Logical representation of XML based on relations
Fortgeschrittenenpraktikum

Amelia Schultheis
Computer Science
University of Munich
Germany

Betreuer:
Prof. François Bry and Dr. Phil. Uta Schwertel
Teaching and Research Unit "Programming and Modelling Languages"
Institute for Informatics
University of Munich

Munich, June 9, 2005

Contents

List of Figures

List of Tables

Abstract

XML is the language of choice for expressing data on the Web. With XML, nested data structures, called "elements" can be expressed. The work reported about here investigates how an XML document can be expressed in terms of binary or, more generally n-ary, relations instead of nested elements. In other words, this report investigates relational and flat representations of the nested structures of XML documents.

This work is a contribution of the project VOXX ("Verbalization of XML and XCerpt") aiming at verbal respresentations of XML documents and queries in the Web query language XCerpt. Indeed, nested structures like XML documents and XCerpt queries are extremely inconvenient for a verbal rendering, while flat relational specifications are well-suited for that.

1 Introduction

1.1 Motivation of VOXX

This work is part of the project VOXX (Verbalisation of XML and XCerpt) [1]. The goal of VOXX is to express and query XML-files, such as web data, using spoken language. This would be useful in many fields, e.g. for warehousing: Somebody could simultaneously drive a fork lift and query orally the content of the stock. It is also of interest for mobile communication: One could orally ask a cell phone for information, such as an address, instead of surfing the web with a rather limited display and keyboard.

The examples illustrate that user-friendly modes of interaction are an essential prerequisite for a broad acceptance of mobile computing technologies working with Web data. As mobile devices become smaller in particular spoken natural language becomes an increasingly attractive front-end. Moreover, end users often have little background in *formal* Web languages (like XML) whereas natural language is familiar and thus more readily accepted as an interface language to query the Web and to render Web data. However, due to the inherent ambiguity of natural language full computational processing of complex sentences and texts is not yet realistic, and in addition would be too time and/or resource consuming.

The project VOXX therefore uses a *controlled* natural language to express Web queries and Web data, called *verbalisation* of Web queries and data. A controlled natural language is a subset of a natural language whose grammar and vocabulary have been restricted in order to reduce and/or eliminate both ambiguity and complexity. VOXX works with (adaptations of) the controlled natural language Attempto Controlled English (ACE) developed at the University of Zurich [2]. ACE can be automatically and unambiguously translated into first-order logic (FOL), i.e. into a formal language that allows for automatic post-processing. In sum, the simplifications suggested by ACE allow to write precise, non-ambiguous texts that are readable by humans and automatically processable with a computer.

1.2 Intended Applications

Since Web data and queries are based on formal languages like XML or XCerpt [3] that are different from first-order logic the idea of VOXX is to build a formal bridge by defining a common underlying formal language based on logical relations of different arities to and from which both XML or Xcerpt constructs and ACE sentences can be unambiguously mapped. Thus VOXX can work with ACE as a user-friendly interface language for the following applications:

- Express XML data directly in the controlled language ACE

  John lives in Oettingenstr.67 in 80539 Munich. He has a private phone number and an office phone number. John's private phone number is 089123456. His office phone number is 089456789.

  Mary's address is Mainstreet 57, 12345 London. 7845637 is her private phone number.

- Use ACE as a verbalisation of the Web query language Xcerpt and thus query the Web using ACE

  Where does John live?
  What is John's address?
  In which city does John live?
  Does John live in London?
  Who lives in London?
  What is John's private phone number?
  Who has the office phone number 089456789?
  Does Mary have an office phone number?

- Translate existing XML documents into ACE, for example to answer queries.

  Yes, John lives in Munich.
  John's private phone number is 089123456.

Note that VOXX does not investigate problems of speech recognition and translating speech to text since these are independent research areas for which powerful solutions already exist.

1.3 Focus of this work

The idea behind the present report is to use ACE as an interface language to access XML documents. This application requires to find a suitable bridge between the structure of an XML document and the logical form generated by the corresponding ACE. The information of XML documents is usually accessed via XML tags on a case-by-case basis without explicitly refering to the logical structure of the document. If the logical structure of a document is known, it could be handled automatically using suitable algorithms. Therefore it would be desirable to transfer the information contained in an XML document into a logical structure, e.g. FOL.

In this report a language is developed to represent the ELEMENT tags and data of XML documents with DTD (Document Type Definition) as relations that can systematically be transformed into the document tree. An algorithm to that aim is introduced in this report.

Thus, the main objective is to find a possibility to express the structure of an XML document by use of n-ary logical relations analogous to the logical relations into which ACE is translated (see below for details). A document tree from an XML document will be created in three steps, which will be discussed in the following sections.

2 First step: Logical representation of XML based on relations

2.1 Notations

Before describing the principles behind this project, some notations used in this report will be introduced. Here 'C' and 'FS' denote the pre-defined relations *child* and *following sibling*. A user-defined relation *address_of_name/3* can be defined as follows:

$$\begin{aligned}&address\_of\_name(\text{“}john\text{“},\ \text{“}main\ street\text{“},\ \text{“}london\text{“}):\\ &\quad C(name,\ \text{“}john\text{“})\ \&\ FS(name,\ street)\ \&\ C(street,\ \text{“}main\ street\text{“})\\ &\quad \&\ city\_of\_person(\text{“}john\text{“},\ \text{“}london\text{“}).\end{aligned} \tag{1}$$

The user will be able to define similar relations using the pre-defined relations *child* and *following sibling* (see chapter 3.1).
All notations are explained in table 1 in three columns, the notation, a short description and an example, which refers to the above relation (1).

2.2 Principles

ACE is automatically translated into first-order logic (FOL), which can be formulated using n-ary logical relations. For translating an XML document into ACE the structure of an XML document will initially be transformed into logical relations. The idea is based on the fact that every XML document using a DTD represents the structure of a graph, which is a tree as long as cross-references are forbidden (see below). It is possible to express the structure of the tree in terms of relations between children and following siblings.

In the following report we assume an XML document [4][5]. Then the leaves correspond to the data and the inner nodes to the tags of that document (see chapter 2.3). For the sake of simplicity, attributes are not considered. Note however, that the approach described in this report can easily be extended so as to accomodate attributes. Let L be the set of leaves and N be the set of inner nodes of a tree T. The leaves of a tree are not in the set of the inner nodes and vice versa, thus $N \cap L = \emptyset$. Information in the tree can be expressed using relations between nodes. Each node can have more than one relation to other nodes. In this work we will use pre-defined and user-defined relations to express the document structure.
The pre-defined relations will be *child* and *following sibling* as known from graph theory:

$child(x, y)$ denotes	y is a child of x, $x \in N$ and $y \in N \cup L$, $x \neq y$.
$following\_sibling(x, y)$ denotes	x and y are siblings, x is the preceding sibling of y, y is the following sibling of x, $x \neq y$.

