UNIVERSITY OF JOENSUU
DEPARTMENT OF COMPUTER SCIENCE
Report Series A

Implementation of Two-Dimensional Filters for Structured Documents in SYNDOC environment

Eila Kuikka, Jouni Mykkänen
Arto Ryynänen, Airi Salminen

Report A-1995-4

ACM X.n.n
UDK nnnn.nn
ISSN ...
ISBN ...

Implementation of Two-dimensional Filters for Structured Documents in SYNDOC environment

Eila Kuikka*
Department of Computer Science
University of Waterloo, Canada

Jouni Mykkänen
Department of Computer Science and Applied Mathematics
University of Kuopio, Finland

Arto Ryynänen
Department of Computer Science
University of Joensuu, Finland

Airi Salminen
Department of Computer Science and Information Systems
University of Jyväskylä, Finland

Abstract

Filtering is used to select a subset, corresponding to the information interests of a user, from a set of information items. The information interests are described in a filter which is created to control the selection. In our earlier work we have described a theoretical framework for specifying filters to express content-based and structure-oriented constraints on structured text. In the filters, the information interests of the user are expressed by constraints and annotations on two-dimensional templates. The templates are created from the grammar associated with the structured text. This report describes a prototype for the filtering method in a syntax-directed document processing system called SYNDOC. In SYNDOC, a filter is applied to documents associated with a common grammar. The application of a filter means finding the documents that match the filter. From the user's point of view, filtering a subset of a given document collection consists of the following six steps. First, a filter for a given grammar is defined; second, a directory containing documents associated with the grammar is chosen; third, indexing is applied to the documents (unless indexed documents were chosen); fourth, the filter is applied to the indexed documents of the chosen directory; fifth, the form of the output is defined; and sixth, the filtered documents are displayed in the specified form. In the current phase of the implementation, the matching test is applied to one document at the time, and in case of matching, the document is displayed using the default output form.

*During the author's stay in the University of Waterloo, Canada. Author's permanent address: University of Kuopio, Department of Computer Science and Applied Mathematics, Kuopio, Finland, email: kuikka@cs.uku.fi

1 Introduction

Filtering is used to select a subset, corresponding to the information interests of a user, from a set of information items. The information interests are described in a filter which is created to control the selection. We have introduced in [KS95] a theoretical framework for defining filters for structured text. According to the method, the structures of documents are defined by context-free grammars, and individual documents are represented as parse trees for grammars. The basis of the framework is a text model which uses constrained productions to specify queries, views and text transformations for structured text [SW92, STSM95, ST95].

According to our filtering method [KS95], a two-dimensional template is created using the grammar to show the structure of a set of textual elements, at a chosen level of detail. The template depicts the hierarchical structure of the elements and indicates also optionality, alternatives, and iteration in the structure. For specifying a subset of the elements, described by the template, constraints are added to the template. A filter consisting of constrained templates and annotations for structure elements allows the user to express complex content-based and structure-oriented conditions on the text and define the elements to be retrieved.

The implementation of the filtering method is made in the SYNDOC environment [KP91, KPV94, KP94]. SYNDOC is a prototype for a declarative document processing system where structures of documents are defined by a context-free grammar. SYNDOC uses grammars and their parse trees for inputting, updating and outputting as well as storing and retrieving documents. Filters are generated interactively under the control of the grammar. Filters are matched to indexed documents generated from SYNDOC documents by creating inverted indices for words and all structure elements of the text. The indices keep track of positions and nesting of index terms. If a match succeeds, the original document is retrieved onto the screen for further processing. In the current implementation, only a subset of the filtering capabilities included in the method and described in [KS95] are functional.

The remainder of this report is organized as follows. Section 2 presents a short overview of the filtering method and defines the terminology used in the rest of the report. Section 3 describes the environment of the implementation, and Section 4 the method from the user's point of view. Section 5 presents the technical issues of the system. It describes the architecture of the retrieval system in SYNDOC as well as algorithms and programmes. Section 6 presents some conclusions. It indicates functions that are missing from the system and discusses issues for the future development of the system.

2 Two-dimensional filters for structured text

A theoretical framework for specifying filtering profiles to express content-based and structure-oriented constraints on structured text is described in detail in [KS95]. An overview of the framework is given in this section. The framework is based on the model represented by Salminen and Tompa in [ST95]. This model first defines the structure of text by a context-free grammar, and then describes queries, views, and text transformations by adding constraints to the productions of the grammar. Constrained productions are applied to a parse tree for the grammar. In the following we first introduce the notions of structured text as defined in [ST95], and then the two-dimensional filters as defined in [KS95].

2.1 Grammars as schemas

In grammar-based data modelling, a context-free grammar is regarded as a schema for textual data, and a parse tree as an instance for the schema. (The basic notions of formal grammars are given, for example, in [AU72]). A context-free grammar defines an alphabet consisting of a set of terminal symbols, a set of names called nonterminal symbols to represent structural elements, a distinguished nonterminal symbol called the start symbol, and a set of productions to show in which way structural elements are composed of other elements. The metasymbols are used as operators to indicate iteration, alternatives and optionality. Iteration is denoted by `*` (zero or more times) and `+` (one or more times), optionality by question mark `?`, and | indicates alternatives. A production whose left side is t is called a t-production.

In our filtering approach, a two-dimensional representation of a grammar is used as a template of a filter, and a filter is derived by adding constraints and annotations to the template. To support the definition of clear partitioning hierarchies by grammars and their simple visualization, some rules concerning the productions of a grammar are required. In [KS95], two kinds of productions are allowed in grammars: aggregate productions and generalization productions. An *aggregate production* describes a structure which, at least potentially, is a composite consisting of several components. An aggregate production has a form $t \to x_1 x_2 \ldots x_n$ where $n > 0$ and each x_i is either a nonterminal symbol, or a nonterminal symbol followed by a metasymbol `?`, `*`, or `+`. If $n = 1$, then x_1 is followed by `*` or `+`. A *generalization production* has a form $t \to t'$ where t' is a type, or a form $|t_1 \ |t_2 \ldots \ |t_n$ where $n > 1$ and each t_i is a text type. In generalization productions, the alternate operand (`|`) is placed in front of every text type, not just between text types as usually in grammars. For example, productions of a grammar for an article are represented as follows.

```
article     --> authors date? title content
authors     --> author+
content     --> abstract section+
section     --> heading paragraph+
paragraph   --> |text_para |itemizelist
itemizelist --> itemize+
```

According to this grammar, an article has a list of authors, an optional date, a title and the content. It contains at least one author. The content consists of an abstract and a list of sections. A section has a heading and more than one paragraph and a paragraph can be either a text paragraph or a list of unnumerated text items. In our grammars it is supposed that the unspecified nonterminal symbols (in the previous grammar `date`, `title`, `author`, `abstract`, `heading`, `text_para` and `itemize`) represent word sequences. Hence, for each unspecified symbol t, there is an implicit production $t \to$ `word+`, where the production for `word` produces a terminal symbol. This symbol is accepted as a word in the language for the grammar.

2.2 Types, parts and properties

Given a parse tree for a grammar, each nonterminal symbol of the grammar represents a set of text entities in the hierarchic structure of the tree: the nodes labelled by the symbol (together with corresponding subtrees) in the parse tree. Hence the nonterminal symbols are called *text types*. To be able to consider a production with t on its left side as a mechanism for defining a type t, the grammar is written such that each nonterminal symbol appears only once as the left side of a production. The text entities in the parse tree, i.e., nodes labelled by nonterminal symbols, are called *parts*. However, a single child labelled by a nonterminal symbol is always regarded as renaming a part, not as a separate part itself. Each of the single child successors of a part, labelled by nonterminal symbols, is called a *renaming node* of the part. If x is a part, the subtree X' with x as its root is the *state* of x, and the string produced by concatenating the terminal symbols of X' (from left to right) is the *value* of x. The part is a *part of type t* (or a t part) if t is the label of x or the label of a node x' in the state of x such that the path from x to x' (including x and x') contains no other parts than a part x. If x and x' are two parts such that x' is a node in the state of x, we say that *part x contains part x'*. Part x' may also be x.

Each of the text types t of a grammar can be considered as a logical operation which tests if a part of a parse tree is a part of type t. In addition to types, also other logical operations may be defined for parts. They are all called *properties*. A property is a predicate which may be applied to any part. Properties are used to define text operations in terms of constraints for text type occurrences in productions and templates. Hence properties are expressed in the form $t\{c\}$ where t is a text type and c a constraint. The property $t\{c\}$ is true for a part of type t if the condition described by c holds for the part. The property $t\{c\}$ is always false to parts which are not parts of type t.

2.3 Templates

A *template* is created for one of text types of the grammar to show the structure defined for the text type, at a chosen level of detail. Formally, a template is defined as an ordered labelled tree whose root is labelled by the given text type for which the template is generated. The child nodes of a parent labelled by a type t correspond to the text type occurrences on the right side of the t-production. Each of the nodes is labelled by the type and the associated metasymbol, if there is any.

A template as a tree is visualized such that each node label is written on its own line, and the parent-child relationship is expressed by indentation. The following is a template for an article generated from the previous grammar:

```
article
    authors
        author+
    date?
    title
    content
```

The root of the template is type `article`. The template shows the main components of an article and the structure of the `authors` component.

2.4 Filters

In [KS95] the correspondence between text type occurrences of a template and parts of a parse tree is defined. To set restrictions to parts corresponding to text types of a template, a *constrained template* is formed by adding constraints to type occurrences of the template. A type t associated with a constraint c indicates the property $t\{c\}$. In [KS95] the following properties defining conditions were represented. In these descriptions t and t_1 are text types, s is a character string, n_1 and n_2 are positive or negative integers and $q_1, q_1, \ldots, q_1$ are constraints. The text in the second column defines the cases in which the property is true for a part of the type t.

$t\{s\}$	contains string s
$t\{= t_1\}$	the value of the part t equals to the value of some part of type t_1
$t\{t_1\}$	contains a part of type t_1
$t\{n_1..n_2\}$	the position of the part among siblings of the same type is indicated by the numbers n_1 and n_2
$t\{q_1 \,\&\, q_2 \,\&\, \ldots \,\&\, q_k\}$	all of the properties $t\{q_i\}$ are true
$t\{q_1 \mid q_2 \mid \ldots \mid q_k\}$	at least one of the properties $t\{q_i\}$ is true
$t\{!\, q_1\}$	the property $t\{q_1\}$ is not true

For iterative text type occurrences, expressed by metasymbols `*` or `+` in templates, a quantity constraint may be added to express the number of parts for which the property associated to the text type must be true. The quantity constraint is one of the following: ALL, n, $> n$, $\geq n$, $< n$ or $\leq n$ where n is a non-negative integer.

