                        <value>male</value>
                <value>female</value>
                </attribute>
        </code>
        <code name="language">
                <attribute name="name" value-type="string"/>
                <attribute name="region" value-type="string"/>
        </code>
```

```
        <code name="age" text-content="true"/>
        </structural-layer>
        </corpus-resource-file>
</corpus-resources>
```

The `path` attribute on the `corpus resources` element tells NITE where to look for resources for this corpus. A corpus resource has a `name` attribute which is unique in the metadata file. Combined with the `name` attribute of an individual resource, we get the filename. The `name` attribute can also be used to refer to this object set from a 'codings' p.23 layer.

The contents of an individual corpus resource are defined in exactly the same manner as 'codings' p.23 layers within codings.

4.4.9 Ontologies (optional)

An ontology is a tree of elements that makes use of the parent/child structure to specify specializations of a data type. In the tree, the root is an element naming some simple data type that is used by some annotations. In an ontology, if one type is a child of another, that means that the former is a specialization of the latter. We have defined ontologies to make it simpler to assign a basic type to an annotation in the first instance, later refining the type. Here's an example of an ontology definition:

```
<ontologies path="../xml/MockCorpus">
        <ontology description="gesture ontology" name="gtypes"
          element-name="gtype" attribute-name="type"/>
</ontologies>
```

The `path` attribute on the `ontologies` element tells NITE where to look for ontologies for this corpus. An ontology has a `name` attribute which is unique in the metadata file and is used so that the ontology can be pointed into (e.g. by a coding layer - see below). It also has an attribute `element-name`: ontologies are a hierarchy elements with a single element name: this defines the element name. Thirdly, there is an attribute `attribute-name`. This names the privileged attribute on the elements in the ontology: the attributes that define the type names.

The above definition in the metadata could lead to these contents of the file gtypes.xml - a simple gesture-type hierarchy.

```
<gtype nite:id="g_1" type="gesture" xmlns:nite="http://nite.sourceforge.net/">
   <gtype nite:id="g_2" type="discursive">
          <gtype nite:id="g_3" type="baton-like"/>
          <gtype nite:id="g_4" type="ideographic"/>
   </gtype>
   <gtype nite:id="g_5" type="topographic">
          <gtype nite:id="g_6" type="deictic"/>
          <gtype nite:id="g_7" type="physiographic">
                 <gtype nite:id="g_8" type="iconographic"/>
                 <gtype nite:id="g_9" type="kinetographic"/>
                </gtype>
        </gtype>
</gtype>
```

An ontology can use any number of additional, un-privileged attributes, as long as they are declared in the metadata for the ontology using an `<attribute>` tag. For example, to extend the ontology above with a new attribute, foo, with possible values bar and baz, the declaration would be as follows:

```
<ontology description="gesture ontology" name="gtypes"
        element-name="gtype" attribute-name="type">
        <attribute  name="foo" type="enumerated">
                <value>bar</value>
                <value>baz</value>
        </attribute>
</ontology>
```

4.4.10 Object Sets (optional)

An object is an element that represents something in the universe to which an annotation might wish to point. An object might be used, for instance, to represent the referent of a referring expression or the lexical entry corresponding to a word

token spoken by one of the agents. When an element is used to represent an object, it will have a data type and may have features, but no timing or children. An object set is a set of objects of the same or related data types. Object sets have no inherent order. Here is a possible definition of an object set - imagine we want to collect a set of things that are referred to in a corpus like telephone numbers and town names:

```
<object-sets path="/home/jonathan/objects/">
        <object-set-file name="real-world-entities" description="">
  <code name="telephone-number">
               <attribute name="number" value-type="string"/>
  </code>
  <code name="town">
          <attribute name="name" value-type="string"/>
  </code>
        </object-set-file>
</object-sets>
```

The `path` attribute on the `object-sets` element tells NITE where to look for object sets on disk for this corpus. Combined with the `name` attribute of an individual object set we get the filename. The `name` attribute is also used to refer to this object set from a coding layer (see below).

The `code` elements describe the element names that can appear in the object set, and each of these can have an arbitrary number of attributes. The above spec describes an object set in file /home/jonathan/objects/real-world-entities.xml which could contain:

```
<nite:root nite:id="root_1">
        <town nite:id="town3" name="Durham"/>
        <telephone-number nite:id="num1" number="0141 651 71023"/>
        <town nite:id="town4" name="Edinburgh"/>
        <town nite:id="town1" name="Oslo"/>
</nite:root>
```

where the contents are unordered and can occur any number of times.

4.4.11 Codings and Layers

Here we define the annotations we can make on the data in the corpus. Annotations are specified using codings and layers, and we start with an example.

```
<codings path="/home/jonathan/MockCorpus">
    <interaction-codings>
        <coding-file name="prosody" path="/home/MockCorpus/prosody">
            <structural-layer name="prosody-layer"
                              draws-children-from="words-layer">
                <code name="accent">
                    <attribute name="tobi" value-type="string"/>
                </code>
            </structural-layer>
        </coding-file>
        <coding-file name="words">
            <time-aligned-layer name="words-layer">
                <code name="word" text-content="true">
                    <attribute name="orth" value-type="string"/>
                    <attribute name="pos" value-type="enumerated">
                        <value>CC</value>
                        <value>CD</value>
                        <value>DT</value>
                    </attribute>
                    <pointer number="1" role="ANTECEDENT"
                             target="phrase-layer"/>
                </code>
            </time-aligned-layer>
        </coding-file>
```

```
    </interaction-codings>
</codings>
```

First of all, the `codings` element has a `path` attribute which (as usual) specifies the directory in which codings will be loaded from and saved to by default. Note that any `coding-file` can override this default by specifying its own `path` attribute (from release 1.3.0 on). Codings are divided into `agent-codings` and `interaction-codings` in exactly the way that signals are (we show only interaction codings here). Each `coding file` will represent one entity on disk per observation (and per agent in the case of agent codings).

The second observation is that codings are divided into layers. Layers contain `code` elements which define the valid elements in a layer. The syntax and semantics of these `code` elements is exactly as described for 'object sets' p.22.

From 25/04/2006 Layers can point to each other using the `draws-children-from` attribute and the name of another layer. If your build is older, use the now-deprecated `points-to` attribute.

For recursive layers like syntax, use the attribute `recursive="true"` on the layer to mean that elements in the layer can point to themselves.

The attribute `recursive-draws-children-from=`*`layer-name`* means that elements in the layer can recurse but they must "bottom out" by pointing to an element in the named layer. With builds pre 25/04/2006, use the now-deprecated `recursive-points-to` attribute.

Layers are further described by their four types which are all described in detail in 'layer definition' p.13.

Layer types

`Time-aligned layer`

elements are directly time-stamped to signal.

`Structural layer`

elements can inherit times from any time-aligned layer they dominate. Times are not serialized with these elements by default. Structural layers can be prevented from inheriting times from their children. This is important as it is now permitted that parents can have temporally overlapping children so long as the times are not inherited. In order to make use fof this, use the attribute `inherits-time="false"` on the `structural-layer` element. Allowing parents to inherit time when their children can overlap temporally may result in unexpected results from the search engine, particularly where precedence operators are used.

`Featural layer`

Elements can have no time stamps and cannot dominate any other elements - they can only use pointers.

`External reference layer`

An external reference layer is one which contains a set of standard NITE elements each of has a standard `nite:pointer` to an NXT object, and an external pointer to some part of a data structure not represented in NXT format. The idea is that when an application program encounters such an external element, it can start up an external program with some appropriate arguments, and highlight the appropriate element in its own data structure.

On disk, the above metadata fragment could describe the file `/home/jonathan/MockCorpus/o1.prosody.xml` for observation o1:

```
<nite:root nite:id="root1">
        <accent nite:id="acc1" tobi="high">
                <nite:child href="o1.words.xml#w_6"/>
                <nite:child href="o1.words.xml#w_7"/>
        </accent>
        <accent nite:id="acc1" tobi="low">
                <nite:child href="o1.words.xml#w_19"/>
                <nite:child href="o1.words.xml#w_20"/>
        </accent>
</nite:root>
```

A note on effective content models: the DTD content model equivalent of this layer definition

```
<structural-layer name="prosody-layer" draws-children-from="words-layer">
        <code name="high"/>
        <code name="low"/>
</structural-layer>
```

Would be `(high|low)*`. However, if a `code` has the attribute `text-content` set to the value `="true"` (as for the element `word` 'codings example' p.23) the content model for this element is overridden and it can contain only text. This is the only way to allow textual content in your corpus. Mixed content is not allowed anywhere.

A metadata fragment looks like this:

```
<coding-file name="external" path="external">
        <external-reference-layer element-name="prop"
                  external-pointer-role="owl_pointer" content-type="text/owl"
                layer-type="featural" name="prop-layer" program="protege">
                <pointer number="1" role="da" target="words-layer"/>
                <argument default="owl_file_1.owl" name="owl_file"/>
                <argument default="arg_value" name="further_arg"/>
        </external-reference-layer>
</coding-file>
```

The corresponding data looks like this:

```
<propara>
   <nite:external_pointer role="owl_pointer"  href="owlid42"/>
   <nite:child href="IS1008a.A.words.xml#id(IS1008a.A.words0)"/>
</propara>
```

In the metadata fragment, you can choose to explicitly name the program that is called using the `program` attribute, or you can specify the content-type of the external file using a `content-type` attribute (not shown in the metadata fragment). Both are treated as String values and not interpreted directly by NXT.

4.4.12 Callable Programs (optional)

To help with housekeeping it's useful to know what programs have been written for the corpus and how to call them. This also allows NXT's top level interface list the programs and run them. Each `callable-program` contains a list of required arguments. for example, a program described thus:

```
<callable-programs>
    <callable-program name="SwitchboardAnimacy" description="animacy checker">
        <required-argument name="corpus" type="corpus"/>
        <required-argument name="prefix" default=""/>
        <required-argument name="observation" type="observation"/>
    </callable-program>
</callable-programs>
```

Would be called `java SwitchboardAnimacy -corpus` *`metadata-path`* `-prefix -observation` *`obs-name`*. The `type` attribute can take one of two values: `="corpus"` meaning that the expected argument is the metadata filename and `observation` meaning the argument is an observation name. Arguments can also have default values. Note also that the argument name or the default values can be empty strings.

4.4.13 Observations

Each observation in a corpus must have a unique name which is used in filenames. This is declared in a list of observations using the `name` attribute, for instance, like this:

```
<observations>
        <observation name="q4nc4"/>
        <observation name="q3nc8"/>
</observations>
```

NXT currently includes the option of declaring two additional types of information for each observation: its categorization according to the observation variables that divide the corpus into subsets, and some very limited data management information about the state of coding for the observation. We expect in future to rethink our approach to data management which will probaby mean removing this facility from the metadata.

It has been pointed out that one might expect observations to have information mapping from agent (roles) to personal information about the individuals filling them in that observation (age, dialect, etc.). We don't propose a specific set of kinds of information one might wish to retain, because in our experience different projects have different needs (but see, for instance, the ISLE/IMDI metadata initiative). We also don't provide a specific way of storing it. This is partly because some of the information that projects retain falls under data protection and some of it doesn't, so there are issues about how it should be designed. At the moment, the best one can do is define a set of `variables` that together give the information one is looking for. We intend further improvements that will allow the corpus designer to specify a structure for the information and will allow private information to be kept in a separate file that is linked to from the metadata. Currently, the query language doesn't give access to the metadata about an observation, which means that it is only useful for deciding programmatically which observations to load as a filter on the entire corpus set, not for any finer-grained filtering. This also is something we hope to look at. Meanwhile, given these shortcomings, sometimes the best option is to store any detailed information in a separate file of one's own design and build variables that link agent roles to individuals in the separate file by idref.

4.5 Dependency Structures

Resource files are an adjunct to metadata files that provide more flexible support for large cross-annotated corpora. A resource is loosely a set of files that instantiate a particular `coding` in the metadata file. For example, if both `fred` and `doris` have taken part in manual dialogue-act annotation, and an automatic process has also been run to derive dialogue acts for all or part of the corpus, all three would have their own resource in the resource file (see example below).

Resource files should also specify dependencies between resources. This helps to ensure that coherent groups of annotations are loaded together. For example, an automatic dialogue act resource that was run over manual transcription must specify a different word-level dependency than a process run over ASR (automatic speech recognition) output.

Once a resource file is present for a corpus, loading multiple versions of the same coding becomes simpler, provided IDs are unique within the corpus. New annotation elements can even be added to the corpus while this kind of *reliability* data is loaded, because of the separation that resources afford us.

There is a simple API for creating resources programatically and it is hoped that a set of higher-level utilities will emerge to make this process easier.

4.5.1 Resource File Syntax

The example resource file below illustrates most of the features provided by resource files.

```
<resources>
  <resource-type coding="da-types">
    <resource id="datypes" description="DA types" type="manual"
      path="ontologies"/>
  </resource-type>
  <resource-type coding="dialog-act" default="true">
    <virtual-resource id="da_gold">
      <dependency observation="IS1008b" idref="da_doris"/>
      <dependency observation=".*" idref="da_fred"/>
      <dependency observation=".*" idref="da_doris"/>
    </virtual-resource>
    <resource id="da_doris" description="Dialogue acts manual version"
         type="manual" annotator="Doris" path="dialogueActs/doris">
      <dependency observation=".*" idref="AMIwordsref1"/>
      <dependency observation=".*" idref="datypes"/>
    </resource>
    <resource id="da_fred" description="Dialogue acts manual version"
         type="manual" annotator="Fred" path="/home/jonathan/fredDAs">
      <dependency observation=".*" idref="AMIwordsref1"/>
      <dependency observation=".*" idref="datypes"/>
```

```
    </resource>
    <resource id="da_auto1" description="Automatic DAs over manual words"
         type="automatic" path="automanda1">
      <dependency observation=".*" idref="AMIwordsref1"/>
      <dependency observation=".*" idref="datypes"/>
    </resource>
    <resource id="da_auto1" description="Automatic DAs over ASR"
         type="automatic" path="autoasrda1">
      <dependency observation=".*" idref="AMIwordsASRa1"/>
      <dependency observation=".*" idref="datypes"/>
    </resource>
  </resource-type>

  <resource-type coding="words">
    <resource id="AMIwordsref1" description="manual transcription"
         type="manual" annotator="various" path="manual">
    </resource>
    <resource id="AMIwordsASRa1" description="ASR transcription"
         type="automatic" annotator="various" path="../auto/ASR_AS1_feb07">
    </resource>
  </resource-type>
</resources>
```

The resource file consists of a set of `resource-type` elements inside a containing `resources` element. Each `resource-type` groups together a set of resources that instantiate the same coding in the metadata file. Note that the word *coding* is used here, but a resource can also instantiate an ontology or similar corpus-level data file (see the `da-types` resource group in the example above).

`Path`s in the resource file can be absolute, but if they are relative, they are relative to the location of the resource file (which itself may be relative to the metadata file). It is important that each resource has a separate directory so that, for example, each annnotator may code the same meeting in order to check inter-annotator agreement. All files will be expected to conform to the NXT naming conventions.

Each resource can have a list of dependencies. Virtual resources are described below, but for a `resource` element, dependencies will be to particular instantiations of the codings they directly dominate and directly point to (there's no need to list more remote descendents as dependencies).

Resources can be grouped into `virtual-resource`s. These groups do not specify a path but instead exist solely to group a particular set of real resources. If a gold standard coding exists for dialogue acts, as in the example above (see the dialogue-act virtual resource called `da-gold`), it can specify by means of dependencies, how it is derived from the set of manual dialogue-act annotations. So the `da-gold` resource derives its gold-standard data for observation IS1008b from Doris; for all other meetings that Fred has annotated, his annotation then takes precedence; and for meetings that Fred has not annotated, we use Doris's annoatation. Note that regular expressions are matched using Java's `java.util.regex`. Note that virtual resources will always have dependencies that point to resources in the same resource-type group.

At most one resource inside each resource-type can be marked as the default which means it will be the one loaded in the absence of any explicit overriding instruction. See the `da-gold` resource in the example above.

Finally, the `notloadedwith` attribute on a `resource` element may specify exactly one other resource which should never be loaded along with this one. This should not be required particularly often, but may be useful if you have clashing IDs and need to avoid co-loads, or if it would just be confusing for users. If NXT is asked to load incompatible resources, it will print a warning message.

4.5.2 Behaviour

Various rules are adhered to when loading files into NXT using a resource file:

- two versions of the same coding are never co-loaded unless there's an explicit request. To request multiple loads, use the `forceResourceLoad(resource_id)` or `forceAnnotatorCoding(annotator, coding_id)` methods of `NOMWriteCorpus`.

- Whenever a particular resource is loaded, its dependents automatically become the preferred resource for their respective codings. This is also true whenever a resource is the default in the resource file, or preferred / forced using an API call.

- If elements in a high level coding are requested and some of its descendents are already loaded (or preferred), the resources that depend (however indirectly) on the already-loaded resources will be preferred.

- Competing resources will be selected by applying these rules in order: choose manually forced resources from API calls; choose manually preferred resources, or those dependent on forced or preferred resources, or dependent on already-loaded resources; choose resources with their `default` attribute in the resource file set to `="true"`; choose virtual resources over ordinary resources.

- If there are multiple choices for which resource to load for a coding, and no single resource is derived using the algorithm above, the user will be asked to select a resource. Users can avoid these questions by selecting a coherent set of defaults in the resource file, or via API calls.

The only way to override this behaviour is to set the Java property `NXT_RESOURCES_ALWAYS_ASK` to `true`. This forces a user decision for every coding to be loaded (unless only one resource instantiates the coding).

4.5.3 Validation

Resource files are validated when they are loaded. To avoid confusion, it is advised but not enforced that paths to data locations should be specified only in the resource file if it is present, rather than allowing a mix of metadata and resource file paths. It is also expected that if a resource file is present it should be complete in terms of its coverage of metadata codings. Warnings will be also be issued when `coding` attributes don't match valid elements in the metadata file.

Resource files can affect data validation as they allow a particular resource to refer to multiple resources that instantiate the same coding. For example, an alignment element can refer to a `word` in the reference transcription as well as a `word` from the ASR version of the transcript. Before resource files were introduced such an alignment would require the ASR and reference word elements to have different types. The disadvantage of using resources in this way is that such data cannot be validated without extra effort.

4.6 Data validation

In any framework, it is a good idea to test data sets to ensure that they are valid against the constraints of that framework's data representation. NXT's metadata file format and use of stand-off annotation to split a data set into a large number of files makes this somewhat trickier than for other XML data. For this reason, NXT comes with a method for validating data sets against their metadata files. NXT validation tests not just the validity of individual XML files from a data set, but also the validity of links between files. We explain how to validate NXT-format data and then describe a utility that helps with the process. In addition to the "off-line" validation described in the section, the methods for loading and editing data in the NITE Object Model perform some validation as they go along; see 'The NITE Object Model' p.11.

4.6.1 Limitations in the validation process

NXT's off-line validation relies on schema validation using a schema that unduly restricts the data format; the metadata format allows a number of options to be configured about the representation for a particular corpus, but the validation can only handle the default values. At present, the undue restrictions are as follows:

- All stream elements (XML file document roots) must be named `nite:root`.

- All ID, Start and End time attributes must use the NITE default names: `nite:id`, `nite:start` and `nite:end`.

- All children and pointers must use XLink / XPointer style links, and ranges cannot be used. These are both handled automatically when `PrepareSchemaValidation` is used.

- Annotations referring to elements of the same type from different resources cannot be validated without extra effort. This kind of annotation can be used for alignment of reference to automatic transcription etc.

In addition, even though the metadata file specifies which elements can occur at the first level under the stream element by constraining the tags that can occur in the file's top layer, the validation process currently fails to check this information, and will allow any elements at this level as long as those elements are valid and allowed somewhere in the data file.

If your data does not use XLink /XPointer style links but it will load into NXT already, you can use NXT itself to change the link style. The program `PrepareSchemaValidation.java` will load the corpus and save it in the correct link style, as well as carrying out a few more of the required validation steps. It is in the `samples` directory of the NXT distribution.

4.6.2 Preliminaries - setting up for schema validation

Before you begin, you need to be set up to perform schema validation using one of the many different methods available.

4.6.2.1 Using Xalan

Schema validation comes as a command line utility from *Apache* in the samples for *Xalan*. Although NXT redistributes *Xalan*, the redistribution does not include the samples, which are in `xalansamples.jar`.

To run schema validation, make sure `xalansamples.jar` and `xalan.jar` are both on your classpath, and run

```
java Validate metadata-file
```

This works for either schema or DTD validation.

4.6.2.2 Using XSV

XSV from is another common choice of validator. To run it, use

```
xsv filename schema
```

Alternatively, you can add the following attributes to the root element of the file you wish to validate

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="schema"
```

to name the schema in the document itself.

4.6.3 The validation process

There are a number of steps in validating an NXT corpus.

4.6.3.1 Validating the metadata file

To validate the metadata file, run it through any ordinary XML validation process, such as the schema validation described in You can choose whether to validate the metadata file against a DTD or a schema, whichever you find more convenient. The correct DTD and schema can be found in the NXT distribution in `lib/dtd/meta-standoff.dtd` and `lib/schema/meta-standoff.xsd`, respectively.

4.6.3.2 Generating a schema from an NXT metadata file

The stylesheet for generating a schema from an NXT metadata file is in the NXT distribution in the `lib` directory. It is called `generate-schema.xsl`.

You can use any stylesheet processor you wish to generate the schema. Since xalan is redistributed with NXT, assuming you have set `$NXT` to be the root directory of your NXT distribution, one possible call is

```
java -cp "$NXT/lib/xalan.jar" org.apache.xalan.xslt.Process -in metadata
     -xsl generate-schema.xsl -out extension.xsd
```

This creates a schema file called `extension.xsd` that relies on two static schema files that are in the `lib` directory of the NXT distribution: `typelib.xsd` and `xlink.xsd`. Put these three files in the same directory.

4.6.3.3 Validating the individual XML files in the corpus

The next step is to validate each of the individual XML files in the corpus.

4.6.3.4 Validating the out-of-file links

To validate an NXT corpus you must check not just the individual XML data files, but also the child and pointer relationships represented by out-of-file links. We do this by transforming each XML data file so that instead of containing an XML element that represents a link to an out-of-file child or pointer target, the file contains the target element itself, and validating the resulting files. The schema you have generated is set up so that it can validate either the actual XML files in the corpus or the files that result from this transformation. The stylesheet `knit.xsl` from the `lib` directory of the NXT distribution does the correct transformation; for more information about knitting, see 'Knitting and Unknitting NXT Data Files' p.76.

4.7 Data Set Design

Because NXT does not itself commit the user to any particular data representation as long as it is expressed in terms of the NITE Object Model, it requires users to design their data sets by expressing formally what the annotations contain and how the annotation relate to each other. For complex data sets with many different kinds of annotations, there can be many different possible arrangements, and it can be difficult to choose among them, particularly for novice users. In this section, we comment on the design choices NXT presents.

4.7.1 Children versus Pointers

Often in NXT data sets, there is a choice between whether to represent a relationship between two nodes as that of parent and child or using a pointer.

In general, prefer parent-child relationships except where that violates the constraints of the NITE Object Model by introducing cycles into the graph structure. If necessary, turn off time percolation within a tree to have parent-child relationships make sense. Trees are both faster to process and easier to access within the query language; the hat operator (`^`) calculates ancestorhood at any distance, but the pointer operator (`>`) is for one level at a time. The one exception is cases where using a pointer seems to fit the semantics of the relationship more naturally and where querying will not require tracing through arbitrary numbers of nodes.