Notation	Definition	Example
relations	all user- and pre-defined relations	the whole relation (1), C(name, “john“), FS(name, street), C(street, “main street“)
relation definition	user-defined relation	the whole relation (1)
definition head	everything before ':'	address_of_name(“john“, “main street“, “london“)
relation name	name of the relation	address_of_name
relation arguments	arguments stated in the definition head	“john“, “main street“, “london“
definition body	everything after ':'	C(name, “john“) & FS(name, street) & C(street, “main street“) & city_of_person(“john“, “london“).
conjuncts of the definition body	all pre-defined relations and relation references used in the definition body	C(name, “john“), FS(name, street), C(street, “main street“), city_of_person(“john“, “london“)
child relation	pre-defined relation child	C(name, “john“), C(street, “main street“)
arguments of a child relation	arguments that occur in a pre-defined relation child	name, “john“, street, “main street“
following sibling relation	pre-defined relation following sibling	FS(name, street)
arguments of a following sibling relation	arguments that occur in a pre-defined relation following sibling	name, street
relation reference	definition head of an user-defined relation	city_of_person(“john“, “london“)
name of the relation reference	name of an user-defined relation	city_of_person
arguments of the relation reference	arguments that occur in a relation reference	“john“, “london“

Table 1: Listing of the notations used in this work

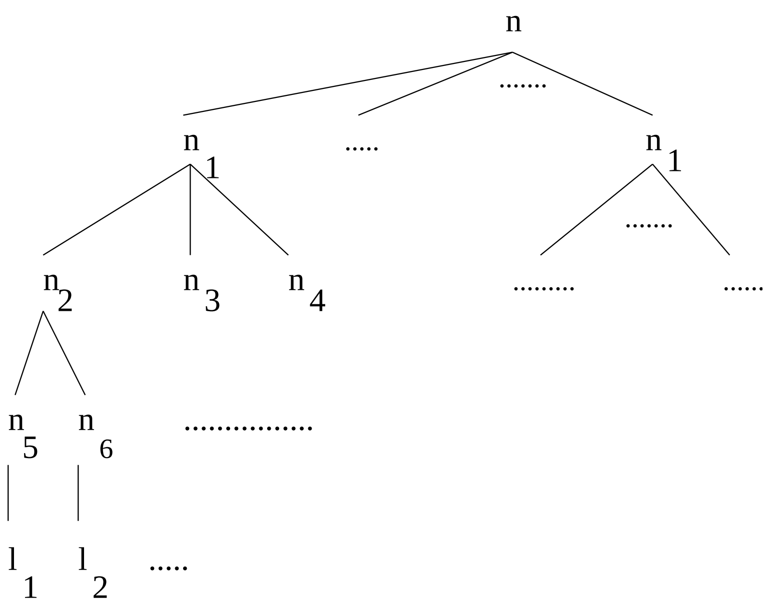

Figure 1: Tree with identified nodes

In this work the expression 'subtree' always implies a part of the tree, which contains only inner nodes and is rooted at a node of depth one (see Figure 1, every tree rooted at n_1). Contrary to the definition commonly used in graph theory, such a subtree does not contain any leaves of the original document tree, but ends with the inner nodes before the leaves (see Figure 1, every tree ending with n_5, n_6 and so on). As the tree is generated from an XML document, every subtree has the same structure as the other subtrees rooted at a node of depth one. This can easily be shown if the document tree is drawn under the assumption that all elements of the DTD are written in every document entry, even if the elements are empty. For the whole work the assumption was made that cross-references are not allowed (for details see below). Thus, to describe the structure of the tree, it is sufficient to build up the structure of one subtree, the connection to the leaves, and the relationship between the root and its children.

If a tree is described using the relations *child* and *following sibling*, the nodes can be identified by indices, in XML usually the tagnames. Let's introduce two types of indices numbering the inner nodes and the leaves within one subtree. Such a tree is shown in the Figure 1. In the practical cases (see chapter 2.3) instead of l_1, $l_2, \ldots$ the nodes will be named as 'street', 'phone' etc.

In an XML document the same tagnames may occur several times, e.g. in an address book with several telephone numbers for one person. In the following it is assumed, that every tagname, thus every name of a node, is unique. That means before using the algorithm introduced in chapter 4 the tagnames have to be converted into unique (e.g. numbered) names. In practice this can get obtained, e.g. labeling the nodes with the position, that they have in the document tree.

Any tree can be defined by a set R of relation definitions, namely the root and its children and the relationship between the nodes within one subtree using conjunctions of *child/2* and *following sibling/2* relations: Let n be the root, $N = \{n_1, \ldots, n_z\}$ a set of the inner nodes of one subtree and $L = \{l_1, \ldots, l_y\}$ a set of the leaves. Then the tree can be defined using the following types of relation definitions:

- Connect the root to the subtree. The *child* relationship between the root n and one node of depth one has to be defined. There may be no explicit connection between the root and the subtrees in a single relation definition. It is sufficient that this connection is stated non-explicitly somewhere within other relation definitions.

- Relation definitions describing one subtree and the leaves: The connection is defined via relation definitions, using only symbols instead of the data of the leaves. Later on, when all relation definitions are stated, the symbols of the leaves will be substituted by the real data.

The inner nodes and leaves, for that the relation should get defined, i.e. that occur as relationarguments, are quantified universally. All other nodes in relation definitions are quantified existentially. The definition body contains conjunctions of the pre-defined relations and relation references as follows:

- *child* relation between an inner node and another inner node (see [6]).

- *child* relation between inner node and leaf (see [6]).

- *following sibling* relation between two inner nodes (see [6]).

- *following sibling* relation between inner node and leaf (see [6]).

- *following sibling* relation between leaf and inner node (see [6]).
- relation references.

Below a formal description is given to specify the above relation references and relations that shows which of the inner nodes and leaves can occur where in the relation definition and which quantifier they must have. The leaves L_3 extracted from the above specified set L of all leaves are divided into two disjunct subsets of universally and existentially quantified leaves L_1 and $L_2 \subseteq L$. The same applies to the inner nodes N_1 and $N_2 \subseteq N$. The subsets 1 and 2 are needed to distinguish universally and existentially quantified nodes and leaves. This makes sure that both types are not mixed in one relation definition. The elements will be defined as follows:

Let y be the total number of leaves in one subtree:
$L_1 = \{a_1, \ldots, a_n\}, 1 \leq n \leq y, \quad n, y \in \mathbb{N}$
$L_2 = \{b_1, \ldots, b_m\}, 1 \leq m \leq y - n, \quad m \in \mathbb{N}$
$L_1 \cap L_2 = \emptyset,\ L_1 \cup L_2 = L_3,\ L_3 \subseteq L$

Let z be the total number of inner nodes in one subtree:
$N_1 = \{c_1, \ldots, c_x\}, 1 \leq x \leq z, \quad x, z \in \mathbb{N}$
$N_2 = \{d_1, \ldots, d_q\}, 1 \leq q \leq z - x, \quad q, z \in \mathbb{N}$
$N_1 \cap N_2 = \emptyset,\ N_1 \cup N_2 = N_3,\ N_3 \subseteq N$

Any relation definition is composed of a conjunction of the above described *child* and *following sibling* relations and relation references (see listing above, each bullet corresponds to one conjunction). A relation definition may contain all or some of this relations and references. Then a relation definition $r \in R$ is given by:

$$\begin{aligned}
\forall a_1, \ldots, a_n, c_1, \ldots, c_x \\
& r(a_1, \ldots, a_n, c_1, \ldots, c_x) \Leftrightarrow \\
& \exists b_1, \ldots, b_m, d_1, \ldots, d_q : \\
\bigwedge_{n_1, n_2 \in N_3} & child(n_1, n_2) \wedge \\
\bigwedge_{n \in N_3, l \in L_3} & child(n, l) \wedge \\
\bigwedge_{n_1, n_2 \in N_3} & following_sibling(n_1, n_2) \wedge \\
\bigwedge_{n \in N_3, l \in L_3} & following_sibling(n, l) \wedge \\
\bigwedge_{n \in N_3, l \in L_3} & following_sibling(l, n) \wedge \\
\bigwedge_{definitonhead \in R} & definitionhead
\end{aligned} \tag{2}$$

Here conjunctions over empty sets have to be omitted.