The semantics of a constrained template is defined in [KS95] by defining those cases where a part of a parse tree matches a constrained template. Constraints are written on the right side of a type occurrence in a template. For example, the following constrained template defines articles where Kero is the first or the second author, and the content consists of an abstract and more than four sections.

```
          article
              authors
                  author+......"Kero" & 1..2
              date?
              title
              content
                 abstract
                 section+......QTY: >4
```

A constrained template with type t as its root specifies a set of parts of type t. A *filter* consists of constrained templates where the specified parts are indicated by annotations. The semantics of filters is described in [KS95] by defining in which cases a part of a parse tree matches an annotation in a filter. Annotations are written on the left side of a type occurrence in a template. A filter may specify conditions concerning the context and content of a set of parts. A *simple filter* is a constrained template where one or more text type occurrences are annotated. An annotation expresses the parts that are filtered, thus, the result of the filtering. Annotation `kero_title` in the following simple filter specifies the titles of articles defined by the previous constrained template, because type `title` is annotated by the name `kero_title`.

```
          article
              authors
                  author+......"Kero" & 1..2
              date?
kero_title.......title
              content
                 abstract
                 section+......QTY: >4
```

A *compound filter* is a sequence of simple filters bound with the use of annotations. The annotation of a filter is regarded as a definition of a *dynamic text type*. The type may be used in constraints of the subsequent filters. Compound filters are needed especially in disjunctive conditions. In the following compound filter, the annotation `art` specifies articles which are either long articles (more than four sections) written by Salminen or short articles (less than or equal to four sections) written by Kuikka. The annotations of the first two filters define dynamic text types `sal_article` and `kui_article`, which are then used to specify the required article in the third filter.

```
sal_article...article
                 authors
                     author+......"Salminen"
                 date?
                 title
                 content
                    abstract
                    section+......QTY: >4

kui_article...article
                 authors
                     author+......"Kuikka"
                 date?
                 title
                 content
                    abstract
                    section+......QTY: =<4

art............article.............sal_article | kui_article
```

In the following compound filter the first filter defines the titles of articles whose first or second author is Kuikka, and where at least one section contains the phrase "text processing system". The second filter defines articles with more than three authors whose titles are same as titles annotated by the first filter.

```
               article
                  authors
                     author+...."kuikka"..............1..2
                  ?date
my_title..........title
                  content
                     abstract
                     section+..."text processing system"

ar.............article
                  authors
                     author+....QTY >3
                  ?date
                  title.........=my_title
                  content
```

More examples about the use of filters can be found in [KS95].

3 The SYNDOC environment

The filtering is implemented in a declarative document processing system whose architecture is described in [KP91, KP94, KPV94]. The prototype system, called SYNDOC (SYNtax-directed DOCument processing system), is based on syntax-directed paradigm. It is implemented in SICStus Prolog (version 2.1 patch #9) [Swe93] with the graphics Manager library to create the X-window user interface and in Tcl (version 7.4) and Tk toolkit (version 4.0) script languages [Ous93]. Tcl (Tool Command Language) [Ous93]) is an interpretive scripting language allowing the possibility to create X window user interfaces. SYNDOC runs in Sun workstations within unix operating system. In SYNDOC the internal representation of a document is a parse tree for a context-free grammar that defines the structure of a document. SYNDOC uses grammars and their parse trees for inputting, updating and outputting as well as storing and retrieving documents. The system is meant to be declarative in the sense that the user is asked only to define what she or he wants. The user does not have to know how to achieve it. In the development of the system, the target has been to focus on the principle of the grammar-based processing as far as possible throughout the system capabilities without any 'ad hoc' extensions, and to create an environment for testing such capabilities.

In the research carried out earlier in the SYNDOC project, grammars and a syntax-directed translation has been used to implement the input and output of text [KP91, KPV94] and the transformation of a parse tree with one structure to a parse tree with another structure [KP94]. In the input phase the input programme proposes the structure of the text to the user and then expands incrementally the parse tree according to the grammar of the document. The system takes care of the validation of the structure. The formatted document for printing is produced by an output programme that is generated from the output grammar of the document. The transformation from a parse tree to another is needed for a document when the user wants to define a new structure for an existing document for the future processing or for the output of the document. The transformations are made by first forming a transformation grammar from grammars for the old and new structures of documents and then generating automatically a transformation programme from a transformation grammar.

This report describes the implementation of the filtering method [KS95] in order to use it for retrieving documents in SYNDOC. The information retrieval from documents is planned to consist of two steps. First, the user will specify her or his information needs in a filter. The filter is created by expanding the template according to the grammar in a way similar to that used in creating a document in the input phase. Second, the user will define the form of the output by a grammar in the same way as is done in printing specific documents. In case that structure modification is needed for the output of a document, the document transformation is specified by transformation grammars. In the current version of SYNDOC, documents are in different files and our filtering method is used to find out in which file a document, described by the filter, exists. The document files are indexed. A filter is matched against the index of a single document. If a document matches it is shown on the screen in the same form as the parse tree of a document for the input phase of documents.

4 Defining and using filters in SYNDOC

We will represent in this section an overview of how to use our filtering method in SYNDOC to select a document which satisfies the constraints defined by a filter. Section 4.1 explains the form of the filter used on the screen. The retrieval process in SYNDOC consists of the generation of a filter, the indexing of documents, the matching of a filter to an indexed document and the displaying of the selected document on the screen. Section 4.2 describes, from the user's point of view, the filter generation phases and gives detailed descriptions for processing commands. Similarly, Section 4.3 describes the indexing process and user commands, and Section 4.4 the matching process and user commands. Section 4.5 explains and shows the display of the selected document.

4.1 Filters on the screen of SYNDOC

On the screen of SYNDOC a filter is represented as a table with five columns (Figure 1). The first column (`RESULT`) defines the result of the filter containing annotations. The annotation

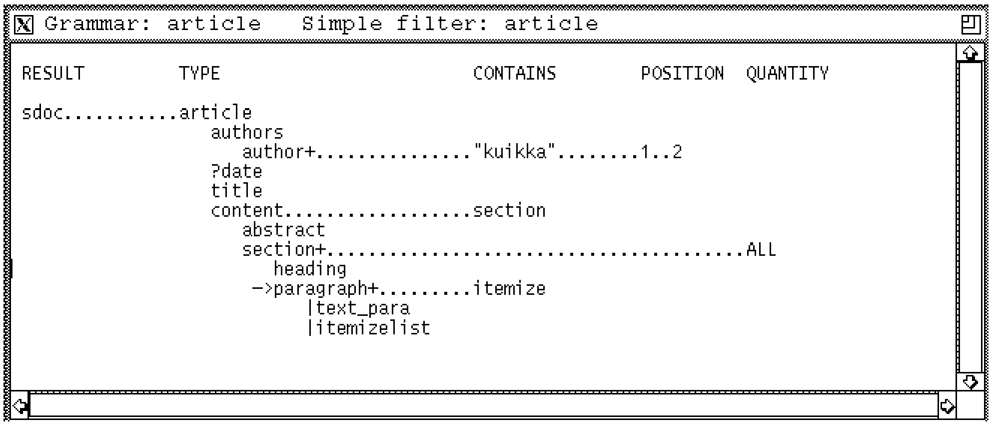

Figure 1: A simple filter on the screen of SYNDOC

is on the same line as its text type occurrence. The second column (`TYPE`) shows a template expressing the hierarchical text structure. In the template every text type occurrence is on its own line and children of a text type node are represented recursively so that every child is also on its own line. The indentation indicates the parent-child relationship in the template. The columns three, four and five (`CONTAINS`, `POSITION` and `QUANTITY`) show containment, position and quantitative constraints, respectively. The constraints are on the same line as their text type occurrence, containment, position and quantity constraints each in a separate column. A containment constraint is a type, a type preceded by $=$, a character string, or a Boolean combination of them. Position constraints are of the form n_1 or $n_1..n_2$ where n_1 and n_2 are positive or negative integers. If both a containment constraint q_1 and a position constraint q_2 has been associated with a type t, it denotes the property $t\{q_1 \,\&\, q_2\}$. Quantity constraints in a form ALL, n, $> n$, $\geq n$, $< n$ or $\leq n$ can be defined only for text types with metasymbols `*` and `+`. In this example a string "kuikka" must be contained in parts of type `author` which, on the other hand, have to be the first or the second `author` part inside the `authors` part in the `article` part. The part of type `content` inside a part of type `article` must contain parts for type `section`. For all `section` parts, `paragraph` parts must contain a part for `itemize` type.

4.2 Generation of filters

The user initiates the generation of a filter by selecting the grammar (a file name containing the grammar), a file name for a filter, and the name of a text type for the root of the template of a filter (Figure 2). The root text type can be any text type in the grammar. Simple filters and compound filters are generated separately. For an existing filter, only the name of the file containing the filter definition is selected, other required information are included in the filter file. By using the **Help** function a list of files in a working directory can be displayed to remind the user about names of existing files.

The initial form of a new simple filter is the root of the template (Figure 3). An arrow as a cursor points to the text type occurrence to be currently processed. A star on a line separates simple filters of a compound filter (Figure 4).

