NXT Search

User's Manual (Draft)

Holger Voormann, Stefan Evert, Jonathan Kilgour,
Jean Carletta

IMS, University of Stuttgart and HCRC Edinburgh

Last modified:2003/7/30

Table of contents

Chapter I - Introduction - The NITE project

The NITE project

NITE (Natural Interactivity Tools Engineering) is a European project carried out under the European Commission's HLT programme in 2001-2003. More infomation about the project and a full list of all partners can be found at the following URL:

http://nite.nis.sdu.dk/

Goal of NITE

The objective of NITE is to develop integrated toolset, for multi-level, cross-level and cross-modality annotation, retrieval and exploitation of multi-party natural interactive human-human and human-machine dialogue data.

Three development strands

In NITE three different strands of software were developed:

1. **NITE Workbench for Windows (NWB)**
   Only available for Windows platform, NWB aims at users who want an easy-to-use interface that requires no programming skills.
2. **NITE XML Toolkit (NXT)**
   A cross-platform toolkit for technically experienced users.
3. **Noldus Observer Software**
   This strand will enable the commercial software from Noldus, *The Observer* to support the annotation of natural interactive communication (cf. *http://www.noldus.com/products/index.html*).

Chapter II - The application NXT Search

1. System requirements

Hardware requirements

- min. 128 MB main memory
- min. 100 MB free hard disk space (includes space required during installation)

Windows

- Intel Pentium II/233 MHz or higher (or compatible)
- Microsoft Windows 98 (SE), 2000 (SP 2), XP, or NT 4.0 (SP6a)

Linux

- Intel Pentium II/233 MHz or higher (or compatible)
- Red Hat Linux 6.2 or higher; SuSE Linux 6.4 or higher

Solaris

- UltraSPARC II or higher
- Solaris 7 (2.7) or 8 (2.8)

Mac

- Power Mac G3, G4, G4 Cube; iMac; PowerBook G3, G4; iBook; eMac
- Mac OS X, version 10.2.4 or higher

⚠**Please note:** For Microsoft Windows, Linux and Solaris, the required Java Runtime Environment (JRE) is bundled with the respective download version of the NXT Search software. For Mac OS X, the JRE is part of the operating system installation.

2. Installation instructions

2.1 Windows

Installation

The installation on the Windows platform is realized by a self-extracting `.exe` file that also comprises the Java Virtual Machine which is necessary for using NXT Search.
To install the application NXT Search and the sample corpora delivered along with the software on the Windows platform the following steps are necessary:

1. Download the installation file.

2. Install NXT Search on your system:

- Double-click the installation file. The installation of NXT Search will be started.
- Choose the destination directory.

  ⚠**Please note:** The data files of corpora are located in subdirectories of *%install_directory%* `/corpora/`

- Choose the shortcut directory. The installation tool will place links to NXT Search and to the HTML and PDF versions of the User's Manual. The User's Manual includes information about the query tool, the NXT Query Language (cf. chapter III), and about the NXT Corpus Format (cf. chapter V). Also you will find a link to the uninstall process, which you will hopefully only use to upgrade to a newer version of NXT Search.
- Now the program files are copied to your system. The end of the installation is reached.
- Start NXT Search (cf. section 3).

Uninstall

There are two ways to start the uninstall process:

- Use the shortcut *Uninstall NXT Search*.
- Enter the Windows System Control. Open the Software Properties. Mark the NXT Search entry in the software list and click the *Remove* button.

2.2 Unix (Linux/Solaris)

Installation

The installation on the Unix platform (Linux and Solaris) is realized by self-extracting `.bin` files that also comprise the Java Virtual Machines which are necessary for using NXT Search on Linux and Solaris.

To install the application NXT Search and the sample corpora delivered along with the software on the Linux or Solaris platform the following steps are necessary:

1. Download the installation file for your platform.

2. Install NXT Search on your system:

- Start the binary installation file.

  ⚠**Please note:** The binary file must be executable, i.e. the file permission must be set to `rwx` (`chmod u+rwx filename`).

- Choose the destination directory.

  ⚠**Please note:** The data files of corpora are located in subdirectories of *%install_directory%* `/corpora/`

- Choose the link directory. The installation tool will place a link to NXT Search in this directory. For example, you might choose `/usr/local/bin`, `/usr/bin` or `~/bin`.

  ⚠**Please note:** The HTML and PDF versions of the User's Manual are placed in the `doc/` subdirectory of the destination directory.

  ⚠**Please note:** The data files of corpora are located in subdirectories of *%install_directory%* `/corpora/`

- Now the program files are copied to your system. The end of the installation is reached.

- Start NXT Search by using the shortcuts created by the installation tool. If you did not create shortcuts, `NXT Search` executables are located in the `bin/` subdirectory of the installation directory. (cf. section 3).

Uninstall

Use the *Uninstall_NXT_Search* link or enter the `UninstallerData/` subdirectory of the destination directory. Start the uninstall script `Uninstall_NXT_Search`.

⚠**Please note:** The script must be executable, i.e. the file permission must be set to `rwx` (`chmod u+rwx filename`).

2.3 Mac OS X

Installation

The installation on the Mac OS X platform is realized by a self-extracting `.zip` file. To install the NXT Search and the sample corpora delivered along with the software on Mac OS X platform the following steps are necessary:

1. Download the installation file:

After downloading, the Mac installer included in the `.zip` file will be automatically recognized and decoded by *StuffIt Expander*. If your system does not handle the file automatically, download and install a current version of the StuffIt Expander software (cf. *http://www.aladdinsys.com/expander/*).

2. Install NXT Search on your system:

- After downloading, a new icon named `nxtsetup` is placed on your desktop. Double-click this icon. The installation of NXT Search will be started.
- Choose the destination directory (typically the directory where programs are installed on your system).