Without loss of generality relation definitions can be restricted to those containing only the logical 'and' ($\wedge$), and no negation or disjunction. The excluded cases can be treated separately making use of the fact that the negation is implicit, using the principle 'negation as failure', and the disjunction can be obtained defining separate definition bodies for one definition head. For a further discussion see chapter 3.1. Recursion will not be covered. That means, that all trees are assumed to have acyclic graphs. The only possibility to get cycles in a graph derived from an XML document are

the identifiers ('id') and references ('idref') in the XML DTD. To avoid this, it is assumed that 'id' and 'idref' are not used in the DTD.

2.3 Example

2.3.1 An XML DTD

The following example shows a DTD for an address book [7]:

```
<!ELEMENT address_book(address_book_entry)>
<!ELEMENT address_book_entry(name, street, city, phone)>
<!ELEMENT phone(private, office)>
<!ELEMENT name(#PCDATA)>
```

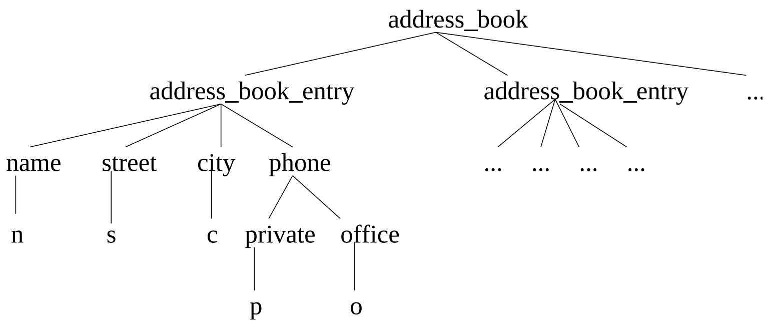

Figure 2: Tree for a document created after the 'address_book'-DTD

<!ELEMENT street(#PCDATA)>
<!ELEMENT city(#PCDATA)>
<!ELEMENT private(#PCDATA)>
<!ELEMENT office(#PCDATA)>

After creating a document using this DTD the tree will have the following structure: The root is formed from a node 'address_book' which contains all the other tags in the XML document. Every entry of an address has the tag 'address_book_entry'. Its children are the nodes 'name', 'street', 'city' and 'phone'. 'phone' has the children 'private' and 'office'. 'n', 's', 'c', 'p', 'o' stand for the data entries that are normally between the tags. Creating a document using this DTD the tree shown in Figure 2 is obtained.

2.3.2 Relation definitions

Now logical relations can be expressed in terms of the set of inner nodes of the subtree, its leaves and the *child* relation between root and subtree. In the following we consider relation definitions, whith $n, s, c, p, o \in L$ denoting the XML data name, street, city, private and office phone number to which the XML tags refer. The advantage of using only symbols instead of the real data is, that once defined the relation definitions for one subtree, it is easily to describe the whole tree substituting the symbols by the real data of the particular subtree. We assume here that every relation definition only expresses relationships of the nodes within the subtree rooted at *address_book_entry*, which belongs to the set of inner nodes, *address_book _entry, name, street, city, phone, private, office* $\in N$. This assumption corresponds to the above exclusion of recursion and acyclic graphs. As discussed in chapter 2.2 all subtrees are structurally identical apart from the leaves. Only the data of the leaves are different. The following relation definition $address\_of\_name(n, s, c)$ gives the relationship between the name and the address of a person. With that relation definition, one could define or the address of a person with a certain name or the name belonging to a certain address.

$$\begin{aligned}
&\forall n, s, c \\
&\quad address_of_name(n, s, c) \Leftrightarrow \\
&\quad \exists address_book_entry, name, street, city : \\
&\quad child(address_book_entry, name) \wedge child(name, n) \wedge \\
&\quad child(address_book_entry, street) \wedge child(street, s) \wedge \\
&\quad child(address_book_entry, city) \wedge child(city, c)
\end{aligned} \tag{3}$$

If the name n is known, one can find the corresponding address in the tree, knowing that the data nodes containing the information are the leaves of the tree and the children of *street* and *city*, which have the same parent node as *name*. The only problem with that formulation is that the information about the order of *name*, *street* and *city* is lost. Therefore the *following sibling* relation is more precise:

$$\begin{aligned}
&\forall n, s, c \\
&\quad address_of_name(n, s, c) \Leftrightarrow \\
&\quad \exists address_book_entry, name, street, city : \\
&\quad child(address_book_entry, name) \wedge child(name, n) \wedge \\
&\quad following_sibling(name, street) \wedge child(street, s) \wedge \\
&\quad following_sibling(street, city) \wedge child(city, c)
\end{aligned} \tag{4}$$

Another relation definition can be defined using the private phone number of a person, who lives at a certain address:

$$\begin{aligned}
&\forall s, c, p \\
&\quad private_phone(s, c, p) \Leftrightarrow \\
&\quad \exists street, city, phone, private : \\
&\quad child(street, s) \wedge following_sibling(street, city) \wedge \\
&\quad child(city, c) \wedge following_sibling(city, phone) \wedge \\
&\quad child(phone, private) \wedge child(private, p)
\end{aligned} \tag{5}$$

As the relation definition only expresses a relationship between nodes, the same definition body could be used for a relation definition called $address\_of\_phone(p, s, c)$ to query or define the address belonging to a private phone number.

The final goal is to be able to construct the tree of a document, using only these relation definitions, without knowing anything about the original DTD and XML document. The missing link in the example is a relation definition that connects the root and all the subtrees with each other. By expanding the relation (4) with the *child* relation between a node $address\_book\_entry$ and the root $address\_book$ the following is obtained:

$$\begin{aligned}
&\forall n, s, c \\
&\quad address_of_name(n, s, c) \Leftrightarrow \\
&\quad \exists address_book, address_book_entry, name, street, city : \\
&\quad child(address_book, address_book_entry) \wedge \\
&\quad child(address_book_entry, name) \wedge child(name, n) \wedge \\
&\quad following_sibling(name, street) \wedge child(street, s) \wedge \\
&\quad following_sibling(street, city) \wedge child(city, c)
\end{aligned} \tag{6}$$

This relation definition is similar to the relation (4). However, $child(address\_book, address\_book\_entry)$ is added, and $address\_book$ is existentially quantified. This guarantees that the subtree is connected to the root of the tree.

3 Second step: Defining a language to represent the relation definitions

In this report, first of all, a new language S will be developed to express the relation definitions derived from a given document. The language must be capable of expressing any set R of such relations. It is the basis of an algorithm that extracts a formal tree structure from the relation definitions and thus the tree structure of the original document (see chapter 4).