The user can generate or modify a simple or a compound filter by applying to the current text type some of the commands **Zoom**, **Unzoom**, **Constraints** and **Result** (Figure 3). In creating compound filters, the user is able to add simple filters to a compound filter using the **AddFilter** command, and to delete simple filters from a compound filter using the **DeleteFilter** command (Figure 4). Further, the user can change the current text type occurrence by commands **Left**, **Right**, **Down** or **Up** (Figures 3 and 4). And finally, the user can match the filter a document,

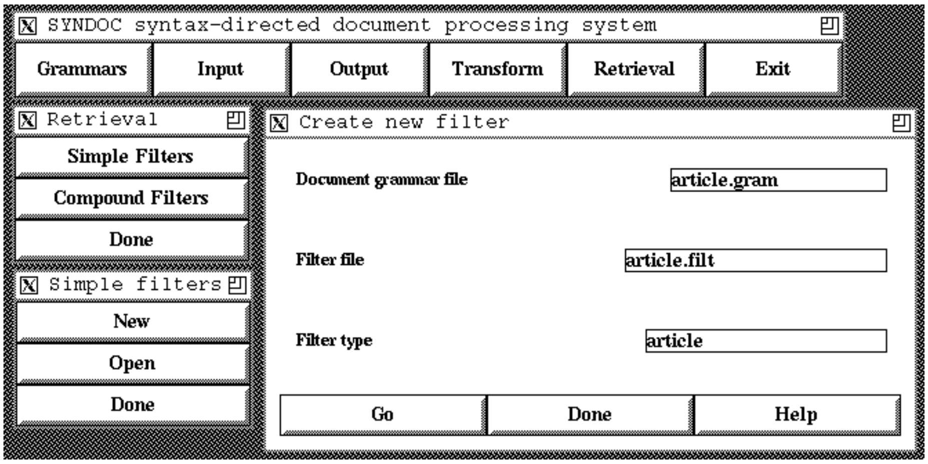

Figure 2: Selecting the grammar, file and root text type for a filter

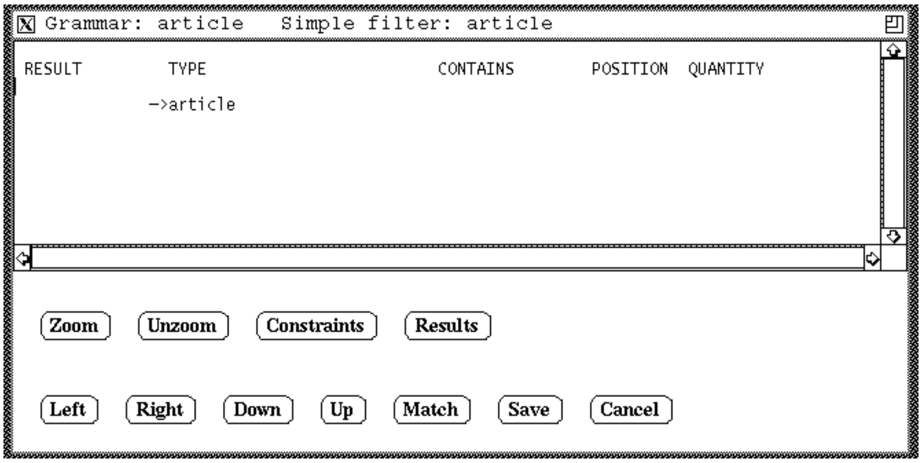

Figure 3: The user interface to start the generation of a simple filter

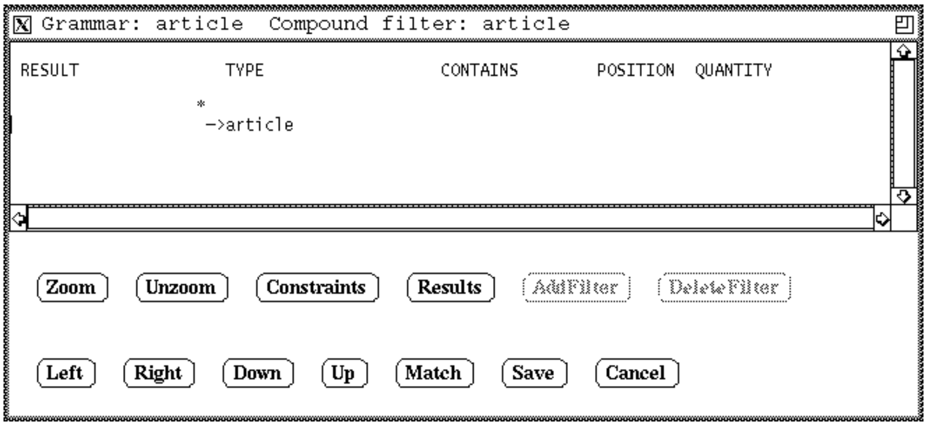

Figure 4: The user interface to start the generation of a compound filter

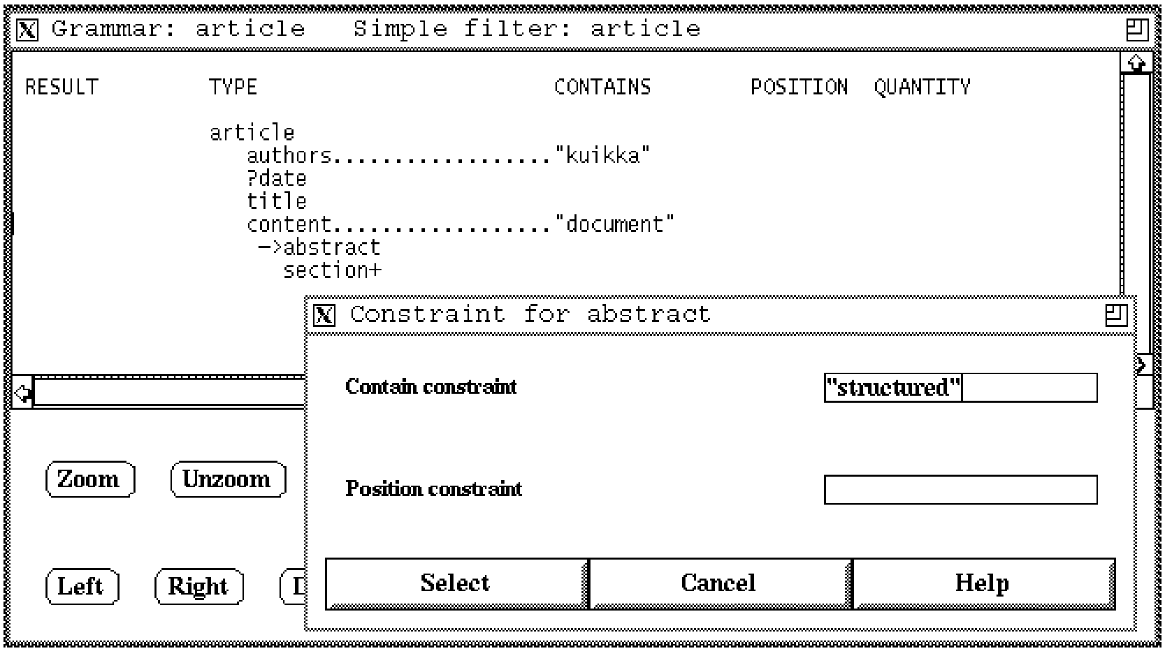

Figure 5: Constraints for a noniterative text type

save the filter into a file and cancel the filter by using the commands **Match**, **Save** and **Cancel**, respectively. A more detailed description for each of the commands is given below.

Zoom. When a text type occurrence of a template pointed at by the cursor is zoomed, a production for the text type is searched from the grammar and text type occurrences of the right side of the production are added to the template as children of the current text type. The cursor is moved to the first child of the current text type. If there is no production for the pointed type the user is informed and the position of the cursor is not changed.

Unzoom. When a text type occurrence pointed at by the cursor is unzoomed it and all its sisters (with the substructures) are removed from the template and the cursor is moved to their parent.

Constraints. When the cursor points at a text type occurrence, constraints can be given on parts of the text type. For a noniterative text type occurrence pointed at by the cursor, only containment and position constraints are asked from the user. Figure 5 shows the window containing fields only for containment and position constraints.

When the user adds text type names in a containment constraint their reasonableness is checked against the grammar, i.e. it is checked that a text type in a constraint is reachable, according to the grammar, from the text type pointed at by the cursor. When the user adds nonreachable text types in a constraint, they are displayed in a message window (Figure 6) and the user is asked to give the constraint again. In the same manner, names for dynamic text types are checked in the compound filter; the name of a dynamic text type in a constraint is required to exist as an annotation in some previous simple filter of the compound filter.

When the cursor points at an iterative text type (a text type with + or *), a containment and a position constraint but also a quantity constraint are requested from the user as shown in Figure 7.

Result. A text type occurrence pointed at by the cursor is annotated by a name of a dynamic