4.7.2 Possible Tag Set Representations

A tag set is a list of types for some annotation. For instance, for a dialogue act annotation, two typical tags in the set would be `question` and `statement`. Dialogue acts would typically be represented as a single layer of nodes that draw their children from a transcription layer, but this still leaves the question of how to represent their tag. There are three possibilities:

- defining a different code in the layer for each tag and using this code, or "node type";
- having one code but using an attribute defined using an enumerated attribute value list containing the tags;
- or having one code that points into a separate ontology or other kind of corpus resource that contains nodes that themselves represent the tags.

The first option might seem the most natural, but it is cumbersome in the query language because in searches over more than one tag, each possible tag must be given in a disjunction for matching the node type (e.g., `($d question | statement)`). A further advantage of the other two representations is that NXT's query language provides regular expression matching over attribute values; that is, if the tag set includes two kinds of questions, `yn-question` and `wh-question` and the tags are given as attribute values on some node, then expressions such as `($d@tag ~ /.*question/)` will match them both.

Of the other two representations, the simpler choice of using an enumerated attribute value makes queries more succinct (e.g., `($s dialogueact):($s@type=="question")` rather than `($s dialogueact)($t da-type):($s>"type"$t):($t@name=="question")`). However, historically the configurable annotation tools could only be set up using ontologies, forcing their use for data created using these tools. The discourse segmentation and discourse entity tools were changed in NXT version 1.4.0 to allow either method, and the signal labeller is expected to follow suit shortly. For new corpora, using an ontology retains two advantages. The first is the ability to swap in different versions without having to change the data itself, for instance, to rename the tags or change the structure of the ontology. The second is for tag sets where the designers wish to encode more information than can be packed clearly into one string for use with regular expression matching --- the nodes in an ontology can contain as many attributes as are required to describe a tag or pointers to and from other data structures, and can themselves form a tree structure instead of a flat list. The latter is useful for testing tags by "supertype"; for instance, if all initiating dialogue act tags are grouped together under one node in the ontology, whether or not an act is an initiation can be tested e.g. by `($d dialogueact)($t da-type)($s da-type):($d >"type" $t) && ($s ^ $t) && ($s@name=="initiation")` .

4.7.3 Orthography as Textual Content versus as an String Attribute

Ordinarily, orthography is represented using the textual content of a node. Alternatively, orthography can be represented in a user-defined string attribute.

There are several advantages to representing orthography as textual content. The first is for processing outside NXT --- since textual content is a common representation, there are a number of XML tools that expect to find orthography there already. The second is that the configurable tools also expect this, and so they will work for data represented this way without the need to modify how they render transcription by writing delegate methods specifically for the corpus. However, using the textual content is not always practicable. This is because in the NITE Object Model, nodes can have either textual content or a set of children, but not both. Although different, rival orthographies for the same signal can be accommodated easily using different, rival transcription layers, complex orthographic encodings which require orthography at different levels in the same tree cannot. As an example, consider cases where a word was said in a reduced form (e.g., "gonna"), but it is felt necessary to break this down further into its component words ("going" and "to"). If this is to be represented as one element decomposed into two, the top one cannot have textual content, and therefore orthography must be represented using a string attribute.

The reason why NITE Object Model nodes cannot have both textual content and children is to make it clear how to traverse trees within it. If a node had both, we would not know whether the textual content should come before the children, after them, or somewhere in the middle. The reason it was included was because it was considered useful to treat the orthography as special within the data representation. One might, for instance, consider the textual content of a node to be a concatenation of the textual content of its children in order using some appropriate separator. We are currently discussing whether the NITE Object Model ought to perform this operation in future versions, or whether we ought to deprecate the use of textual content altogether in favour of string attributes that concatenate except where overridden at a higher level in the tree.

4.7.4 Use of Namespaces

Ordinarily, we would recommend the use of namespaces throughout a data set, particularly where a corpus is expected to attract many different annotations from different contributors. However, NXT's query language processor has historically contained a bug which means that it is unable to parse types and attribute names that contain namespaces. This is not a problem for default use of the `nite` namespace because the query language exposes the namespaced attributes using functions (e.g., `ID($x)` for `$x@nite:id`, but it is for user-defined uses of namespacing. We expect this problem to be resolved at some point in the future (after version 1.3.7) but not as a high priority.

4.7.5 Division into Files for Storage

The set of intersecting trees in a set of annotations constrains how the data will be stored, since each tree must be stored in a different file (or set of files, in the case of agent annotations). However, it does not fully specify the division into files, since NXT has both in-file and out-of-file representations for parent-child relationships. Data set designers can choose to store an entire tree in one file (or set of files) or to split the tree into several files, specifying which layers go in each.

Dividing a tree into several files can have several benefits. The first is that since NXT lazy loads one file at a time as it needs it, it can mean less data gets loaded over all, making the processing quicker and less memory-intensive. The second is that each file is simpler, making it easier to process as plain old XML, especially the base layers, since these have no out-of-file child links. The third is that during corpus creation, assuming some basic ground rules about building layers up from the bottom, each file can be edited independently. However, dividing a tree into several files also has some drawbacks --- the data takes more space on disk it parent-child relationships are represented as out-of-file links rather than using the structure of the XML file, and there is a processing overhead involved in the operation of loading each file.

A good general rule of thumb is to consider whether use of one layer means that the other layer will also be needed. Unless most users will always want the two layers together, both inside and outside NXT, then store them in different files.

4.7.6 Skipping Layers

Sometimes, the layering model imposed by NXT metadata is more rigid than a corpus designer would like. It can be useful to allow some nodes in a layer to draw children not from the declared next layer, but from the layer directly below it, skipping a layer completely. This can be the case when the middle layer contains some phenomenon that covers only some of the lowest level nodes. For instance, suppose one intends a corpus with dialogue act annotation and some kind of referring expression markup, both over the top of words. Referring expressions are within acts, so it would be possible to have them between acts and words in one tree, if the layer structure allowed acts to mix words and referring expressions as children.

Some NXT corpora have simply violated the layer model in this way, and at present, for the most part it still works; however, because NXT by default uses "lazy loading" to avoid loading data files when it knows they are not needed, users of these corpora must turn lazy loading off - a major cost for memory and processing time that is untenable for larger data sets - or else consider for each data use whether lazy loading is safe in that instance. In addition, relying on NXT's current behaviour in this regard may be risky as the software develops. There are several other options.

The first is to separate out the middle and top layers into separate trees, making them both draw children from the bottom layer, but independently. (This has the side effect of serializing them into different files.) Then nodes in the top layer are no longer parents of nodes in the middle layer, but they can be related to each other via the nodes from the bottom layer that they both contain.

The second is to wrap all of the unannotated node spans from the bottom layer with a new node type in the middle layer, and have the top layer draw children from the bottom layer indirectly via these new nodes. The additional nodes would take some storage space and memory. The one thing that might trip corpus users up about this arrangement is if they use distance-limited ancestorhood in queries, since they would be likely to forget the nodes are there.

The third is to declare the top and middle layers together as one recursive layer, which means as a side effect that they must be serialized to the same file. This method behaves entirely as desired, but prevents NXT from validating the data correctly, since NXT allows any node type from a recursive layer to contain any other node type from the layer as a child.

4.8 Data Builds

This section explains how to use the *`Build`* utility of NXT to produce packaged-up versions of your corpus in a way that other users can unpack and use. You can specify which annotations and observations are included, and an appropriate metadata file will be produced to go along with the data you select.

To specify a data build for an NXT corpus you need to follow these steps:

- Write a build specification file conforming to the simple DTD file in your distribution (see `lib/dtd/build.dtd`)

- With NXT and its various lib jar files on your `CLASSPATH`, run

  ```
  java net.sourceforge.nite.util.Build yourfile
  ```

  This creates an `ant` file to actually do the build. You will also be told what command to issue...

- Run `ant -f your_antfile` to produce a data file (you'll be told what the data file is called).

4.8.1 Examples and explanation of format

First, here's a valid build specification. The resulting ant file extracts a set of words and abstractive summaries from all observations matching the regular expression `Bed00*`. We take the standard words from the corpus, but for the abstractive summary, we decide we want to use the files from the annotator `sashby`.

```
<build metadata="Data/ICSI/NXT-format/Main/ICSI-metadata.xml"
      description="ICSI extract" name="jonICSI"
      type="gold" corpus_resources="off" ontologies="on" object_sets="off">
```

```
   <extras dir="/home/jonathan/configuration" includes="*.html" dir="config"/>
   <coding-file name="words"/>
   <coding-file name="abssumm" annotator="sashby"/>
   <observation name="Bed00*"/>
</build>
```

There are two types of build: `gold` and `multi-coder`. The first of these is for builds where we want only one set of XML files for each coding and for that set to be treated as the *gold-standard*. Note that in the example above we actually chose a specific annotator's abstractive summary: in the resultant build, that annotator's abstractive summaries will replace any existing 'gold-standard' abstractive summaries. Multi-coder builds result in corpora which may have *gold-standard* codings, but can also have all the different annotators' data included. There's an example below.

Output of any of the corpus-wide information can be toggled on or off, using attributes of the same name: `corpus_resources`; `ontologies`; `object_sets`. They are all output by default. Arbitrary *extras* can also be included in the build. These are essentially specs that are passed straight through to ant.

```
<build metadata="Data/ICSI/NXT-format/Main/ICSI-metadata.xml"
          description="ICSI multi-coder extract" name="jonICSImulti"
          type="multi_coder">
   <coding-file name="words"/>
   <coding-file name="abssumm"/>
   <coding-file name="extsumm" resource="autoextract1"/>
   <observation name="Bmr*"/>
</build>
```

This requests the words and abstractive summary codings as before, but for a different set of observations. This time we'll end up with the gold-standard words (since that's all there is in our corpus), but the entire tree of abstractive summaries including any 'gold-standard' files plus subdirectories of all the annotators' abstractive summaries. Note that if an annotator is named in multi-coder mode, only that annotator's data is included but it is not raised to the gold-standard location. Finally (from NXT release 1.4.2; CVS date 14/09/2007), note the extra `coding-file extsumm` which has an associated `resource` attribute. This attribute will have an efect on the build if there's a resource file referred to in the metadata file and it contains a resource for extractive summaries called `="autoextract1"`. In that case, the resource file will be included in the build, and the selected resource is the one used for extractive summaries. You can have multiple instances of the same coding-file to include multiple competing resources.

One extra element allowed is a `default-annotator` element before any of the `coding-file` elements: the `name` attribute of will be the name of the annotator that is used by default (where not overriden by an `annotator` element on a `coding-file` element).

Note:

In any circumstance where a specific annotator's data has been requested, but there is none present, the 'gold-standard' data (if present) will be used instead.

---

5 The NXT Query Language (NQL)

NXT has its own query language: the NXT Query Language, or NQL,. In this section, we describe NQL, list its operators, and give example queries.

5.1 General structure of a simple query

NQL queries describe n-tuples of nodes, possibly constrained by type, and a set of conditions that expresses further constraints, for instance, on the attributes that a node contains or how two nodes relate to each other. If an n-tuple of nodes with the required types satisfies the conditions expressed in the query, it is said to be a match.

Syntactically, a query consists of two parts separated by a colon (`:`). The first part declares the variables for the query, and the second part expresses the conditions.

Example showing general query structure

`($a)($b word): $a ^ $b`	This query matched pairs (2-tuples) of nodes in which the first, bound to `$a`, can be any type and the second, bound to `$b` must be of type word, and where the first dominates the second. In this example `($a)($b word)` is the declaration part and the dominance relation `$a ^ $b` is the only condition.

Formal definition:

query := *declarations* : *match_condition*

5.1.1 Declaration part

The declaration part of the query must contain a variable declaration for every variable mentioned in the conditions. Each variable declaration is enclosed in parentheses. The components of a variable declaration are separated by whitespace. The first component is an optional quantifier; the possible quantifiers are `forall` and `exists`, which have their usual logical meanings. The second component is the name of the variable. A variable name is a `$` character followed by an arbitrary number of letters and digits, which can include underscore (_) and language-specific characters. The final component is an optional type restriction. This is either a simple string expressing the type of a node in the NOM or a disjunction of these types separated using the pipe symbol (`|`).

Example declaration parts

`($a)`	The query matches singletons (1-tuples) in which `$a` is be bound to nodes of any type for which the conditions are true.
	Tip: An empty type definition may slow down query processing drastically.

`($a word)($b sentence)`	The query matches pairs (2-tuples) in which `$a` is bound to nodes of type word and `$b` is bound to nodes of type sentence for which the conditions are true.
`($a word)($b word)`	The query matches pairs (2-tuples) in which `$a` is bound to nodes of type word and `$b` is bound to nodes of type word for which the conditions are true. **Caution:** In a pair, both variables might be bound to the same node.
`($a word | phrase | sentence)`	The query matches singletons (1-tuples) in which `$a` is bound to nodes of type word, phrase, or sentence for which the conditions are true.
`($b sentence)(forall $a word)`	The query matches singletons (1-tuples) in which `$b` is bound to any node that has type sentence for which the conditions are true for every possible way of binding `$a` to individual nodes that have type word.

Formal definition:

```
declarations     :=  declarations var_declaration
declarations     :=  var_declaration
var_declaration  :=  ( variable )
var_declaration  :=  ( variable typedeclaration )
typedeclaration  :=  types
typedeclaration  :=  type
types            :=  types | type
```

5.1.2 Condition part

The condition part is a Boolean expression over property tests, structural relations, and temporal relations. Parentheses are only needed if a lower precedence relation should be executed first. The strongest binding operator is negation (`!`). The operators are listed in the order of their precedence below:

- Negation: `not` or `!`
- Conjunction: `and` or `&` or `&&`
- Disjunction: `or` or `|` or `||`
- Implication: `->`

For convenience, there are a number of syntactic literals for each operator. The different forms for the same operator are completely synomynous. The implication operator `->` is the weakest binding operator and, again, is provided for convenience; `$a -> $b` is logically equivalent to `!a | b`.

Formal definition:

```
match_condition    :=
match_condition    := ( match_condition )
match_condition    := ! match_condition
var_declaration    := match_condition & match_condition
var_declaration    := match_condition | match_condition
var_declaration    := match_condition -> match_condition
var_declaration    := property_test
```

```
var_declaration    := structural_relation
var_declaration    := time_relation
```

Tip:

The condition part may be empty, in which case the query always evaluates to true.

5.2 Property tests

A property test either tests for the existence of some property or compares the values of two properties.

5.2.1 Simple and functional property expressions

A property may be an attribute value, a constant, or the result of a function applied to an element.

The query language contains a number of functions that take elements and return some property of the element. These functions have to do with properties that are special in NXT: timing, identification, and textual content. Function names can be given in upper or lower case, and the operators = and == are synonymous. Where the data storage format for a corpus stores these properties as XML attributes, they can also be queried using the form *variable* `@`*attribute*. The functional form is preferred because it is the same across all data sets and leaves the user in no doubt that these are the attributes holding the special properties.

The functions are as follows:

Query functions

`TEXT($w)`	Returns the text contained by the element matched by `$w`.
`ID($w)`	Returns the unique identifier of the element matched by `$w`.
`TIMED($w)`	Returns true if the element matched by `$w` has start and end times, and false otherwise.
`START($w)`	Returns the start time of the element matched by `$w`.
`END($a)`	Returns the end time of the element matched by `$w`.
`DURATION($a)`	Returns the duration of the element matched by `$w` (that is, the end time minus the start time).
`CENTER($a)`	Returns the temporal center of the element matched by `$w` (that is, the end time minus the start time, divided by 2).

Formal definition:

```
property    := " number_or_string "
property    := variable @ attribute
property    := TEXT(  variable )
property    := ID( variable )
property    := TIMED( variable )
property    := START( variable )
property    := END( variable )
property    := DURATION( variable )
property    := CENTER( variable )
```

Note:

Numbers and string values must be placed in quotes. Placing a number in quotes does not mean it will be treated as a string. Either single or double quotes can be used, but if you are passing a query as an argument at the command line, your choice must be compatible with your choice of quotes for the shell.

5.2.2 Property existence tests

Property existence tests check for the existence of some property.

Node property tests

`$w@pos`	True if and only if the element matched by `$a` has a pos attribute.
`TIMED($a)`	True if and only if the element matched by `$a` is timed, either because it has both start and end times or because it inherits them from their children.
`START($a)`	True if and only if the element matched by `$a` has a start time, either in its own right or by inheritance from its children.
`END($a)`	True if and only if the element matched by `$a` has an end time, either in its own right or by inheritance from its children.
`TEXT($a)`	True if and only if the element matched by `$a` contains text.

It is actually possible to test for the existence of any property, but the other possible existence tests are not useful; `ids`, `string`, and `numbers` always exist, and `duration` and `center` properties exist for elements that are timed.

Formal definition:

```
property_test    := variable @ attribute
property_test    := TIMED( variable )
property_test    := START( variable )
property_test    := END( variable )
property_test    := TEXT( variable )
```

5.2.3 String and number comparisons using==, !=, <=, <, >, and >=

The next set of property tests compare the equality and order of two values. Property expressions are weakly typed. The value resulting from an expression will be interpreted as a floating-point number when possible. If it cannot be converted into a number or if the value is compared to a pattern given by a regular expression, the value will be treated as a string. A number is always unequal to a string. Strings are themselves alphabetically ordered, and are case-sensitive. Strings starting with upper case letters are less than strings with upper case letters. As a result, `"2" == "2.0"` is true, while `"Two" == "two"` is false.

Equality and order tests

`($x): $x@cat=="NP"`	Matches elements with a category attribute containing the string value "NP".
`($x)($y): $x@cat==$y@cat`	Matches pairs of elements with the same cat attribute (including the pair where `$x` and `$y` are bound to the same element).
`($x)($y): $x@cat==$y@cat & $x!= $y`	Matches pairs of elements with the same `cat` attribute (excluding the pair where `$x` and `$y` are bound to the same element).

Tip:

= and == are synonymous.

Formal definition:

```
property_test    := property  ==  property
property_test    := property  !=  property
property_test    := property  <  property
property_test    := property  >  property
property_test    := property  <=  property
property_test    := property  >=  property
```

5.2.4 Regular expression comparisons

The final set of property tests compare string values against regular expressions. Regular expressions are enclosed by slashes (`/`). NXT's regular expression implementation uses Java regular expressions underneath, so it is not possible to give a definitive syntax for the patterns here, instead, see the Java 1.5 Regular Expression Documentation or equivalent documentation for your Java version.

Regular expression examples

`($a): text($a) ~ /th.*/`	Words starting with `th`. Dot (`.`) means any single character, and `*` means 0 or more repetitions of whatever it follows.
`($a): text($a) ~ /[dD](as|er)/`	The words `das` and `der`, whether capitalized or not.
`($a): text($a) ~ /.+([0-9A-Z])+.*/`	Words which contain at least one uppercase letter or number at a non-initial position. The plus (`+`) means 1 or more repetitions of whatever it follows, and the square brackets (`[]`)specify a character class.
`($a): text($a) ~ /\.*/`	A possibly empty sequence of dots, where in contrast `/.*/` matches every word (assuming it contains text). The backslash (`\`) means the dot (`.`) is interpreted literally.

Note:

Your regular expression must match the entire string, not some substring contained in it. `/x/` in NQL notation means `/^x$/` in the Perl notation.

Formal definition:

```
property_test    := property ~ / pattern /
property_test    := property !~ / pattern /
```

5.3 Comments

Comments are allowed in the form of line comments and block comments. *Line comments* start with the symbol `//` and include the remainder of the current line. *Block comments* begin with `/*` and end with `*/`, and may extend over multiple lines.

Comment examples

`($a) // all elements`	line comment: *all elements*
`($a)($b word): /*$a@pos="NN" & */ $a ^ $b`	only `($a)($b word): $a ^ $b` will be processed

5.4 Structural relations

5.4.1 Identity

The simplest structural relation asserts the identity or non-identity of two elements. Since the default evaluation strategy allows different variables to be bound to the same element, the `!=` operator is sometimes necessary to exclude unwanted results. The `==` operator is less useful and was mainly added for the sake of symmetry.

```
structural_relation    :=    variable == variable
structural_relation    :=    variable =! variable
```

5.4.2 Dominance

The basic structural relation is the dominance relation ^. To describe that an element *a* dominates an element *b* the dominance operator ^ is be used. In other words *a* is an ancestor of *b*.

```
structural_relation    :=   variable ^ variable
structural_relation    :=   variable ^ distance variable
```

Note:

The expression *a*^*a* is always true! Use the non-identity operator to exclude these special case.

5.4.3 Precedence

Two elements are in a precedence relation if they have a common ancestor element, which can be a normal element or the root element of a layer. An element `$x` precedes another element `$y` if some ancestor of `$x` (or `$x` itself) is a preceding sibling of some ancesor of `$y` (or `$y` itself).

```
structural_relation    :=    variable <> variable
```

Note:

The expression *a*<>*a* is always false!

Some examples:

Structural relations examples

`($a)($b): $a ^ $b & $a != $b`	all combinations of two different elements in a dominance relation
`($s syntax)($w word): $s ^1 $w`	all combinations of syntax and word elements, where the syntax element dominates directly the word element
`($a)($b): $a ^0 $b`	equal to `$a == $b`
`($a)($b): $a ^-2 $b`	equal to `$b ^2 $a`
`($a word)($b word): $a <> $b`	two words, `$a` precedes `$b`

5.5 Temporal relations

Temporal relations examples

Op., short	Operator, lexical	Definition
`%`	`overlaps.left`	`(start($a) <= start($b)) and (end($a) > start($b)) and (end($a) <= end($b))`
`[[`	`left.aligned.with`	`start($a) == start($b)`
`]]`	`right.aligned.with`	`end($a) == end($b)`
`@`	`includes inclusion`	`(start($a) <= start($b)) and (end($a) >= end($b))`
`[]`	`same.extent.as`	`(start($a) == start($b)) and (end($a) == end($b))`

Op., short	Operator, lexical	Definition
#	overlaps.with	(end($a) > start($b)) and (end($b) > start($a))
][	contact.with	end($a) == start($b)
<<	precedes	end($a) <= start($b)
	starts.earlier.than	start($a) <= start($b)
	starts.later.than	start($a) >= start($b)
	ends.earlier.than	end($a) <= end($b)
	ends.later.than	end($a) >= end($b)

5.6 Quantifier

To express complex structural relations in some cases auxiliary elements are required, which should not be part of the query result. Sometimes it is sufficient that one such element satisfies the match condition, sometimes all auxiliary elements must match.

The mathematical solution to this problem are the existential and universal quantifiers. In NQL variables can be existential quantified or universal quantified. In both cases elments which are bound to a quantified variable are not part of the result.

The formal definition of 'Condition part' p.35 is now extended with quantifiers:

```
var_declaration     :=  ( exists variable )
var_declaration     :=  ( exists variable typedefinition )
var_declaration     :=  ( forall variable )
var_declaration     :=  ( forall variable typedefinition )
```

In queries with quantifiers the implication operator `->` could be useful (see 'Condition part' p.35).

Some examples:

Quantifier Examples

($a)(exists $b): $a ^1 $b	elements with children
($root)(forall $null): !$null ^1 $root	root elements

5.7 Query results

The result of a query is a list of *n*-tuples of elements (or, more precisely, variable bindings) satisfying the match condition, where *n* is the number of variables declared without quantifiers (cf. 'Quantifier' p.40). The query result is returned in the form of an XML document (or, abstractely, a new tree structure adjoined to the corpus). Each query match corresponds to a *match* element, with pointers representing variable bindings and the variable name given by the pointer's role.