3.1 The language S

All relation definitions of chapter 2.3 follow the same notation: Every relation definition starts with a universal quantifier with all the inner nodes and leaves for which the relation is defined, i.e. all the inner nodes and leaves that occur in the definition head. The universal quantifier is followed by the definition head. The next term is an existential quantifier over those inner nodes and leaves that occur in the definition body and are not universally quantified earlier. The definition body contains conjunctions of *child* and *following sibling* relations and relation references. Thus it has the following form:

$$\begin{aligned}&\forall relationArguments\\ &\quad relationName(relationArguments) \Leftrightarrow\\ &\quad \exists\, nodes,\ leaves:\ definitionBody\end{aligned} \tag{7}$$

To improve readability relation (7) will be written in the form

$$relationName(relationArguments) \Leftrightarrow definitionBody \tag{8}$$

using the following simplifications:

- The relation arguments appear twice in (7). They each comprise all the universally quantified nodes and leaves and may be rather long expressions. For example, one single leaf may consist of the abstract of an article. To shorten the writing, the first part is dropped starting the relation definition with the relation name.

- Every node in the definition body of (7) that does not occur in the definition head must be existentially quantified. Thus also this part of the notation can be dropped without loss of information.

Let's review the example (5) mentioned above. The explicit notation is the following:

$$\begin{aligned}&\forall s, c, p\\ &\quad private\_phone(s, c, p) \Leftrightarrow\\ &\quad \exists street, city, phone, private : child(street, s) \wedge\\ &\quad following\_sibling(street, city) \wedge child(city, c) \wedge\\ &\quad following\_sibling(city, phone) \wedge child(phone, private) \wedge\\ &\quad child(private, p)\end{aligned}$$

The above simplifications reduce this to:

$$\begin{aligned}&private\_phone(s, c, p) \Leftrightarrow\\ &\quad child(street, s) \wedge following\_sibling(street, city) \wedge\\ &\quad child(city, c) \wedge following\_sibling(city, phone) \wedge\\ &\quad child(phone, private) \wedge child(private, p)\end{aligned} \tag{9}$$

To make the relation definition easier to type and to parse computationally, the following changes in syntax are made:

- A point at the end of every relation definition makes parsing easier.
- A colon ':' replaces the '$\Leftrightarrow$' symbol.
- An ampersand '&' replaces the '$\wedge$' symbol.
- Leaves start and end with double quotes ' " '.
- 'C' replaces '*child*'.
- 'FS' replaces '$following\_sibling$'.

The last replacements have the additional advantage that the words '*child*' and '$following\_sibling$' are not longer forbidden words for the user. He may use them in user-defined relation names. Capital letters are no drawback because user-defined relation names must be lower case anyway. With these shortcuts the relation definition (9) becomes:

$$\begin{aligned}&private\_phone(\text{“}s\text{“},\ \text{“}c\text{“},\ \text{“}p\text{“}) :\\ &\quad C(street,\ \text{“}s\text{“})\ \&\ FS(street,\ city)\ \&\ C(city,\ \text{“}c\text{“})\ \&\\ &\quad FS(city,\ phone)\ \&\ C(phone,\ private)\ \&\ C(private,\ \text{“}p\text{“}).\end{aligned}$$

3.2 Grammar of *S*

The language *S* has the following context-free grammar:

Relations	→ RelationDefinition*.
RelationDefinition	→ DefinitionHead ':' DefinitionBody'.'.
DefinitionHead	→ RelationName '(' InnerNodeOrLeaf ')'.
InnerNodeOrLeaf	→ InnerNode | Leaf.
InnerNodeOrLeaf	→ InnerNode ',' InnerNodeOrLeaf
	| Leaf ',' InnerNodeOrLeave.
DefinitionBody	→ Child
	| FollowingSibling
	| RelationReference
	| Child '&' DefinitionBody
	| FollowingSibling '&' DefinitionBody
	| RelationReference '&' DefinitionBody.
Child	→ 'C' '(' InnerNode ',' InnerNode ')'
	| 'C' '(' InnerNode ',' Leaf ')'.

FollowingSibling → 'FS' '(' InnerNode ',' InnerNode ')'
| 'FS' '(' InnerNode ',' Leaf ')'
| 'FS' '(' Leaf ',' InnerNode ')'.
RelationReference → DefinitionHead.
RelationName → (Character | Digit)+.
Leaf → ' “ ' (Character | Digit | WhiteSpace)* ' “ '.
Character → (_ | a | ... | z).
Digit → (0 | ... | 9).
WhiteSpace → ' '.
InnerNode → ... .

The place holders '...' in the last rule denote the tagnames that occur in the original XML DTD. The insertion of tag names will be performed automatically by a suitable program.
The grammar has been implemented using Definite Clause Grammars (DCG) of Prolog [8]. The advantage is, that this grammar can be used directly to check the syntax of the relations. A separate tokenizer code rewrites the user's relation definitions deleting all white spaces (except the ones in leaves). A parser derived from the DCG Prolog code checks the syntax using the grammar via the prolog console. If it accepts the relation definitions, then the syntax is considered correct.

3.3 Defining suitable relations

There are some points to bear in mind when defining a suitable set of relation definitions of a given document:

- For some relation definitions there are several ways of defining the definition body, using both *child* and *following sibling* relations (see (4)) or using only relations of type *child* (see (3)). It is important to analyze, which set of relations should be taken.

- When describing a tree with *child* relations only, normally information about the order of the tree is lost. Order however may be important in certain cases. Consider an address book in which each entry contains the office and the private phone number of the person. Then a person can be called first at his office simply dialing the second phone number without checking the label of the father node. This does not work if the tree does not have the same order as the original document, because the children of a node are out of order. It is left to the user, if he wants to keep the order in the tree or not.

- Another critical decision has to be made about how many and which relation definitions should be used to express the DTD in an unambigous manner. It would be desirable to consider only such relation definitions that contain information which is not covered in other relation definitions. This would avoid unnecessary loops in the algorithm to rule out redundant relation definitions. In practice, however, it is difficult to decide, if a set of relation definitions R is minimal and complete, in the sense that all information of the document is covered.

- Additionally it has to be examined that the relation definitions really express the original DTD, which for now can be made comparing the original tree with the new created one.

- *Following sibling* relations should be preferred instead of *child* relations, if possible. As already mentioned, the order of the siblings is lost if only *child* relations are used in a definition body.

- Every relation reference must be defined somewhere in the relation definitions.

- All inner nodes and leaves that occur in the definition head must also occur in the definition body. Otherwise they would be undefined.

- Different relation definitions that have the same definition body and vary only in the relation names are allowed. The name of the relation definition is not used in the algorithm constructing the tree. Double names are inconsequential in the algorithm (but may disturb the user).

- The same applies to using the same name for several different relation definitions.

4 Third step: Algorithm for expressing the relation definitions as a tree

This chapter presents an algorithm that determines the document tree from the relation definitions stated by the user, splits them into tokens, and classifies those. The algorithm requires that the user's definitions are correct, complete and unambiguous in the following sense:

- All definitions are correct i.e. all *child* or *following sibling* relations stated match the original tree of the document. As the algorithm has no access to the document, discrepancies between user-defined relations and the original document will remain undetected apart from certain grammatical errors.