  ⚠**Please note:** The data files of corpora are located in subdirectories of *%install_directory%* `/corpora/`

- Choose the alias folder. We recommend the option *Place into Home Folder*. The installation tool will place links to NXT Search and to the HTML and PDF versions of the User's Manual. The User's Manual includes information about the query tool, the NXT Query Language (cf. chapter III), and about the NXT Corpus Format (cf. chapter V). Also you will find a link to the uninstall process, which you will hopefully only use to upgrade to a newer version of NXT Search.
- Now the program files are copied to your system. The end of the installation is reached.
- Start NXT Search by clicking the corresponding icon in your dock.

Uninstall

Use the uninstall alias *Uninstall_NXT_Search* or enter the `UninstallerData/` subdirectory of the destination directory. Double-click the icon `Uninstall_NXT_Search`.

3. Launching NXT Search

The way you can start the NXT Search tool depends on your operating system. On Windows machines, a program group called *NXT Search* has been created during the installation - so you just have to select the *NXT Search* program in the start menu.

On Unix machines, symbolic links have been created. If your general path is set properly, you may just need to type in `NXT_Search`. However, the NXT Search start program can always be found in the NXT Search installation path:
%install_directory% `/bin/NXT_Search`

⚠**Please note:** Any relative path specified in a dialog window is evaluated with regard to the so-called working directory. On Unix machines this directory is defined as the NXT Search starting directory (i.e. the directory which NXT Search has been started from). On Mac and Windows machines the working directory is defined as the user's home directory.

When you start the NXT Search application, the NXT Search window pops up (cf. screenshot).

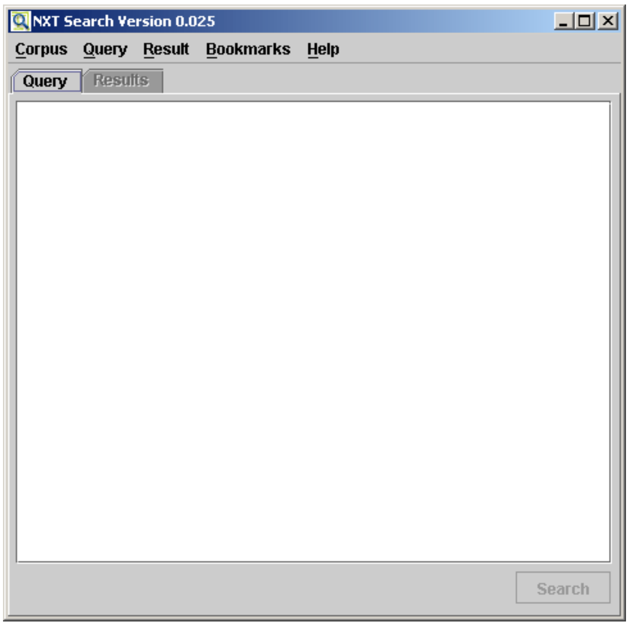
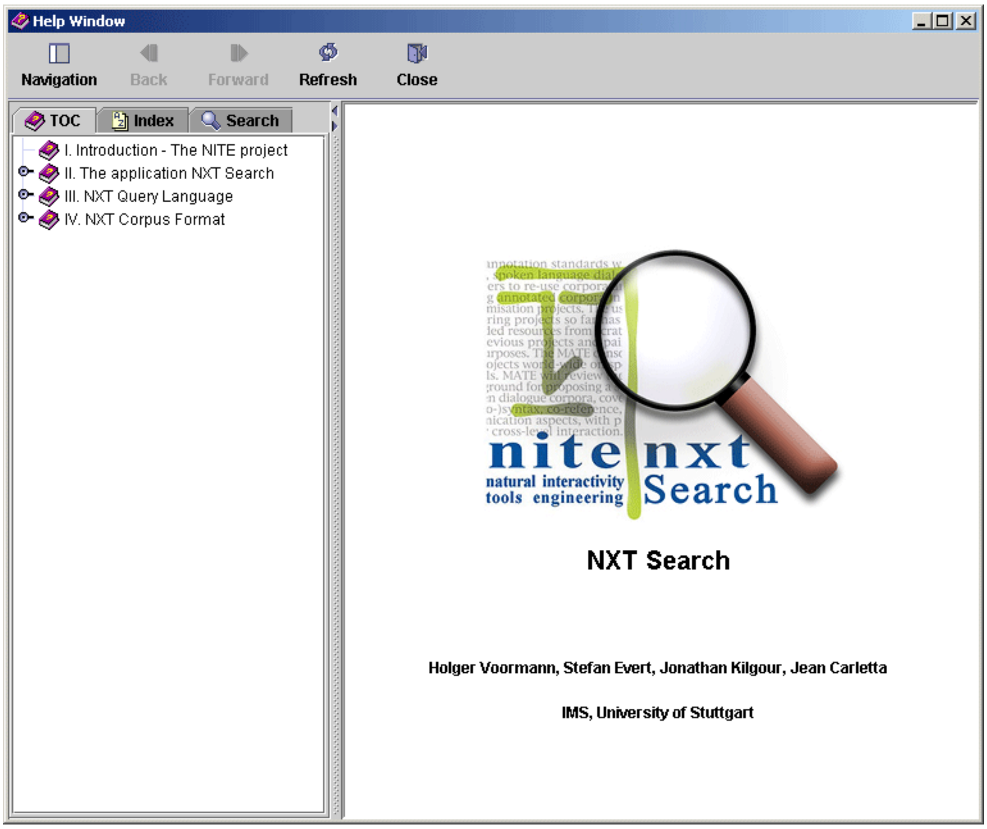


Figure: The NXT Search window

At the top of the window you find the menus: *Corpus* (cf. section 5), *Query* (cf. section 6), *Result* (cf. section 6), *Bookmarks* (cf. section 7), and *Help* (cf. section 4). The main, white area is the query input field. Here you can type in your query. Below the input field, in the bottom right corner of the window, the *Search* button is placed, used to submit a query.