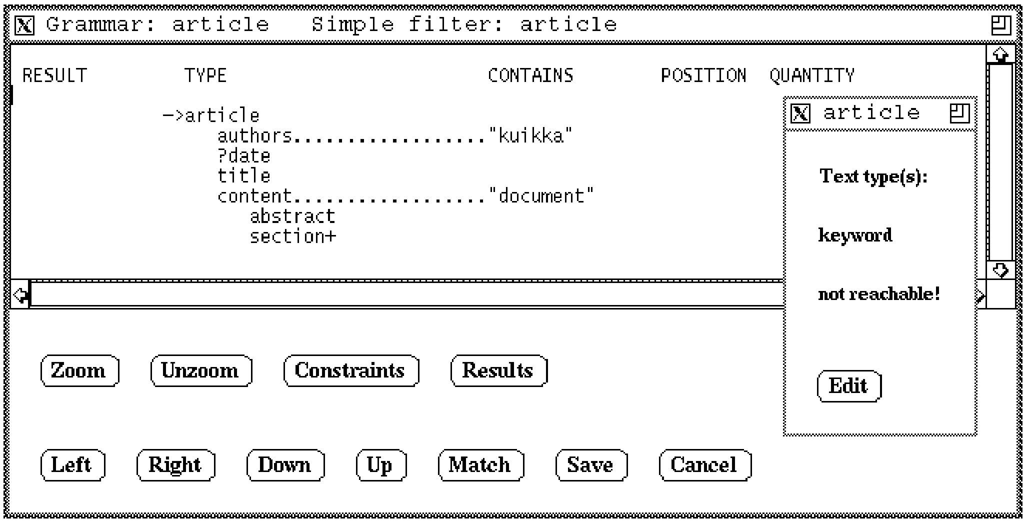

Figure 6: The message about non-reachable text types

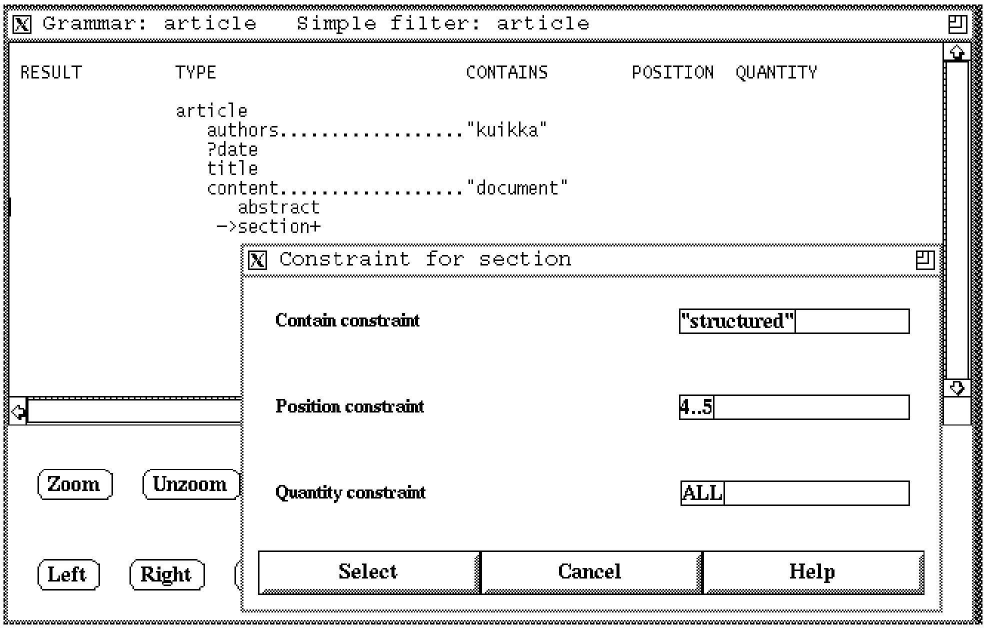

Figure 7: Constraints for an iterative text type

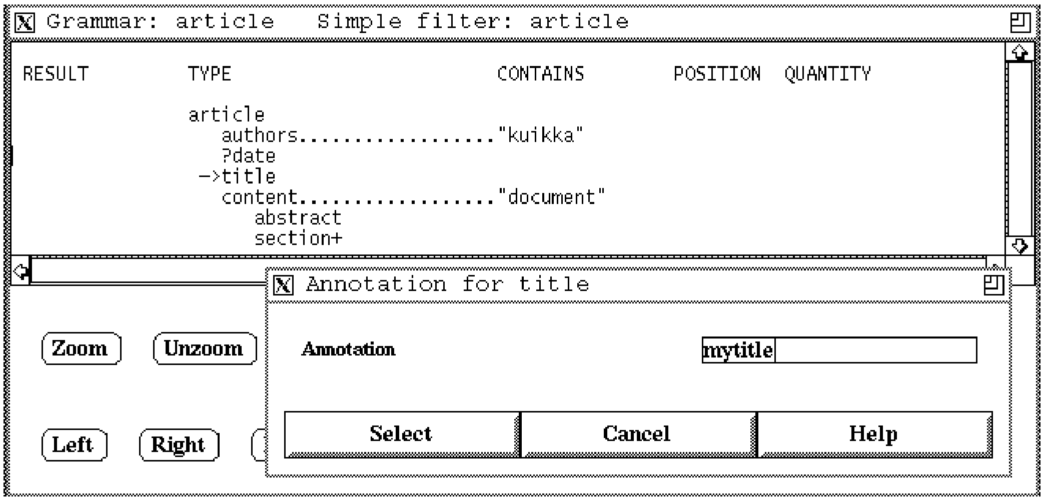

Figure 8: An annotation for a text type

text type (Figure 8). If the user gives as a dynamic text type a name of a text type in the grammar, a message is displayed and the dynamic text type is not added to the filter.

AddFilter. When the cursor points at a star on a line of a compound filter the user can add a new simple filter before the star. Also a star is added in front of the new filter. The user can either give the name of the file that contains a simple filter, or a name for the root text type of a new simple filter (Figure 9). When an existing filter is added, the filter is copied to the compound filter and possibly modified afterwards. The modified filter is saved only as a part of the compound filter. The cursor is now placed to point at the root of the added filter.

DeleteFilter. When the cursor points at a star on a line of a compound filter the user can delete a simple filter beneath the star. Also the star is deleted. The cursor is placed to point at the star before the subsequent filter.

Left. The cursor is moved to the parent of a text type occurrence pointed at by a cursor. If there is no parent a message is displayed and the position of the cursor is not changed.

Right. The cursor is moved to the first child of a text type occurrence pointed at by the cursor. If there is no children a message is displayed and the position of the cursor is not changed.

Down. The cursor is moved to a text type occurrence beneath (on the screen) a text type occurrence pointed at by the cursor. If there is no text type, a message is displayed and the position of the cursor is not changed.

Up. The cursor is moved to a text type occurrence above (on the screen) a text type occurrence pointed at by the cursor. If there is no text type, a message is displayed and the position of the cursor is not changed.

Match. The mathing test is started and the match window containing a menu is displayed (Figure 10). From this menu the user can initiate either the indexing of documents with the use of **Indexing** selection or the matching of a document with the use of **Matching** selection. **Cancel** selection returns the process to the filter generation window.

Save. The filter is written to a file and the filter generation window is closed. The existing filter in the file is replaced by a modified filter.

Cancel. The filter or modifications made in the active filter are ignored and the filter generation window is closed.

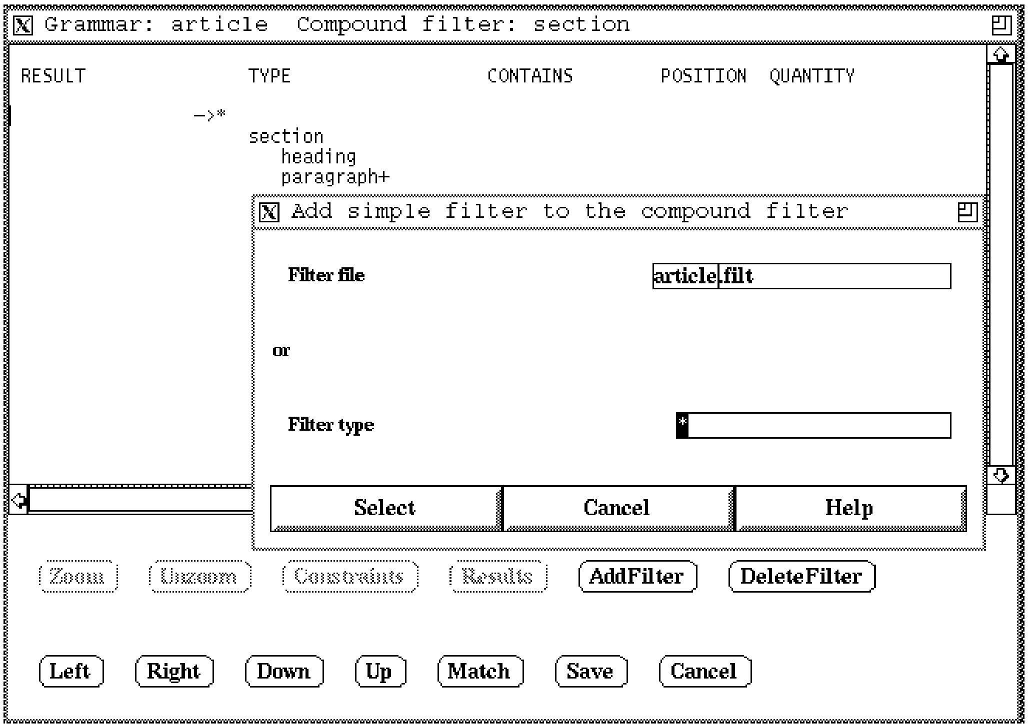

Figure 9: Adding a simple filter to a compound filter

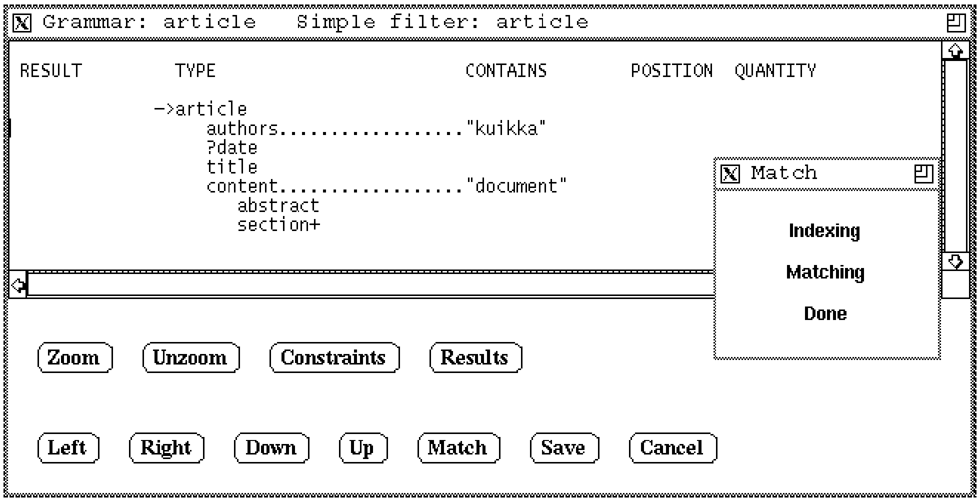

Figure 10: Match window with a menu to start the indexing and matching actions

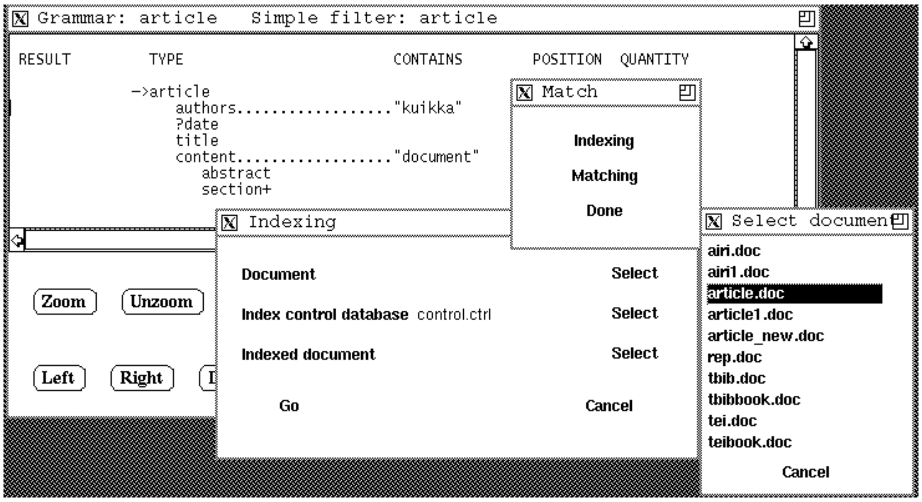

Figure 11: Starting the indexing of a document

4.3 Indexing documents

The indexing of a document is started from the match window (Figure 10) which is displayed after selecting **Match** from the filter generation window. The indexing of documents is started by selecting **Indexing** from the menu in match window and two windows, and indexing window and a document selection window, as shown in Figure 11 are displayed. From the document selection window, the user selects a document to be indexed by double clicking the name of the file in a window. In the indexing window, the file name of the selected document, and default file names of the control file and the indexed document are displayed. In the indexing window the user can select other names for all these files using **Select** selection, start the indexing using **Go** selection or cancel the indexing using **Cancel** selection. **Select** selection shows either the files with the same ending as the default file or, by giving a star as a default file name, all files in the directory. After the indexing is completed, or if the user cancels the indexing, the processing returns to the match window.

4.4 Matching a filter

The matching test is started by selecting **Match** in the filter generation window causing the display of the match window on the screen (Figure 10). Then the actual matching of the active filter is started by selecting **Matching** from the menu of the match window. This causes a document selection window containing a list of names of indexed document files to be displayed as shown in Figure 12. The user starts the matching of the active filter by double-clicking the document name in a list. By selecting **Cancel** the user can cancel the matching test in which case the processing returns to the match window. When the matching test has been executed, the document selection window and the match window are closed.