An example result for a query involving variables *$w* and *$p* is:

```
<matchlist size="2">
        <match n="1">
                <nite:pointer role="w" xlink:href="..."/>
                <nite:pointer role="p" xlink:href="..."/>
        </match>
        <match n="2">
                <nite:pointer role="w" xlink:href="..."/>
```

```
                <nite:pointer role="p" xlink:href="..."/>
        </match>
</matchlist>
```

Note:

The matches are not ordered. The ordering of the results of two similar but not identical queries can be very different.

5.8 Complex queries

A complex query consists of a sequence of simple queries seperated by `::` markers.

complex_query := *complex_query* :: *query*
complex_query := *query*

For a complex query, the leftmost query is evaluated first. Each query in the sequence operates on the result of the previous query. This means that for every match, the following query is evaluated with the variable bindings of the previous queries. The fixed variable bindings may be used anywhere in the ensuing queries. This evaluation strategy produces a hierarchically structured query result, where each match of the leftmost simple query includes a matchlist for the second query, etc.

In the example

```
($w word): $w@orth ~ /S.*/ :: ($p phone): $w ^ $p
```

the query result has the following structure:

```
<matchlist size="2">
        <match n="1">
                <nite:pointer role="w" xlink:href="..."/>
                <matchlist type="sub" size="2">
                        <match n="1">
                                <nite:pointer role="p" xlink:href="..."/>
                        </match>
                        <match n="2">
                                <nite:pointer role="p" xlink:href="..."/>
                        </match>
                </matchlist>
        </match>
        <match n="2">
                <nite:pointer role="w" xlink:href="..."/>
                <matchlist type="sub" size="1">
                        <match n="1">
                                <nite:pointer role="p" xlink:href="..."/>
                        </match>
                </matchlist>
        </match>
</matchlist>
```

Note:

There are no empty submatches. If for a variable binding the following single query has no matches, the variable binding will be removed from the result. So the number of matches for a complex query is less than or equal to the number of matches for the first part.

5.9 Known Problems

At Feb 05, there are a number of known problems with the current querylanguage implementation.

5.9.1 Multiple observations and timings

There is a bug when querying over multiple observations - the implementation considers times in different observations to be comparable, so that it's possible to get the result that an element in one observation is before some element in another. This is easy to get around: query on one observation at a time, or declare the reserved attribute for observation names for your corpus and add a test for the same observation as an extra query term - e.g. `($f@obs = $g@obs)`, if the attribute declared is `obs`.

5.9.2 Search GUI and forall

The search GUI (whether called stand-alone or from a search menu) can't display results if some subquery in a complex query only has query matches that are bound with `forall` - e.g. `($f foo):($f@att="val")::(forall $g bar):!($g ^ $f)`

5.9.3 Immediate Precedence

The immediate precedence operator is missing. Immediate precedence is equivalent to `($f foo)($g foo)(forall $h foo): ($f<<$g) &amp;amp; (($h=$f) || ($h=$g) || ($h<<$f) || ($g<<$h))`

but, of course, this is cumbersome and can be too slow and memory-intensive for practical purposes, depending on the data set. Some common uses of the operator are covered by the `NGramCalc` utility.Another work-around is to create one XML tree from the NXT data thatrepresents the information required and query it using XPath. Export to *LPath* and *tgrep2* would also be reasonable and are not difficult to implement. If you need to match on regular expressions of XML elements in order to add markup, (so, for instance, saying "find syntactic constituents with one determiner, followed by one or more adjectives, followed by one noun, and wrap a new tag around them"), but you can always use something like *fsgmatch* (from the LTG; new release, currently in beta, is called *lxtransduce*) and then modify the metadata to match. Remember, the data is just XML, amenable to all of the usual XML processing techniques.

5.9.4 Arithmetic

The arithmetic operators are missing.

At present, users who need them add new attributes to theirdata set and then carry on as normal. For instance, a researcher looking at how often *bar* elements start in the 10 seconds after *foo* elements end might add an `"adjusted start"` attribute to *bar* elements that take 10 secondsoff their official start times, and then use the query `($f foo)($b bar):(START($b) > END($f)) && ($b@adjustedstart < END($foo))`

This stylesheet, run on a specific individual coding in the context of the MONITOR project, is an example of how this can be done. It just copies everything, adding new attributes to feedback gaze codes. We used this general technique on the Switchboard data to get lengths for syntactic constituents, and on the Monitor data to get durations.

This method is inconvenient, particularly for the sort of exploratory study that wishes to consider several different time relationships. We don't think it is worth adding special loading routines that addtemporary attributes for adjusted start and end times, but we could include some utilities for command line searching based on adjustments passed in on the command line. For instance, `java CountWithTimeOffset -q '($t turn)($f feedback):($t # $f)' -t feedback -d 50` could mean to count overlaps after feedback elements have been displaced 50 seconds forward. We are considering whether this would be useful enough to supply.

5.9.5 Inability to handle namespacing

At present (Apr 05) the query language parser fails to handle namespacing properly, so any elements and attributes that are namespaced will be difficult to work with. For the timing and id attributes, where the default names are in the `nite:` namespace, this doesn'tmatter, since they are exposed to query via e.g. `START($x)`, but namespacing other tags and attributes would make working with them difficult until this is fixed.

5.9.6 Speed and Memory Use

NXT's query engine is slow and uses a great deal of memory. For instance, some of our more complicated syntactic querieson the Switchboard corpus take 10 seconds per dialogue, or over an hour and a half for the entire corpus.

This is partly a consequence of what it does - the query languageis solving a harder problem than languages that operate on trees and/or are limited in their use of left and right context. It istrue that the current implementation is not fully optimized, but this is not something we intend to look at in the immediate future. Our first choice strategy for addressing this

problem is to look at mapping NQL queries to XQuery for implementation, and addition of the missing operators, that way. Meanwhile, most of *NXT*'s users are not actually engaged in real-time processing, and find that if they develop queries on a few observations using a GUI, they can then afford to run the queries at the command line in batch. The more they are interested in sparse phenomena, the less suitable this strategy is. For some query-based analyses, it is also useful to consider direct implementation using the NOM API, since the programmer can optimize for the analysis being performed.

Meanwhile, an hour and a half is OK for batch mode, but some of our queries areso common that we really want easy access to the results. We can get this by indexing. Using indices rather than the more complex syntactic queries makes querying roughly ten times faster. This will be even faster if one then selects not to load the syntax at all, which is possible if one doesn't need it for other parts of the subsequent query. You can choose not to load any part of the data by commenting out the `<coding-file>` tag for it in your local copy of the metadata file, or after NXT 1.3.0, by enabling lazy loading in your applications.

It's faster to use string equality than regular expression matching in the query language, and keep in mind the regular expressions have to match the entire string they are compared against, not just a substring of it.

The very desperate can write special purpose applications to evaluate their queries, which is faster especially for queries involving quantification. For instance,one user has adapted `CountQueryResults` to run part of the query he wants, but instead of returning the results, then checks the equivalent of hisforall conditions using navigation in the NOM.

5.10 Helpful hints

We recommend refining queries using `display.bat/.sh` on a single dialogue (probably spot-checking on a couple more, since observations vary), and running actual counts using the command line utilities. Build up queries term by term - the syntax error messages aren't always very easy to understand. Missing dollar signs, quotation marks, and parentheses are the worst culprits. Get around the bookmark problems and the lack of parenthesis and quote matching in the searchGUI by typing the query into something else that's handier (such as *emacs*) and pasting what you've written into the query window. You canand should include comments in queries if they are at all complicated. Queries have to be expressed on one line to run them at the command line, but you shouldn't try to author them this way - instead, postprocess a query developed in this more verbose style by taking out

Analysis of query results can be expedited by thinking carefully aboutthe battery of tools that are available: `knit`, LT-XML, stylesheets, xmlperl, shell script loops, and so on. One interesting possibility is importing the query results into the data set, which would be a fancier, hierarchically structured form of indexing. At May 2004, the metadata `<coding-file>` declaration required to do this would be a little different for every query result, but we intend minor syntactic changes in both the query result XML and what `knit` produces to make this declaration static.

5.11 Related documentation

The main documentation is the query language reference manual . Virtually the same information can be found on the helpmenu of the search window (if you don't find it there, it's an installation problem). An older document with more contextual information can be found here.

At September 2006, we plan a revised version of the manual. The current version fails to give details about the operator for finding out whether two elements are linked via a pointer with a role. `($a <"foo" $b)` is true if `$a` points to `$b` using the *"foo"* role; the role name can be omitted, but if it is specified it can only be given as a textual string, not as a regular expression. The current version also fails to make clear that the regular expression examples given are only a subset of the possibilities. The exact regular expression syntax depends on your version of Java, since it is implemented using the java.util.regex package. Java 1.5 regular expression documentation can be found here .

Here are some worked examples for the Switchboard data sample and the Monitor data sample.

Computer scientists and people familiar with first order predicate calculus have tended to be happy with the reference manual plus the examples; other people need more (so, for instance, don't know what implication is or what `forall` is likely to mean) and we're still thinking about what we might be able to provide for them.

At Nov 2004, there are a few things described in the query documentation that haven't been implemented yet (and aren't on the workplan for immediate development). This includes arithmetic operators and temporal fuzziness. We thought this included versions of `^` and `<>` limited by distance, but users report that these (or some of these?) work. Also, some versions of the query documentation show `;` instead of `:` as the separator between bindings and match conditions. The only major bug we've run into (at Nov 2004) is that temporal operators will perform comparisons across observations, even though time in different observations is meant to be independent. After NXT-1.2.6, 05 May 04, one can in the metadata

declare a reserved attribute to use for the observation name that will be added automatically for every element, providing a work-around.

There's a nifty visual demo that runs on a toy corpus and might be useful for deciding whether this stuff is useful in the first place.

6 Analysis

The fundamental tool for analysis in NXT is the NXT Query Language used through the command line tools. Query development can very usefully be done using the GUI tools, but corpus-wide analysis will normally require command line tools. Some helper tools exist for special cases like the study of reliability.

6.1 Command line tools for data analysis

This section describes the various command line utilities that are useful for searching a corpus using NXT's query language. Command line examples below are given in the syntax for `bash`. It is possible to run NXT command line utilities from the DOS command line without installing anything further on Windows, but many users will find it easier to install *cygwin*, which comes with a bash that runs under Windows. The command line tools can be found in the XXXX directory of the NXT source, and are useful code examples.

6.1.1 Preliminaries

Before using any of the utilities, you need to set your classpath and perhaps consider a few things about your local environment.

6.1.1.1 Setting the classpath

The command line utilities require the classpath environment variable to be set up so that the shell can find the software. Assuming $NXT is set to the top level directory in which the software is installed, this can be done as follows (remove the newlines):

```
if [ $OSTYPE = 'cygwin' ]; the
n         export CLASSPATH=".;$NXT/lib;$NXT/lib/nxt.jar;$NXT/lib/jdom.jar;
            $NXT/lib/xalan.jar;$NXT/lib/xercesImpl.jar;$NXT/lib/xml-apis.jar;
            $NXT/lib/jmanual.jar;$NXT/lib/jh.jar;$NXT/lib/helpset.jar;
            $NXT/lib/poi.jar"
else
         export CLASSPATH=".:$NXT/lib:$NXT/lib/nxt.jar:$NXT/lib/jdom.jar:
            $NXT/lib/xalan.jar:$NXT/lib/xercesImpl.jar:$NXT/lib/xml-apis.jar:
            $NXT/lib/jmanual.jar:$NXT/lib/jh.jar:$NXT/lib/helpset.jar:
            $NXT/lib/poi.jar"
fi
```

This is not the full classpath that is needed for running NXT GUIs, but contains all of the methods used by the command line tools.

It is possible instead to specify the classpath on each individual call to java using the -cp argument.

6.1.1.2 Shell interactions

You'll need to be careful to use single quotes at shell level and double quotes within queries - although we've found one shell environment that requires the quotes the other way around. Getting the quoting to work correctly in a shell script is difficult even for long-time Unix users. There is an example shell script that shows complex use of quoting in the sample directory of the NXT distribution called "quoting-example.sh".

Don't forget that you can use redirection to divert warning and log messages:

```
java CountQueryResults -corpus swbd-metadata.xml -query '($n nt):' 2> logfile
```

Diverting to /dev/null gets rid of them without the need to save to a file.

6.1.1.3 Memory usage

It is possible to increase the amount of memory available to java for processing, and depending on the machine set up, this may speed things up. This can be done by using flags to java, e.g.

```
java -Xincgc -Xms127m -Xmx512m -Xfuture CountQueryResults ...
```

but also as an edit to the java calls in any of the existing scripts. This is what they mean:

Java Arguments Controlling Memory Use

`-Xincgc`

use incremental garbage collection to get back unused memory

`-Xms`*size*

initial memory heap size

`-Xmx`*size*

maximum memory heap size

The best choice of values will depend on your local environment.

6.1.2 Common Arguments

Where possible, the command line tools use the same argument structure. The common arguments are as follows.

Common Arguments for Command Line Tools

`-corpus` *corpus*

the path and filename specifying the location of the metadata file

`-observation` *obs*

the name of an observation. If this argument is not given, then the tools process all of the observations in the corpus

`-query` *query*

a query expressed in NXT's query language

`-allatonce`

an instruction to load all of the observations for a corpus at the same time. This can require a great deal of memory and slow down processing, but is necessary if queries draw context from outside single observations.

6.1.3 SaveQueryResults

`java SaveQueryResults-c` *corpus*`-q` *query*`-o` *observation*`-allatonce-f` *outputfilename*`-d` *directoryname*

`SaveQueryResults` saves the results of a query as an XML document whose structure corresponds to the one displayed in the search GUI and described in 'Query results' p.40. Saved query results can be knit with the corpus to useful effect (see 'Knitting and Unknitting NXT Data Files' p.76) as well as subjected to external XML-based processing.

If no output filename is indicated, the output goes to System.out. (Note that this isn't very sensible to do unless running `-allatonce`, because the output will just concatenate separate XML documents.) In this case, everything else that could potentially be on System.out is redirected to `System.err`.

If *outputfilename* is given, output is stored in the directory *directoryname*. If running `-allatonce` or if an *observation* is specified, the output ends up in the file *outputfilename*. Otherwise, it is stored is a set of files found by prefixing *outputfilename* by the name of the observation and a full stop (`.`).

Caution:

Under cygwin, `-d` takes Windows-style directory naming; e.g., `-d "C:"` not `-d "/cygdrive/c"`. Using the latter will create the unexpected locatio n`C:/cygdrive/c`.

In distributions before 05 May 2004 (1.2.6 or earlier), the default was -allatonce, and the flag -independent was used to indicate that one observation should be processed at a time.

6.1.4 CountQueryResults

`java CountQueryResults-c corpus-q query-o observation-allatonce`

CountQueryResults counts query results for an entire corpus, showing the number of matches but not the result tree. In the case of complex queries, the counts reflect the number of top level matches (i.e., matches to the first query that survive the filtering performed by the subsequent queries - matches to a subquery drop out if there are no matches for the next query). Combine CountQueryResults with command line scripting, for instance, to fill in possible attribute values from a nenumerated list.

When running `-allatonce` or on a named *observation*, the result is a bare count; otherwise, it is a table containing one line per observation, with observation name, whitespace, and then the count.

In versions before NXT-1.2.6, CountQueryResults runs -allatonce and a separate utility, CountOneByOne, handles the independent case.

6.1.5 MatchInContext

`java MatchInContext-c corpus-q query-o observation-allatonce-context contextquery-textatt textattribute`

MatchInContext evaluates a query and prints any orthography corresponding to matches of the first variable in it, sending the results to standard output. It was developed for a set of users familiar with tgrep. *contextquery* is a noptional additional query expressing surrounding context to be show nfor matches. If it is present, for each main query match, the context query will be evaluated, with the additional proviso that the match for the first variable of the main query must dominate (be an ancestor of) the match for the first variable of the context query. If any such match for the context query is found, then the orthography of the for the first variable of the first match found will be shown, and the orthography relating to the main query will be given completely in upper case. Where the context query results in more than one match, a comment is printed to this effect. The context query must not share variable names with the main query.

By default, the utility looks for orthography in the textual content of a node. If *textattribute* is given, the nit uses the value of this attribute for the matched node instead. This is useful for corpora where orthography is stored in attributes and for getting other kinds of information, such as part-of-speech tags.

Since not all nodes contain orthography, MatchInContext ca nproduce matches with no text or with context but no main text. There is no clean way of knowing where to insert line breaks, speaker attributions, etc. in a general utility such as this one; for better displays write a tailored tool.

In versions before NXT-1.2.6, MatchInContext means -allatonce and a separate utility, MatchInContextOneByOne, handles the independent case.

6.1.6 NGramCalc: Calculating N-Gram Sequences

`java NGramCalc-c corpus-q query-o observation-tag tagname-att attname-role rolename-n n`

6.1.6.1 Background

An n-gram is a sequence of n states in a row drawn from an enumerated list of types. For instance, consider Parker's floor state model (Journal of Personality and Social Psychology 1988). It marks spoke nturns in a group discussion according to their participation i npairwise conversations. The floor states are newfloor (first to establish a new pairwise conversation), floor (in a pairwise conversation), broken (breaks a pairwise conversation), regai n(re-establishes a pairwise conversation after a broken), and nonfloor (not in a pairwise conversation). The possible tri-grams of floor states are newfloor/floor/broken, newfloor/floor/floor, regain/broken/ nonfloor, and so on. We usually think of n-grams as including all ways of choosing a sequence of n types, but in some models, not all of them are possible; for instance, in Parker's model, the bi-gram newfloor/newfloor can't happen. N-grams are frequently used i nengineering-oriented disciplines as background information for statistical modelling, but they are sometimes used in linguistics and psychology as well. Computationalists can easily calculate n-grams by extracting data from NXT into the format for another tool, but sometimes this is inconvenient or the user who requires the n-grams may not have the correct skills to do it.

6.1.6.2 Operation

NGramCalc calculates n-grams from NXT format data and prints on standard output a table reflecting the frequencies of the resulting n-grams for the given *n*. The default value for *n* is 1 (i.e., raw frequencies). NGramCalc uses as the set of possible states the possible values of *attribute* for the node type *tag*; the attribute must be declared in the corpus metadata as enumerated. NGramCalc then determines a sequence of nodes about which to report by finding matches to the first variable of the given *query* and placing them in order of start time. If *role* is given, it then substitutes for these nodes the nodes found by tracing the first pointer found that goes from the sequenced nodes with the given role. (This is useful if

the data has been annotated using values stored in an external ontology or corpus resource.) At this point, the sequence is assumed to contain nodes that contain the named attribute, and the value of this attribute is used as the node's state.

Tag is required, but *query* is itself optional; by default, it is the query matching all nodes of the type named i n*tag*. Generally, the query's first variable will be of the node type specified in *tag*, and canonically, the query will simply filter out some nodes from the sequence. However, as long as a state can be calculated for each node in the sequence using the attribute specified, the utility will work. There is no `-allatonce` option; if no *observation* is specified, only one set of numbers is reported but the utility loads only one observation at a time when calculating them.

6.1.6.3 Examples

```
java NGramCalc -c METADATA -t turn -a fs -n 3
```

will calculate trigrams of fs attributes of turns and output a tab-delimited table like

```
500         newfloor         floor         broke
n0         newfloor         newfloor          newfloor
```

Suppose that the way that the data is set up includes an additional attribute value that we wish to skip over when calculating the tri-grams, called "continued".

```
java NGramCalc -c METADATA -t turn -a fs -n 3 -q '($t turn):($t@fs != "continued")'
```

will do this. Entries for "continued" will still occur in the output table because it is a declared value, but will have zero in the entries.

```
java NGramCalc -c METADATA -t gesture-type -a name -n 3 -q '($g gest):'
     -r gest-target
```

will produce trigrams where the states are found by tracing the gest-target role from gest elements, which finds gesture-type elements (canonically, part of some corpus resource), and further looking at the values of their name attributes. Note that in this case, the tag type given in `-t` is what results from tracing the role from the query results, not the type returned in the query.

6.1.7 FunctionQuery: Time ordered, tab-delimited output, with aggregate functions

`java FunctionQuery-c` *`corpus`*`-q` *`query`*`-o` *`observation`*`-att` *`attribute_or_aggregate`*

FunctionQuery is a utility for outputting tab-delimited data. It takes all elements resulting from the result of a query, as long as they are timed, and put them in order of start time. Then it outputs one line per element containing the values of the named attributes or aggregates with a tab character between each one.

The value of `-atts` must be a space-separated list of attribute and aggregate specifiers. If an attribute or aggregate does not exist for some matched elements, a blank tab-stop will be output for the corresponding field.

6.1.7.1 Attribute Specifiers

Attribute values can be specified using the form *var@attributename* (e.g., `$v@label`, where label is the name of the attribute). If the variable specifier (e.g., `$v`) is omitted, the attribute belonging to the first variable in the query (the "primary variable") is returned. If the attribute specifier (e.g.. `label`) is omitted, the ntextual content for the node will be shown. Nodes may have either direct textual content or children; in the case of children, the textual content shown will be the concatenated textual content of its descendants separated by spaces. For backwards compability with a norder utility called SortedOutput, instead of specifying it in the list of attributes, `-text` can be used to place this textual content in the last field, although this is not recommended.

6.1.7.2 Aggregate Specifiers

Aggregate functions are identified by a leading '`@`' character. The first argument to an aggregate function is always a query to be evaluated in the context of the current result using the variable bindings from the main query. For instance, if `$m` has bee nbound in the main query to nodes of type `move`, the context query `($w w):($m ^ $w)` will find all `w` nodes descended from the move corresponding to the current return value, and the context query `($g gest):($m # $g)`, all `gest` nodes that temporally overlap with it. The list of returned results for the context query are then used in the aggregation.

For the following functions, optional arguments are denoted by an equals sign followed by the default value of that argument. There are currently four aggregate functions included in FunctionQuery.

Aggregate Functions

`@count(`*conquery*`)`

returns the number of results from evaluating *conquery*

`@sum(`*conquery*`, `*attr*`)`

returns the sum of the values of *attr* for all results of *conquery*. *attr* should be numerical attribute.

`@extract(`*conquery*`, `*attr*`, `*n*`=0, `*last*`=n+1)`

returns the *attr* attribute of the *n*th result of *conquery* evaluated in the context of query. If *n* is less than 0, extract returns the *attr* attribute of the *n*th last result. If *last* is provided, the *attr* value of all results whose index is at least *n* and less tha n *last* is returned. If *last* is less than 0, it will count back from the final result. If *last* equals zero, all items between *n* and the end of the result list will be returned.

`@overlapduration(`*conquery*`)`

returns the length of time that the results of *conquery* overlap with the results of the main query. For some *conquery* results, this number may exceed the duration of the main query result. For example, the duration of speech for all participants over a period of time may exceed the duration of the time segment if there are multiple simultaneous speakers. This can be avoided, for example, by using *conquery* to restrict matches to a specific agent.

6.1.7.3 Example

```
java FunctionQuery -c corpus -o observation -q '($m move)'
         -atts type nite:start nite:end '@count(($w w):$w#$m)' '$m'
```

will output a sorted list of moves for the observation consisting of type attribute, start and end times, the count of w (words) that overlap each move, and any text included in the move, or any children.

6.1.8 Indexing

`java Index-c `*corpus*`-q `*query*`-o `*observation*`-t `*tag*`-r `*role*

Index modifies a corpus by adding new nodes that index the results of a query so that they can be found quickly. If *observation* is omitted, all observations named in the metadata file are indexed in turn. One new node is created for each query match. The new nodes have type *tag*, which defaults to "markable". If -r is omitted, the new node is made a parent of the match for the first unquantified variable of the query. If -r is included, then the new node will instead use the role names to point to the nodes in the n-tuple returned at the top level of the query, using the role names in the order given and the variables in the order used in the query until one of the two lists is exhausted. Index does not remove existing tags of the given type before operatio nso that an index can be built up gradually using several different queries.