- All relation definitions are complete. That means they are sufficient to construct the tree. Connections between the inner nodes and between the inner nodes and their leaves must be defined.

- The relation definitions are unambigous in the sense that it is not possible to obtain two different trees from the set of relation definitions.

4.1 Preprocessor

Before starting with the algorithm the input data have to be transformed in a list suitable for the algorithm. For this purpose first all relation references in the relation definitions stated by the user are substituted by their definitions. Then a list of lists is constructed. Each list consists of the pre-defined relations that occur in one relation definition. The algorithm works on that list of lists as input.

4.2 First step of the algorithm

The algorithm consists of two steps. First only the root and one subtree without leaves is created. In the second step the leaves and further subtrees, if any, will be added.

4.2.1 Lists and nodes

The algorithm reads the above list of lists of relation definitions, extracts the pre-defined relations, which have inner nodes only, and sorts them into two lists:

- All relations of type *C(inner node, inner node)* are stored in a 'C-list'. It contains at least one element. If not, the tree would not have any root.

- All relations of type *FS(inner node, inner node)* are stored in a 'FS-list'. This list may be emtpy.

The algorithm scans these lists to build a tree. Unevaluated relations are buffered into two lists, called 'C1-list' and 'FS1-list', one for the *child* relations and one for the *following sibling* relations.
To process the lists the following Prolog predicates are used:

head(list)	get the first element of 'list'
tail(list)	get the rest of the list, ignoring the first element of 'list'
member(element,list)	true, if 'element' is an element of 'list'
append(list,element)	insert 'element' into 'list'
delete(element,list)	delete 'element' from 'list'

The representation of the nodes is a left-child-right-sibling-representation, as given by Corman et al. [9]. Every node has two pointers, one to its left child and one to the right sibling of the node. According to [9] this is sufficient to represent the tree. For convenience additional pointers from child to father can be defined that faciliate searches in the tree. The root has no pointer to the father. Thus a node is represented by the Prolog fact *node(ID, FatherID, ChildID, Following siblingID).*

4.2.2 Pseudo code

At the beginning the algorithm inserts the two nodes of the first *child* relation of the C-list into the empty tree and deletes the child relation in the list.

```
if (tree empty)
{ \\ get first element of the list
  C(father, child) = head(C)
  insert father in the tree above child
  \\ delete first element of the list
  C = tail(C)
}
```

Then the next child relation is evaluated.

- The tree remains unchanged, if father and child already exist.
- If neither father nor child exist, the *child* relation is added to the C1-list.
- If no father (but the child) is found, the father is inserted into the tree with a connection to the child. Connections between father and other nodes will be added later.
- If the father exists and the child is missing, there are several cases:

- If the algorithm finds the child node in the FS-list, it uses the method 'nodeOnTheRight' if the child is a following sibling and the method 'nodeOnTheLeft' if some other node follows. Both methods try to insert the child below its father. The details will be described below.
- If no child node is found in the FS-list or if both of the above methods fail, the child is inserted below its father. Following siblings, if any, are then treated later.

In any case, the *child* relation is deleted from the C-list. If the relation cannot be evaluated it is saved in the C1-list for further use.

In this manner the entire C-list is evaluated and erased. This leads to a pseudo code as follows:

```
while (C or FS not empty)
{   while (C not empty)
    { \\ get first element of the list
      C(father, child) = head(C)
      if(C(father, child) in tree)
      {   C = tail(C)
      }
      else if (father is in tree)
      {   \\ test1 and test2 are boolean values and show, whether the tree
          \\ has changed in the particular method 'nodeOnTheRight' and
          \\ 'nodeOnTheLeft'. In that case the node 'child' was inserted
          \\ into the tree which means that C(father, child) can be
          \\ deleted from C
          test1=false
          test2=false
          if(member(FS(left, child), FS))
          {     nodeOnTheRight(left, child, test1)
          }
          if (member(FS(child, right), FS))
          {     nodeOnTheLeft(child, right, test2)
          }
          if(test1==false && test2==false))
          {     insert child below father
                C = tail(C)
          }
```

```
            else
            {      C = tail(C)
            }
        }
        else if (child is in tree)
        {   insert father above child
            C = tail(C)
        }
        \\ no hint, where the nodes are connected to the tree
        else
        {   append(C1, C(father, child))
            C = tail(C)
        }
    }
}
```

At the end the C-list is empty, as either the arguments of the relations were inserted into the tree, or the *child* relation was added to the C1-list. The tree contains at least two nodes, namely those of the first *child* relation of the original C-list. At this point the algorithm walks through the FS-list doing the following:

- If both nodes exist in the tree, the tree remains unchanged
- If the left node of the relation definition is in the tree, the second argument, the following sibling, is inserted on the right.
- If the right node is in the tree, the left node is inserted into the tree.
- If neither of the nodes is in the tree, the *following sibling* relation is added to the FS1-list.

In any case, the relation is deleted from the FS-list. The corresponding pseudo code reads:

```
while (FS not empty)
{   \\ get first element of the list
    FS(left, right) = head(FS)
    if(FS(left, right) in tree)
    {  FS = tail(FS)
    }
```

```
    else if (left is in tree)
    { insert right on the right hand side of left
      \\ delete first element of the list
      FS = tail(FS)
    }
    else if (right is in tree)
    { insert left on the left hand side of right
      FS = tail(FS)
    }
    else
    { append(FS1, FS(left, right))
      FS = tail(FS)
    }
}
```

Finally, the C1-list becomes the new C-list and the FS1-list becomes the FS-list, and this is iterated until both lists are empty.

It remains to explain the two methods 'nodeOnTheRight' and 'nodeOneTheLeft'. They are similar but differ in where the new node is inserted and which *following sibling* relations are observed.

The method 'nodeOnTheLeft' is called if the father is in the tree and the child is not. It controls, wether there are further following siblings of the child. The method has three parameters: the child node, its following sibling selected from FS-list, and a false boolean that becomes true if the child node has been inserted. The method does the following:

- If the right node is already in the tree, the child is inserted to the left, the *following sibling* relation is deleted from the FS-list and the boolean value is set to true.
- If the right node is not in the tree, the method attempts to insert the right node recursively. If this works, the child is also inserted and the boolean is set to true.

The boolean remains false if the method fails to insert a node.

```
nodeOnTheLeft(left, right, test2)
{   if(right is in tree)
    {   insert left on the left hand side of right
        delete(FS(left, right), FS)
```

```
        test2=true
    }
    else if member(FS(right, rightNeighbour), FS))
    {   nodeOnTheLeft(right, rightNeighbour, test2)
        if(test2 = true)
        {   test2=false
            if(right is in tree)
            {   insert left on the left hand side of right
                delete(FS(left, right), FS)
                test2=true
            }
        }
    }
}
```

The method 'nodeOnTheRight' is essentially the 'mirror' of 'nodeOnTheLeft'. Essentially this two methods are symetrical. E.g. instead of inserting the node on the left, it is inserted on the right hand side.

```
nodeOnTheRight(left, right, test1)
{   if(left is in tree)
    {   insert right on the right hand side of left
        delete(FS(left, right), FS)
        test1=true
    }
    else if member(FS(leftNeighbour, left), FS))
    {   nodeOnTheRight(leftNeighbour, left, test1)
        if(test1 = true)
        {   test1=false
            if(left is in tree)
            {   insert right on the right hand side of left
                delete(FS(left, right), FS)
                test1=true
            }
        }
    }
}
```

4.3 Second step of the algorithm

The second step of the algorithm completes the structure obtained in the first step by adding leaves and further subtrees. The input to the second step consists of the list produced by the preprocessor and the main structure of the tree generated in the first step. The algorithm results in a tree, that represents the original structure of the XML document.