4.5 Displaying a document

If the matching succeeds the document is displayed on the screen in the input form (Figure 13) and the user can process it in a usual way. The document can be edited or it can only be viewed or browsed. After closing the document window by either saving the document or cancelling the document the user can continue to process the active filter on the screen. If the filter does not match the document a message containing the file name of the document is displayed and the user can continue the processing of the active filter on the screen.

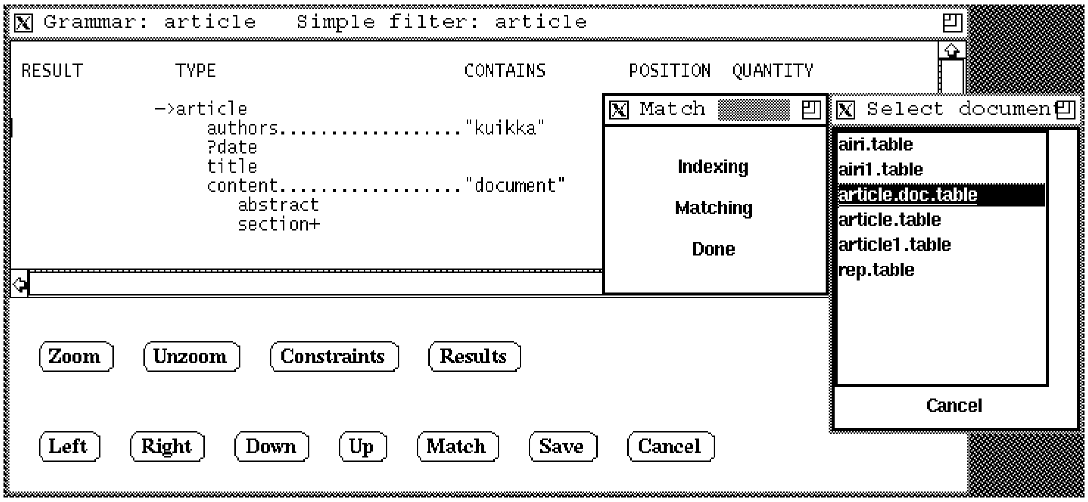

Figure 12: Starting the matching of the filter a document

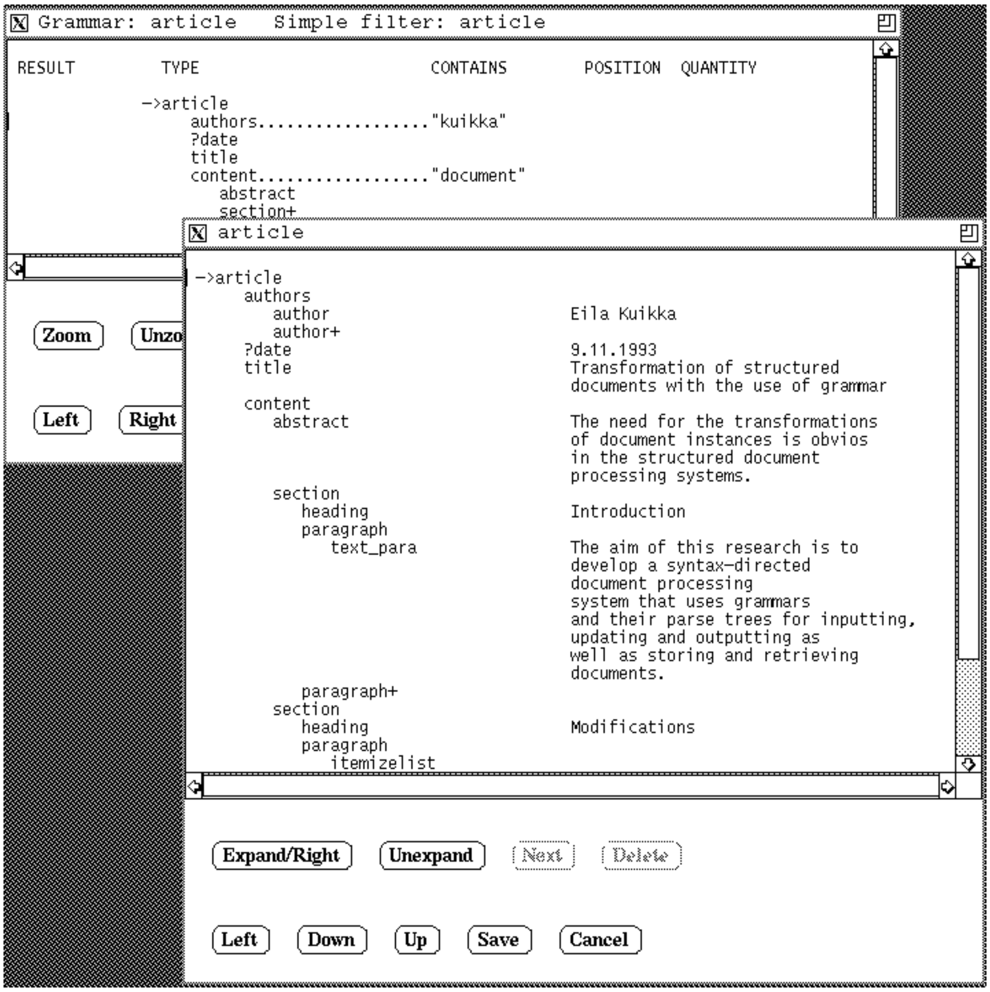

Figure 13: Selected document on the screen

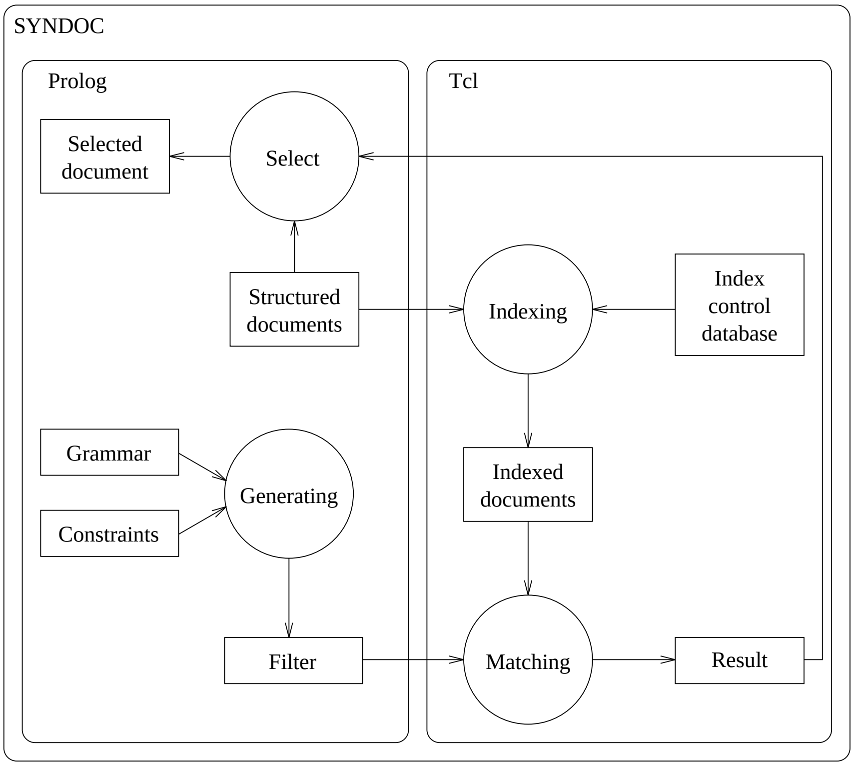

Figure 14: The system architecture

5 Technical implementation of the filtering method

This section contains detailed descriptions about techniques, algorithms, and programmes used in the implementation. First, Section 5.1 describes the architecture of the filtering system and Section 5.2 the user interface of the filtering module in SYNDOC. Then, the rest of the sections explain technical details for different modules of the filtering system. Section 5.3 describes the generation of a filter and contains descriptions for the data structure, operations and the external representation of a filter as well as validity checking that is done during the generation process. Section 5.4 describes the indexing of documents and consists of the representation of the index for SYNDOC documents, the indexing algorithm, and the indexing programme. Similarly, Section 5.5 describes the matching of a filter to a document and contains the descriptions for the matching algorithm and programme. And finally, Section 5.6 describes how the selection of a document has been implemented.

5.1 Architecture of the filtering system in SYNDOC

The filtering system in SYNDOC (Figure 14) consists of four modules: a module for generating filters, a module for indexing documents, a module for matching filters with indexed documents, and a module for displaying the result of matching.

The filter generation module uses an existing grammar and constraints input by the user. The filter is generated interactively and saved into a temporary file to be used in the matching process. The indexing module takes a named document file and a file that contains nonindexed words as the input, and produces a file that contains an indexed document. The matching module uses the temporary filter file and the indexed document file as its input and produces the result of the matching. If the matching succeeds the document selection module displays the selected document on the screen. In other cases the user is informed about the result of the

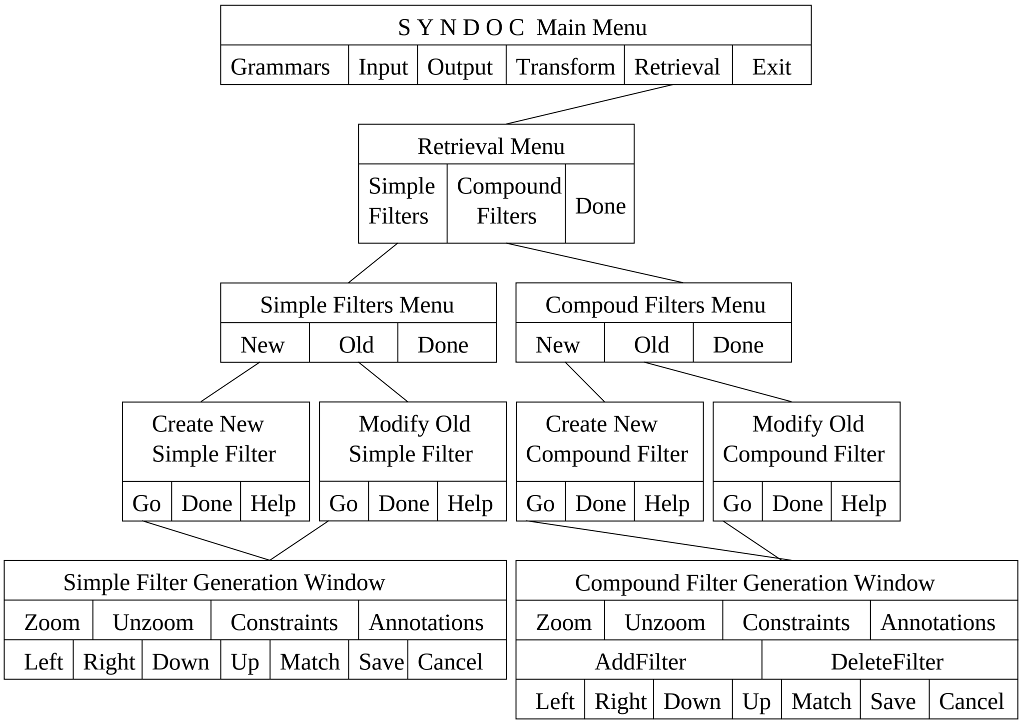

Figure 15: Windows, their menus and relations for the document retrieval in SYNDOC

matching.