Note that the same node can be indexed more than once, if the query returns n-tuples that involve the same node. The tool does nothing to check whether this is the case even when creating indices that are parents of existing nodes, which can lead to invalid data if you are not careful. Using roles, however, is always safe, as is using parents when the top level of the given query matches only one unquantified variable.

Note that if you want one pointer for every named variable in a simple query, or you want tree-structured indices corresponding to the results for complex queries, you can use SaveQueryResults and load the results as a coding. For cases where you could use either, the main difference is that SaveQueryResults doesn't give control over the tag name and roles.

6.1.8.1 Metadata requirements

The tool assumes that a suitable declaration for the new tag have already bee nadded into the metadata file. It is usual to put it in a new coding, and it would be a bad idea to put in a layer that anything points to, since no work is done to attach the indices to prospective parents or anything else besides what they index. If the indexing adds parents, then the type of the coding file (interaction or agent) must match the type of the coding file that contains the matches to the first variable. If an observation name is passed, it creates a index only for the one observation; if none is, it indexes each observation in the metadata file by loading one at a time (that is, there is no equivalent to `-allatonce` operation).

The canonical metadata form for an index file, assuming roles are used, is an interaction coding declared as follows:

```
<coding-file name="foo">
  <featural-layer name="baz">
      <code name="tag">
         <pointer number="1" role="role1" target="LAYER_CONTAINING_MATCHES"/>
          ...
      </code>
  </featural-layer>
</coding-file>
```

The name of the coding file determines the filenames where the indices get stored. The name of the featural-layer is unimportant but must be unique. The tags for the indices must not already be used in some other part of the corpus, including other indices.

6.1.8.2 Example of Indexing

To add indices that point to active sentences in the Switchboard data, add the following `coding-file` tag to the metadata as an interaction-coding (i.e., as a sister to the other coding file declarations).

```
<coding-file name="sentences">
    <featural-layer name="sentence-layer">
        <code name="sentenceindex">
            <pointer number="1" role="at"/>
        </code>
    </featural-layer>
</coding-file>
```

This specifies that the indices for sw2005 (for example) should go in sw2005.sentences.xml. Then, for example,

```
java Index -c swbd-metadata.xml -t active -q '($sent nt):($sent@cat=="S")'
```

After indexing,

```
($n nt)($i sentenceindex):($i >"at" $n)
```

gets the sentences.

6.2 Projecting Images Of Annotations

Sometimes even though an annotation layer draws children from some lower layer, it's useful to know what the closest correspondence is between the segments in that layer and some different lower layer. For instance, consider having both hand transcription and hand annotation for dialogue acts above it, and also ASR output with automatic dialogue act annotation on top of that. There is no relationship apart from timing between the hand and automatic dialogue acts, but to find out how well the automatic process works, it's useful to know whether it segments the hand transcribed words the same way, and with the same categories, as the hand annotation does.

`ProjectImage` is a tool that allows this comparison to be made. Given some source annotation that segments the data by drawing children from a lower layer, and the name of a target annotation that is defined as drawing children from a different lower layer, it creates the target annotation by adding annotations that are just like the source but with the other children. A child is inside a target segment if its timing midpoint is after the start and before the end of the source segment. If there are no such children, then the target element will be empty. `ProjectImage` adds a pointer from each target element back to its source element so that it's easy to check categories etc.

Note:

ProjectImage was committed to CVS on 21/11/2006 and will be in all subsequent NXT builds.

- 'Compiling from Source' p.9 (or use a build if there is one post 21/11/06).

- Edit your metadata file and prepare the ground. You need to decide what NXT element is being projected onto which other. As an example we'll look at Named Entities on the AMI corpus: imagine we want to project manually generated NEs onto ASR output to take a look at the result. You'll already have the manual NEs and ASR transcription declarations in your metadata:

```
<coding-file name="ne" path="namedEntities">
    <structural-layer draws-children-from="words-layer" name="ne-layer">
        <code name="named-entity" text-content="false">
            <pointer number="0" role="type" target="ne-types"/>
        </code>
    </structural-layer>
</coding-file>

<!-- ASR version of the words -->
<coding-file name="asr" path="ASR">
    <time-aligned-layer name="asr-words-layer">
        <code name="asrword" text-content="true"/>
        <code name="asrsil"/>
    </time-aligned-layer>
</coding-file>
```

  and now you need to add the projection layer into the metadata file, remembering to add a pointer from the target to source layer:

```
<!-- ASR Named entities -->
<coding-file name="ane" path="ASRnamedEntities">
    <structural-layer draws-children-from="asr-words-layer" name="asr-ne-layer">
        <code name="asr-named-entity" text-content="false">
            <pointer number="0" role="source_element" target="ne-layer"/>
            <pointer number="0" role="type" target="ne-types"/>
        </code>
    </structural-layer>
</coding-file>
```

- Using a standard NXT `CLASSPATH` or just using the `-cp` argument to the java command below like this: `-cp lib/nxt.jar:lib/xercesImpl.jar`, run `ProjectImage`:

```
java net.sourceforge.nite.util.ProjectImage -c /path/to/AMI-metadata.xml
          -o ES2008a -s named-entity -t asr-named-entity
```

  The arguments to `ProjectImage` are:

  - `-c` metadata file including definition for the target annotation
  - `-o` Optional observation argument. If it's not there the projection will be done for the entire corpus
  - `-s` source element name
  - `-t` target element name

The output is a (set of) standard NXT files that can be loaded with the others. To get textual output, use `FunctionQuery` on the target annotation resulting from running `ProjectImage` (see 'FunctionQuery' p.48).

6.2.1 Notes

`ProjectImage` can be used to project any type of data segment onto a different child layer, and so has many uses beyond the one described. The main restriction is that the segments must all use the same tag name. Although it might be more natural to define the imaging in terms of a complete NXT layer, the user would have to specify at the command line a complete mapping from source tags to target tags, which would be cumbersome. Moreover, many current segmentation layers use single tags. In future NXT versions we may consider generalizing to remove this restriction.

6.3 Reliability Testing

This section contains documentation of the facility for loading multiply-annotated data that forms the core of NXT's support for reliability tests, plus a worked example from the AMI project, kindly supplied by Vasilis Karaiskos. For more information, see the JavaDoc corresponding to the NOM loading routine for multiply-annotated data, for `CountQueryMulti`, and for `MultiAnnotatorDisplay`.

The facilities described on this page are new for NXT v 1.3.3.

6.3.1 Generic documentation

Many projects wish to know how well multiple human annotators agree on how to apply their coding manuals, and so they have different human annotators read the same manual and code the same data. They then need to calculate some kind of measurement statistic for the resulting agreement. This measurement can depend on the structure of the annotation (agreement on straight categorization of existing segments being simpler to measure than annotations that require the human to segment the data as well) as well as what field they are in, since statistical development for this form of measurement is still in progress, and agreed practice varies from community to community.

NXT 1.3.3 and higher provides some help for this statistical measurement, in the form of a facility that can load the data from multiple annotators into the same NOM (NXT's object model, or internal data representation, which can be used as the basis for Java applications that traverse the NOM counting things or for query execution).

This facility works as follows. The metadata specifies a relative path from itself to directories at which all coding files containing data can be found. (The data can either be all together, in which case the path is usually given on the `<codings>` tag, or it can be in separate directories by type, in which case the path is specified on the individual `<coding-file>` tags.) NXT assumes that if there is annotation available from multiple annotators, it will be found not in the specified directory itself, but in subdirectories of the directory specified, where the subdirectories is called by the names (or some other unique designators) of the annotators. Annotation schemes often require more than one layer in the NOM representation. The loading routine takes as arguments the name of the highest layer containing multiple annotations; the name of a layer reached from that layer by child links that is common between the two annotators, or null if the annotation grounds out at signal instead; and a string to use as an attribute name in the NOM to designate the annotator for some data. Note that the use of a top layer and a common layer below it allows the program to know exactly where the multiply annotated data is - it is in the top layer plus all the layers between the two layers, but not in the common layer. (It is possible to arrange annotation schemes so that they do not fit this structure, in which case, NXT will not support reliability studies on them.) The routine loads all of the versions of these multiply-annotated layers into the NOM, differentiating them by using the subdirectory name as the value for the additional attribute representing the annotator.

NXT is agnostic as to which statistical measures are appropriate. It does not currently (June 05) implement any, but leaves users to write Java applications or sets of NXT queries that allow their chosen measures to be calculated. (Being an open source project, of course, anyone who writes such applications can add them to NXT for the benefit of others who make the same choices.) Version 1.3.3 provides two end user facilities that will be helpful for these studies, which are essentially multiple annotator versions of the `GenericDisplay GUI` and of `CountQueryResults`.

6.3.2 MultiAnnotatorDisplay

This is a version of the `GenericDisplay` that takes additional command line arguments as required by the loading routine for multiply-annotated data, and renders separate windows for each annotation for each annotator. The advantage of using the GUI is, as usual, for debugging queries, since queries can be executed, with the results highlighted on the data display.

To call the GUI:

```
java net.sourceforge.nite.gui.util.MultiAnnotatorDisplay -c METADATAFILE
                  -o OBSERVATION -tl TOPLAYER [-cl COMMONLAYER] [-a ANNOTATOR]
```

`-c` *METADATAFILENAME* names a metadata file defining the corpus to be loaded.
`-tl` *TOPLAYER* names the data layer at the top of the multiple annotations to be loaded.

`-cl` *COMMONLAYER* is required only if the multiple annotations ground out in a common layer, and names the first data layer, reached by descending from the toplayer using child links, that is common between the multiple annotations.
`-a` *ANNOTATOR* is the name of the attribute to add to the loaded data that contains the name of the subdirectory from which the annotations were obtained - that is, the unique designator for the annotation. Optional; defaults to `coder`.

6.3.3 CountQueryMulti

To call:

```
java CountQueryMulti -corpus METADATAFILE -query QUERY
           -toplayer TOPLAYER -commonlayer COMMONLAYER
           [-attribute ANNOTATOR] [-observation OBSERVATION][-allatonce]
```

where arguments are as for `MultiAnnotatorDisplay`, apart from the following (which are as for `CountQueryResults`):

`-observation` *OBSERVATION*: the observation whose annotations are to be loaded. Optional; if not given, all observations are processed one by one with counts given in a table.
`-query` *QUERY*: the query to be executed.
`-allatonce`: Optional; if used, then the entire corpus is loaded together, with output counting over the entire corpus. This option is very slow and memory-intensive, and assuming you are willing to total the results from the individual observations, is only necessary if queries draw context from outside single observations.

6.3.4 Example reliability study

The remainder of this web page demonstrates an annotation scheme reliability test in NITE. The example queries below come from the agreement test on the named entities annotation of the AMI corpus. Six recorded meetings were annotated by two coders, whose marking were consequently compared. The categories and attributes that come into play are the following:

named-entity: new named entities - the data for which we are doing the reliability test. These are parents of words in the transcript. They are in a layer called `ne-layer`.
w: the words in the transcript. They are in a layer called `word-layer`.
ne-type: the categories a named entity can be assigned to. They are in an ontology, with the named entities pointing to them, using the `type` role.
name: an attribute of a named entity type that gives the category for the named entity (e.g., `timex`, `enamex`).
coder:an attribute of a named entity, signifying who marked the entity.

6.3.4.1 Loading the data into the GUI

The tests are being carried out by loading the annotated data on the NXT display `MultiAnnotatorDisplay` (included in nxt_1.3.3 and above). The call can be incorporated in a shell script along with the appropriate classpaths. For example, the following is included in our `multi.sh` script run from the root of the NXT install (`% sh multi.sh`). All the `CLASSPATH`s should be in a single line in the actual script.

```
#!/bin/bash
# Note that a Java runtime should be on the path.
# The current directory should be root of the nxt install.
# unless you edit this variable to contain the path to your install
# then you can run from anywhere. CLASSPATH statements need to be
# in a single line
NXT="."

# Adjust classpath for running under cygwin.
if [ $OSTYPE = 'cygwin' ]; then

export CLASSPATH=".;$NXT;$NXT/lib;$NXT/lib/nxt.jar;$NXT/lib/jdom.jar;
  $NXT/lib/JMF/lib/jmf.jar;$NXT/lib/pnuts.jar;$NXT/lib/resolver.jar;
  $NXT/lib/xalan.jar;$NXT/lib/xercesImpl.jar;$NXT/lib/xml-apis.jar;
  $NXT/lib/jmanual.jar;$NXT/lib/jh.jar;$NXT/lib/helpset.jar;$NXT/lib/poi.jar;
  $NXT/lib/eclipseicons.jar;$NXT/lib/icons.jar;$NXT/lib/forms-1.0.4.jar;
  $NXT/lib/looks-1.2.2.jar;$NXT/lib/necoderHelp.jar;$NXT/lib/videolabelerHelp.jar;
```

```
  $NXT/lib/dacoderHelp.jar;$NXT/lib/testcoderHelp.jar"

else

export CLASSPATH=".:$NXT:$NXT/lib:$NXT/lib/nxt.jar:$NXT/lib/jdom.jar:
  $NXT/lib/JMF/lib/jmf.jar:$NXT/lib/pnuts.jar:$NXT/lib/resolver.jar:
  $NXT/lib/xalan.jar:$NXT/lib/xercesImpl.jar:$NXT/lib/xml-apis.jar:
  $NXT/lib/jmanual.jar:$NXT/lib/jh.jar:$NXT/lib/helpset.jar:$NXT/lib/poi.jar:
  $NXT/lib/eclipseicons.jar:$NXT/lib/icons.jar:lib/forms-1.0.4.jar:
  $NXT/lib/looks-1.2.2.jar:$NXT/lib/necoderHelp.jar:$NXT/lib/videolabelerHelp.jar:
  $NXT/lib/dacoderHelp.jar:$NXT/lib/testcoderHelp.jar"

# echo "CLASSPATH=.:$NXT:$NXT/lib:$NXT/lib/nxt.jar:$NXT/lib/jdom.jar:
    $NXT/lib/JMF/lib/jmf.jar:$NXT/lib/pnuts.jar:$NXT/lib/resolver.jar:
    $NXT/lib/xalan.jar:$NXT/lib/xercesImpl.jar:$NXT/lib/xml-apis.jar:
    $NXT/lib/jmanual.jar:$NXT/lib/jh.jar:$NXT/lib/helpset.jar:$NXT/lib/poi.jar:
    $NXT/lib/eclipseicons.jar:$NXT/lib/icons.jar:lib/forms-1.0.4.jar:
    $NXT/lib/looks-1.2.2.jar:$NXT/lib/necoderHelp.jar:$NXT/lib/videolabelerHelp.jar:
    $NXT/lib/dacoderHelp.jar:$NXT/lib/testcoderHelp.jar\n";
fi

java net.sourceforge.nite.gui.util.MultiAnnotatorDisplay -c Data/AMI/AMI-metadata.xml
       -tl ne-layer -cl words-layer
```

A GUI with a multitude of windows will load (each window contains the data of one of the various layers of data and annotations), thus allowing comparisons between the choices of these coders. In our examples below the annotators are named Coder1 and Coder2.

Selecting Search off the menu bar will bring up a small GUI where the queries such as the ones below can be written. Clicking on any of the query results, highlights the corresponding data in the rest of the windows (words, named entities, coders' markings etc.). Simultaneously, underneath the list of matches, the query GUI expands whichever *n*-tuple is selected. For a the low-down on the NITE query language (NiteQL), look at the 'The NXT Query Language' p.34 or the Help menu in the query GUI.

6.3.4.2 Querying data related to a single annotator

```
($a named-entity) : $a@coder=="Coder1"
```

Give a list of all the named entities marked by `Coder1`.

```
($w w)(exists $a named-entity) : $a@coder="Coder1" && $a ^ $w
```

Give a list of all the words marked as named entities by `Coder1`.

```
($a named-entity): $a@coder=="Coder1" :: ($w w): $a ^ $w
```

Gives all the named entities marked by `Coder1` showing the words included in each entity.

```
($a named-entity)($t ne-type) : ($a >"type"^ $t) && ($t@name == "EntityType") && ($a@coder
== "Coder1")
```

Gives the named entities of type `EntityType` annotated by `Coder1`. The entity types (and their names) to choose from can be seen in the respective window in the GUI (titled "Ontology: ne-types" in this case).

```
($a named-entity)($t ne-type) : ($a >"type"^ $t) && ($t@name == "EntityType") && ($a@coder
== "Coder1") :: ($w w): $a ^ $w
```

Like the previous query, only each match also includes the words forming the entity.

```
($t ne-type) :: ($a named-entity) : $a@coder=="Coder1" && $a >"type"^ $t
```

Gives a list of all the named entity types (including `root`), and for each type, the entities of that type annotated by `Coder1`. By writing the last term of the query as `$a >"type" $t`, the query will match only the bottom level entity types (the ones used as actual tags), that is it will display `MEASURE` entities, but not `NUMEX` ones (assuming here that `MEASURE` is a sub-type of `NUMEX`).

```
($a named-entity)($t ne-type) : $a@coder=="Coder1" && $a >"type"^ $t :: ($w w): $a ^ $w
```

Like the previous query, only each match (*n*-tuple) also includes the words forming the entity.

6.3.4.3 Querying data related to two annotators

Checking for co-extensiveness

The following examples check for agreement between the two annotators as to whether some text should be marked as a named entity:

```
($a named-entity)($b named-entity): $a@coder=="Coder1" && $b@coder=="Coder2" :: ($w1 w)
(forall $w w) : ($a ^ $w1) && ($b ^ $w1) &&(($a ^ $w) -> ($b ^ $w)) && (($b ^ $w) -> ($a
^ $w))
```

Gives a lost of all the co-extensive named entities between `Coder1` and `Coder2` along with the words forming the entities (the entities do not have to be of the same type, but they have to span exactly the same text).

```
($a named-entity)($b named-entity): $a@coder=="Coder1" && $b@coder=="Coder2" :: ($w1 w)
(exists $w w) : ($a ^ $w1) && ($b ^ $w1) &&(($a ^ $w) -> ($b ^ $w)) && (($b ^ $w) -> ($a
^ $w))
```

Like the previous query, but includes named entities that are only partially co-extensive. The words showing in the query results are only the ones where the entities actually overlap.

```
($a named-entity)(forall $b named-entity)(forall $w w): $a@coder=="Coder1" &&
(($b@coder=="Coder2" && ($a ^ $w))->!($b ^ $w))
```

Gives the list of entities that only `Coder1` has marked, i.e. there is no corresponding entity in `Coder2`. Switching `Coder1` and `Coder2` in the query, gives the respective set of entities for `Coder2`.

```
($a named-entity)(forall $b named-entity)(forall $w w): $a@coder=="Coder2" &&
(($b@coder=="Coder1" && ($a ^ $w))->!($b ^ $w)) || $a@coder=="Coder1" &&
(($b@coder=="Coder2" && ($a ^ $w))->!($b ^ $w))
```

Like the previous query, only this time both sets of non-corresponding entities is given in one go.

Checking for categorisation agreement

The following examples check how the two annotators agree on the categorisation of co-extensive entities:

```
($a named-entity)($b named-entity) ($t ne-type): $a@coder=="Coder1" && $b@coder=="Coder2"
&& ($a >"type" $t) && ($b >"type" $t) :: ($w1 w) (forall $w w) : ($a ^ $w1) && ($b ^ $w1)
&&(($a ^ $w) -> ($b ^ $w)) && (($b ^ $w) -> ($a ^ $w))
```

Gives all the common named entities between `Coder1` and `Coder2` along with the entity type and text; the entities have to be co-extensive (fully overlapping) and of the same type.

```
($a named-entity)($b named-entity) ($t ne-type): $a@coder=="Coder1" && $b@coder=="Coder2"
&& ($a >"type" $t) && ($b >"type" $t) :: ($w1 w) (exists $w w) : ($a ^ $w1) && ($b ^ $w1)
&&(($a ^ $w) -> ($b ^ $w)) && (($b ^ $w) -> ($a ^ $w))
```

Like the previous query, but includes partially co-extensive entities. The words showing in the query results are only the ones that actually do overlap.

```
($a named-entity)($b named-entity) ($t ne-type): $a@coder=="Coder1" && $b@coder=="Coder2" && ($a >"type" $t) && ($b >"type" $t) :: ($w2 w):($a ^ $w2) && ($b ^ $w2) :: ($w w):(($b ^ $w) && !($a ^ $w)) || (($a ^ $w) && !($b ^ $w))
```

Gives the list of entities which are the same type, but only partially co-extensive. The results include the entire set of words from both codings.

```
($a named-entity)($b named-entity) ($t ne-type)($t1 ne-type): $a@coder=="Annotator1" && $b@coder=="Annotator2" && ($a >"type" $t) && ($b >"type" $t1) && ($t != $t1) :: ($w1 w) (exists $w w) : ($a ^ $w1) && ($b ^ $w1) &&(($a ^ $w) -> ($b ^ $w)) && (($b ^ $w) -> ($a ^ $w)) :: ($w2 w): ($b ^ $w2)
```

Gives the list of entities, which are partially or fully co-extensive, but for which the two coders disagree as to the type.

```
($a named-entity)($b named-entity)($c ne-type)($d ne-type): $a@coder=="Coder1" && $b@coder=="Coder2" && $c@name="EntityType1" && $d@name="EntityType2"&& $a>"type"^ $c && $b>"type"^ $d :: ($w2 w):($a ^ $w2) && ($b ^ $w2)
```

Gives the list of entities which are partially or fully co-extensive, and which `Coder1` has marked as `EntityType1` (or one of its sub-types) and `Coder2` has marked as `EntityType2` (or one of its sub-types). This checks for type-specific disagreements between the two coders.

```
($t ne-type): !($t@name="ne-root") :: ($a named-entity)($b named-entity): $a@coder=="Coder1" && $b@coder=="Coder2" && (($a >"type"^ $t) && ($b >"type"^ $t)) :: ($w1 w) (forall $w w) : ($a ^ $w1) && ($b ^ $w1) &&(($a ^ $w) -> ($b ^ $w)) && (($b ^ $w) -> ($a ^ $w))
```

The query creates a list of all the entity types, and slots in each entry all the (fully) co-extensive entities as marked by the two coders. The actual text forming each entity is also included in the results.

```
($t1 ne-type): !($t1@name="ne-root") :: ($a named-entity)($b named-entity): $a@coder=="Coder1" && $b@coder=="Coder2" && (($a >"type"^ $t1) && ($b >"type"^ $t1)) :: ($w1 w) (exists $w w) : ($a ^ $w1) && ($b ^ $w1) &&(($a ^ $w) -> ($b ^ $w)) && (($b ^ $w) -> ($a ^ $w))
```

Like the previous query, but includes partially co-extensive entities. The words showing in the query results are only the ones that actually do overlap.

7 Graphical user interfaces

NXT gives three different levels of support for graphical user interfaces. The first is a very basic data display that will always work for data in the correct format. The second is a set of configurable end user tools for common coding tasks that covers simple timestamped labelling plus a range of discourse coding types. Finally, NXT contains a number of libraries that can be used to build tailored end user interfaces for a particular corpus.

7.1 Preliminaries

7.1.1 Invoking the GUIs

Most NXT corpora come with a script for invoking the GUIs that work with the data; look for a top level file with the extension `.bat` (for Windows), `.sh` (for Linux), or `.command` (for Mac OSX). Where these scripts fail to work, it is usually because you need to edit them because you have put the data in a different place than where the script author expected. These start-up scripts will give as options the standard search gui and generic display gui, plus any other interfaces that have been registered for the corpus by editing the callable-programs section of the metadata. For corpora with many different annotations, the generic display as accessed in this way is unusable because by default, it tries to load and display everything - the command line call will give better control.