4. The help window

The NXT Search User's Manual can be accessed directly within the NXT Search user interface. The NXT Search help window can be activated by selecting one of the items in the *Help* menu.



NXT Search

Holger Voormann, Stefan Evert, Jonathan Kilgour, Jean Carletta

IMS, University of Stuttgart

Figure: The NXT Search help window

The help window is divided into two parts: On the left there is the manual navigation area. Here you find a table of contents, an index, and a search engine. Just browse through the table of contents or through the index and click on the topic you are interested in. If you are looking for a special topic which you do not find in the table of contents or in the index, you might use the search engine. Just type in your search item(s) and the search engine finds all topics in which one of the items is mentioned.

The help window toolbar comprises the following buttons: The *Navigation* button lets you hide the navigation bar. The *Back* and *Forward* buttons let you navigate through the topics you have viewed before. The *Refresh* button reloads a topic, and the *Close* button allows you to leaves the help window.

Corpus queries are displayed as green-colored hyperlinks. If you click on a query hyperlink, the query text is automatically copied into the query editor of the NXT Search main window.

5. Corpus menu

Open corpus

To open a corpus select the first item named *Open ...* in the *Corpus* menu. A file chooser dialog will appear, showing the content of the *corpora* directory, a subdirectory of your installation directory (cf. section 2). Select the metadata file (cf. section 2, chapter V) of the corpus you want to load. If the corpus has been successfully loaded, the name (of the metadata file) will be shown in the window title. If loading fails, an error message will appear in the status bar at the bottom of the window.

⚠**Please note:** At any given time only one corpus can be loaded. A currently open corpus will be closed before opening a new corpus.

⚠**Please note:** Loading a big corpus can take a long time, and loading cannot be interrupted.

Reload corpus

If the current corpus has been edited, the corpus has to be reloaded to make the changes visible.

Close corpus

If *Autoload* (see next paragraph) is enabled and you don't want to load automatically a corpus next time you launch NXT Search, you have to close the current corpus before exiting.

Autoload

An interesting feature is the so-called *corpus autoload*. If this feature has been activated, the corpus open when leaving the tool will be automatically loaded when the tool is started for the next time. The autload feature is not active by default and can be activated by selecting the *Autoload* menu item in the *Corpus* menu.

Corpus shortcuts

If a corpus has been successfully loaded, a shortcut will be added to the *Corpus* menu. The shortcut list contains up to 8 links of the last corpora you loaded. Use one of the links to load the respective corpus immediately.

Exit

To leave the tool, select the *Exit* menu item.

6. Querying

Submit a query

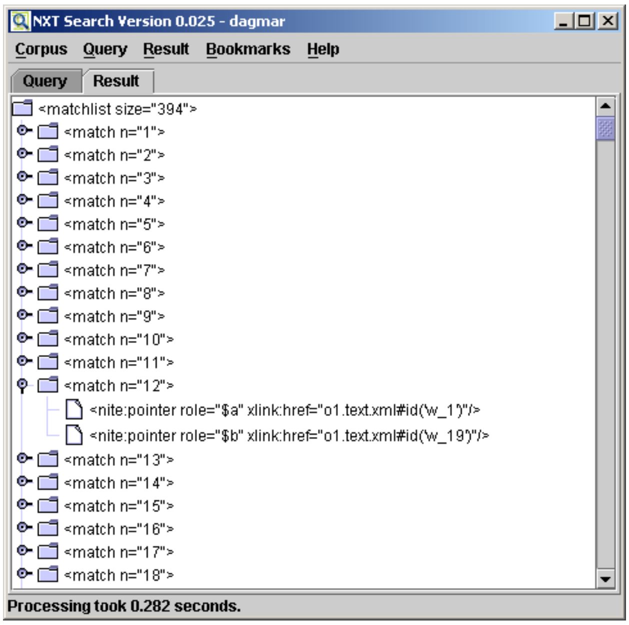
Before you can submit a query, a corpus must be loaded (cf. section 5).
You can submit a query by selecting *Search* in the *Query* **menu**, by clicking the *Search* **button**, or by hitting the *ENTER* **key** while the *CTRL* key is pressed.

Syntax error

If the submitted query contains syntax errors, an error message alert will appear. This window contains a description of the first detected syntax error. The window is closed by clicking the *OK* button. The cursor will be set to the suspected error position.

Viewing results

After successfully proccessing a query, the result is shown in the *Result* tab as a tree.



```
<matchlist size="394">
  <match n="1">
  <match n="2">
  <match n="3">
  <match n="4">
  <match n="5">
  <match n="6">
  <match n="7">
  <match n="8">
  <match n="9">
  <match n="10">
  <match n="11">
  <match n="12">
    <nite:pointer role="$a" xlink:href="o1.text.xml#id('w_1')"/>
    <nite:pointer role="$b" xlink:href="o1.text.xml#id('w_19')"/>
  <match n="13">
  <match n="14">
  <match n="15">
  <match n="16">
  <match n="17">
  <match n="18">
Processing took 0.282 seconds.
```

Figure: Result visualisation.

Saving a result

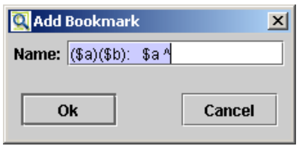
You can save a result as an XML file. Select *Save ...* in the *Result* menu. A file chooser window will open, where you can specify the name and location of the XML result file.

7. Bookmarks

To store your favourite corpus queries, NXT Search provides a simple bookmarking facility.

Adding a bookmark

To save a query, select *Add Bookmark* ... in the *Bookmark* menu. A window pops up. Here you can name the bookmark.



Name: ($a)($b): $a ^

Figure: Add Bookmark window.