The generation of filters and selecting of documents are implemented in SICStus Prolog with the Graphics Manager library to create the X window user interface. The indexing and matching modules are implemented in Tcl and Tk toolkit. At the moment, the filter definition and document selection modules work independently from the indexing and matching modules. Both generate their own X window user interfaces using their own methods and the data is transferred from the Prolog interpreter to the Tcl interpreter via temporary files.

5.2 The user interface for retrieving documents in SYNDOC

The user interface of SYNDOC is window-based and consists of menus as well as question, document, filter and message windows. Menus are used to select operations. Through question windows the user input information to the system, for example, about file names. Message windows inform the user about exceptions in the processing. Document and filter windows display tree representations for their data, either a document or a filter.

The architecture of the windows used in filtering documents is represented in Figure 15. The retrieval of documents is one of the alternatives in the main menu of SYNDOC. From its submenu, the generation of simple filters or compound filters can be selected. For both of these alternatives either a new filter will be generated or an existing filter modified. For each filter a grammar, a file name and the type of the root text type is needed. Figure 2 shows the windows that are used to give this information for a new simple filter. The filter generation window shows a filter in the form that is described in Section 2 with the menu containing operations that can be applied to the filter. In addition to filter modification operations, which are described in the next subsection, a filter can be matched, saved or canceled using operations **Match**, **Save** or **Cancel**, respectively.

5.3 Generation of filters

5.3.1 Data structure and operations of a filter

On the screen the template of a filter is represented as a tree as described in Section 4.1. The internal data structure of the template is the same as the data structure of the SYNDOC document (the detailed description is represented in [KP91, KPV94]). The data structure represents a template as a combination of Prolog terms and lists of lists. Regard the node that is pointed at by the cursor as the current node of the tree. Subtrees whose roots are the current node and its sisters are represented as terms and the rest of the tree, the context of the current node, is represented as lists containing lists. Basic operations of the data structure move elements from a list to another or add or delete terms and elements of lists.

A filter is expanded by **Zoom** operation that adds new structures to the tree according to grammar productions. The traversal in the tree structure is made using operations that move the cursor in the tree from a current node pointed at by the cursor to its first child (operation **Right**), to its parent (operation **Left**), or to a node which is its right sister (operation **Down**), or left sister (operation **Up**). Existing structures can be deleted from the tree using **Unzoom** operation which removes a pointed node and its sisters along with their substructures from the tree. Filters of a compound filter are represented as sister trees. New sister trees can be added using the operation **AddFilter** or existing sister trees can be removed with the use of the operation **DeleteFilter**. The operations **Constraints** and **Result** modify constraints and annotations, respectively, not the tree structure of a filter.

Values for constraints and annotations as an attribute type of data are concatenated as strings to the name of a text type and closed in braces. Their default values are empty strings. Different values are separated by semicolons. For example, in the filter in Figure 1 the text type `article` is represented in a form `article{;;}{sdoc}` because `article` type has no constraints but has an annotation indicated by the name of a dynamic text type `sdoc`. Whereas, the text type `author` is represented in a form `author+{"kuikka";1..2;}{}` because the `author` text type has a containment constraint expressed by a word "kuikka" and a position constraint presented by a numbers `1..2` but has no quantity constraint or no annotation.

5.3.2 External representation of the filter

The external representation of a simple filter in a file is a Prolog term. The character string representation of a term consists of text type names as functors and left and right parenthesis and commas to describe the structure of the term. Thus, the external representation of the simple filter in Figure 1 is the following:

```
article{;;}{sdoc}(authors{;;}{}(author+{"kuikka";1..2;}{}),?date{;;}{},
title{;;}{},content{section;;}{}(abstract{;;}{},section+{;;ALL}{}(
heading{;;}{},paragraph+{itemize;;}{}(|text_para{;;}{},|itemizelist{;;}{})))).
```

Different simple filters of a compound filter are represented in the order they exist on the screen. In a filter file they are separated by two line feeds. The filter file contains, in addition to the external representation of the filter, a prefix that defines the names of the grammar, the root text type of a filter and the text type used for the character string of the content in the grammar.

5.3.3 Checkings in the filter generation

The use of grammar offers many possibilities to help the user. In the current implementation two kinds of checks, when generating a filter, are made with the use of the grammar.

First, a check is done to guarantee that a name of a dynamic text type given as an annotation by the user is not a name of a text type in the grammar. According to the method the name should be different from any text type name in the grammar. The system does not allow illegal names.

Second, a check is done to ensure that a text type given by the user in a containment constraint of the current text type is such that, according to the grammar, parts for this given text type can exist legally inside parts of the current text type. This means that the given text type must be reachable from the current text type. A text type t' is reachable from an other text type t according to the grammar if from t it is possible to derive a sentential form that contains t' (see definitions of the derivation and sentential form for example in [AU72]). Thus, it is possible to generate a parse tree according to the grammar whose root is t and which contains a path from t to t'. The checking of the reachability is made with the use of following algorithm.

Algorithm 1. Checks that a text type t' is reachable from a text type t according to a context-free grammar G.
Input: Text types t and t' and grammar G.
Output: Yes or No.
Method:

```
Step 1:  IF t' = t THEN RETURN Yes.
Step 2:  FOREACH nonterminal t" IN the right side of the production of t
             IF t' = t" THEN
                 RETURN Yes
             ELSE
                 apply Step 2 to a production for t"
             ENDIF
         ENDFOREACH
Step 3:  RETURN No.
```

In our implementation in addition to these two checks using grammars of documents, a check is made of the existence of names of dynamic text types used in annotations to bind separate filters in compound filters. Nonexisting names are not allowed to be added in the containment constraint. Whenever the user adds an annotation the name of the dynamic text type is saved temporarily.

Many kinds of other checks using the grammar of a document could be implemented. Defining filters requires a certain amount of information about the grammar of the document being filtered. It cannot be supposed that the user would know it accurately in all situations. However, this information can be extracted from the grammar and shown to the user. When a current text type is zoomed text types on the right side of the grammar production of the current text type are added to the filter. Thus, the hierarchical structure is taken from the grammar and the user does not need to know it. But when the user adds constraints concerning text types the reachability checking described above is not sufficient to guarantee that the constraint is formed to satisfy the meaning of the user. For example, suppose the grammar defines an article such that type `authors` represents in the grammar both the authors of the article, and the authors of references and both of the `authors` parts are optional. Suppose the user writes the filter

```
authored_article.....article...........authors
```

A check can be made whether the user is aware that an article matches the filter if it contains authors in references, even if no authors are given for the article. The intention of the user probably is to specify the articles which have authors. The filter above is not suitable for the purpose. The user should first annotate the authors of an article, and then define the articles containing annotated parts.

```
'article.gram'.
article.
text.
article(authors([author(text('Eila Kuikka')),'author+']),'?date'
(text('9.11.1993')),title(text('Transformation of structured
documents with the use of grammar')),content(abstract(text('The
need for the transformations
of document instances is obvious
in the structured document
processing systems.')),[section(heading(text('Introduction')),[p
aragraph(text_para(text('The aim of this research is to
develop a syntax-directed
document processing
system that uses grammars
and their parse trees for inputting,
updating and outputting as
well as storing and retrieving
documents.'))),'paragraph+']),section(heading(text('Modification
s')),[paragraph(itemizelist([itemize(text('reordering elements')
),itemize(text('deleting elements')),itemize(text('adding elemen
ts')),itemize(text('renaming elements')),'itemize+'])),'paragrap
h+']),'section+'])).
```

Figure 16: A SYNDOC document

5.4 Generation of an index

In the indexing module the inverted index for documents is made using the method represented by Burkowski in [Bur92]. An inverted index is a function that maps the index terms into positions in the documents where the terms occur. Index terms being searchable words and structure elements of the text are defined as contiguous extents (either word extents or element extents). For each index term a concordance list is generated that keeps track at the position and nesting of various contiguous extents. Thus, elements of the concordance list indicate occurrences of words or parts of text types. We will call those lists later occurrence lists. The occurrence of a word is specified by an integer (the position of the word in the text) and the occurrence of a part is specified by two integers, the first integer for the position of the first word of the part and the second integer for the position of the last word of the part. Elements of occurrence lists are ordered by the first (or only) integer. The indexing creates occurrence lists using an index control database containing words which are not indexed (so called stopwords).

The indexing programme is controlled by the Tcl programme, which creates a common X-window user interface for indexing and matching programmes. The programme creates a menu from which the indexing programme is called. A document file for indexing is selected from a list that is generated by another Tcl programme.

5.4.1 Description of the index for a SYNDOC document

The content of a SYNDOC document is represented as a Prolog term. The character string representation of a term (Figure 16) consists of text type names (with or without metasymbols or apostrophes) as functors, left and right parenthesis and commas to describe the term structure, left and right brackets for lists of parts for a same text type, and characters, spaces and line feeds for the content (the content is not hyphenated). The file of a SYNDOC document (Figure 16) contains a prefix that defines the names of the grammar of the document, the text type for the root of the parse tree, and the text type for character strings in the content of a document.

The indexing demands (but does not check) that a document is complete, i.e. ground terms (atomic text types) in the content of the document are not allowed in a document. For a list of parts for a same text type a ground term with a list symbol "*" or "+" indicates only the end of the list and is not a functor for a part of the content.

The words (except stopwords) of the content included in parts for a text type indicating character strings of the content are numbered to create occurrence lists for words and parts for text types in a document. Because there is no need to index all words (for example, 'the') the index control database is used to reject words that are not significant considering the search. Thus, if the index control database is used an indexed document is not complete and the original document cannot be created from its indexed version. Elements in occurrence lists generated for the document in Figure 16 will start as follows.

```
document:           article(authors([author(text('Eila Kuikka')) ...
word occurrences:                                  1    2
part occurrences:   1,54    1,2      1,2
```

5.4.2 Indexing algorithm

Algorithm 2. Indexing of a SYNDOC document to create occurrence lists for words and parts of text types of a document. Unindexed words are listed in an index control database.
Input: A structured SYNDOC document, an index control database.
Output: Occurrence list tables.
Method:

```
set global word counter N = 1
FOREACH contiguous extent IN document
  CASE contiguous extent
    text type:            set start number to N
    content text type:    skip
    word:                 set word number to N and set N = N + 1
    stop word:            skip
    end of text type:     set  end number to N - 1
    others:               skip
  END CASE
END FOREACH
```

The "`end of text type`" character is the right parenthesis. The "`others`" case takes care of left and right brackets and atomic terms, i.e. text types indicating the end of a list for parts of the same text type (a text type with a + or * metasymbol).