7.1.2 Time Highlighting

7.1.3 Search Highlighting

In the search highlighting, if there isn't a direct representation of some element on the display, then there's nothing to highlight. For instance, in many data sets timestamped orthographic transcription consists of `w` and `sil` elements but the `sil` elements are not rendered in the display, so the query `($s sil):` won't cause any highlighting to occur. This can be confusing but it is the correct behaviour. Good interface design will have a screen rendering for any elements of theoretical importance.

7.2 Generic tools that work on any data

Any corpus in NXT format is immediately amenable to two different graphical interfaces that allow the corpus to be searched, even without writing tailored programs. The first is a simple search GUI, and the second is a generic data display program that works in tandem with the search GUI to highlight search results.

7.2.1 The NXT Search GUI

The search GUI can be reached either by using `search.bat/search.sh` and specifying which corpus to load or by using the `.bat/.sh` for the specific corpus (if it exists) and choosing the Search option. It has two tabbed windows. The query tab allows the user to type in a query. Cut and paste from other applications works with this window. The query can also be saved on the Bookmark menu, but at May 2004 this doesn't work well for long queries. There is a button to press to do the search, which automatically takes the user either to a pop-up window with an error message explaining where the syntax of the query is incorrect, or, for a valid query, to the result tab. This window shows the results as an XML tree structure, with more information about the element the user has selected (with the mouse) displayed below the main tree.

The GUI includes an option to save the XML result tree to a file. This can be very handy in conjunction with `knit` for performing data analysis. It also includes an option to save the results in a rudimentary *Excel* spreadsheet. This is less handy, especially in the case of complex queries, because the return value is hierarchically structured but the spreadsheet just contains information about each matched element dumped into a flat list by performing a depth-first, left-to-right traversal of the results. However, for relatively simple queries and people who are used to data filtering and pivot tables in *Excel*, it can be the easiest first step for analysis.

The search GUI works on an entire corpus at once. This can make it slow to respond if the corpus is very large or if the query is very complicated (although of course it's possible to comment out observations in the metadata to reduce the amount of information it loads). Sometimes a query is slow because it's doing something more complicated than what the user intended. A query can be interrupted mid-processing and will still return a partial result list, which can be useful for checking it.

At May 2004, when the user chooses to open a corpus from the File menu, the search GUI expects the metadata file to be called *something*`.corpus`, although many users are likely to have it called *something*`.xml` (so that it behaves

properly in other applications like web browsers). Choose the All files option (towards the bottom of the open dialogue box) in order to see `.xml` files as well as `.corpus` ones.

7.2.2 The Generic Display

NXT comes with a generic display so that it can at least display and search any corpus in NXT format "out of the box", without having to configure the end user coding tools or build a tailored tool. It provides the absolute basics. It isn't meant for serious use, but it can be useful to test out new data or if you don't need GUIs often enough to spend time getting something better set up.

The Generic Display works on one observation at a time. It can be invoked at the command line as follows:

```
java net.sourceforge.nite.gui.util.GenericDisplay -c CORPUS -o OBS -f FONTSIZE -q QUERY
```

In the call, *CORPUS* gives a path to a metadata file and *OBS* names an observation that is listed in that metadata file. These are mandatory. You may optionally specify a font size for the data rendering. You may also specify a query that will be used to choose kinds of data for display. Only the variable type information will be used in the processing; the display will show data just from the files that include data that matches variables of those types. For instance, -q '($w word)($d dialogue-act):' will render display windows for words and dialogue-acts only, ignoring all other data. This is particularly useful for corpora with many different kinds of annotation, where it would create too busy a display to show everything. For larger corpora, NXT is unable to render all of the annotations at once because this would take too much memory. It is only possible to run the generic display for such corpora with the -q option.

The Generic Display simply puts up an audio/video window for each signal associated with an observation, plus one window per coding that shows the elements in an `NTextArea`, one element per line, with indenting corresponding to the tree structure and a rendering of the attribute values, the PCDATA the element contains, and enough information about pointers to be able to find their targets visually on the other windows. It doesn't try to do anything clever about window placement. As with other NXT GUIs, there is a Search menu, and the display shows both search and time highlights.

7.3 Configurable end user coding tools

There are currently three built-in and configurable end user GUIs for common interface requirements.

7.3.1 The signal labeller

The signal labeller is for creating timestamped labels against signal, with the labels chosen from an enumerated list. This can be used for a very wide range of low-level annotations, such as gaze direction, movement in the room, rough starts and ends of turns, and areas to be included in or excluded from some other analysis. The tool treats the labels as mutually exclusive and exhaustive states; as the user plays the signal, whenever a new label is chosen (either with the mouse or using keyboard shortcuts), that time is used both for the beginning of the new label and the end of the old one. Although there are several similar tools available, this tool will work on either audio or video signals, including playing a set of synchronized signals together, and works natively on NXT format data, which is of benefit for user groups that intend to use NXT for further annotation. It does not, however, currently include either the palette-based displays popular from *Anvil* and *TASX*, and the signal control is meant for the coarser style of real-time coding, not for the precision timing that some projects require. It also does not contain waveform display, and therefore is unsuitable for many kinds of speech annotation.

Java class: `net.sourceforge.nite.tools.videolabeler.ContinuousVideoLabeling`

7.3.2 The discourse entity coder

The second end user GUI is for coding discourse entities above an existing text or speech transcription. Coding is performed by sweeping out the words in the entity and then mousing on the correct entity type from a static display of the named entity type ontology, or choosing it by keyboard shortcut. It can be used for any coding that requires the user to categorize contiguous stretches of text (or of speech by one person) using labels chosen from a tree-shaped ontology. In addition, it allows the user to indicate directional relationships between two coded entities, with the relationship categorized from a set of labels. The most common uses for this style of interface are in marking up named entities and coreferential relationships.

Java class: `net.sourceforge.nite.tools.necoder.NECoder`

7.3.3 The discourse segmenter

The final GUI is for segmenting discourse into contiguous stretches of text (or of speech by one person) and categorizing the segments. The most common use for this style of interface is a dialogue act coder. Coding is performed by marking the end of each discourse segment; the segment is assume to start at the end of the last segment (or the last segment by the same speaker, with the option of not allowing segments to draw words across some higher level boundary, such as previously marked speaker turns). A permanent dialogue box displays information about the currently selected act and allows a number of properties to be specified for it beyond simple type. The coding mechanisms supported include a tick-box to cover boolean properties such as termination of the act before completion, free text comments, and choice from a small, enumerated, mutuallly exclusive list, such as might be used for noting the dialogue act's addressee. Although this structure covers some styles of dialogue act coding, this tool is not suitable for schemes such as MRDA where dual-coding from the same act type list is allowed. This tool additionally allows the user to indicate directional relationships between acts using the same mechanism as in the discourse entity coder, although for current dialogue act schemes this is a minority requirement.

Java class: `net.sourceforge.nite.tools.dacoder.DACoder`

7.3.4 The non-spanning comparison display

This is the first in a series of tools to display multiple versions of a particular type of annotation. The *non-spanning comparison display* can show two different annotators' data over the same base-level transcription. We use *annotator* loosely to mean any human or machine process that results in an annotation. This is a display only, not an annotation tool. Display details are controlled using a configuration file much like the other end user GUIs, though there are two extra settings required (see below). The display shows the two annotators' names, with the first underlined and the second italicised, in a small Annotator Legend window. Every annotation by the first annotator is underlined and every annotation by the second is italizized so that the two can be distinguished. The types of annotations will be distinguished in the same way as for the discourse entity coder.

Java class: `net.sourceforge.nite.tools.comparison.nonspanning.NonSpanningComparisonDisplay`

7.3.5 The dual transcription comparison display

This is the second in a series of tools to display multiple versions of a particular type of annotation. The *dual transcription comparison display* displays two transcriptions side-by-side using a different configuration for each. For example, a manual transcription on the left and an automatic transcription on the right. This is a display only, not an annotation tool. Display details are controlled using a configuration file much like the other end user GUIs. Any annotations on the transcriptions will be displayed in the same way as for the non spanning comparison display.

Java class: `net.sourceforge.nite.tools.comparison.dualtranscription.DualTranscriptionComparisonDisplay`

7.3.6 How to configure the end user tools

There are two basic steps to configure one of these end-user tools for your corpus:

7.3.6.1 Edit the Metadata File

Consider what you want to code and which tool you want to use. Edit the codings and layers in the metadata file for your new annotation, then add something like this to the `callable-programs` section of your metadata file:

```
<callable-program description="Named Entity Annotation"
  name="net.sourceforge.nite.tools.necoder.NECoder">
    <required-argument name="corpus" type="corpus"/>
    <required-argument name="observation" type="observation"/>
    <required-argument name="config" default="myConfig.xml"/>
    <required-argument name="corpus-settings" default="my-corpus-settings-id"/>
    <required-argument name="gui-settings" default="my-gui-settings-id"/>
</callable-program>
```

This tells NXT to allow the use of the built-in Named Entity coder on this corpus. When you start up `net.sourceforge.nite.nxt.GUI` on this metadata, a new entry will appear called `Named Entity Annotation`. The `required-argument`s require first that the `corpus` (metadata file name) is passed to the tool and than an observation is chosen by the user. The third `required-argument`, `config` tells NXT where to find the configuration file for this tool, relative to the metadata, and the last two tell it which settings to use within that file (see next section).

7.3.6.2 Edit or Create the Configuration File

Configuration files can look complicated but the requirements to get started are really quite simple. One example configuration file is included in the NXT distribution as `lib/nxtConfig.xml`. It contains extensive comments about what the settings mean. Below is a full discussion of the elements and attributes of the configuration files, but to continue with the above example, here is a configuration file (according to the above metadata fragment, it should be called `myConfig.xml` and located in the same directory as the metadata). This configures the named entity coder:

```
<NXTConfig>
  <DACoderConfig>
    <!-- Corpus settings for the ICSI corpus -->
    <corpussettings
        id                        = "my-corpus-settings-id"
        segmentationelementname = "segment"
        transcriptionlayername  = "words-layer"
        transcriptiondelegateclassname = "MyTranscriptionToTextDelegate"
        neelementname             = "named-entity"
        neattributename           = "type"
        annotatorspecificcodings= "nees"
    />

    <guisettings
        id                        = "my-gui-settings-id"
        gloss                     = "My Corpus settings"
        applicationtitle          = "My Corpus Tool"
    />

  </DACoderConfig>
</NXTConfig>
```

Note the `corpussettings` element with the `ID ="my-corpus-settings-id"` as referred to in the metadata file, and similarly a `guisettings` element named `my-gui-settings-id`. In this way, a configuration file can contain any number of different configurations for different corpora as well as different tools, though it's normally clearer to have at least one config file per corpus.

Some Important Settings

`neelementname`

the name of the element, which must be present in the metadata file, that will be created by the named entity tool

`neattributename`

if this is present, we are using an enumerated attribute directly on the `neelementname` rather than a pointer into a type hierarchy. The attribute must be present in the metadata file and must be enumerated. To use a pointer into a type hierarchy you should specify at least the `neontology`, `neroot`, `nenameattribute` and `netyperole` instead of this single attribute. Note: this feature is only available in NXT versions after March 2006.

`segmentationelementname`

the element used to split the transcription into 'lines'. It is normally assumed this is an agent-coding and if so, the agent associated with each speaker is placed in front of each line.

`transcriptionlayername`

the layer that contains the transcription to be printed. How it actually appears can be specified using `transcriptiondelegateclassname`.

`transcriptiondelegateclassname`

if this is absent, any element in the `transcriptionlayername` with text content will be displayed as transcription. If it is present, each element is passed to the delegate class in order to display the transcription. Any such delegate

class has to implement the Java interface `TranscriptionToTextDelegate` which contains the single method `getTextForTranscriptionElement(NOMElement nme)`.

7.3.6.3 Config File Detail

This section is a detailed look at the settings in the NXT configuration files. Note: some of these settings can only be used in NXT builds after 1.3.5 (9/5/06). The details may not be entirely static.

At the top level, in the `NXTConfig` element, there are currently two possible subelements: `DACoderConfig` and `CSLConfig`. The first is for configuring discourse coder tools (dialogue act coder; named entity coder etc). The second is for configuring the 'Continuous Signal Labeller' p.58 tool

Both `CSLConfig` and `DACoderConfig` can contain any number of `corpussettings` and `guisettings` elements, each of which has an `id` attribute to uniquely identify it: often these `ID`s will be used in the `CallableTools` section of an NXT metadata file. `guisettings` are preferences that affect the overall look of the interface and `corpussettings` tell NXT about the elements to be displayed and annotated. The detail of what goes where is described in each subsection below.

DACoderConfig

`guisettings` attributes

`id`

Unique identifier

`gloss`

Example element containing short explanation of all possible settings

`showapwindow`

If true, the Adjacency Pair (or relation) window is shown in the discourse entity coder. Defaults to `true`.

`showlogwindow`

If `true`, the log feedback window is shown. Defaults to `true`.

`applicationtitle`

The title that you want to see in the main frame

`wordlevelselectiontype`

This determines what units are selectable on the speech transcriptions (assuming `transcriptselection` is not `="false"`). The are currently five valid strings - anything else will result in the default behaviour: `in_segment_phrase`. The values and their meanings are: `one_word`: only single words can be selected at a time; `one_segment`: only single segments can be selected; `multiple_segments`: multiple complete sements can be selected; `in_segment_phrase`: contiguous words that lie within a single segment can be selected; `cross_segment_phrase`: contiguous words across segments can be selected (note that the selection can in fact be discontiguous if `multiagentselection` is not true).

`transcriptselection`

This determines whether you can select speech transcription elements. If this is `false` no speechtext selection will take place, regardless of settings such as `allowMultiAgentSelect` or `wordlevelSelectionType`. Defaults to `="true"`.

`annotationselection`

This determines whether you can select annotation elements. If this is `false` no annotation selection will take place, regardless of other settings. Defaults to `true`.

`multiagentselection`

This determines whether you can select data from more than one agent. If this is `true` such selection can take place. Defaults to `false`.

`corpussettings` attributes

`id`

Unique identifier

`gloss`

Example element containing short explanation of all possible settings

`segmentationelementname`

Element name of the segmentation elements that pre-segments the transcription layer. Used for the initial display of the text.

`segmenttextattribute`

Name of the attribute on the segment element to use as the header of each transcription line. Use a delegate (below) for more complex derivation. If neither delegate nor attribute is set, the agent is used as the line header (if agent is specified).

`segmenttextdelegateclassname`

full class name of a `TranscriptionToTextDelegate` that derives the text of the segment header from each segment element. Note this is not the transcription derivation, just the derivation of the header for each line of transcription. If neither this delegate nor `segmenttextattribute` is set, the agent is used as the line header (if agent is specified).

`transcriptionlayername`

LAYER name of the transcription layer

`transcriptionattribute`

Name of the attribute in which text of transcription is stored. Leave out if text not stored in attribute.

`transcriptiondelegateclassname`

full class name of `TranscriptionToTextDelegate`. Leave out is no delegate is used. `net.sourceforge.nite.gui.util.AMITranscriptionToTextDelegate` is an example delegate class that works for the AMI corpus. For a new corpus you may have to write your own, but it is a simple process.

`daelementname`

element name of dialogue act instances

`daontology`

ontology name of dialogue acts

`daroot`

nite-id of dialogue act root

`datyperole`

role name of the pointer from a dialogue act to its type

`daattributename`

The enumerated attribute on the DA element used as its 'type'. If this attribute is set, the `daontology`, `daroot` and `datyperole` attributes are ignored.

`dagloss`

the name of the attribute of the dialog act types that contains some extra description of the meaning of this type

`apelementname`

element name of adjacency pair instances

`apgloss`

the name of the attribute of the relation types that contains some extra description of the meaning of this type

`apontology`

ontology name of adjacency pairs

`aproot`

nite-id of adjacency pair root

`defaultaptype`

nite-id of default adjacency pair type

`aptyperole`

role name of the pointer from a AP to its type

`apsourcerole`

role name of the pointer from a AP to its source

`aptargetrole`

role name of the pointer from a AP to its target

`neelementname`

element name of named entity instances

`neattributename`

The enumerated attribute on the NE element used as its 'type'. If this attribute is set, the `neontology`, `neroot` and `netyperole` attributes are ignored.

`neontology`

ontology name of named entities

`neroot`

nite-id of named entities root

`neontologyexpanded`

set to `false` if you want the ontology to remain in un-expanded form on startup. The default is to expand the tree.

`nenameattribute`

attribute name of the attribute that contains the name of the named entity

`netyperole`

role name of the pointer from a named entity to its type

`nenesting`

Set to `true` to allow named entities to nest inside each other. Defaults to `false`.

`nemultipointers`

if this is *true* each span of words can be associated with multiple values in the ontology. Note that this only makes sense when the `neattributename` is not set - this setting is ignored if `neattributename` is set. It also requires that the `nenesting` attribute is *true*.

`abbrevattribute`

name of the attribute which contains an abbreviated code for the named entity for in-text display

`nelinkelementname`

The element linking NEs together. Used by `NELinker`.

`nelinkattribute`

The enumerated attribute on the NE link element used as its 'type'. If this attribute is set, the `nelinkontology`, `nelinkroot` and `nelinkrole` attributes are ignored, and the `nelinktypedefault` if present is the default string value of the type. Used by `NELinker`.

`nelinkontology`

The type ontology pointed to by the NE link element. Used by `NELinker`.

`nelinkroot`

The root of the type ontology pointed into by the NE link element. Used by `NELinker`.

`nelinktyperole`

The role used to point into the type ontology by the NE link element. Used by `NELinker`.

`nelinktypedefault`

The default type value for NE link elements. Used by `NELinker`.

`nelinksourcerole`

The role of the pointer from the link element to the first (or source) NE element. Used by `NELinker`.

`nelinktargetrole`

The role of the pointer from the link element to the second (or target) NE element. Used by `NELinker`.

`annotatorspecificcodings`

the semi-colon-separated list of codings that are annotator specific, i.e. for which each individual annotator will get his or her own datafiles. Usually these are the codings for all layers that will be annotated in the `DACoder`; see AMI example. This setting only has effect when the tool is started for a named annotator or annotators.

`nsannotatorlayer`

Only used by `NonSpanningComparisonDisplay` this specifies the layer containing elements to compare. This is the top layer passed to the multi-annotator corpus load.

`nscommonlayer`

Only used by `NonSpanningComparisonDisplay` this is the layer that is common between all annotators - it will normally be the same layer as `transcriptionlayername`.

In the above settings, `da` and `ap` prefixes are used in the attribute names here (standing for 'dialogue act' and 'adjacency pair'), these can refer to any kind of discourse elements and relations between them you wish to annotate.

CSLCoderConfig

`guisettings` attributes

`id`

Unique identifier

`gloss`

Example CSL settings, giving an explanation for every entry.

`autokeystrokes`

Optional (default `false`): if `true`, keystrokes will be made automatically if no keystroke is defined in the corpus data or if the defined keystroke is already in use.

`showkeystrokes`

Optional (default `off`): set to `off` (keystroke won't be shown in the GUI), `tooltip` (keystroke will be shown in the tooltip of a control) or `label` (keystroke will be shown in the label of a control).

`continuous`

Optional (default `true`): if `true`, the CSL tool will ensure that annotations remain continuous (prevent gaps in the time line)

`syncrate`

Optional (default `200`): the number of milliseconds between time change events from the NXT clock

`timedisplay`

Optional (default `seconds`): the type of display of coding times in the annotation window: if `minutes` then the format is like that of the clock h:mm:ss.ms

`corpussettings` attributes

`id`

Unique identifier

`gloss`

Example CSL settings for Dagmar demo corpus

`annotatorspecificcodings`

pose

For the 'Continuous Signal Labeller' p.58 we expect the `corpussettings` element to contain a number of `layerinfo` elements, each of which can contain these attributes. Each layer named within the current `corpussettings` element can be coded using the same tool: users choose what they're annotating using a menu.

`corpussettings / layerinfo` attributes

`id`

Unique identifier

`gloss`

Textual description of this layer

`codename`

Name of the elements that are annotated in the given layer

`layername`

The name of the layer that you want to code in the video labeler

`layerclass`

Delegate `AnnotationLayer` class. Defaults to `net.sourceforge.nite.tools.videolabeler.LabelAnnotationLayer`

`controlpanelclass`

Delegate `TargetControlPanel` class. Defaults to `net.sourceforge.nite.tools.videolabeler.LabelTargetControlPanel`

`enumeratedattribute`

Either this or `pointerrole` are required for `LabelAnnotationLayer`: name of the attribute that should be set - attribute must exist on the `codename` element and must be enumerated - currently no flexibility is offered in the keyboard shortcuts - they always start at "1" and increase alphanumerically.

`pointerrole`

Either this or `enumeratedattribute` are required for `LabelAnnotationLayer`: role of the pointer that points to the object set or ontology that contains the labels.

`labelattribute`

Required for `LabelAnnotationLayer`: name of the attribute of an object set or ontology element that contains the label name.

`evaluationattribute`

Required for `FeeltraceAnnotationLayer`: name of the double value attribute that contains the evaluation of an emotion.

`activationattribute`

Required for `FeeltraceAnnotationLayer`: name of the double value attribute that contains the activation of an emotion.

`showlabels`

Optional (default true) for `FeeltraceTargetControlPanel`: if `true`, labels for some predefined emotions will be shown in the Feeltrace circle.

`clickannotation`

Optional (default `false`) for `FeeltraceTargetControlPanel`: if `true`, the user can click to start and end annotating; if `false`, the user should keep the mouse button pressed while annotating.

7.4 Libraries to support GUI authoring

Please refer to the NXT *Javadoc* and the example programs in the `samples` directory.

7.4.1 The NXT Search GUI as a component for other tools

It's often useful for applications to be able to pop up a search window and react to search results as they are selected by the user. Using any in-memory corpus that implements the `SearchableCorpus` interface (for example `NOMWriteCorpus`), you can very simply achieve this. If *nom* is a valid `SearchableCorpus` we could use:

```
net.sourceforge.nite.search.GUI searchGui = new GUI(nom);
searchGui.registerResultHandler(handler);
...
searchGui.popupSearchWindow();
```

In this extract, the first line initializes the search GUI by passing it a `SearchableCorpus`. The second line tells the GUI to inform *handler* when search results are selected. *handler* must implement the `QueryResultHandler` interface. This simple interface is already implemented by some of NXT's own GUI components like `NTextArea`, but this mechanism allows you complete freedom to do what you want with search results. There is no obligation to register a result handler at all, but it may result in a less useful interface.

The third line of the listing actually causes the search window to appear and will normally be the result of a user action like selecting the Search menu item or something similar.

8 Using NXT in conjunction with other tools

This section contains specific information for potential users who need NXT's features about how to record, transcribe, and otherwise mark up their data before up-translation to NXT. NXT's earliest users have mostly been from computational linguistics projects. This is partly because of where it comes from - it arose out of a collaboration among two computational linguistics groups and an interdisciplinary research centre - and partly because for most uses, its design assumes that the projects that use it will have access to a programmer to set up tailored tools for data coding and to get out some kinds of analysis, or at the very least someone on the project will be willing to look at XML. However, NXT is useful for linguistics and psychology projects based on corpus methods as well. This web page is primarily aimed at them, to tell them problems to look out for, help them assess what degree of technical help they will need in order to carry out the work successfully, and give a sense of what sorts of things are possible with the software.

8.1 Recording Signals

8.1.1 Signal Formats

For information on media formats and JMF, see 'How to Play Media in NXT' p.7.