The algorithm converts the above list of lists into a new list R (of lists) by dropping those *child* and *following sibling* relations that contain inner nodes only. The new list R contains terms of the form *C(inner_node, leaf)*, *FS(inner_node, leaf)* and *FS(leaf, inner_node)*. The algorithm uses a list *R1* for unevaluated lists as in 4.2 and another list *R2* for buffering.

A copy of the original list R is saved in *R2* to check if the list R has been modified. No modification occurs if the relation definitions are ambigous, so that the tree cannot be build. In this case the algorithm is aborted. The algorithm has no possibility to control, if the relation definitions are defined in a complete and correct way. If they are not, the algorithm will terminate, but it maybe delivers a wrong output, resulting from the incorrect input.

Now the leaves of the first element of R are inserted into the tree.

```
\\R2 is for tests to prevent the algorithm from non-termination
R2 = R
if (tree has no leaf)
{ E = head(R)
  while(E not empty)
  {  X = head(E)
     insertLeaf(X, subtree)
     delete(X, E)
  }
  R = tail(R)
}
```

Now the main part of the algorithm starts: further subtrees - if needed - are generated and all XML-data of the relation definitions are inserted as leaves. The algorithm sequentially checks if the leaves of the elements of R fit into the existing tree. This is the case if one of the leaves can be found in the tree. Then the other leaves are inserted according to the predefined relation. If the leaves definitely do not fit into the tree because they are in contradicition to the existing structure, a new subtree is created. If the decision is unclear the element of *R* is stored in *R1*. In any case the element is deleted in *R*. After renaming *R1* into *R* the algorithm continues in the above manner until *R* is empty. In detail this requires the following steps:

1. Compare the *R*-list to the *R2*-list, which is a copy of the *R*-list of the previous round. If the lists are identical, then the relation definitions are ambigous, then exit after a warning. Save the first element of the list, which is itself a list, and delete it from the list.

2. Then use 'searchSubtree' (see below) to check, wether the leaves of that relation definition:

   (a) do not fit into the subtree, because some inconsistency occurs, then continue with 3.

   (b) are contained in the subtree, then continue with 4.

   (c) can be inserted to the subtree, then continue with 4.

3. Proceed to the next subtree, as described in step 2. If the leaves do not fit into any subtree, create a new subtree and insert the leaves.

4. The method 'interpretResult' (see below) evaluates the list created by 'searchSubtree' and fits them into the subtree, if possible.

5. If the *R*-list and the *R1*-list are empty the algorithm ends. Otherwise save the *R* in *R2*, rename *R1* into *R* and empty the *R1*-list and continue with 1.

This corresponds to the following pseudo code:

```
while(R not empty)
{  \\ test, wether the list R is the same as R2 (which corresponds
   \\ to the R of the last round). If this is the case, i.e. the relation
   \\ definitions are ambigous, give a warning stating the critical
   \\ relations. Otherwise, continue
   if(R == R2)
   { give a warning about the ambiguity of some relations
     state R2
   } \\test is in order to control, if E fits in a subtree
   test = false
   \\get the first element (itself a list) E of R
```

```
    E = head(R)
    result = searchSubtree(E, subtree)
    \\if the result is not false, then it is a list
    if(result != false)
    { interpretResult(result, E, R)
      test = true
    }
    \\if result is false, search the next subtrees
    else if (nextSubtree)
    { while(nextSubtree)
      {  subtree = nextSubtree
         result = searchSubtree(E, subtree)
         if(result != false)
         {  interpretResult(result, E, R)
            test = true
            break
         }
      }
    }
    \\ if searchSubtree returnes false for all subtrees, that is, that something
    \\ in the list E is in conflict with every subtree
    if(test=false)
    { make new subtree S
      while(E has elements)
      {  X = head(E)
         insertLeaf(X, S)
         delete(X, E)
      }
      R = tail(R)
    }
    if(R empty)
    { R2 = R
      R = R1
      R1 = [ ]
    }
}
```

The method 'searchSubtree' compares a given list with a given subtree. If the subtree contains all elements of the list, the method returns an empty list. If there are elements in the list, that are not in the subtree, it returns a list containing these elements. If the list contains an element that is in conflict with leaves in the subtree, the method returns false:

```
searchSubtree(list, subtree)
{  L = head(list)
   if(L in subtree)
   {  list = tail(list)
      if(list not empty)
      {   searchSubtree(list, subtree)
      }
   }
   \\ L does not fit into the tree, because the place is just occupied
   \\ by another value
   else if(L doesn't fit in the tree)
   {  return false
   }
   else
   {  append(newList, L)
      list = tail(list)
      if(list not empty)
      {   searchSubtree(list, subtree)
      }
   }
   return newList
}
```

The method 'interpretResult' interprets the list, that 'searchSubtree' has returned. If the returned list is empty, all leaves of the original list are in the tree and need not be inserted. If the list is not empty, there are two cases:

1. The list returned contains the same elements as the original list. This means that none of the leaves has been found in the subtree and therefore no leaf could be connected to the subtree without unambiguity. The list is then stored in *R1* for further handling.

2. The list has less elements than the original list, at least one leaf was contained in the subtree, so the other leaves belong to the same subtree. They are inserted using the method 'insertLeaf' (see below).

```
interpretResult(result, E, R)
{  \\ if all elements of the list are already in the tree
   if(result is empty)
   {  R = tail(R)
   }
   else if(result = E)
   {  \\ if the result is the same as E, there are the possibilities of
      \\ at least two subtrees,(one with leaves and another one with
      \\ leaves or an empty subtree),where the result would fit in
      append(result, R1)
      R=tail(R)
   }
   else
   {  while(result has elements)
      {  X = head(result)
         insertLeave(X, subtree)
         delete(X, result)
      }
      R = tail(R)
   }
}
```

The method 'insertLeaf' inserts the leaf of a predefined *child* or *following sibling* relation, that is, the string-argument of that relation, into a subtree after the following scheme:

- If the leaf is already in the tree, no action is necessary.
- If the leaves come from a *child* relation (the leaf has to be the child in this case), check if there are any following siblings to that leaf, or if the leaf is a following sibling, and insert it at the correct position. If there are no siblings, insert the child below its father.
- If the leaf comes from a *following sibling* relation, then there are two possibilities: Either the leaf is a following sibling or it is followed by a sibling. It can be inserted correspondingly.

```
insertLeaf(ChildOrSiblingRelation X, subtree)
{  if(X is in tree)
```

```
    {
    }
    else if(X = C(inner_node, leaf))
    {  test=false
       while(member(C(inner_node, another_inner_node), tree)
       {   if(member(FS(another_inner_node, leaf), R)
           {  insert leaf next to another_inner_node in subtree
              test=true
              break
           }
           else if(member(FS(leaf, another_inner_node), R)
           {  insert leaf in front of another_inner_node in subtree
              test=true
              break
           }
       }
       if (test==false)
       {   insert leaf below inner_node in subtree
       }
    }
    else if(X = FS(inner_node, leaf))
    {  insert leaf next to inner_node in subtree
    }
    else if(X = FS(leaf, inner_node))
    {  insert leaf in front of inner_node in subtree
    }
}
```