The beginning of the occurrence lists for a document in Figure 16 are represented in Tables 1 and 2. The first table presents words and their occurrences, and the second table presents text types and occurrences of parts of text types.

5.4.3 Indexing programme

The indexing programme processes each line of a document separately. Words are numbered, lowercased (e.g. 'A' is changed to 'a') and written along with its position number into a word occurrence table. If the word exists in the index control file it is neither numbered nor added to the table. The name of the text type for a part and the position number of the first word of

Table 1: The beginning of word occurrence list for a document in Figure 3

Word	Occurrences of words
11	4
1993	5
9	3
adding	51
aim	23
deleting	49
develop	27
document	14 19 29
documents	8 45
eila	1
elements	48 50 52 54
...	

Table 2: The beginning of part occurrence list for a document in Figure 3

Text type	Occurrences of parts of text type
article	1,54
authors	1,2
author	1,2
content	11,54
date	3,5
heading	22,22 46,46
itemize	47,48 49,50 51,52 53,54
itemizelist	47,54
...	

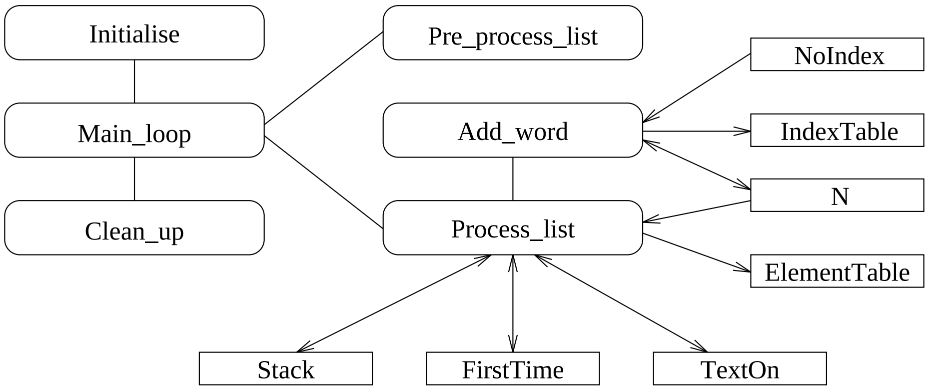

Figure 17: The structure of the indexing programme

the part will be pushed onto a stack. After the last word of the part has been read, the element on the top of the stack (a text type name and the position number of its first word) is removed. The text type name with its start and end position numbers is written into the part occurrence table. If a text type name already exists in the table only the position numbers are added. In the word occurrence table (see Table 1), every word is followed by sequence of numbers indicating its occurrences in a document. In the part occurrence table (see Table 2), the text type name is followed by a sequence of pairs of integers separated by a comma for occurrences of parts of the text type indicating their occurrences. Both tables are written into a file. The file contains as a prefix the name the grammar file of the document, every word and its occurrence list in the word occurrence table separated by a line feed and every text type name with its occurrence list in part occurrence table separated by a line feed. Two line feeds separate the prefix and the these two tables in a file, respectively.

The parameters for the indexing programme are names of a document file and an index control file. The index control file is optional. The format of the document file is as described in Figure 16. The index control file contains words separated by line feeds.

The structure of the programme is represented in the Figure 17. In the main loop, the programme processes each separate input line. After the procedure *Pre_process_list* has removed all question marks to avoid backtracking confusions the main loop splits the input line into a list using a left parenthesis, a comma, a period and an apostrophe as separating characters. An element of the list is either a text type name, a sequence of characters (not containing separating characters) or a sequence of backtracking characters (i.e. a right parenthesis). The list and the name of the text type for the character strings of the content of a document are given as arguments to the indexing procedure *Process_list*. The indexing procedure checks every item from the list. If the item is not empty, it is lowercased and trimmed. Words are numbered and stored to the word occurrence table. Names of text types are pushed onto the stack with position numbers of their first words. When backtracking, elements (the name of a text type and a start position number) are popped from the stack and stored with position numbers of their last words to the part occurrence table. The content text types controls whether the processing word is a word or a name of a text type. If a text type or a word is already in the table, the new position number or the position number pair is added to the corresponding place to the table. The indexing programme procedures (Figure 17) are as follows.

Initialise: This procedure checks the existence of needed parameters, opens the files and sets the input streams.

Main_loop: The global variables are initialised. The index control database, if used, is read into a list. Every separate line from the input stream is processed calling first *Pre_process_list* procedure and then *Process_list* procedure to do the indexing. If the stack used by the procedure is empty, the indexing succeeds and both, `IndexTable` and `ElementTable` tables are written to the output stream. Otherwise, an error message is written.

Pre_process_list: The procedure gets a text variable (a line from a SYNDOC document) as a parameter and removes all question marks '?' and returns the modified text.

Process_list: The procedure does the indexing for words and parts of text types for a given list. A parameter is used to identify the content text type, which is used to recognise words of the content from a list. The `ElementTable` is updated by this procedure. The `IndexTable` is updated calling procedure *Add_word*.

Add_word: The procedure adds a new word into the `IndexTable` table. If the word is found from the global list `NoIndex`, the word is not added to the word occurrence table. If the word does not exist in the table, it is added to the table with its position number. If the word exists in the table only its position number is added to the table.

Clean_up: Closes open files.

5.5 Matching of a filter

The matching of a simple filter to an indexed document means that the filter is matched to all parts of the root text type in the template of the filter. The matching of a compound filter checks all parts of the root text type in the last filter of a compound filter. Matching a filter is implemented in a top-down manner for a template of the filter and for a document. Single filters of a compound filter are checked sequentially starting from the first. If a single filter matches to a part of its root text type a temporary table containing occurrence lists for parts for annotated text type occurrences in a filter is created. The result of matching informs the user if the filter matches a document. The answer of matching contains a positive result and the name of the file of the matched document or a negative answer and the name of the file of the unmatched document.

The matching programme is controlled by the Tcl programme, which creates a common X-window user interface for indexing and matching programmes. The programme creates a menu from which the matching programme is called. The indexed document file is selected from the list generated by another Tcl programme.

5.5.1 Matching algorithm

Algorithm 3. Matching a filter against an indexed SYNDOC document.
Input: An indexed document containing word and part occurrence tables, a simple or compound filter.
Output:
1) If the filter matches the document a string 'MATCH' and occurrence integers for the matched part of the root text type of the filter.
2) If the filter does not match the document a string 'NO_MATCH'.
Method:

```
FOREACH  filter IN compound filter
  FOREACH part corresponding the root type of the filter
          found from the part occurrence table
    WHILE text type occurrences in the filter or FAIL
      IF part  corresponding the text type occurrence is the root or
         includes to the part of its parent text type occurrence
```

```
        AND
        check constraint
      THEN
        save annotation if exists
      ENDIF
    ENDWHILE
    IF NOT FAIL THEN
      make annotations
    ENDIF
    add result to the result list
  ENDFOREACH
ENDFOREACH
display the last item from result list
```

In this algorithm "`save annotations`" stores occurrence lists of annotated text type occurrences temporarily and "`make annotation`" moves stored occurrence lists of annotated text types to an annotation table. The following algorithm is for "`check constraints`" action.

Algorithm 4. Checking constraints of a text type occurrence.
Input: Constraints and occurrence numbers of a part for the text type, a list of words in the word occurrence table, list of a text type names in the part occurrence table and a list of dynamic text type names of annotated text type occurrences.
Output: Boolean values T (true) or F (fail).
Method:

```
define the operand between the different items of the constraint
set result = F
FOREACH separate item IN constraint
  CASE item
    word:          IF word in the word list THEN
                       set result = check word
                   ELSE
                       set result = F
                   ENDIF
    annotation:    IF annotation in the annotation list THEN
                      set result = check annotation
                   ELSE
                      set result = F
                   ENDIF
    text type:     IF text type in the text type name list THEN
                      set result = check element
                   ELSE
                      set result = F
                   ENDIF
    otherwise:     result = F
  ENDCASE
  IF 'negation' in front of the item THEN
     result = NOT result
  ENDIF
  IF result =  T THEN
    IF operand in constraint is 'or' THEN RETURN T
```

```
      ELSE IF operand in constraint is ' ' THEN
              check sequentiality of the item
      ENDIF
    ELSE IF operand in constraint is 'and' OR ' ' THEN RETURN F
    ENDIF
  ENDFOREACH
  RETURN T
```

In this algorithm "`check word`" checks whether the word is contained in the given part; "`check annotation`" determines if a part of an annotated text type is contained in a part of the given text type, and "`check element`" determines if a part of the text type is contained in a part of given text type. If a phrase (a sequence of words) exists in the constraints a space is used as an operand to indicate that words of the phrase must follow each others in a document. To check this in "`check sequentiality of the item`" the start position of the part occurrence is updated.

As an example of the matching process, let us consider the matching of the following filter the document in Figure 16.

```
section..........."adding" & "elements"
```

Beginnings of occurrence tables of the article in Figure 16 are represented in Tables 1 and 2. From Table 2 we will see that for the `section` text type there exists two parts in the document. The first part starts from the word whose position number is 22 and ends at the word with the position number 45. Correspondingly, for the second part the start and end positions are 46 and 54. The first part from words 22 to 45 fails because, as we will see from Table 1, the word "adding" has the position number 51 which is not included in the interval [22,45]. The second word in not checked because the operand is "and" which demands that both words be included in the interval. The second part starting from the position number 46 and ending at 54 succeeds because the word "adding" at the position number 51 and first occurrence of the word "elements" (at the position numbers 48, 50, 52 and 54) are both included in the interval [46,54].

5.5.2 The matching programme

The structure of the matching programme is presented in Figure 18. The parameters of the programme are the names of the indexed document file containing occurrence tables and the file containing a simple filter or a compound filter. The matching programme returns the occurrence integers of the part for the root text type of the filter and the result of the matching (a string "MATCH" or "NO_MATCH").

All occurrence integer pairs for parts of the text type of the root of a simple filter are retrieved from the part occurrence table `ElementTable`, if any exists. The filter is checked against these occurrences, separately. The annotations of text types are recorded into the annotation table after the filter matches. The matching result of every processed filter is added to the result list. For a compound filter results are added to the list in the processing order of simple filters. Thus, the last item in the list indicates the result of the matching which is saved to a temporary file.