Opening a bookmark

A bookmarked query can be opened by selecting one of the bookmark menu items in the *Bookmark* menu. The current query will be replaced by the opened query.

⚠**Please note:** Queries are mostly written for a given corpus. When you open a bookmark, be sure that a suitable corpus is loaded.

Deleting a bookmark

To delete a bookmark open the *Delete Bookmark* submenu in the *Bookmark* menu. Select the bookmark you want to delete. The bookmark will be deleted without prompting.

Chapter III - The NXT Query Language (NQL)

The *NITE* (Natural Interactivity Tools Engineering) XML Toolkit (NXT) has its own query language: the NXT Query Language (NQL). It is based on the *MATE* (Multilevel Annotation Tools Engineering) *query language*.

1. General structure of a query

A query consists of a variable declaration and a match condition part. Variables represent elements (usally of a certain type). The match condition is a Boolean expression over attribute tests, structural and temporal relations (cf. section 2). The result of a query is a list of variable bindings (mapping each variable to a specific element in the corpus) which satisfy the match condition.
The declarations part is separated from the match condition by a `:` character.

***query* :=** *declarations* `:` *match_conditions*

An example:

```
($a)($b word): $a ^ $b
```

This query looks for an element of arbitrary type that dominates a word element. In this example *($a)($b word)* is the declaration part and the dominance relation *$a* ^ *$b* is the only condition.

Declaration part

In the declaration part every variable of the query must be declared. Each variable declaration is enclosed in parentheses. The first item of a variable declaration is the name of the variable. A variable name starts with a *$* character followed by an arbitrary number of letters and digits (including '_' and language-specific characters).
Usually, a variable is assigned to a certain element type (i.e. it may only be bound to elements of this type). Also it is allowed to specify none or a list of element types:

`($a)`	*$a* will be bound to elements of any type ⚠**Please note:** An empty type definition may slow down query processing drastically.
`($a word)`	*$a* will be bound to elements of type *word*.
`($a type1 | type2 | ... )`	*$a* will be bound to elements of one of the types *type1*, *type2*, etc.

⚠**Please note:** Two or more different variables may be bound to the same element.

Formal definition:

***declarations* :=** *declarations var_declaration*
| *var_declaration*
***var_declaration* :=** (*variable*)
| (*variable typedefiniton*)
***typedefiniton* :=** *typedefiniton type*
| *type*

Condition part

The condition part is a Boolean expression over attribute tests, structural, and temporal relations. For details see section 2.

⚠**Please note:** The condition part may be empty, in which case the query always evaluates to true.

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.

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

2. Match condition

The **match condition** is a Boolean expression over attribute tests (cf. section 3), structural relations (cf. section 4), and temporal relations (cf. section 5).
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:

1. Negation: `!` `not`
2. Conjunction: `&` `&&` `and`
3. Disjunction: `|` `||` `or`
4. Implication: `->`

The following constructions are possible for a match condition:

***match_conditions* :=** (*match_conditions*)

| ! *match_conditions*

| *match_conditions* & *match_conditions*

| *match_conditions* | *match_conditions*

| *match_conditions* -> *match_conditions*

| *attribute_test*

| *structural_relation*

| *time_relation*

⚠**Please note:** The negation ! may also be written as `not`, the conjunction & may be replaced by && or by the word `and`, and the disjunction symbol | may also be replaced by `||` or by the word `or`.

Implication relation

The implication operator `->` is the weakest binding operator. The Boolean expression

a -> *b*

is logically equivalent to the expression:

! *a* | *b*

It is mainly used in conjunction with the *forall* quantifier (cf. section 6).

3. Attribute tests

An attribute test compares two expressions. An expression may be an attribute value, a constant, or the result of an element function like `start($x)`, `duration($x)`, etc.
Expressions are weakly typed. An expression will be interpreted as a floating-point number when possible. If it cannot be converted into a number or if the expression is compared to a pattern given by a regular expression, the expression is a string. For example `"2" == "2.0"` is true, while `"Two" == "two"` is false.

attribute_test **:=** *expression* == *expression*
| *expression* != *expression*
| *expression* < *expression*
| *expression* <= *expression*
| *expression* >= *expression*
| *expression* >= *expression*
| *variable* @ *attribute*
| `TIMED(` *variable* `)`
| *expression* ~ / *pattern* /
| *expression* !~ / *pattern* /

expression **:=** *number_or_string*
| *variable* @ *attribute*
| `TEXT(` *expression* `)`
| `ID(` *expression* `)`
| `START(` *expression* `)`
| `END(` *expression* `)`
| `DURATION(` *expression* `)`
| `CENTER(` *expression* `)`

⚠**Please note:** The functions *TIMED(), TEXT(), ID(), START(), END(), DURATION()*, and *CENTER()* can also be written with lower case letters.

Tests: has attribute, is timed

For testing if an element has a particular attribute, use the notation `$x@cat`, where *$x* is the variable and *cat* the name of the requested attribute.
With the function `timed($x)` it can be tested whether the element *$x* has timestamps. A timed element must have a start and end timestamp of its own or inherits the time information from its children. For more details on temporal relations see section 5.

`($x): $x@cat`	elements with the *cat* attribute (syntactic category)
`($x): timed($x)`	element *$x* is timed, means *$x* has a start time and an end time value

Element functions

There are temporal element functions to get information about start and end time, the center of the time interval (average of start and end times), and the duration of an element *$x*:

- **Start time:** `start($x)`
  Returns the start time stamp of the timed element *$x*.
- **End time:** `end($x)`
  Returns the end time stamp of the timed element *$x*.
- **Center of start and end time:** `center($x)`
  Returns `(end($x) + start($x))/2` of *$x*.
- **Duration:** `duration($x)`
  Returns `end($x) - start($x)` of *$x*.