5 Implementation

The implemented solution reads a given XML DTD, extracts the tag names and writes them at the end of the parser, which is equivalent to the grammar discussed above (see 3.2). The tag names will be the names of future inner nodes. Then the user-defined relation definitions are compared with the grammar. If the syntax is correct, the implemented algorithm constructs the document tree out of the relation definitions. The order of the subtrees may differ from the order of the entries of the document. This is tolerable, because the order of the entries of a document is unconsequential. In an address book, for example, it does not matter, if the entry 'John' comes before or after the entry 'Sam'. Only the order within one entry is important.

5.1 How to use the program

The main algorithm has been programmed in Prolog [10] [11]. The source files are available on the CD that accompanies this report. For programming details see these files. This section explains how to use the program.

Scope of the program *representRelations.pl* is to represent user-defined relations describing an XML document as a tree. The other files are needed to format the relation definitions (see *transformRelations.pl*) and to check if they are written in the correct way (see *tokenizer.pl* and *parser.pl*). Before using the parser another program is needed, that extracts from the XML DTD the tag names and inserts them as inner node names to the parser (see *getInnerNodesToGrammar.java*), as the parser needs to know, which names of inner nodes can be used for the document to control if the relation definitions are written in the correct way.

First the files are described, that are needed to run the program. Then a step-by-step instruction follows.

- Files needed:
  - User supplied files:
    - One file containing the XML DTD according to which the document was created. It should contain ELEMENT definitions only - all others will be ignored. Every line must start with an XML tag definition. Leading white space characters and empty or blank lines are not allowed.
    - A second file containing the relation definitions describing the XML document created after the above DTD.
    - Apart from the usual extensions '.dtd' and '.txt' the user is free to choose any file names. For simplicity the user file names 'UserFile.dtd' and 'UserFile.txt' will be used in this text.
  - Files on CD:
    - *getInnerNodesToGrammar.class* is a small Java program that appends the ELEMENT names of an XML DTD as inner nodes to the program *parser.pl*.

* *representRelations.pl* is a batch-type Prolog program that calls the following Prolog files:
  - *tokenizer.pl* reads the text file of relations 'UserFile.txt' and splits it into a list of atomic terms.
  - *parser.pl* parses the result and checks its content against a built-in grammar.
  - *substituteReferences.pl* replaces all relation references in 'UserFile.txt' by their definitions.
  - *transformRelations.pl* simplifies the relation definition as described in chapter 4.1. The result is the list of lists of section 4.1.
  - *algorithm.pl* transforms this list into a tree-structured list, the document tree. The tree is a list of lists. Each list contains one or more terms *node(name, father, left-most child, following sibling).* The first list contains the root node, every following list contains a subtree (in terms of its nodes).

- Instructions:

  1. Append inner nodes to the grammar:
     java getInnerNodesToGrammar UserFile.dtd parser.pl
  2. Start SWI-Prolog using the maximum available global stack (-G0 for an Unix console):
     pl -G0
  3. Compile the Prolog program:
     consult('representRelations.pl').
     Two warnings will appear refering to a redefined procedure *trim_list/2* and *remove_white_spaces/2.* They result from replacing code by identical code. The warnings can be ignored.
  4. Run the program:
     start('UseFile.txt').

5.2 Limitation of the implementation

The algorithms described in this report are designed to be applicable to any kind of XML document that is created from a DTD. As stated above, only ELEMENT tags are evaluated, others are ignored.

Some of the Prolog codes described in 5.1 tend to require rather large global stacks, in particular the codes *substituesReferences.pl* and for replacing references and the main algorithm *algorithm.pl* that creates the document tree.

Therefore, in practice, the available global stack space restricts the amount of data in one document that can be handled by the algorithm without leading to a stack overflow. The limitations has been studied in a number of XML DTD's. They are included on the CD accompanying this text. It appears that the list processing within Prolog requires a large global stack. However, this could not be traced in all cases studied, because the Prolog tracer [12] produces an overflow of the local stack.

There is no well-defined limit to the size of XML documents that can be processed because the required Prolog stack space depends much on the manner how the user defines the relation definitions of the input file. In general the execessive use of relation references, that is, calling an user-defined relation within another user-defined relation (see table 1) tends to lead to long relation definitions and long lists, which, in term, require much stack space.

5.3 Suggested solutions

As the Prolog stack overflow seems to arise from list processing an obvious work-around is to avoid lists as much as possible and to keep unavoidable lists small. For instance, the lists of lists *R*, *R1* and *R2* of section 4.3 are processed sequentially, and could easily be replaced by files with one list per record. However, this would require to rewrite most of the Prolog codes because they are based on list processing predicates that do not apply to the file processing.

One could also try to store only branches that are filled with leaves rather than subtrees with few leaves only. This would reduce the overhead of unused information in the code on the expense of making the procedure of inserting leaves more complex. The terms *node(ID, fatherID, childID, following siblingID)* could be stored in a file, if the subtrees are distinguished in some manner. Alternatively, one could attempt to use a Prolog database for adding and deleting nodes. However, it is difficult to estimate a priori to which extent the required stack space can be diminished in these cases.

6 Outlook

The algorithm described in this report transforms the ELEMENT tags of an XML document into a document tree. This makes the information extracted from the document accessible to any kind of software that deals with tree structures without recourse to the original XML document. If other types of tags such as ATTLIST, ID, IDREF and NOTATION can also be extracted in such a manner, the complex analysis of the logic of XML documents on a case-by-case basis can be simplified. It is reduced to the well known logic of tree structures that can be treated automatically using existing algorithms.

Acknowledgement

I would like to thank François Bry, Uta Schwertel, Sacha Berger and Norbert Eisinger for their help and the time they spent on explaining the project and answering my questions.