In the current implementation the matching process works with one document. However, the system could be changed to select a list of documents, match them sequentially and display to the user a list of names of matched documents. Procedures of the programme (Figure 18) are as follows:

Initialise: The procedure checks parameters, opens input files and sets input streams and variables.

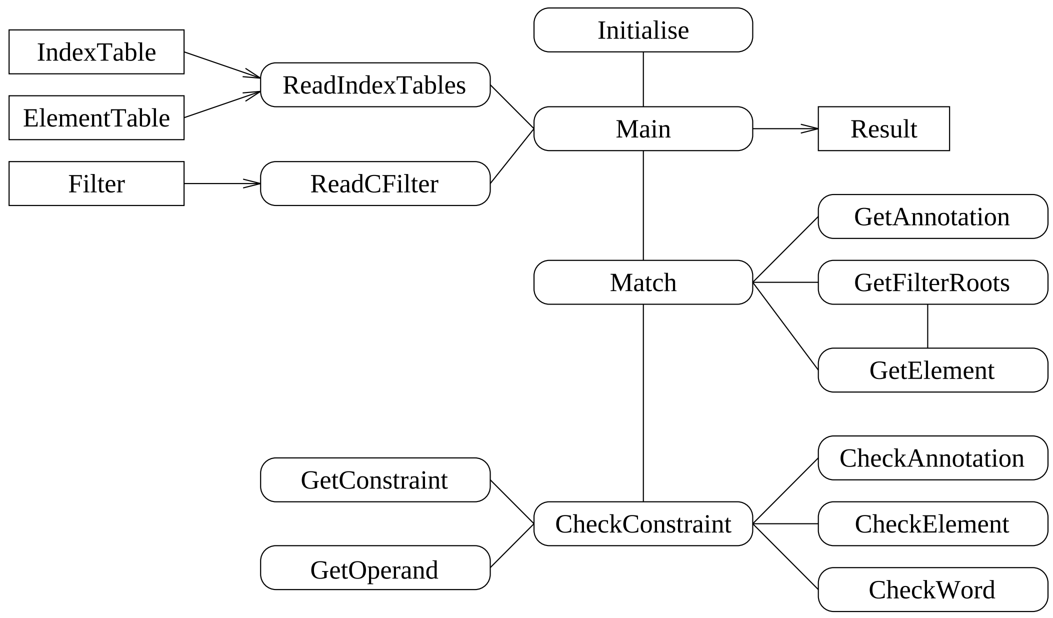

Figure 18: The structure of matching programme

Main: The procedure calls *ReadIndexTables* and *ReadCFilter* procedures. The procedure processes every simple filter from the compound filter by first splitting the filter into a list using the left parentheses as a separating characters and then calling the procedure *Match* using this list and the default occurrence numbers for the database as parameters. The result of every filter is added to the result list. The procedure writes the result of the last (or only) filter with the name of the document file to a temporary file.

ReadIndexTables: The procedure reads the word index table and the element index table from a given file into the corresponding global variables `ElementTable` and `IndexTable`.

ReadCFilter: The procedure reads a simple filter or a compound filter from a given file into the filter list.

Match: The procedure processes the filter list sequentially. The position numbers of the parts of the root text type of the filter (an element of the filter list before the filter) are retrieved by calling procedure *GetFilterRoots*. If an empty string in returned the filter fails. Those occurrences of parts that are found are checked separately. Every filter in a filter list is split using a comma as a separating character into a sublist and the sublist is processed sequentially. Procedure *GetElement* is called to extract the name of the text type from the sublist element. The part of the text type is checked from the `ElementTable` to ensure that the part is a part for the root text type or that it is included in its parent part. The filter fails if this test fails. The procedure *GetAnnotation* is called to extract the annotation from the element of the sublist. The procedure *CheckConstraint* is called to validate constraints on the element of the sublist for a part of the text type occurrence. Results of the matching are stored in the result list which is returned.

GetFilterRoots: The procedure retrieves the text type name and occurrence integers of parts for the root text type of the filter from `ElementTable`. Its parameters are the default occurrence numbers of the database and a list of names of text types in the database. The empty string is returned, if the root text type does not exist in the database.

GetElement: The procedure extracts the text type name from a given string.

GetAnnotation: The procedure extracts the annotation from a given string. If found, it is returned, otherwise the empty string is returned.

CheckConstraint: The procedure checks the constraint of a given list element inside parts for its given text type occurrence. Its parameters are lists of words and names of text types of the database and annotations of the filter. The constraint is extracted calling the procedure *GetConstraint* and the operand calling *GetOperand*. The constraint items are found using the operand as the separating character and every item is checked separately using the procedures *CheckWord*, *CheckElement* and *CheckAnnotation*. If the constraint is empty, the Boolean value T is returned.

GetConstraint: The procedure extracts the constraint from a given string.

GetOperand: The procedure defines the used operand from a given constraint. The found operand or "NONE" is returned.

CheckWord: The procedure checks the position of given word against given the start and end position numbers of the given part and the given operand. The Boolean value T or F is returned.

CheckElement: The procedure checks the position numbers of parts of a text type against given the start and end position numbers of the given part and the given operand. The Boolean value T or F is returned.

CheckAnnotation: The procedure checks the position numbers of parts of annotated text types against the start and end position numbers of the given part and the given operand. The Boolean value T or F is returned.

5.6 Selecting and displaying a document

The document selecting module retrieves the answer from the matching process. If the document matches, it is retrieved using its file name and shown on the screen. The document is shown using the document window of SYNDOC whose generation and operations are implemented in the input module of SYNDOC [KPV94]. The user can edit the document and save it in the modified form or the user can only view or browse the document. If the document does not match, a message is displayed to indicate the file name of the document.

Instead of seeing the whole document in the input form, the user would perhaps be interested to see it in some formatted form or see only a part of the document. Using the formatting and translation methods of SYNDOC [KPV94, KP94, KP95] it is possible produce other forms for the selected document. If only a special layout is needed the user defines a layout grammar and SYNDOC will produce the new representation for the selected document using a programme that is generated from the layout grammar. If only some parts of a document are displayed, the selected document is first transformed to satisfy the new structure defined by a new grammar. A new layout of the transformed document is defined using the new structure grammar. Document transformations in SYNDOC are defined with the use of grammars of the old and new document representations automatically or with the help of the user, depending on the differences in old and new structure of the document. From the definitions SYNDOC produces the transformation programmes and makes the transformations automatically.

6 Conclusions

This report introduced the implementation of the filtering method for the structured text described in [KS95] in SYNDOC environment. The prototype implemented only a part of functionalities of the method. The generated filters contain all features described in [KS95] but the matching of a filter was implemented only for containment constraints without position and quantity constraints. Further, annotations of filters were not used to define the parts to be retrieved; instead a whole document that matches a filter was displayed. The filtering was used to check if a filter matches a given indexed SYNDOC document. If the matching succeeds the whole document was shown on the screen.

The implementation was created to test ideas of the method, but it has many restrictions. As we mentioned above, the prototype needs to be developed to express position and quantity constraints in matching filters, to display only annotated parts of documents, to allow the user to defined that layout of selected parts, and to filter a specified set of documents. In the future our aim is to execute user tests for the method using the prototype. For this purpose it should be changed in two ways. First, documents used by the prototype should be SGML tagged documents. Using SGML tagged texts experiments could be made with large existing document collections. Second, the user interface of the prototype should be more flexible than it is in the current version. In order be able to use the whole power of the X window user interface we have already started the development of the new interface using Tcl. This kind of a possibility exists, in some Prolog compilers, for example, in the new version 3.0 of SICStus Prolog compiler and in the BinProlog compiler. Our aim is to use the latter one.

Acknowledgements

The research for this report was carried out as a subproject of the research project on Structured Documents and Text Databases financed by the Academy of Finland. The authors would like to thank all persons who have given helpful advices during this work, especially Martti Penttonen, Jorma Sajaniemi, Pekka Savolainen, Heikki Mannila, Pekka Kilpeläinen, Erja Nikunen and Helena Ahonen for their constructive criticism and comments in the different stages of this research. Special thanks go also to Frank Tompa for the possibility of one of the authors to continue the work in his department and to Betty Blake for revising the English manuscript. This work was also supported by grants of the Saastamoinen Foundation and the Emil Aaltonen Foundation, which are gratefully acknowledged.

References

[AU72] A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling, Vol. I: Parsing.* Prentice-Hall, Inc., Englewood Cliffs, N.J., USA, 1972.

[Bur92] F.J. Burkowski. An algebra for hierarchically organized text-dominated databases. *Information Processing & Management*, 28(3):333–348, 1992.

[KP91] E. Kuikka and M. Penttonen. Designing a syntax-directed text processing system. In K. Koskimies and K.-J. Räihä, editors, *Proceedings of the Second Symposium on Programming Languages and Software Tools*, pages 191–204. University of Tampere, Department of Computer Science, A-1991-5, Tampere, Finland, 1991.

[KP94] E. Kuikka and M. Penttonen. Transformation of structured documents with the use of grammar. *Electronic Publishing - Origination, Dissemination and Design*, 6(3):373–383, 1993 (published April 1994).

[KP95] E. Kuikka and M. Penttonen. Transformation of structured documents. Technical Report CS-95-46, University of Waterloo, Department of Computer Science, 1995.

[KPV94] E. Kuikka, M. Penttonen, and M.-K. Väisänen. Theory and implementation of SYNDOC document processing system. In *Proc. of the Second Int. Conf. on Practical Application of Prolog*, pages 311–327, London, UK, April 1994.

[KS95] E. Kuikka and A. Salminen. Two-dimensional filters for structured text. Technical report, University of Waterloo, Department of Computer Science, 1995. Submitted for publication.

[Ous93] J.K. Ousterhout. Tcl and the tk toolkit. Draft version of ISBN 0-201-63337-X, 1993.

[ST95] A. Salminen and F.Wm. Tompa. Grammars++ for modelling information in text. Technical report, UW Centre for the New OED and Text Research, University of Waterloo, Department of Computer Science, 1995. Manuscript, unpublished.

[STSM95] A. Salminen, J. Tague-Sutcliffe, and C. McClellan. From text to hypertext by indexing. *ACM Transactions on Information Systems*, 13(1):69–99, 1995.

[SW92] A. Salminen and C. Watters. Two-level structure for textual databases to support hypertext access. *Journal of the American Society for Information Sciences*, 43(6):432–447, 1992.

[Swe93] Swedish Institute of Computer Science. *SICStus Prolog User manual and SICStus Prolog Library manual (Version 2.1)*, 1993.