See section 5 for more temporal relations.
Also there are functions get the ID of an element *$x* and its textual content:

- **ID:** `id($x)`
  Returns the value of the NITE identifier attribute, which every element must contain (cf. subsection 2.2, chapter V).
- **Text:** `text($x)`
  Returns the text embedded in the element *$x*.

Comparison operators: ==, !=, <, <=, >, >=

Expressions are tried to interpreted as numbers. Numbers can compared with the operators ==, `!=`, <, <=, >, and >=. But also strings could be compared by these operators. A number is always unequal to a string. Strings are alphabetically ordered. Strings starting with upper case letters are < strings with upper case letters. Here are some examples:

`($x): $x@cat=="NP"`	elements with syntactic category *NP*
`($x)($y): $x@cat==$y@cat`	all combinations of two elements with identical syntactic categories
`($x)($y): $x@cat==$y@cat & $x != $y`	all combinations of two *different* elements with identical syntactic categories
`($a): $a@orth > "s" & $a@orth < "u"`	*$a@orth* must be a string starting with the lower case letter *t*

Regular expressions

An attribute value and the results of a function can be tested against a regular expressions. Regular expressions are enclosed by slashes `/`. The syntax of regular expressions used in this query language is compatible with the syntax of regular expressions in Perl 5.003. In a *pattern* the following operators could be used:

`.`	unspecified character
`*`	unrestricted repetition
`+`	one or more
`[ ... ]`	character set
`[^ ... ]`	negated character set
`( ... )`	grouping
`|`	disjunction

⚠**Please note:** Regular expression are always anchored at the start and end of strings: `/x/` in NQL notation means `/^x$/` in the Perl notation.

`($a): text($a) ~ /th.*/`	words starting with *th*
`($a): text($a) ~ /[dD](as|er)/`	the words *das* and *der*, irrespective of capitalization of the first letter
`($a): text($a) ~ /.+([0-9A-Z])+.*/`	words which contain at least one uppercase letter or a figure at a non-initial position, i.e. hyphenated compounds, and potential abbreviations and product names
`($a): text($a) ~ /\.*/`	a possibly empty sequence of dots, where in contrast /.*/ matches every word

4. Structural relations

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*
| *variable* `!=` *variable*

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*
| *variable* ^ *distance variable*

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

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*

⚠**Please note:** The expression *a* <> *a* is always false!

Some 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. Temporal relations

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))`
`#`	`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)`

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 section 2 is now extended with quantifiers:

var_declaration := (`exists` *variable*)
| (`exists` *variable typedefiniton*)
| (`forall` *variable*)
| (`forall` *variable typedefiniton*)

In queries with quantifiers the implication operator `->` could be useful (see section 2).

Some examples:

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

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. section 6). The query result is returned in the form of an XML document (or, abstractly, 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>
```

⚠**Please note:** The matches are not ordered. The ordering of the results of two similar but not identical queries can be very different.

8. Complex queries

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

***complex_query* :=** *complex_query* `::` *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>
```

⚠**Please 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.

Chapter IV - Corpora Sampler and Queries

In this section sample queries for the corpora distributed with NXT Search can be found.

1. Floorplan Corpus Sampler

The Floorplan Corpus is based on a video, in which two persons discuss about their new office looking at some floor plans. The video screen is split into three parts. The upper split screen shows both persons sitting at a table. The left bottom screen shows the face of the male, and the right bottom screen the face of the female speaker.

Figure: The split screen video.

1.1 Annotation Schema

Word layer

File: o1.words.xml.
Element **word**:

- **text()**
  the word itself is containt in the element as text
- **@hlem**
  head lemma
- **@pos**
  must be one of:

CC	conjunction: and, but, or
CD	card. numeral: eight, fifteen
DT	determiner: a, an, all, any, this
EX	existential there: there (in *there is*)
IN	preposition: about, against, before, in
JJ	adjective (pos.): awful, detailed, enormous, happy
JJR	adjective (comp.): easier, larger

MD	modal verb: should, might
NN	singular noun: area, block, coffetable, division
NNS	plural noun: architects, offices, drawings
NP	person name: Thomas
PDT	pronoun or det.: half
PP	personal pron.: I, he, us, they
PP$	possessive pron.: our, their
RB	adverb (pos.): actually, elsewhere, also
RBR	adverb (compar.): better, less
RP	verb particle: out, over
TO	preposition *to*
UH	interjection: no, oh, yep
VB	verb (inf or pres): forget
VBD	verb (part): happened
VBG	verb (-ing-form): comparing
VBN	verb participle: compared
VBZ	verb 3rd p. sing.: corresponds
WDT	rel./interr. pron: that, whatever
WRB	rel./interr. advb: how, when

Turn layer

File: o1.turns.xml.
Element **turn**:

- timed: **start()**, **end()**, **duration()**, **center()**

Element **speaker**, child of *turn* element,
has word children in the word layer:

- **@n**
  speaker: either *1* or *2*

Element **comment**, child of *speaker* element:

- **@desc**
  *desc*ription: comment text

Element **event**, child of *speaker* element:

- **@desc**
  *desc*ription
- **@type**
  one of: noise, lexical, pronounce, language
- **@extent**
  one of: previous, next, instantaneous, start, end

Non-verbal comunication layers

Gesture files (*sp*eaker 1 or 2, hand: *left*, *right*, *both*): o1.sp1left.xml, o1.sp1right.xml, o1.sp1both.xml, o1.sp2left.xml, o1.sp2right.xml, o1.sp2both.xml.
Elements: **sp1left**, **sp1right**, **sp1both**, **sp2left**, **sp2right**, **sp2both**:

- **gesture types ontology pointer(s)**
- optional: **@desc**
  *desc*ription

Facial expression files: o1.sp1face.xml, o1.sp2face.xml.
Elements: **sp2face**, **sp2face**:

- **facial expression types ontology pointer**
- optional: **@desc**
  *desc*ription

Gaze files: o1.sp1gaze.xml, o1.sp2gaze.xml.
Elements: **sp1gaze**, **sp2gaze**:

- **@type**
  one of: emblem, baton, other
- optional: **@desc**
  *desc*ription
- optional: **@meaning**
  meaning of the gaze

Bodily posture files: o1.sp1body.xml, o1.sp2body.xml.
Elements: **sp1body**, **sp2body**:

- **@type**
  one of: emblem, baton, other
- optional: **@desc**
  *desc*ription

Gesture types ontology

File: gtypes.xml.