A Pseudo code of the algorithm

A.1 First step

```
if (tree empty)
{   \\ get first element of the list
    C(father, child) = head(C)
    insert father in the tree over child
    \\ delete first element of the list
    C = tail(C)
}

while (C or FS not empty)
{   while (C not empty)
    { \\ get first element of the list
      C(father, child) = head(C)
      if(C(father, child) in tree)
      {   C = tail(C)
      }
      else if (father is in tree)
      {   \\ test1 and test2 are boolean values and show, whether the tree
          \\ has changed in the particular method 'nodeOnTheRight' and
          \\ 'nodeOnTheLeft'. In that case the node 'child' was inserted
```

```
        \\ into the tree which means that C(father, child) can be
        \\ deleted from C
        test1=false
        test2=false
        if(member(FS(left, child), FS))
        {     nodeOnTheRight(left, child, test1)
        }
        if (member(FS(child, right), FS))
        {     nodeOnTheLeft(child, right, test2)
        }
        if(test1==false && test2==false))
        {     insert child below father
              C = tail(C)
        }
        else
        {     C = tail(C)
        }
    }
    else if (child is in tree)
    {   insert father above child
        C = tail(C)
    }
    \\ no hint, where the nodes are connected to the tree
    else
    {   append(C1, C(father, child))
        C = tail(C)
    }
}
while (FS not empty)
{ \\ get first element of the list
    FS(left, right) = head(FS)
    if(FS(left, right) in tree)
    {   FS = tail(FS)
    }
    else if (left is in tree)
    {   insert right on the right hand side of left
        \\ delete first element of the list
        FS = tail(FS)
    }
    else if (right is in tree)
```

```
        {   insert left on the left hand side of right
            FS = tail(FS)
        }
        else
        {   append(FS1, FS(left, right))
            FS = tail(FS)
        }
    }
    C=C1
    C1=[ ]
    FS=FS1
    FS1=[ ]
}

nodeOnTheLeft(left, right, test2)
{   if(right is in tree)
    {   insert left on the left hand side of right
        delete(FS(left, right), FS)
        test2=true
    }
    else if member(FS(right, rightNeighbour), FS))
    {   nodeOnTheLeft(right, rightNeighbour, test2)
        if(test2 = true)
        {   test2=false
            if(right is in tree)
            {   insert left on the left hand side of right
                delete(FS(left, right), FS)
                test2=true
            }
        }
    }
}


nodeOnTheRight(left, right, test1)
{   if(left is in tree)
    {   insert right on the right hand side of left
        delete(FS(left, right), FS)
        test1=true
    }
```

```
    else if member(FS(leftNeighbour, left), FS))
    {   nodeOnTheRight(leftNeighbour, left, test1)
        if(test1 = true)
        {   test1=false
            if(left is in tree)
            {   insert right on the right hand side of left
                delete(FS(left, right), FS)
                test1=true
            }
        }
    }
}
```

A.2 Second step

```
\\R2 is for tests to prevent the algorithm from non-termination
R2 = R
if (tree has no leaf)
{   E = head(R)
    while(E not empty)
    { X = head(E)
      insertLeaf(X, subtree)
      delete(X, E)
    }
    R = tail(R)
}

while(R not empty)
{   \\ test, wether the list R is the same as R2 (which corresponds
    \\ to the R of the last round). If this is the case, i.e. the relation
    \\ definitions are ambigous, give a warning stating the critical
    \\ relations. Otherwise, continue
    if(R == R2)
    { give a warning about the ambiguity of some relations
      state R2
    } \\test is in order to control, if E fits in a subtree
    test = false
    \\get the first element (itself a list) E of R
    E = head(R)
    result = searchSubtree(E, subtree)
```

```
    \\if the result is not false, then it is a list
    if(result != false)
    { interpretResult(result, E, R)
      test = true
    }
    \\if result is false, search the next subtrees
    else if (nextSubtree)
    { while(nextSubtree)
      {  subtree = nextSubtree
         result = searchSubtree(E, subtree)
         if(result != false)
         {  interpretResult(result, E, R)
            test = true
            break
         }
      }
    }
    \\ if searchSubtree returnes false for all subtrees, that is, that something
    \\ in the list E is in conflict with every subtree
    if(test=false)
    { make new subtree S
      while(E has elements)
      {  X = head(E)
         insertLeaf(X, S)
         delete(X, E)
      }
      R = tail(R)
    }
    if(R empty)
    { R2 = R
      R = R1
      R1 = [ ]
    }
}

searchSubtree(list, subtree)
{  L = head(list)
   if(L in subtree)
   {  list = tail(list)
      if(list not empty)
```

```
        {   searchSubtree(list, subtree)
        }
    }
    \\ L does not fit into the tree, because the place is just occupied
    \\ by another value
    else if(L doesn't fit in the tree)
    {   return false
    }
    else
    {   append(newList, L)
        list = tail(list)
        if(list not empty)
        {   searchSubtree(list, subtree)
        }
    }
    return newList
}

interpretResult(result, E, R)
{   \\ if all elements of the list are already in the tree
    if(result is empty)
    {   R = tail(R)
    }
    else if(result = E)
    {   \\ if the result is the same as E, there are the possibilities of
        \\ at least two subtrees,(one with leaves and another one with
        \\ leaves or an empty subtree),where the result would fit in
        append(result, R1)
        R=tail(R)
    }
    else
    {   while(result has elements)
        {   X = head(result)
            insertLeave(X, subtree)
            delete(X, result)
        }
        R = tail(R)
    }
}
```

```
insertLeaf(ChildOrSiblingRelation X, subtree)
{  if(X is in tree)
   {
   }
   else if(X = C(inner_node, leaf))
   {  test=false
      while(member(C(inner_node, another_inner_node), tree)
      {   if(member(FS(another_inner_node, leaf), R)
          {  insert leaf next to another_inner_node in subtree
             test=true
             break
          }
          else if(member(FS(leaf, another_inner_node), R)
          {  insert leaf in front of another_inner_node in subtree
             test=true
             break
          }
      }
      if (test==false)
      {   insert leaf below inner_node in subtree
      }
   }
   else if(X = FS(inner_node, leaf))
   {  insert leaf next to inner_node in subtree
   }
   else if(X = FS(leaf, inner_node))
   {  insert leaf in front of inner_node in subtree
   }
}
```

References

[1] Reasoning on the Web: Language Prototypes and Perspectives. Sacha Berger, François Bry, Bernhard Lorenz, Hans Jürgen Ohlbach, Paula-Lavinia Pătrânjan, Sebastian Schaffert, Uta Schwertel, Stephanie Spranger, Institute for Informatics, University of Munich, 2004, see `http://rewerse.net/publications/download/REWERSE-RP-2004-39.pdf` (web page accessed on March 31, 2005)

[2] Attempto Controlled English (ACE). Requirements Engineering Research Group, Department of Informatics, University of Zurich, see `http://www.ifi.unizh.ch/attempto/` (web page accessed on March 31, 2005)

[3] XCerpt, see *http://www.xcerpt.org/* (web page accessed on June 07, 2005)

[4] XML Reference. SelfHTML, see *http://de.selfhtml.org/xml/index.htm* (web page accessed on April 7, 2005)

[5] E.T. Ray, J. McIntosh, Perl & XML, O'Reilly, 1st edition 2003

[6] XML Reference. DevX, see `http://www.devx.com/projectcool/Article/20014/1763/page/2` (web page accessed on March 31, 2005)

[7] T. Kobert, bhvCo@ch: XML, verlag moderne industrie, 1st edition, 2004

[8] Fernando C.N. Pereira, David H.D. Warren, Definite Clause Grammar for Language Analysis. In: Artificial Intelligence 1980, issue 13, pp. 231 - 278.

[9] T.H. Corman, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, 2001, MIT Press, pp. 214

[10] W.F. Clocksin, C.S. Mellish, Programming in Prolog, 5th edition, 2003, Springer-Verlag

[11] SWI-Prolog can be accessed under *http://www.swi-prolog.org* (web page accessed on April 7, 2005)

[12] SWI-Prolog Reference Manual, *http://gollem.science.uva.nl/SWI-Prolog/Manual* (web page accessed on April 7, 2005)