It is a good idea to produce a sample signal and test it in NXT (and any other tools you intend to use) before starting recording proper, since changing the format of a signal can be confusing and time-consuming. There are two tests that are useful. The first is whether you can view the signal at all under any application on your machine, and the second is whether you can view the signal from NXT. The simplest way of testing the latter is to name the signal as required for one of the sample data sets in the NXT download and try the generic display or some other tool that uses the signal. For video, if the former works and not the latter, then you may have the video codec you need, but NXT can't find it - it may be possible to fix the problem by adding the video codec to the JMF Registry. If neither works, the first thing to look at is whether or not you have the video codec you need installed on your machine. Another common problem is that the video is actually OK, but the header written by the video processing tool (if you performed a conversion) isn't what JMF expects. This suggests trying to convert in a different way, although some brave souls have been known to modify the header in a text editor.

We have received a request to implement an alternative media player for NXT that uses *QT Java* (the QuickTime API for Java) rather than JMF. This would have advantages for Mac users and might help some PC users. We're currently considering whether we can support this request.

8.1.2 Capturing Multiple Signals

Quite often data sets will have multiple signals capturing the same observation (videos capturing different angles, one audio signal per participant, and so on). NXT expresses the timing of an annotation by offsets from the beginning of the audio or video signal. This means that all signals should start at the same time. This is easiest to guarantee if they are automatically synchronized with each other, which is usually done by taking the timestamp from one piece of recording equipment and using it to overwrite the locally produced timestamps on all the others. (When we find time to ask someone who is technically competent exactly how this is done, we'll insert the information here.) A distant second best to automatic synchronization is to provide some audibly and visibly distinctive event (hitting a colourful children's xylophone, for instance) that can be used to manually edit the signals so that they all start at the same time.

8.1.3 Using Multiple Signals

Most coding tools will allow only one signal to be played at a time. It's not clear that more than this is ever really required, because it's possible to render multiple signals onto one. For instance, individual audio signals can be mixed into one recording covering everyone in the room, for tools that require everyone to be heard on the interface. Soundless video or video with low quality audio can have higher quality audio spliced onto it. For the purposes of a particular interface, it should be possible to construct a single signal to suit, although these might be different views of the data for different interfaces (hence the requirement for synchronization - it is counter-productive to have different annotations on the same observation that use different time bases). The one sticking point is where combining multiple videos into one split-screen view results in an unacceptable loss of resolution, especially in data sets that do not have a "room view" video in addition to, say, individual videos of the participants.

From NXT 1.3.0 it is possible to show more than one signal simultaneously by having the application put up more than one media player. If one signal is selected as the master by clicking the checkbox on the appropriate media player, that signal will control the time for all the signals: it will be polled for the current time that will be sent to the other signals (and anything else that monitors time). The number of signals that can successfully play in sync on NXT depends on the spec

of your machine and the encoding of the signals. Where sync is seriously out, NXT will attempt to correct the drift by pausing all the signals and re-aligning. If this happens too often, it's a good sign your machine is struggling. If you intend to rely on synchronization of multiple signals, you should test your formats and signal configuration on your chosen platform carefully.

8.2 Transcription

One of the real benefits of using NXT is the fact that it puts together timing information and linguistic structure. This means that most projects transcribing data with an eye to using NXT want a transcription tool that allows timings to be recorded. For rough timings, a tool with a signal (audio or video) player will do, especially if it's possible to slow the signal down and go back and forth a bit to home in on the right location (although this greatly increases expense over the sort of "on-line" coding performed simply by hitting keys for the codes as the signal plays). For accurate timing of transcription elements - which is what most projects need - the tool must show the speech waveform and allow the start and end times of utterances (or even words) to be marked using it.

NXT does not provide any interface for transcription. It's possible to write an NXT-based transcription interface that takes times from the signal player, but no one has. Providing one that allows accurate timestamping is a major effort because NXT doesn't (yet?) contain a waveform generator. For this reason, you'll want to do transcription in some other tool and import the result into NXT.

8.2.1 Using special-purpose transcription tools

There are a number of special-purpose transcription tools available. For signals that effectively have one speaker at a time, most people seem to use *Transcriber* or perhaps *TransAna*. For group discussion, *ChannelTrans* which is a multi-channel version of *Transcriber*, seems to be the current tool of choice. *iTranscribe* is a ground-up rewrite of it that is currently in pre-release.

Although we have used some of these tools, we've never evaluated them from the point of view of non-computational users (especially whether or not installation is difficult or whether in practice they've required programmatic modification), so we wouldn't want to endorse any particular one, and of course, there may well be others that work better for you.

Transcriber's transcriptions are stored in an XML format that can be up-translated to NXT format fairly simply. *TransAna*'s are stored in an SQL database, so the up-translation is a little more complicated; we've never tried it but there are NXT users who have exported data from SQL-based products into whatever XML format they support and then converted that into NXT.

8.2.2 Using programs not primarily intended for transcription

Some linguistics and psychology-based projects use programs they already have on their computers (like *Microsoft Word* and *Excel*) for transcription, without any modification. This is because (a) they know they want to use spreadsheets for data analysis (or to prepare data for importation into *SPSS*) and they know how to get there from here, (b) because they can't afford software licenses but they know they've already paid for these ones; and (c) they aren't very confident about installing other software on their machines.

Using unmodified standard programs can be successful, but it takes very careful thought about the process, and we would caution potential users not to launch blindly into it. We would also argue that since there are now programs specifically for transcription that are free and work well on Windows machines, there is much less reason for doing this than there used to be. However, whatever you do for transcription, you want to avoid the following.

- hand-typing times (for instance, from a display on the front of a VCR), because the typist *will get them wrong*

- hand-typing codes (for instance, `{laugh}`, because the typist *will get them wrong*

In short, avoid hand-typing *anything* but the orthography, and especially anything involving numbers or left and right bracketing. These are practices we still see regularly, mostly when people ask for advice about how to clean up the aftermath. Which is extremely boring to do, because it takes developing rules for each problem (`{laughs}, {laugh, laugh), laugh, {laff}, {luagh}`... including each possible way of crossing nested brackets accidentally), inspecting the set as you goes to see what the next rule should be. Few programmers will take on this sort of job voluntarily (or at least not twice), which can make it expensive. It is far better (...easier, less stressful, better for staff relations, less expensive...) to sort out your transcription practices to avoid these problems.

More as a curiosity than anything else, we will mention that it is possible to tailor *Microsoft Word* and *Excel* to contain buttons on the toolbars for inserting codes, and to disable keys for curly brackets and so on, so that the typist can't easily get them wrong. We know of a support programmer who was using these techniques in the mid-90s to support corpus projects, and managed to train a few computationally-unskilled but brave individuals to create their own transcription and coding interfaces this way. If you really must use these programs, you really should consider these techniques. (Note to the more technical reader or anyone trying to find someone who knows how it works these days: the programs use Visual Basic and manipulate *Word* and *Excel* via their APIs; they can be created by writing the program in the VB editor or from the end user interface using the Record Macro function, or by some combination of the two.) In the 90's, the Microsoft platform changed every few years in ways that required the tools to be continually reimplemented. We don't know whether this has improved or not.

Up-translating transcriptions prepared in these programs to NXT can be painful, depending upon exactly how the transcription was done. It's best if all of the transcription information is still available when you save as "text only". This means, for instance, avoiding the use of underlining and bold to mean things like overlap and emphasis. Otherwise, the easiest treatment is to save the document as HTML and then write scripts to convert that to NXT format, which is fiddly and can be unpalatable.

8.2.3 Using Forced Alignment with Speech Recognizer Output to get Word Timings

Timings at the level of the individual word can be useful for analysis, but they are extremely expensive and tedious to produce by hand, so most projects can only dream about them. It is actually becoming technically feasible to get usable timings automatically, using a speech recognizer. By "becoming", we mean that computational linguistics projects, who have access to speech specialists, know how to do it well enough that they think of it as taking a bit of effort but not requiring particular thought. This is a very quick explanation of how, partly in case you want to build this into your project and partly because we're considering whether we can facilitate this process for projects in general (for instance, by working closely with one project to do it for them and producing the tools and scripts that others would need to do forced alignment, as a side effect). Please note that the author is not a speech researcher or a linguist; she's just had lunch with a few, and not even done a proper literature review. That means that we don't guarantee everything in here is accurate, but that we are taking steps to understand this process and what we might be able to do about it. For better information, one possible source is Lei Chen, Yang Liu, Mary Harper, Eduardo Maia, and Susan McRoy, Evaluating Factors Impacting the Accuracy of Forced Alignments in a Multimodal Corpus, LREC 2004, Lisbon Portugal.

Commercial speech recognizers take an audio signal and give you their one best guess (or maybe n best guesses) of what the words are. Research speech recognizers can do this, but for each segment of speech, they can also provide a lattice of recognition hypotheses. A lattice is a special kind of graph where nodes are times and arcs (lines connecting two different times) are word hypotheses, meaning the word might have been said between the two times, with a given probability. The different complete things that might have been said can be found by tracing all the paths from the start time to the end time of the segment, putting the word hypotheses together. (The best hypothesis is then the one that has the highest overall probability, but that's not always the correct one.) If you have transcription for the speech that was produced by hand and can therefore be assumed to be correct, you can exploit the lattice to get word timings by finding the path through the lattice for which the words match what was transcribed by hand and transferring the start and end times for each word over to the transcribed data. This is what is meant by "forced alignment". *HTK*, one popular toolkit that researchers use to build their speech recognizers, comes with forced alignment as a standard feature, which means that if your recognizer uses it, you don't have to write a special purpose program to get the timings out of the lattice and onto your transcription. Of course, it's possible that other speech recognizers do this to and we just don't know about it.

The timings that are derived from forced alignment are not as accurate as those that can be obtained by timestamping from a waveform representation, but they are much, much cheaper. Chen et al. 2004 has some formal results about accuracy. Speech recognizers model what is said by recognizing phonemes and putting them together into words, so the inaccuracy comes from the kinds of things that happen to articulation at word boundaries. This means that, to hazard a guess, the accuracy isn't good enough for phoneticians, but it is good enough for researchers who are just trying to find out the timing relationship between words and events in other modalities (posture shifts, gestures, gaze, and so on). The timings for the onset and end of a speech segment are likely to be more accurate than the word boundaries in between.

The biggest problem in producing a forced alignment is obtaining a research speech recognizer that exposes the lattice of word hypotheses. The typical speech recognition researcher concentrates on accuracy in terms of word error rates (what percentage of words the system gets wrong in its best guess), since in the field as a whole, one can publish if and only if the word error rate is lower than in the last paper to be published. (This is why most people developing speech recognizers don't seem to have immediate answers to the question of how accurate the timings are.) Developing increasingly accurate recognizers takes effort, and once a group has put the effort in, they don't usually want to give their recognizer away. So if you want to used forced alignment, you have the following options:

- Persuade a speech group to help you. Lending the speech recognizer for your purposes doesn't harm commercial prospects or their research prospects in any way, but they might never have thought about that. This does require contact with a group that is either charitable or knows the benefits of negotation. Since speech groups are always wanting more data and since hand-transcription is expensive, one reasonable deal is that if they provide you with the timings for your research, they can use your data to improve their recognizer. This only works if your recordings are of high enough quality for their purposes and speech groups may have specific technical constraints. For instance, speech recognizers work better on data that is recorded using the same kind of microphones as the data the recognizer was trained on. This means that the best time to broker a deal is before you start recording. The easiest arrangement is usually for them to bung your data through the recognizer at their site and pass you the results rather than for you to install the recognizer.

- Build your own recognizer. One of the interesting things about forced alignment is that you don't actually need a good recognizer - you just need one that can get the correct words somewhere in the lattice of word hypotheses. Knowing the correct words also makes it possible to make it much more likely that the correct hypothesis will be in the lattice somewhere, since you can make sure that none of the words are outside of the speech recognizer's vocabulary. A quick poll of speech researchers results in the estimate that constructing a speech recognizer that works OK but won't win any awards using *HTK* takes 1-3 person-months. More time lowers the word error rate but isn't likely to affect the timing accuracy. The researchers involved found it difficult to think about how bad the recognizer could be and still work for these purposes, so they weren't sure whether spending less time was a possibility. It does take someone with a computational background to build a recognizer, although they didn't feel it took any particular skill or speech background to build a bad one.

- Find a speech recognizer floating around somewhere that's free and will work. There must be a project student somewhere who has put together a recognizer using HTK that is good enough for these purposes.

Finally, here are the steps in producing a forced alignment:

- Produce high quality speech recordings. You must have one microphone per participant, and they must be close-talking microphones (i.e., tabletop PZMs will not do - you need lapel or head-mounted microphones). If you are recording conversational speech (i.e., dialogue or small groups), it's essential that the signal on each participant's microphone be stronger when they're speaking than when other people are. Each participant must be recorded onto a separate channel.

- Optionally, find the areas of speech on each audio signal automatically. The energy on the signal will be higher when the person is speaking; you need to figure out some threshhold above which the person is speaking and write a script to mark those. This is often done in *MATLAB*.

- Hand-transcribe, either scanning each entire signal looking for speech or limiting yourself to the areas found to the automatic process. Turns (or utterances, depending on your terminology) don't have to be timestamped accurately, but can include extra silent space before or after them that will be corrected by the forced alignment. However, it's important that the padding not include cross-talk from another person that could confuse the recognizer.

- Optionally, add to the speech recognizer's dictionary all of the words in the hand-transcription that aren't in it already. (This is so that it can make a guess at what speech matches them even though it has never encountered the words before, rather than treating them as out-of-vocabulary, which means applying some kind of more general "garbage" model.)

- Run the speech recognizer in forced alignment mode and then a script to add the timings to the hand transcription.

8.2.4 Time-stamped coding

Although waveforms are necessary for timestamping speech events accurately, many other kinds of coding (gestures, posture, etc.) don't really require anything that isn't available in the current version of NXT, except possibly the ability

to advance a video frame by frame. People are starting to use NXT to do this kind of coding, and we expect to release some sample tools of this style plus a configurable video labelling tool fairly soon. However, there are many other ways of getting time-stamped coding; some of the video tools we encounter most often are *The Observer*, *EventEditor*, *Anvil*, and *TASX*. *EMU* is audio-only but contains extra features (such as format and pitch tracking) that are useful for speech research.

Time-stamped codings are so simple in format (even if they allow hierarchical decomposition of codes in "layers") that it doesn't really matter how they are stored for our purposes - all of them are easy to up-translate into NXT. In our experience it takes a programmer .5-1 day to set up scripts for the translation, assuming she understands the input and output formats.

8.3 Importing Data into NXT

NXT has been used with source data from many different tools. The import mechanisms used are becoming rather less ad-hoc, and this section has information about importing from some commonly-used tools. As transforms for particular formats are abstracted enough to be a useful starting point for general use, they will appear in this document, and also in the NXT distribution: see the `transforms` directory for details.

8.3.1 Transcriber and Channeltrans

Transcriber and *Channeltrans* have very similar file formats, *Channeltrans* being a multi-channel version of *Transcriber*.

See the `transforms/TRStoNXT` directory for the tools and information you will need. The basic transform is run by a perl script called `trs2nxt`. The perl script uses three stylesheets and an NXT program. Before running the transform, compile the `AddObservation` Java program using the standard NXT `CLASSPATH`. Full instructions are included, but the basic call to transform is:

```
trs2nxt  -c metadata_file -ob observationname -i in_dir -o out_dir -n nxt_dir
```

where you need to point to your local NXT directory using `-n`, and your local editable metadata copy using `-c`. The Java part of the process is useful as it checks validity of the transform and saves the XML in a readable format.

Note:

: there are many customizations you can make to this process using command line arguments, but if you have specific transcription conventions that you need to be converted to particular NXT elements or attributes, you will need to edit the script itself. The transcription conventions assumed are those in the AMI Transcription Guidelines.

8.3.2 EventEditor

EventEditor is a free Windows-only tool for direct time-stamping of events to signal.

See the `transforms/EventEditortoNXT` directory for the tools and information you will need. The basic transform is a Java program which needs to be compiled using the standard NXT `CLASSPATH` (comprising at least `nxt.jar` and `xalan.jar`). To transform one file, use

```
java EventEditorToNXT -i input_file -t tagname -a attname -s starttime
        -e endtime -c comment [ -l endtime ]
```

The arguments are the names of the elements and attributes to be output. Because *EventEditor* is event based, the last event does not have an end time. If you want an end time to appear in the NXT format, use the `-l` argument.

`ID`s are not added to elements, but you can use the provided `add-ids.xsl` stylesheet for that:

```
java -classpath $NXTDIR/lib/xalan.jar org.apache.xalan.xslt.Process
        -in outputfromabove -out outfile
        -xsl add-ids.xsl -param session observationname
        -param participant agentname
```

where *NXTDIR* is your local NXT install, or you can point to anywhere you happen to have installed *Xalan*. At least the session parameter, and really the participant one too, should be used as these help the `ID`s to be unique.

8.3.3 The Observer

The Observer is a commercial Windows-only tool for timestamping events against signal.

Output from *The Observer* is the textual `odf` format and this is transformed to NXT format using the `observer2nxt` perl script in the `transforms/ObserverToNXT` directory. It will be necessary to specify your own transform between *Observer* and NXT codes by editing the lookup tables in the perl script.

8.3.4 Other Formats

For data in different formats it's worth investigating how closely your transform might resemble one of those above: often it's a fairly simple case of tailoring an existing transform to your circumstances. If you manage this successfully, please contact the NXT developers: it will be worth passing on your experience to other NXT users. If your input format is significantly different to those listed, the NXT developers may still have experience that can be useful for your transform. We have also transformed data from *Anvil* and *ELAN* among others.

8.4 Exporting Data from NXT into Other Tools

'The NXT Query Language' p.34 is a good query language for this form of data, but it is necessarily slower and more memory-intensive than some others in use (particularly by syntacticians) because it does not restrict the use of left and right context in any way (in fact, it's possible to search across several observations using it). This isn't really as much of a problem for data analysts as they think - they can debug queries on a small amount of data and then run them overnight - but it is a problem for real-time applications. And sometimes users already know other query languages that they would prefer to use. This page considers how to convert NXT data for use in two existing search utilities, *tgrep2* and *TigerSearch*. Our recommended path to *tgrep2* is via Penn Treebank format, which can be useful as input to other utilities as well. Besides the speed improvements that come from limiting context, *tgrep2* has a number of structural operators that haven't been implemented in NQL, including immediate precedence and first and last child (although we expect to address this in 2006). We haven't gone through it looking at whether it has functionality that is difficult to duplicate in XPath; if it doesn't, then using XPath is likely to be the better option for those who already know it, but *tgrep2* already has a user community who like the elegance and compactness of the language. *TigerSearch* has a nice graphical interface and again supports structural operators missing in NQL.

Tgrep2 is for trees, and *TigerSearch*, for directed acyclic graphs with a single root. NXT represents a directed acyclic graph with multiple roots and additionally, some arbitrary graph structure thrown over the top that can have cycles. The biggest problem in conversion is determining what tree, or what single-rooted graph, to include in the conversion. This is a design problem, since it effectively means deciding what information to throw away. Every NXT corpus has its own design, so there is no completely generic solution - conversion utilities will require at least corpus-specific configurations.

8.4.1 TGREP2 via Penn Treebank Format

Penn Treebank format is simply labelled bracketing of orthography. For instance,

```
(NP (DET the) (N man))
```

Tgrep2 can load Penn Treebank format, but other tools use it as well. This means that it's reasonable to get to *tgrep2* via Penn Treebank format, since some of the work on conversion can be dual purposed.

Most users of Penn Treebank format treat the labels simply as labels. *Tgrep2* users tend to overload them with more information that they can get at using regular expressions. So, for instance, if one has markup for NPs that are subjects of sentences, one might mark that using `NP` for non-subjects and `NP-SUBJ` for subjects. The hyphen as separator is important to the success of regular expressions over the labels, especially where different parts of the labelling share substrings.

Some users of Penn Treebank format additionally overload the labels with information about out-of-tree links that can't be used in *tgrep2*, but that they have other ways of dealing with. For instance, suppose they wish to mark a coreferential link between "the man" and "he". One way of doing this is using a unique reference number for link:

```
(NP/ANTEC1 (DET the) (N man)) ... (PRO/REF1 he)
```

We recommend dividing conversion into two steps: (1) deriving a single XML file that represents the tree you want from the NXT format data, where the XML file structure mirrors the tree structure for the target treebank and any out of tree links are representing using `id`s and `idref`s; and (2) transforming that tree into the Penn Treebank format.

(1) is specific to a given corpus and set of search requirements. For some users, it will be one coding file from the original data, or the result of one knit operation, in which case it's easy. It might also be a simple transformation of a saved query result. Or it might be derived by writing a Java program that constructs it using the data model API. Once you know what tree you want, the search page will give hints about how to get it from the NXT data.

(2) could be implemented as a generic utility that reads a configuration file explaining how to pack the single-file XML structure into a Penn Treebank labelling and performs it on a given XML file. Assume that each label consists of a basic label (the first part, before any separator, usually the most critical type information), optionally followed by some rendering of attribute-value pairs, optionally followed by some rendering of out-of-tree links. The configuration file would designate separators between different kinds of information in the Treebank label and where to find the roots of trees for the treebank. (The latter is unnecessary, since anything else could be culled from the tree in step 1, but it makes it more likely that a single coding file from the NXT data format will be a usable tree for input to step 2.) For each XML tag name, it would also designate how to find the basic label (the first part, before any separator), which attribute-value pairs and links to include, and how they should be printed.

Below is one possible design for the configuration file. Note that the configuration uses XPath fragments to specify where to find roots for the treebank and descendants for inclusion. Our assumption is that those who don't know XPath can at least copy from examples, and those who do can get more flexibility from this approach.

```
<NXT-to-tgrep-config>
   <!-- specify where the treebank roots are. We will tree-walk
        the XML from these nodes, printing as we go -->
   <treebank-roots match="//foo"/>
   <!-- what to use as brackets -->
   <left-bracket value="("/>
   <right-bracket value=")"/>
   <!-- string with a separator to use between base label and atts;
     if none give, none used -->
   <base-label-sep value="-"/>
   <!-- string with a separator to use between att name and value -->
   <att-value-sep value=":"/>
   <!-- string with a separator to use between different atts -->
   <att-sep value="*"/>
   <!-- string with a separator to use between attributes and links -->
   <link-sep value="/"/>
   <!-- don't bother printing attribute names or the separator between
        the names and the values -->
   <omit-attribute-names/>
   <!-- if a node matches the expression given, skip it, moving
        on to its children -->
   <omit match="baz"/>
   <!-- transformation instructions for nodes matching the expression given -->
   <transform match="nt">
       <!-- the base-label comes first in the label, again an XPath
       fragment.  name() for tag name, @cat for value of cat attribute -->
       <base-label value="name()"/>
       <!-- where to find the orthography, if any (usually the textual content,
       sometimes a particular attribute) -->
       <orthography value="text()"/>
       <!-- leave out the start attribute -->
       <omit-attribute name="start"/>
       <!-- we assume all other attributes are printed in a standard
       format with the name, att-value-sep, and then the attribute-value.
       If we need individual control for how attributes are printed,
       we'll need to allow configuration of that here.
       -->
   </transform>
   <!-- how to print out-of-tree links represented by id/idref in
```

```
    the input.  This example says expect foo tags
    to be linked to bar tags where the refatt attribute has the same
    value as the foo's idatt attribute.  For the foo label, add the
    link separator followed by ANTEC followed by the value of idatt,
    and for the bar label, add the link separator followed by REF
    followed by the value of refatt (which is the same value).  -->
   <link>
       <antecedent match="foo"/>
       <antecedent-id name="@idatt"/>
       <referent match="bar"/>
       <referent-idref name="@refatt"/>
       <link-anteclabel value="ANTEC"/>
       <link-reflabel value="REF"/>
   </link>
</NXT-to-tgrep-config>
```

A few example tags that can be generated from this configuration from

```
<nt cat="NP" subcat="SUBJ" id="1"/>
```

where this serves as an antecedent in a link:

- `NP`
- `NP-subcat:SUBJ/ANTEC1`
- `NP-subcat:SUBJ/1`
- `nt-cat:NP*subcat:SUBJ/ANTEC1`
- `nt-NP:SUBJ`
- `nt-NP`

and so on.