Elements: **gtype**.

```
gesture or movement
 ├ task-oriented movement
 └ gesture
     ├ adaptor
     |   ├ self-directed
     |   |   ├ self-directed attitude
     |   |   └ self-directed emotion
     |   ├ alter-directed
     |   |   ├ alter-directed attitude
     |   |   └ alter-directed emotion
     |   └ object-directed
     |       ├ object-directed attitude
     |       └ object-directed emotion
     ├ emblem
     |   ├ part of body
     |   ├ function of body
     |   ├ based on task-oriented movement
     |   ├ perception of body
     |   ├ artefact-handling
     |   ├ pointing (deictic)
     |   ├ socially conventionalised
     |   └ other emblem
     └ illustrator
         ├ baton
         ├ deictic
         |   ├ abstract deictic
         |   └ concrete deictic
         ├ iconic
         └ metaphoric
```

Facial expression types ontology

File: ftypes.xml.
Elements: **ftype**.

```
facial expression
 ├ adaptor
 ├ emblem
 └ affect display
     ├ happiness
     ├ surprise
     ├ sadness
     ├ anger
     ├ disgust
     ├ interest
     └ fear
```

1.2 Sample queries

1a) All *iconic* gestures:

```
($x)(exists $type gtype): $x > $type and $type@name =
"iconic"
```

1b) All *iconic* gestures by speaker 1:

```
($x sp1left, sp1right, sp1both)(exists $type gtype): $x >
$type and $type@name="iconic"
```

1c) All gestures which are *illustrator*s or any subtype thereof by speaker 1:

```
($x sp1left, sp1right, sp1both)(exists $type gtype): $x >^
$type and $type@name="illustrator"
```

2a) Gestures which are classified simultaneously by more than one type:

```
($g)($gt1 gtype)($gt2 gtype): $g > $gt1 and $g > $gt2 and
$gt1 != $gt2
```

2b) Turn(s), which temporally overlap(s) the first result of 2a).

```
($t turn)(exists $g): id($g) = "s1r_31" and $t # $g
```

⚠**Please note:** Manual inspection of the results of query 2b) has shown that the identifier of the first result is *s1r_31*. This ID can be used in a subsequent query. Similarly, cascades of queries can be realized to stepwise refine search.

3) Turns temporally overlapping with gestures of (sub)type *illustrator*:

```
($t turn)($g)(exists $type gtype): $t # $g and $g >^ $type
and $type@name = "illustrator"
```

4) *Deictic* gestures near (= overlapping with the turn of) the word *this* or *these* respectively:

```
(exists $w word)($t turn) (exists $g sp1left, sp1right,
sp1both)(exists $type gtype): ( text($w) = "this" or
text($w) = "these" ) and $t ^ $w and $g # $t and $g > $type
and $type@name = "deictic"
```

5) Descriptions (annotated in the attribute *@desc*) containing the word *paper*:

```
($x):$x@desc ~ /.*paper.*/
```

6) *Batons* of any modality:

```
($x)(exists $type gtype): $x@type="baton" or ($x > $type
and $type@name="baton")
```

⚠**Please note:** *Baton* is a subtype of the gesture type ontology. In the gaze, facial expression, and bodily posture modalities, it is annotated as a feature (*@type="baton"*).

2. Dagmar Corpus Sampler

Both-handed batons:

```
($a gesture)($g gtype): ($a@hand == "both") && ($a > $g) &&
($g@name == "baton")
```

Gestures which are neither batons nor iconic:

```
($a gesture)($g gtype): ($a > $g) && ($g@name != "baton")
&& ($g@name != "iconic")
```

Same as above, but in addition we want a word *diese* to be **in** the time frame of the gesture:

```
($w word)($a gesture)($g gtype): ($a # $w) && ($w@orth ==
"diese") && ($a > $g) && ($g@name != "baton") && ($g@name
!= "iconic")
```

Gestures overlapping with the words *Geste*; or *Handbewegung*:

```
($w word)($g gesture): (($w@orth == "Handbewegung") ||
($w@orth == "Geste")) && ($g # $w)
```

PPs with an embedded personal pronoun (PPER):

```
($s syntax)($w word): ($s@cat == "PP") && ($w@pos ==
"PPER") && ($s ^ $w)
```

The same situation, but with the embedded NP expressed and marked:

```
($s syntax)($s2 syntax)($w word): ($s@cat == "PP") &&
($s2@cat == "NP") && ($w@pos == "PPER") && ($s ^1 $s2) &&
($s2 ^ $w)
```

The same with the embedded NP just postulated as being existent:

```
($s syntax)(exists $s2 syntax)($w word): ($s@cat == "PP")
&& ($s2@cat == "NP") && ($w@pos == "PPER") && ($s ^1 $s2)
&& ($s2 ^ $w)
```

Which nouns are said while some gesture takes place (has already started)?:

```
($g gesture)($w word): ($w@pos == "NN") && ($g % $w)
```

Which mentions of a personal pronoun come with an accompanying (overlapping) gesture, which is not a baton?:

```
($g gesture)($w word)(exists $t gtype): ($w@pos == "PPER")
&& ($g % $w) && ($g > $t) && ($t@name != "baton")
```

Which sequences of *machen* and *Geste* or *Handbewegung* are found in one sentence?:

```
($o syntax)($v syntax)(exists $s syntax): ($s@cat == "S")
&& ($v@hlem == "machen") && (($o@hlem == "Geste") ||
($o@hlem == "Handbewegung")) && ($v << $o) && ($s ^ $v) &&
($s ^ $o)
```

Chapter V - NXT Corpus Format

1. File naming conventions

We assume you are familiar with the terms *observation*, *agent*, *coding*, *signal*, *object set*, *ontology* and *style* as used by NXT data model. If not, these terms are described in the paper *The NITE Object Model Library for Handling Structured Linguistic Annotation on Multimodal Data Sets* (download as PDF: *http://www.ltg.ed.ac.uk/~jeanc/nlpxml2003.final.pdf*) and also in section 2.

General storage policy

Annotations will be stored in the directory named in the *path* attribute of the *codings* element in section 2. Similarly, signals, ontologies and object sets should be stored in the directories as analogously defined in the metadata. There will be no subdirectory structure for any of these directories.

Codings and annotations

For all codings, the directory is specified by the *path* attribute of the *codings* element in the metadata.

There are two kinds of coding: interaction codings and agent codings (see section 2 for details). for interaction codings, the full filename is derived from:

```
observation-name.coding-name.xml
```

Example: `obs1.words.xml`

For agent codings there will be one file per agent according to this pattern:

```
observation-name.agent-name.coding-name.xml
```

Example: *obs1.g.words.xml*

Object sets

for all object sets, the directory is specified by the *path* attribute of the *object-sets* element in the metadata.
The full filename is simply:

```
objectset-name.xml
```

Ontologies

For all ontologies, the directory is specified by the *path* attribute of the *ontologies* element in the metadata.
The full filename is simply:

```
ontology-name.xml
```

Signals

For all signals, the directory is specified by the *path* attribute of the *signals* element in the metadata.

There are two kinds of signals: interaction signals and agent signals (see section 2 for details). For interaction signals, the full filename is derived from:

`observation-name.signal-name.signal-extension`

Example: `obs1.interaction-video.avi`

For agent signals there will be one file per agent conforming to this formula:

`observation-name.agent-name.signal-name.signal-extension`

Example: `obs1.g.audio.au`

⚠**Please note:** The signal name and signal extension are both part of the signal definition in the metadata file.

Styles

For all styles, the directory is specified by the *path* attribute of the *styles* element in the metadata.

The full filename is simply:

`style-name.style-extension`

For example *display.xsl* could be the name of a display stylesheet using the NITE Interface engine. The style name and extension are both attributes of the style in the metadata file.

2. Metadata detailed content description

This page describes in detail each component that goes into a NXT *metadata* file. See *http://www.ltg.ed.ac.uk/NITE/metadata/meta.html* for a more general overview and example metadata files. It will be useful to have an example metadata file as well as a copy of the metadata DTD file handy while reading this guide.

2.1 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">
  ...
</corpus>
```

The important attributes of the *corpus* element are *links* and *type*. The *type* attribute can take one of the two values: *simple* or *standoff*. Simple corpora have one tree of data per observation, whereas standoff corpora have multi-rooted infosets with links between files.

If the corpus is *standoff*, the *links* attribute defines the syntax of the links between the files. It can be one of: *ltxml1* or *xpointer*. The former looks like this:

```
<nite:child href="q4nc4.g.timed-units.xml#id('q4nc4g.1')"/>
```

The latter looks like this:

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

2.2 Reserved Attributes

The reserved attributes section of the metdata file describes the names of those attributes in the NXT corpus that we consider to be *privileged* in some manner. Example of setting reserved attributes:

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

If no *reserved-elements* entry appears in the metadata file, or the specific element is not overriden, the values will default (see table). The *name* values are expected to be namespace-qualified.

	metadata tag name	**Default value**
Element identifier	identifier	nite:id
Element start time	starttime	nite:start
Element end time	endtime	nite:end
Agent responsible	agentname	agent

Identifiers are required on all elements in a NXT corpus. start and end times may appear on time-aligned elements. A time-aligned element in the corpus with the above description might look like this:

```
<word identifier="word_1" starttime="1.3" endtime="1.5">the</word>
```

The attribute describing an agent is a special case. Normally, the agent won't explicitly be named on an element at the level of words since, if agents are involved, the agent will normally be derivable from the metadata and the filename from which the word came. Explicit agent names are mainly useful for elements that describe interactions between agents. If you use the method *getAgentName* on a NOM element, it will transparently use either the agent attribute if one exists or derive the agent from the metadata if it doesn't.

2.3 Reserved Elements

The reserved elements section of the metdata file describes the names of those elements in the NXT corpus that we consider to be *privileged* in some manner. Example of setting reserved element names:

```
<reserved-elements>
    <pointername name="mypointer"/>
    <child name="mynamespace:child"/>
    <stream name="stream"/>
</reserved-elements>
```

If no *reserved-elements* entry appears in the metadata file, or the specific element is not overriden, the values will default (see table). The *name* values are expected to be namespace-qualified.

	metadata tag name	**Default value**
Pointer	pointername	nite:pointer
Child (pointing to remote child)	child	nite:child
Stream element	stream	nite:root

A stream of word elements may look like this with the above example:

```
<stream>
  <word nite:id="word_1">
    <mypointer role="antecedent" href="doc2.xml#ante_2"/>
    <mynamespace:child href="doc2#syllable_1"/>
  </word>
</stream>
```

Pointers and children will have an unqualified *href* attribute that specifies the pointed-to element unless XLink links are being used in which case an xlink:href attribute will be assumed to be used. This attribute name is not changeable. More information on pointers and children below.

2.4 Independent Variables on Observations

A corpus is a set of observations all of which conform to the same basic format. Each observation can have a number of independent variables associated with it, and this part of the metadata file describes those variables for a corpus. An example of an independent observation variable in a dialogue corpus might record whether eye-contact is permitted between agents.

There are three kinds of variable in the NXT world:

- String - free text
- Number - any kind of numeric value is permitted
- Enumeration - only values listed in the enclosed *value* elements are permitted

These three types are also used to describe attributes on elements below. Here is an example of the definition of some observation variables:

```
<observation-variables>
    <observation-variable name="eye-contact" type="enumerated">
        <value>no eye</value>
        <value>eye</value>
    </observation-variable>
    <observation-variable name="temperature" type="number"/>
    <observation-variable name="weather" type="string"/>
</observation-variables>
```

See subsection 2.12 for more information.

2.5 Agents

A corpus is a set of observations all of which conform to the same basic format and have the same number of agents being observed, with the same basic roles. An agent is one interactant in an observation. Agents can be human or artificial. The following table shows how to fit some well-known corpus types into this agent categorization:

Corpus	**Agents**
Map task corpus:	giver,follower
Smartkom:	system,user
Wall Street Journal articles:	writer
Five person discussion:	1,2,3,4,5

For group discussion corpora of mixed size, the user must define agents for the maximum size and fail to use some of them for the observations with fewer people.

Here's a sample agent description (as used for the MapTask corpus):

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

The *name* attribute must be a string with no spaces as it is used to derive filenames.

2.6 Signals

Each observation in a corpus will have been recorded separately using some signal or set of signals. Signals can either be for a single agent (like a video trained exclusively on the route giver), or of the interaction as a whole (like an overhead video that captures the whole group, or at least part of it). All signals for the same observation are assumed to start at the same time. This can be achieved through pre-editing. Note that because there could be several video signals associated with the same corpus, any GVM (video overlay markup) needs to know which signal it applies to.

Signal specification in the metadata file will tell NXT what signals are present, and where they reside on disk. Here's an example of some signal definitions:

```
<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>
```

The *path* attribute on the *signals* element specifies where the media files can be located on disk. If the path is a relative pathname, it is relative *to the metadata file*. Signals are divided into *agent-signals* and *interaction-signals* as discussed above. The *name* attribute of the signal is used in filenames so must not include any spaces.

In this example, imagining there is an observation named *o1* and agents *g* and *f*, we would expect to find the media files:

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

2.7 Ontologies

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 NXT 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 elemnt 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. Note: the *ontology* element can contain any number of *attribute* tags that can define further un-privileged attributes on the ontology where necessary (these attributes are specified in exactly the same way as subsection 2.4.

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>
```

2.8 Object sets

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 NXT 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 refere 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, which are described very much like observation variables. 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

2.9 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">
                <structural-layer name="prosody-layer"
                                  points-to="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. Codings are divided into *agent-codings* and *interaction-codings* in exactly the way that signals (cf. subsection 2.6) 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 (cf. subsection 2.8).

Layers can point to each other using the *points-to* attribute and the name of another layer. There's an alternative syntax for recursive layers (like syntax): the attribute *recursive="true"* on a layer means that elements in the layer can point to themselves. The attribute *recirsive-points-to="layer-name"* means that elements in the layer can recurse but they must "bottom out" by pointing to an element in the named layer.

Layers are further described by their three types which are all described in detail in *this paper*:

- **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.
- **Featural layer** - elemnts can have no time stamps and cannot dominate any other elements - they can only use *pointers*.

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" points-to="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* above) 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.

2.10 Styles

Styles are the files that allow either NIE (NITE interface engine) of OTAB (observable track annotation board) to produce an appropriate display. In the case of NIE, these files are stylesheets and in the case of OTAB they are specification files. Styles may be grouped into views (cf. subsection 2.11). An example of a definition of some styles:

```
<styles path="/home/styles/">
    <style application="nie"
           description="basic syntax coder"
           extension=".xsl"
           name="maptask-editor"
           type="editor"/>
    <style application="otab"
           description="annotation board"
           extension=".xml"
           name="maptask-annotation-board"
           type="editor"/>
</styles>
```

As with many other elements in the metadata file, the *styles* element has a *path* attribute whose value is the directory in which style files for this corpus exist. The *names* of the individual *style*s act as the filename as well as allowing them to be referred to from a views (cf. subsection 2.11). So in this example, we will expect to have a stylesheet in the file */home/styles/maptask-editor.xsl* which is a basic syntax coder. The *type* attribute describes whether the style is an *editor* or just a *display*.

2.11 Views

Views are combinations of displays that combine to produce an editing or display environment for a particular purpose. Views can comprise zero or one NIE displays, zero or one OTAB displays, and any number of video and audio windows. Here's an example combining a styled display and an audio window:

```
<views>
    <view description="basic transcription" type="editor">
        <styled-window nameref="maptask-editor"/>
        <audio-window nameref="audio" sound="yes"/>
    </view>
</views>
```

2.12 Observations

The list of observations tells us the number of actual observations in the corpus and the types of codings that have been done on them. Each observation can contain attributes as defined in the observation variables section above. As an example, an observation list could lok like this:

```
<observations>
    <observation name="q4nc4">
        <variables>
            <variable name="eye-contact" value="eye"/>
            <variable name="familiarity" value="non-familiar"/>
        </variables>
        <user-codings>
            <user-coding coder="cathy" date="sep98"
                            name="games" status="final"/>
            <user-coding coder="gwyneth" date="oct01"
                            name="move" status="final"/>
        </user-codings>
    </observation>
</observations>
```

Each observation in a corpus must have a unique *name* attribute which is used in filenames. After the independent variables are defined for this observation, we have a list of *user-codings*: the actual work that has been done on this observation. The idea is that the *name* of a user-coding points to a coding. The *status* can be one of: *unstarted*, *draft*, *final* or *checked*. If the status is *checked* it is expected that there will be a further attribute *checker* containing the name of the checker.

Index