The utility should have defaults for everything so that it does something when there is no configuration file, choosing standard separators, not omitting any tags or attributes, printing attribute names, and failing to print any out-of-tree links. It also should not require a DTD for the input data. One thing to note: this design assumes we print separate `id`s for every link, but some nodes could end up linked in two ways, to two different things, causing labels like `FOO-BARANTEC1-BAZANTEC1`. This is the more general solution, but if users always have the same `id` attribute for both types of links, we can make the representation more compact.

We have attracted funding to write this utility, with the work to be scheduled sometime in the period Oct 05 to Oct 06, and so we are consulting on this design to see whether it is flexible enough, complete, too complicated for the target users, and actually in demand. Note that a converter like this couldn't guarantee safety of queries given that the Penn Treebank labels get manipulated using regular expressions - the user could easily get matches on the wrong part of the label by mistake because these regular expressions are hard to write to preclude this, unless you devise your attribute values and tag names carefully so that no pair of things matches an obvious reg exp you might want to search on. The users who have requested this work expect to get around this problem by running conversion from NXT format into several different tgreppable formats for different queries that omit the information that isn't needed.

Our biggest concern with the utility is how implementation choices could affect usability for this user community. It tends to be the less computational end of the *tgrep* user community who most want *tgrep* conversion, with speed and familiarity as the biggest issues. (Familiarity doesn't really seem to be an issue for the more computational users, and speed

is slightly less of an issue since they're more comfortable with scripting and batch processing, but it's still enough of a problem for some queries that they want conversion. This may change when we complete work on a new NXT search implementation that evaluates queries by translating them to XQuery first, but that's a bigger task.) Serving the needs of less computational users introduces some problems, though. The first one is that since they know nothing about XML, and are used to thinking about trees but not more complex data models, they won't be able to write the configuration file for the utility. The second is that it may be difficult to find an implementation for the converter that runs fast, is easy to install, and doesn't require us to make executables for a wide range of platforms. (We think it needs to run fast because the users expect to create several different *tgrep*pable forms of the same data, but if they have to get someone else to do it because it requires skills they don't have to write the configuration file, this is no longer important - the real delay will be in getting someone's time.)

We're still wrestling with this design; comments about our assessment of what's required and acceptable solutions welcome. The implementations we're considering are (a) generating a stylesheet from the configuration file and applying that to the data or (b) direct implementation reading the data and configuration file at the same time, in either perl with xml parsing and xpath modules, Java with Apache libraries, or LT-XML2.

8.4.2 TigerSearch

We have put less thought into conversion into *TigerSearch*, but that doesn't mean the conversion is less useful. The fact that *TigerSearch* supports a more general data structure than trees means that it will be more useful for some people. NXT uses the XML structure to represent major trees from the data, but *Tiger*'s XML is based on graph structure, with XML tags like `nt` (non-terminal node), `t` (terminal node), and `edge`. On the other hand, since *Tiger* can represent not just trees but directed acyclic graphs with a single root, it would be more reasonable to specify a converter, again using a configuration file, in one step from NXT format. The configuration file would need to specify what to use as roots, where to find the orthography, a convention for labelling edges, and which links to omit to avoid cycles, but otherwise it could just preserve the attribute-value structure of the original. The best implementation is probably in Java using the NXT data model API to walk a loaded data set.

8.5 Knitting and Unknitting NXT Data Files

By "`knit`ting", we mean the process of creating a larger tree than that in an individual coding or corpus resource by traversing over child or pointer links and including what is found. Knitting an XML document from an NXT data set performs a depth-first left-to-right traversal of the nodes in a virtual document made up by including not just the XML children of a node but also the out-of-document children links (usually pointed to using `nite:child` and `nite:pointer`, respectively, although the namimg of these elements is configurable). In the data model, tracing children is guaranteed not to introduce cycles, so the traversal recurses on them; however, following links could introduce cycles, so the traversal is truncated after the immediate node pointed to has been included in the result tree. For pointers, we also insert a node in the tree between the source and target of the link that indicates that the subtree derives from a link and shows the role. The result is one tree that starts at one of the XML documents from the data set, cutting across the other documents in the same way as the `^` operator of the query language, and including residual information about the pointer traces. At May 2004, we are considering separating the child and pointer tracing into two different steps that can be pipelined together, for better flexibility, and changing the syntax of the element between sources and targets of links.

Unknitting is the opposite process, involving splitting up a large tree into smaller parts with stand-off links between them.

Knitting NXT data can create standard XML files from stand-off XML files. This can be essential for downstream processing that is XML aware but does not deal with stand-off markup. 'Data Storage' p.14 describes NXT's stand-off annotation format.

There are two distinct approaches for kitting data: using an XSLT stylesheet, or using the LT XML2 toolkit.

8.5.1 Knit using Stylesheet

To resolve the children and pointers from any NXT file there is a stylesheet in NXT's `lib` directory called `knit.xsl`. Stylesheet processor installations vary locally. Some people use *Xalan*, which happens to be redistributed with NXT. It can be used to run a stylesheet on an XML file as follows.

```
java org.apache.xalan.xslt.Process -in INFILE -xsl lib/knit.xsl -param idatt id
   -param childel child -param pointerel pointer -param linkstyle ltxml
   -param docbase file:///my/file/directory 2> errlog > OUTFILE
```

The docbase parameter indicates the directory of the INFILE, used to resolve the relative paths in child and pointer links. If not specified, it will default to the location of the stylesheet (NOT the input file!). Note that if you're using the absolute

location of the INFILE, it is perfectly fine to just set docbase to the same thing, because the entity resolver will take its base URL (according to xslt standard) for document function calls.

Note:

This means you may have to move XML files around so that all referred-to files are in the same directory.

The default `linkstyle` is `="LT XML"`, the default `id` attribute is `="nite:id"`, the default indication of an out-of-file child is `nite:child`, and the default indication of an out-of-file pointer is `nite:pointer`. These can be overridden using the parameters `linkstyle`, `idatt`, `childel`, and `pointerel`, respectively, and so for example if the corpus is not namespaced and uses xpointer links,

```
java org.apache.xalan.xslt.Process -in INFILE -xsl STYLESHEET
        -param linkstyle xpointer -param idatt id
        -param childel child -param pointerel pointer
```

A minor variant of this approach is to edit `knit.xsl` so that it constructs a a tree that is drawn from a path that could be knitted, and/or document calls to pull in off-tree items. The less the desired output matches a knitted tree and especially the more outside material it pulls in, the harder this is. Also, if a subset of the knitted tree is what's required, it's often easier to obtain it by post-processing the output of `knit`.

8.5.2 Knit using LT XML2

`Knit.xsl` can be very slow. It follows both child links and pointer links, but conceptually, these operations could be separate. We have implemented separate "knits" for child and pointer links as command line utilities with a fast implementation in LT XML2: `lxinclude` (for children) and `lxnitepointer` (for pointers).

`lxinclude -t nite` *FILENAME* reads from the named file (which is really a URL) or from standard input, writes to standard output, and knits child links. (The `-t nite` is required because this is a fuller XInclude implementation; this parameterizes for NXT links). If you haven't used the default `nite:child` links, you can pass the name of the tag you used with `-l`, using `-xmlns` to declare any required namespacing for the link name:

```
lxinclude -xmlns:n=http://example.org -t nite -l n:mychild
```

This can be useful for recursive tracing of pointer links if you happen to know that they do not loop. Technically, the `-l` argument is a query to allow for constructions such as `-l '*[@ischild="true"]'`.

Similarly,

```
lxnitepointer FILENAME
```

will trace pointer links, inserting summary traces of the linked elements.

8.5.2.1 Using stylesheet extension functions

As a footnote, LT XML2 contains a stylesheet processor called `lxtn`, and we're experimenting with implementing extension functions that resolve child and pointer links with less pain than the mechanism given in `knit.xsl`; this is very much simpler syntactically and also faster, although not as fast as the LT XML2 based implementation of `knit`. This approach could be useful for building tailored trees and is certainly simpler than writing stylesheets without the extension functions. Edinburgh users can try it as

8.5.3 Unknit using LT XML2

Again based on LT XML2 we have developed a command line utility that can `unknit` a knitted file back into the original component parts.

```
lxniteunknit -m METADATA FILE
```

`Lxniteunknit` does not include a command line option for identifying the tags used for child and pointer links because it reads this information from the metadata file.

8.6 General Approaches to Processing NXT Data

Suppose that you have data in NXT format, and you need to make some other format for part or all of it - a tailored HTML display, say, or input to some external process such as a machine learning algorithm or a statistical package. There are an endless number of ways in which such tasks can be done, and it isn't always clear what the best mechanism is for any particular application (not least because it can depend on personal preference). Here we walk you through some of the ones we use.

The hardest case for data processing is where the external process isn't the end of the matter, but creates some data that must then be re-imported into NXT. (Think, for instance, of the task of part-of-speech tagging or chunking an existing corpus of transcribed speech.) In the discussion below, we include comments about this last step of re-importation, but it isn't required for most data processing applications.

8.6.1 Option 1: Write an NXT-based application

Often the best option is to write a Java program that loads the data into a NOM and use the NOM API to navigate it, writing output as you go. For this, the iterators in the NOM API are useful; there are ones, for instance, that run over all elements with a given name or over individual codings. It's also possible from within an application to evaluate a query on the loaded NOM and iterate over the results within the full NOM, not just the tree that saving XML from the query language exposes. (Many of the applications in the sample directory both load and iterate over query results, so it can be useful to borrow code from them.) For re-importation, we don't have much experience of making Java communicate with programs written in other languages (such as the streaming of data back and forth that might be required to add, say, part-of-speech tags) but we know this is possible and that users have, for instance, made NXT-based applications communicate with processes running in C (but for other purposes).

This option is most attractive:

- for those who write applications anyway (since they know the NOM API)
- for applications where drawing the data required into one tree (the first step for the other processing mechanisms) means writing a query that happens to be slow or difficult to write, but NOM navigation can be done easily with a simpler query or no query at all
- for applications where the output requires something which is hard to express in the query language (like immediate precedence) or not supported in query (like arithmetic)

8.6.2 Option 2: Make a tree, process it, and (for re-importation) put it back

Since XML processing is oriented around trees, constructing a tree that contains the data to be processed, in XML format, opens up the data set to all of the usual XML processing possibilities.

8.6.2.1 First step: make a tree

Individual NXT codings and corpus resources are, of course, tree structures that conveniently already come in XML files. Often these files are exactly what you need for processing anyway, since they gather together like information into one file. Additionally, you can use the knitting and knit-like tree construction approaches described in 'Knitting and Unknitting NXT Data Files' p.76.

As an alternative to knitting data into trees, if you evaluate a query and save the query results as XML, you will get a tree structure of matchlists and matches with `nite:pointers` at the leaves that point to data elements. Sometimes this is the best way to get the tree-structured cut of the data you want, since it makes many data arrangements possible that don't match the corpus design and therefore cannot be obtained by knitting.

The query engine API includes (and the search GUI exposes) an option for exporting query results not just to XML but to *Excel* format. We recommend caution in exercising this option, especially where further processing is required. For simple queries with one variable, the *Excel* data is straightforward to interpret, with one line per variable match. For simple queries with n variables, each match takes up n spreadsheet rows, and there is no way of finding the boundaries between *n*-tuples except by keeping track (for instance, using modular arithmetic). This isn't so much of a problem for human readability, but it does make machine parsing more difficult. For complex queries, in which the results from one query are passed through another, the leaves of the result tree and presented in left-to-right depth-first order of traversal, and even

human readability can be difficult. Again, it is possible to keep track whilst parsing, but between that and the difficulty of working with *Excel* data in the first place, its often best to stick to XML.

8.6.2.2 Second step: process the tree

Stylesheets

This is the most standard XML transduction mechanism. There are some stylesheets in the `lib` directory that could be useful as is, or as models; `knit.xsl` itself, and `attribute-extractor.xsl`, that can be used in conjunction with `SaveQueryResults` and `knit` to extract a flat list of attribute values for some matched query variable (available from Sourceforge CVS from 2 July 04, will be included in NXT-1.2.10).

This option is most attractive:

- for those who write stylesheets anyway (since they know XSLT)
- for operations that can primarily be carried out on one coding at a time, or on knitted trees, or on query language result trees, limiting the number and complexity of the document calls required
- for applications where the output requires something which is not supported in query but is supported in XSLT (like arithmetic)

Xmlperl

Xmlperl gives a way of writing pattern-matching rules on XML input but with access to general perl processing in the action part of the rule templates.

This option is most attractive:

- for those who write xmlperl or at least perl anyway
- for operations that can be carried out on one coding at a time, or on knitted trees, or on query language result trees
- for applications where the output requires something which is not supported in query (like arithmetic)
- for applications where XSLT's variables provide insufficient state information
- for applications where bi-directional communication with an external process is needed (for instance, to add part-of-speech tags to the XML file), since this is easiest to set up in xmlperl

Xmlperl is quite old now. There are many XML modules for perl that could be useful but we have little experience of them.

In the LT XML2 release, see also `lxviewport`, which is another mechanism for communication with external processes.

ApplyXPath/Sggrep

There are some simple utilities that apply a query to XML data and return the matches, like *ApplyXPath* (an Apache sample) and `sggrep` (part of LT XML2). Where the output required is very simple, these will often suffice.

Using lxreplace

This is another transduction utility available that is distributed more widely with LT XML2. It is implemented over LT XML2's stylesheet processor, but the same functionality could be implemented over some other processor.

```
lxreplace -q query -t template
```

template is an XSLT template body, which is instantiated to replace the nodes that match *query*. The stylesheet has some pre-defined entities to make the common cases easy:

- `&this;` expands to a copy of the matching element (including its attributes and children)
- `&attrs;` expands to a copy of the attributes of the matching element
- `&children;` expands to a copy of the children of the matching element

Examples:

To wrap all elements `foo` whose attribute `bar` is `="unknown"` in an element called `bogus`:

```
lxreplace -q 'foo[@bar="unknown"]' -t '&this;'
```

(that is, replace each matching `foo` element with a `bar` element containing a copy of the original `foo` element).

To rename all `foo` elements to `bar` while retaining their attributes:

```
lxreplace -q 'foo' -t '&attrs;&children;'
```

(that is, replace each `foo` element with a `bar` attribute, copying the attributes and children of the original `foo` element).

To move the (text) content of all `foo` elements into an attribute called `value` (assuming that the `foo`s don't have any other attributes):

```
lxreplace -q 'foo' -t ''
```

(that is, replace each `foo` element with a `foo` element whose `value` attribute is the text value of the original `foo` element).

8.6.2.3 Third step: add the changed tree back in

Again based on LT XML2 we have developed a command line utility that can `unknit` a knitted file back into the original component parts.

```
lxniteunknit -m METADATA FILE
```

`Lxniteunknit` does not include a command line option for identifying the tags used for child and pointer links because it reads this information from the metadata file. With `lxniteunknit`, one possible strategy for adding information to a corpus is to knit a view with the needed data, add information straight in the knitted file as new attributes or a new layer of tags, change the metadata to match the new structure, and then unknit.

Another popular option is to keep track of the data edits by `id` of the affected element and splice them into the original coding file using a simple perl script.

8.6.3 Option 3: Process using other XML-aware software

NXT files can be processed with any XML aware software, though the semantics of the standoff links between files will not be respected. Most languages have their own XML libraries: under the hood, NXT uses the Apache XML Java libraries. We sometimes use the XML::XPath module for perl, particularly on our import scripts where XSLT would be inefficient or difficult to write.

8.7 Manipulating media files

A wide variety of media tools can be used to create signals for NXT and to manipulate them for use with other tools. Here we mention a few that we use regularly. The cross-platform tool *mencoder* is good for encoding video for use with NXT; *VirtualDub* in conjunction with *AviSynth* (Windows only) are useful for accurately chopping up video files for use in other programs. Using the latter approach is better if you need frame-accurate edits, *mencoder* is only accurate to the nearest keyframe.

As an example, we are sometimes asked to provide video extracts that show certain NXT phenomena. The first task is to find the phenomena using an NXT tool like FunctionQuery. This results in a tab-delimited result set each line of which will identify the video file to use, along with the start and end time of the segment. Using a scripting language like *perl* it's easy to transform this into a set of *AviSynth* format files. These files simply describe a set of media actions to take like loading a video file and adding an audio soundtrack, then chopping out the appropriate section (these would use the *AviSynth* functions `AVISource`, `WAVSource`, `AudioDub` and `Trim`). These files can then be loaded into *VirtualDub* which treats them like any other video file, and the result saved as an AVI file with whatever video / audio compression you choose. The useful thing about *VirtualDub* is that these actions can be applied to a batch of files and left to run with no further user-action.

A. FAQ

End user and developer questions for NXT still tend to be dealt with by private email, although we do realize that we should move over to using public forums for this. When we receive a question more than once, we try to make time to change the web pages to make the answer clear in the correct location. This page is for frequently asked questions that haven't yet found a proper home, plus their answers.

Namespacing

- **Q:** Exactly what does `xmlns:nite="http://nite.sourceforge.net/"` do in the xml files? Is it necessary?
- **A:** It declares the nite namespace. If you use it in your data, then you have to include this attribute on the root element of the data files that include elements and attributes from this namespace. In NXT format data, users typically namespace the reserved attributes and element names to avoid naming conflicts (e.g., attributes for ids, start and end times, and elements for document roots, out-of-file children, and pointers).
- **Q:** Can I use namespacing in my data set?
- **A:** In theory namespacing is a good idea, but there is a bug in NXT's query language parser that means it can't handle namespaced element names and attributes. For this reason, you should avoid namespacing, with the possible exception of XML document roots (which aren't available to query anyway) and the reserved attributes that have their own special meaning to NXT and dedicated query language syntax (the id, available as `ID($x)`, the start time, available as `START($x)`, and the end time, available as `END($x)`).

Fonts and Font Sizes

- **Q:** How do I change the font in an NXT GUI?
- **A:** You can do whatever you want in a customized tool. The standard and configurable NXT GUIs don't specify a font, so what you get depends on your java installation. Getting different fonts for different parts of the displayed data requires you to write customized tools or to contribute code to the project that allows the user to specify in the configuration file what font to use for a particular element, attribute, or element's textual content.
- **Q:** How do I change the font size in an NXT GUI?
- **A:** You can do whatever you want in a customized tool. The standard and configurable NXT GUIs have a font size (usually 12 point) wired in, with the exception (at September 2006) of the `GenericDisplay`, which allows a font size to be passed in at the command line. The simplest change would be to recompile other GUIs with the font size you want, although it would be better to contribute code that allows users to specify the font size in the configuration file. Some previous customized tools have allowed the end user to change the font size for a display from a menu. If you wish to revive this code for general use, contact us. The main NXT GUI (`net.sourceforge.nite.nxt.GUI`) that allows the user to choose among the registered programs for a data set (those mentioned in the metadata under `<callable-programs/>`) automatically adds a `GenericDisplay` to the list. This automatic addition uses the default font size (12 point). If you want a menu entry for a different font size, you need to register the generic display with the font size you require. The declaration to do this is, e.g.:

```
<callable-programs>
   <callable-program description="20 point GenericDisplay" name="net.sourceforge.nite.gui.util
       <required-argument name="corpus" type="corpus"/>
       <required-argument name="observation" type="observation"/>
       <required-argument name="fontsize" default="20"/>
   </callable-program>
</callable-programs>
```

To pop up a window asking the user to enter the fontsize they require, use:

```
<callable-programs>
   <callable-program description="20 point GenericDisplay" name="net.sourceforge.nite.gui.util
       <required-argument name="corpus" type="corpus"/>
       <required-argument name="observation" type="observation"/>
       <required-argument name="fontsize" default="20"/>
   </callable-program>
</callable-programs>
```

GUIs

- **Q:** Why is the `GenericDisplay` unusable? / Why does the `GenericDisplay` run out of memory?
- **A:** The `GenericDisplay` is designed to throw up windows corresponding to every XML tree in the data set for the observation chosen. If your data set has many different annotations, this will be too many windows for the user to handle, and if it's really big, you many not even be able to load them all at once. You can cut it down using the query argument to specify the kinds of things you actually want to see in the display. The `GenericDisplay` is designed to be something that will work, badly, for any NXT format data set - for actual work you will almost certainly want to set up one of the configurable interfaces or write your own customized display.

Data Model

- **Q:** Are filenames case sensitive?
- **A:** Yes.
- **Q:** Can I use the same element name in two different layers?
- **A:** No. NXT needs each element to belong to exactly one layer because otherwise it doesn't know how to serialize the data set, or what files to load when it requires elements of a specific type.
- **Q:** Can I use the same attribute name for two different elements?
- **A:** Yes.
- **Q:** What kinds of properties can elements inherit from their children?
- **A:** Only timing information using the reserved start and end time attributes, and this only if time inheritance is enabled for the element type involved.
- **Q:** What are ids for, and what constraints are there on the values for ids?
- **A:** An id can be any string that's globally unique. If you are importing data and don't have ids on it yet, you can get NXT to generate ids for you by loading the data and then saving it. Ids are used to manage the relationship between display elements in a GUI and the underlying data, and for specifying out-of-file child and pointer links.
- **Q:** Can elements in two structural layers point to each other?
- **A:** Yes. In general, any element can point to any other element, as long as all the elements from a given layer point to elements from the same layer, and this relationship is declared in the metadata. Pointers do not have to be in featural layers; the featural layer is just useful conceptually for the kind of layer that only relates to the rest of the data set via pointers.

Data Set Design

- **Q:** What if I want elements from one layer to be able to draw children from either some layer or the the layer that layer draws children from, skipping straight to what is usually a grandchild?
- **A:** This violates the NXT data model. Suppose the `phrase-layer` contains the element `phrase`, which draws children from the `subphrase-layer`, which contains the element `subphrase`, which draws children from the `word-layer`, which contains the element `word`. There are two standard ways to encode the relationship you want:

  - Wrap non-subphrase runs of `word` elements in some new tag, say, `nosubphrase`, and use these as the children for phrases, so that you get get strict decomposition in the layers. Then the data conforms completely, but users who are used to distance limited operators like `^1` will need to know that the intermediate `nosubphrase` tag is there in the structure.

  - Serialize `phrase` and `subphrase` elements in the same file, and declare them as two tags within the same recursive layer. Then either can contain `word`s, but also either can contain each other This has the disadvantage that the data model design is declared to be less restrictive than it should be for the data set, so data validation wouldn't catch `subphrase` elements that contain `phrase` elements, for instance.

  - Declare `phrase-layer` to draw children from `subphrase-layer`, have `phrase` elements point to `word`s directly whenever you want, and either store all three layers in the same file or never use code that lazy loads.

  The first one is what was designed in as the preferred solution; the others are what data sets usually do. The third one may not be robust against future NXT development.

- **Q:** When should I use pointers and when should I use children?
- **A:** Use children whenever this is acceptable in the data model (i.e., when it doesn't create loops or require an element to have multiple, conflicting sets of children), turning off the temporal inheritance if you need to - it's much easier to query elements related by hierarchy than by pointer.
- **Q:** How much data should I put in one XML file?
- **A:** Divide your data into files by thinking about typical uses of the data. If one layer draws children from another, and the two layers always get used together (both within NXT and in external processing), then you can save some loading overhead by putting them in the same file. If, however, users may want one without the other, separate them into two files so that lazy loading can minize the data set size in working memory. If you have an element with many attributes, most of which are rarely used, consider putting the information conveyed by the attributes in one or more files containing elements that use the old, reduced elements as children, or that point to them. This makes querying the rarely used information more cumbersome, but saves overhead in the more common uses.
- **Q:** Should I represent my orthography in textual content, or use an attribute?
- **A:** The original NXT developers were split between some who wanted to preserve the TEI-ish notion that the textual content is the base text and some who didn't want any privileged textual content at all. Both designs have strengths for different kinds of data sets, so it depends. Most current data sets seem to use textual content. For NXT, textual content has the following special properties:

  - In query, you can get at it using e.g. `TEXT($w)` Some users find this more intuitive than having to remember a specific attribute name.

  - Some of the libraries for building GUIs based on text or transcription expect textual content, and so e.g. coding tools and transcription-based displays (which you haven't been using so far) can require less setup if the data is laid out this way - but adding a delegate function that displays based on an attribute isn't hard.

- Some command line utilities, like `SortedOutput`, treat an element as having textual content equal to the whitespace-delimited concatenation of its children in order. This can make it easier to extract some kinds of tables out of an NXT data set (for instance, a list of phrases by syntactic type) It's possible to get the text out in such tables if it is in attributes on words lower down in the hierarchy using `FunctionQuery` with the extract function, but cumbersome.

- In future, it's possible that the query language will always treat an element as having textual content equal to the whitespace-delimited concatenation of its children in order. This was part of our original design and we have recently had someone complain that NXT doesn't do this, but we haven't made a decision about whether to make this extension or committed resource to it yet. If we do this work we could consider adding a reserved attribute for orthography so that we can treat it equivalently to textual content and suit both choices.

There are cases where using textual content is less elegant, as, for instance, in parallel corpora, where there are two rival versions of the orthography of equal importance.

- **Q:** What's special about ontologies? Can I search for the "top-level" code and get all the child codes? How is it reflected in the underlying data structure?
- **A:** Ontologies are a way of providing type or attribute value information that isn't just a string, but where the types or values fit into a hierarchical structure in their own right. Suppose your ontology contains:

```
[ontol.xml]
<foo id="id0" name="animal">
    <foo id="id1" name="bird">
       <foo id="id2" name="sparrow"/>
       <foo id="id3" name="chickadee"/>
    </foo>
    <foo id="id4" name="dog">
       <foo id="id5" name="mutt"/>
    </foo>
</foo>
```

Your elements can point into the ontology:

```
<el>
    <nite:pointer href="ontol.xml#id3"/>
</el>
```

to get type information. You can test for chickadees:

```
($a el)($b foo):($a > $b) &&  ($b@name="chickadee")
```

but you can also test for birds in general:

```
($a el)($b foo):($a > $b)::($c foo):($c@name="bird") && ($c ^ $b)
```

Elements in ontologies have searchable relationships just like everything else. In another sense, ontologies aren't at all special, because you could encode the same information as a corpus-resource and still be able to access the information from the query language. Using an ontology is more restrictive because it assume one tag name throughout the hierarchy.

Query Language

- **Q:** Is there a "not dominates" operator, like `!^`?
- **A:** Use e.g. `!($a ^ $b)`.

Performance

- **Q:** What are the memory limits to NXT in loading data?
- **A:** The in-memory data representation uses around 7 times the disk storage space for the same data, or a bit less. If lazy loading is on, only the files that are actually needed are loaded.

B. How To Use Metadata

The main investment involved in allowing your own data to be used by the NITE XML toolkit is the production of a metadata file and the provision of your data in a conformant fashion (especially as regards file-naming). Understanding the format of metadata files will be important if you wish to import your data, though we provide several example metadata files to help. Once you have a metadata file that describes your data, you will be able to use all the NITE tools to validate, analyse and edit your data.

B.1 What metadata files do

Metadata files describe all aspects of a corpus including:

- where on disk the parts of the corpus reside;
- the codings that can validly be made on the data;
- the observations that have been made already along with their status;
- the NITE editors and viewers that can be used on the corpus;
- much more (see below).

B.2 What metadata files look like

Metadata files are XML and conform to a DTD. There is one metadata DTD for simple (single file) corpora and one for standoff corpora. They both share much in common, so import the same basic DTD. The set of DTDs (zipped) can be downloaded here. If you are more familiar with XML Schema and have a schema validator installed you may prefer this set of zipped schemas.

B.3 Metadata examples

Save these to disk and have a look at them in your favourite XML or text editor.

- Metadata for NITE's simple example (you may also want to see the data it describes - 5K zip)
- Metadata for the Maptask corpus (here is a single maptask observation - 165K zip)
- Metadata for the Smartkom corpus (simple corpus case) (here's a single Smartkom interaction file - 15K XML)

B.4 Using Metadata to validate data

Since metadata describes the format of the data and where to find it on disk, it is used by the NITE software to validate the data as it is loaded and edited. This sort of direct validation is useful, but we also provide schema validation of data using a schema derived automatically from the metadata (via a stylesheet).

Assuming you have already " p.6, you already have the schema-generating stylesheet (it's in the `lib` directory). Armed with this and a stylesheet processor (xalan is also in the NOM distribution), you can run this command on your metadata file:

`java org.apache.xalan.xslt.Process -in <your-metadata> -xsl generate-schema.xsl -out extension.xsd`

If you have a schema validator (I use *xsv*) you are now ready to validate some data files. Try putting these declarations:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="extension.xsd"
```

in the root element of your data file and then execute:

```
xsv <your-file>
```

One of the major reasons behind this approach to schema validation is that we can validate data that is either a single file "as-serialized" by NITE, or files that have been transformed to replace their `nite:child` elements with the pointed-to elements recursively, and also replacing pointers with their actual elements. This is useful for validating the types of elements that can be children of a specific element and pointed to by that element. In this way an entire corpus could be schema validated. You have a stylesheet that does this transformation in the `lib` directory of your NOM distribution.

If this all seems rather involved, and your data already loads into the NOM, the program `PrepareSchemaValidation.java` will make a new directory for you which is fully ready for schema validation.

Validation limitations

- all stream elements *must* be named `nite:root`;
- all ID, Start and End time attributes must use the NITE default names: `nite:id`, `nite:start` and `nite:end`.
- all children and pointers *must* use XLink / XPointer style links.

- stream elements will be permitted to contain inadvisably mixed elements (so long as all those elements are valid and defined themselves)

C. Comparison to other efforts

Several other tools and frameworks exist that offer some functionality which overlaps with that of NXT. This section describes these other tools and how they relate to NXT.

Note:

Any errors in the descriptions below are our own. These descriptions are based on published information at March 2004.

C.1 Annotation Graph Toolkit (AGTK)

The *Annotation Graph Toolkit*, or *AGTK*, employs a data model, the annotation graph, which is a directed acyclic graph where edges are labeled with feature-value pairs and nodes can be labeled with time offsets. Structural relationships can be expressed by convention in the edge labelling, but they are not exposed directly in the API as they are in NXT; instead, the focus is on the efficient handling of temporal information. AGTK is written in C++ and comes with a Java port. A query language is planned for AGTK but is not yet available. Although AGTK does not provide direct support for writing graphical user interfaces, it does include wrappers for Tcl/Tk and Python, two scripting languages in which writing such interfaces is easier than in C++ itself. The developers expect interfaces to call upon WaveSurfer, a compatible package, to display waveforms and play audio files.

C.2 ATLAS

ATLAS is intended to generalize the annotation graph and differs in two main ways. First, it allows richer relationships between annotation and signal. In annotation graphs, the only relationship between annotation and signal that is supported in the data handling is the timespan on the signal to which the annotation refers, given as a start and end time. NXT is similar to *AGTK* in this regard. *ATLAS*, however, defines more generic signal regions which can refer to other properties besides the timing. For example, on a video signal, a region could pinpoint a screen location using X and Y coordinates. Second, *ATLAS* explicitly represents structural relationships by allowing annotations to name a set of "children", without constraining how many "parents" an annotation may have. The framework for defining the semantics of this relationship and for specifying which types of annotations expect which other types as children, MAIA, is still under development. It has the potential to be very flexible, especially if the semantics of the parent-child relationship can vary depending on the types of data objects that they link. The *ATLAS* data model is implemented in Java, and the developers plan both a query language and direct support for writing graphical user interfaces.

C.3 MMAX

MMAX2 is primarily used for annotation of text, but it has the facility to play some kinds of audio signal in synchrony with its data display. Timing information is represented in the stylesheet that *MMAX2* uses to specify a data display format declaratively and not in the data itself. *MMAX2*'s data model is rather simpler than NXT's, but it allows one to specify different types of annotation all of which point independently to the base documents, and links between annotations. *MMAX2* also has a query language based on the idea of intersections between paths. *MMAX2* is easier to set up than NXT but in general NXT is more useful, the more one's work relies on crossing hierarchies, complex structural relationships, or timing information.

C.4 *EMU*

EMU also shares some properties with NXT, in that it allows time-aligned labelling of speech data including hierarchical decomposition across different tiers of labels and specifically supports query of the label sets. (This differentiates *EMU* from tools such as *Anvil* and *TASX* that are just coding tools without more general support, although given the availability of XML query languages to deal with their data formats, it's not clear that this really makes a difference.)

C.5 Others

Other tools and frameworks worth considering: *GATE*, *WordFreak* and *CALLISTO* if your data is textual (i.e., you don't need signal playing to annotate) and you can tolerate stand-off using character offsets; *The Observer*, *EventEditor* *TASX*, *Anvil*, and *ELAN* for simple time-stamped labelling of signals (with some tools offering linking between elements).

C.6 Relationship to the Text Encoding Initiative

The TEI is not a tool like the others on this list, but we have been asked about the relationship between NXT and the Text Encoding Initiative, and in particular, whether it is possible to produce an annotation for spoken dialogue compliant with the TEI standards using NXT GUIs. (Although NXT does get used on text, we have not considered the relationship between NXT and the TEI on textual materials yet, but we expect there to be fewer issues that arise for them.) These are our thoughts on the issue so far. We have made some reference to the P5 documentation in writing them, although we are also relying partly on memory and have not thoroughly checked our work, so it is not definitive. Corrections are welcome. Note also that the TEI states that their guidelines are under revision in this area.

Summary of Answer

If one has TEI-compliance in mind from the start, then it should be possible to design the NXT storage format for the data set so that it only requires a simple transform to be TEI-compliant, and for some data sets it may be possible to make it TEI-compliant as is. However, designing the NXT data representation for maximum TEI-compliance loses the main benefits of using NXT. If the data has crossing hierachies of annotation, using a TEI-compliant representation means losing the search facility that handles these nicely. If the data represents temporal relationships, using a TEI-compliant representation means losing the ability of NXT browsers to highlight the current annotations as a signal plays. In addition, the configurable interfaces for dialogue acts and named entities currently constrain the NXT data representation in ways that violate TEI recommendations, which means that data sets which aim for TEI-compliance would either need to write their own tailored GUIs for everything or contribute (fairly modest) changes to them. If one wants to make use of NXT's best properties, then it would be better to

develop a data path for getting between the NXT and TEI-compliant data formats than to build TEI-compliance into the NXT format. If one doesn't need NXT's facilities for crossing hierachies or timing, then there may be a simpler framework upon which annotation tools can be built.

Data without crossing hierarchies or timing

The TEI recommends particular tag names for orthographic transcription element. These are not a problem for NXT, which has no constraints on tag naming - it just requires the tags to be formally defined in the NXT "metadata" using the TEI's set. The TEI recommends the use of markup within one XML tree as the orthography for the representation of dialogue acts, named entities, turns, and the like. For instance, dialogue acts are represented in the TEI as `<seg>`'s and named entities as `<rs>`'s (or similar non-segmenting spans of transcription elements, such as `<persName>`). One hierarchy of `<seg>`'s over the transcription can be represented in NXT, again by authoring the metadata to match, but the metadata will not be particularly useful for data validation because it will simply have the semantics that all `<seg>`'s draw from the transcription elements as children; if there is internal structure among the segments, NXT will not by itself enforce or check that. Similarly, `<rs>` and similar tags can be used, but technically they violate NXT's data model unless hey are either defined within the orthographic transcription tag set (with recursive descent through that set of tags). This is because strictly speaking, NXT requires "layers" of annotation to span the layers beneath them (in this case, the layer of transcription elements). However, this is a only a weak data model violation, and NXT copes with it by allowing tags to contain either the element types declared as their children or skip directly to the ones declared as their children's children. If one's data does not have crossing hierarchies or a relationship to signal, this suggests that TEI-compliance is either possible or very close. There may be a problem with the representation of links. The TEI practice for relating data elements uses `IDREF` or `IDREFS` or in-file links. Some NXT data sets use string matching on attribute values which is similar to using `IDREF`s, but there is nothing in the attribute declarations which lets NXT validate that relationship. NXT currently writes in-file links using a syntax that (redundantly) contains the filename, although this could be changed without much difficulty. There may also be differences in what's expected at file roots. NXT doesn't require a particular tag name at the root (although it does currently warn if an unexpected one is used), but it doesn't expect headers and bodies in the same file, and the metadata declaration won't allow different content models for two tags at the same depth from the root in the same file, weakening the data validation where they are stored together (since then the content model must specify a disjunction of the possible types at that depth). Every NXT element must have an `id`, which may be a burden for some data sets.

Crossing hierarchies

The main difference between NXT's representation and that of the TEI is whether or not overlapping (crossing) hierarchies pointing down to the same elements are expected. NXT is designed specifically for cases where they are; the TEI contains mechanisms for dealing with crossing hierarchies, but because this is not their primary concern, the mechanisms are more cumbersome. NXT's data representation is based on the idea of multi-rooted trees; in the data model, individual nodes can have one set of children, but multiple parents from different upward trees. A typical use of for this representation in the annotation of spoken dialogue (which makes up NXT's largest user group) is to have time-aligned orthographic transcription at the bottom, and then separate hierarchies for, say, named entities, dialogue acts, prosodic phrases, turns, or whatever that use the words as children. The data is serialized into XML by divided the multi-rooted tree into convenient trees where the XML structure mirrors the data structure and representing the remaining connections between nodes using stand-off links in XLink format. NXT also allows arbitrary additional links to be represented on top of the multi-rooted tree, again using XLinks, but ones that have a different semantics within NXT. The TEI representation for a data set with crossing hierarchies would choose one hierarchy as the primary one, mirror that in the XML structure, and use milestone tags for the other hierarchies. This keeps everything in one file. For extreme cases, one could use the TEI's recommended form for representing graphs, which gives a list of nodes and links where the XML structure does not mirror any part of the graph. Either of these styles of representation can be defined in NXT's "metadata" describing the set of tags, and as long as everything fits into one XML tree they can be kept in one file, but the NXT data validation won't be particularly useful then, and there are no existing GUIs or search facilities that will help in creating or using this data, which means building new ones using the GUI library.

Timing data

The other main difference between NXT and the TEI is in the representation of timing relationships. The TEI gives a choice of mechanisms, ranging from the coarse statement that an element is overlapped via `trans="overlap"`, through the use of `<anchor>` tags that link to overlapping events, to the representation of complete timelines that give time points which then can be used to indicate the start and end times for an element. Any of these representations can be defined in NXT's data storage format, but none of them will get the timing data recognized as time in NXT, which disables one of the most useful features of NXT browsers (the ability to play signals and show which annotations are current as they play). NXT's format for timing information is closest to the last one, but is not TEI-compliant; where annotations of a particular type for different speakers ("agents") can overlap temporally, NXT requires them to be stored in separate files. This is in aid of the temporal semantics inherent in NXT's data model which allows timings to percolate up trees. This requirement can only be circumventing by failing to declare the attributes as times.

Standardized GUIs

NXT comes with some configurable tools for annotating dialogue acts and named entities. These currently rely on an NXT data representation in which the dialogue act and named entity tags point into an external ontology of act or entity types, rather than allowing the type to be expressed as an attribute value. That means that if a data set is represented to be as TEI-compliant as possible in the NXT format itself, these tools cannot be used. We are considering making it possible to configure the tools to use an enumerated attribute, but we don't have an immediate need for the result so the work hasn't been scheduled yet. If there is more than one type of `<seg>` in the data, this will cause problems for setting up the tool because the NXT metadata will have no way of specifying which types go together into one set to be annotated together (so, for instance, making dialogue act annotation different from some other segmentation and classification task).

Other frameworks

The difficulties in mapping between the TEI and NXT arise from the fact that NXT is designed for data that is rather esoteric for the TEI. If one doesn't need crossing hierachies or relationships to signal, there may be other annotation frameworks that are closer to TEI-compliance in their native data formats. We have never considered other frameworks in this light. *MMAX2* uses multiple file stand-off, so probably isn't any closer. Other key words to search on are *AGTK*, *CALLISTO*, *ATLAS*, and *WordFreak*.

D. Information and Further Reading

Updated printable documentation: March 07

Download v.0.2

Notes on version 0.2

In October 2006, we decided to move over NXT documentation from being completely web-based to being written in DocBook so that we can generate HTML, JavaDoc, and PDF at will. We are rewriting much of the documentation at the same time. Versions of the documentation numbered before v1.0 are incomplete, although the outline gives some idea of our intentions for it. In this version, version 0.2, not all of the information has been checked for accuracy yet. The most likely difficulties concern the following areas: corpus resources, ontologies, and object

sets; validation; incomplete description of data set concepts. In addition, not all the formatting works, and the query reference manual has not been fully converted over to DocBook, so it is incomplete and hard to read in this version.

D.1 NXT's history and funding

The NITE XML Toolkit is software that arose out of a European Commission-funded collaboration between the University of Edinburgh's The Language Technology Group, the University of Stuttgart's Institut för Maschinelle Sprachverarbeitung (IMS), and the Deutsches Forschungszentrum för Könstliche Intelligenz (DFKI). Although the NITE project itself finished in 2003, the software is now being maintained and further developed via Sourceforge; the University of Twente has been a particularly active contributor. NXT is in use on a number of large distributed projects including JASTand TALK. NXT is in use on a wide range of corpora, representing everything from Biblical text structure to the relationship between deictic expressions and gestures in multimodal referring expressions. Its users range from individual PhD students up to large multi-site projects, many of whom contribute to development in some way. The AMI consortium is its biggest user and also the largest current contributor to its development. Other past and current funders are The Engineering and Physical Sciences Research Council (UK) , The Economic and Social Research Council (UK), and Scottish Enterprise, via The Edinburgh-Stanford Link.

D.2 Technical Documents

These are project internal documents that the NXT partners have agreed to make web-accessible.

- Formal specification of the NITE Object Model, the abstract data model used by the NITE XML Toolkit.
- End user documentation of NiteQL , the query language that operates over data conforming to the NITE Object Model.
- Formal specification of NiteQL, the query language that operates over data conforming to the NITE Object Model.
- Metadata information describing the metadata format required by the NITE Object Model, and how to produce one for your data.
- A document introducing our display object library and describing how to use display objects as building blocks for data display from stylesheets (pdf format)

D.3 Documentation for Programmers

Bug reports and feature requests are kept on Sourceforge. The sample programs that come with NXT provide commented example code.

However, there is a space here for a general overview of the programmers' API provided by NXT, but currently there's only the Javadoc that comes with the NXT dowload. We will generally provide Javadoc from the latest CVS build which will not necessarily tally with the NXT version you downloaded, so don't rely on it.

D.4 Academic publications

Because we're academics, it helps us if you cite one of our papers when making use of NXT. Please be aware that there were other software products to arise out of the NITE project, and be sure to credit the correct one.

Papers about the NITE XML Toolkit or development concerns

CARLETTA J.EVERT S.HEID U.KILGOUR J. (in press)
The NITE XML Toolkit: data model and query
. Language Resources and Evaluation Journal

CARLETTA J.EVERT S.HEID U.KILGOUR J.ROBERTSON J.VOORMANN H. (2003)
The NITE XML Toolkit: flexible annotation for multi-modal language data
. Behavior Research Methods, Instruments, and Computers, special issue on Measuring Behavior, 35(3), 353-363.

CARLETTA J.KILGOUR J.O'DONNELL T.EVERT S.VOORMANN H. (2003)
The NITE Object Model Library for Handling Structured Linguistic Annotation on Multimodal Data Sets
.

MAYO N.KILGOUR J.CARLETTA J. (2006)
Towards an alternative implementation of NXT's query language via XQuery
. EACL Workshop on Multi-dimensional Markup in Natural Language Processing, Trento, Italy, April 4th.

REIDSMA D.JOVANOVIC H.HOFS D. (2005)
Designing annotation tools based on properties of annotation problems
. Measuring Behavior 2005 , 5th International Conference on Methods and Techniques in Behavioral Research, 30 August - 2 September 2005, Wageningen, The Netherlands.

Research papers that mention NXT in use (more than in passing)

BLAYLOCK N.SWAIN B.ALLEN J. (2009)

TESLA: A Tool for Annotating Geospatial Language Corpora
. In Proceedings of the North American Chapter of the Association for Computational Linguistics - Human Language Technologies (NAACL HLT) 2009, Toronto, Canada, May 2009.

CALHOUN S.NISSIM M.STEEDMAN M.BRENIER J. (2005)
A framework for annotating information structure in discourse
. In Frontiers in Corpus Annotation II: Pie in the Sky, ACL2005 Conference Workshop, Ann Arbor, Michigan, June 2005.

CARLETTA J.C.KILGOUR J. (2005)
The NITE XML Toolkit Meets the ICSI Meeting Corpus: Import, Annotation, and Browsing.
. In MLMI'04: Proceedings of the Workshop on Machine Learning for Multimodal Interaction., SamyBengio and HerveBourlard eds.
Springer-Verlag Lecture Notes in Computer Science Volume 3361
. ISBN: 3-540-24509-X. This is an updated version of a workshop paper.

CARLETTA J.DINGARE S.NISSIM M.NIKITINA T. (2004)
Using the NITE XML Toolkit on the Switchboard Corpus to study syntactic choice: a case study
. In Fourth Language Resources and Evaluation Conference, Lisbon, Portugal, May.

GUT U.MILDE J-T.VOORMANN H.HEID U. (2004)
Querying annotated speech corpora
. In Speech Prosody (International Conference), Nara, Japan, March 23-26, ed. by BernardBel and IsabelleMarlien, ISCA, 569-572.

HEID U.VOORMANN H.MILDE J-T.GUT U.ERK K.PADO S. (2004)
Querying both time-aligned and hierarchical corpora with NXT Search
. In Fourth Language Resources and Evaluation Conference, Lisbon, Portugal, May.

ISARD A.BROCKMANN C.OBERLANDER J. (2005)
Re-Creating Dialogues from a Corpus
. In Proceedings of the Corpus Linguistics 2005 Workshop on Using Corpora for Natural Language Generation, July 2005 Birmingham, U.K.

Pre-NITE paper motivating the concept

CARLETTA J.MCKELVIE D.ISARD A.MENGEL A.KLEIN M. (2005)
A generic approach to software support for linguistic annotation using XML
.
G.Sampson D.McCarthy
Corpus Linguistics: Readings in a Widening Discipline
Continuum International
London and NY
ISBN: 082648803X

Paper about the NITE project in general

SORIA C.BERNSEN N. O.CADEE N.CARLETTA J.DYBKJAER L.EVERT S.HEID U.ISARD A.KOLODNYTSKY M.LAUER C.LEZIUS W.NOLDUS L.PIRRELLI V.REITHINGER N. (2002)
Advanced tools for the study of natural interactivity
Third International Conference on Language Resources and Evaluation (LREC 2002)
Las PalmasSpain
May