Optimizing the evaluation of XPath using Description Logics

U. Furbach, M. Gross-Hardt, T. Kleemann, P. Baumgartner

eMail
{uli | margret | tomkl | peter}@uni-koblenz.de

Abstract: The growing use of XML in commercial as well as non-commercial domains to transport information poses new challenges to concepts to access these information. Common ways to access parts of a document use XPath-expressions. We provide a transformation of DTDs into a knowledge base in Description Logic. We will use reasoning capabilities in Logic to decide, if a given XPath may be satisfied in a document, and to guide the search of XML-Processors into possibly successful branches of the document, avoiding parts of the document, that will not yield results. The extension towards objectoriented subclassing schemes opens this approach towards OODB-queries. Opposed to other approaches we will not use some kind of graph representing the document structure, and no steps towards incorporation of the XML/OODB-processor itself will be taken.

Keywords: XML, XPath, Description Logics, automated reasoning, DTD, Schema

Introduction

Within a short period of time XML has become a widely accepted standard for information interchange. Starting as a subset of SGML to transport structured text, the ease of understanding and using XML promoted it's use as an interchange format of rather large documents. This evolution created the needs for a validation of documents against an according definition. Most common and basic validation is accomplished by XML-processors referring a Document Type Definition (DTD). A DTD defines the structure of elements of the document, that references the DTD. We will detail the terms element and structure in following chapters.

Beyond the need to validate data several attempts have been made or are currently made to access parts of a document. One common idea in these attempts is the access of parts of a document following a path from it's root to some subtrees. Generally these paths are not specified completely from the root to the subtree, but leaves unspecified gaps to be filled by the XML/query-processor. Usually a XML-processor will traverse the document tree, to find instances of the specified parts of a path.

Based on DTD and XPath-expressions we will provide a way to optimize this traversal. This will be accomplished by translating the DTD into a set of description logics (DL) formulae and a subsequent query of this logic representation. Questions

about the satisfiability of a XPath in a document will be answered as well as Queries of the starting element of subtrees, that might contain fillers for the path specification.

Due to some shortfalls of DTDs we will show the compatibility of the translation and reasoning with some sorts of objectoriented extension. This opens our approach to uses in pathcompletion of objectoriented databases as well as schema-based definitions of XML-documents.

XML Documents

Starting as a specialisation of the Standard Generalized Markup Language (SGML) the eXtensible Markup Language (XML) was supposed to provide a better way of document markup than the widespread HyperText Markup Language (HTML). The *normative* definition of XML is available from the w3c [XML00]. Opposed to the fixed markup, and its interpretation, of HTML the XML approach offers a standardized way to markup arbitrary documents. This may include redefined HTML documents but is not limited to this application.

A XML document consists of a prolog an an element, optionally followed by miscelaneous comments and processing instructions, that are not in the scope of this paper. An element is either an empty element or a sequence of a starting tag followed by content and an ending tag. Taken from [XML00]:

[1] document ::= prolog element Misc*
[39] element ::= EmptyElemTag | STag content Etag
[40] STag ::= '<' Name Attribute* '>'
[41] Attribute ::= Name '=' Attvalue
[42] ETag ::= '</' Name '>'
[43] content ::= CharData? ((element | Reference | CDSect | PI | Comment) CharData?)*
[44] EmptyElemTag ::= '<' Name Attribute* '/>'

Content by itself may contain among others elements, that will be called child-elements. Tags are identified by their names. We will use this name as the name of an element.

The w3c cares about character codings, white spaces and miscelaneous components. Because we are merely interested in the structure of the document we will omit these otherwise important details. The topmost element will be called root element. It spans almost all of the document, especially it contains all other elements. We expect all documents to be wellformed and as explainf in the following section valid.

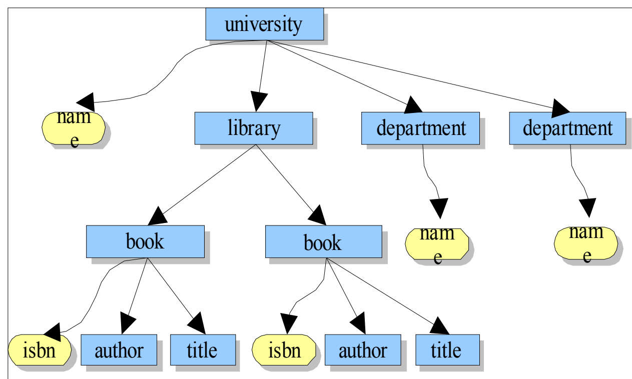
Sample Document

According to the above mentioned productions and constraints a wellformed document may look like this:

```
<?xml version="1.0"?>
<!DOCTYPE university SYSTEM "university.dtd">
<university name="Universität Koblenz">
  <library>
```

```
    <book isbn="978123123">
      <author> </author>
      <title> </title>
    </book>
    <book isbn="978234234">
      <author> </author>
      <title> </title>
    </book>
  </library>
  <department name="cs"/>
  <department name="math"/>
  ...more descriptions...
</university>
```

The prolog specifies this document to be a XML-document according to version 1.0. This is currently the only possible version. The second line specifies a document type definition. University ist the root element of the document. This university element contains a library element, that contains several book elements, and several department elements. The department elements are empty elements. Empty elements are empty with respect to the content, but may contain attributes. The university, department and book elements contain attributes, i.e. name and isbn.



A XML document may be represented by a tree. The root element corresponds to the root of this tree. The elements are represented as nodes of the tree. An element is linked to its child elements and attributes. The data of elements will not be represented in our trees to reduce the information to what we need. We are focused on the structure of documents. The following section introduces a common way to define the possible structures.

Validation of Documents

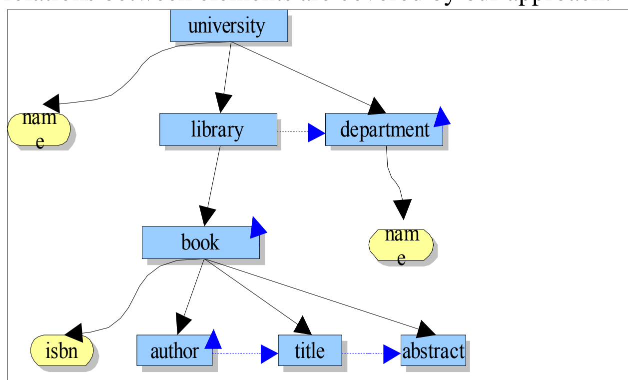
Whenever XML is used to transport information between independent applications, that is most common in business-to-business communication, there is a need to validate the document structure. Validating XML processors offer standardized ways to implement this validation process. These processors use a Document Type Definition (DTD) or a schema as a description of accepted elements, nesting of

elements and type information. In the example document above (2nd line) we already introduced the reference to an external DTD. This Document Type Declaration names the root element, university in this case. DTDs may as well be inline. The advantage of external DTDs derives from onetime central storage of the DTD. Beyond these differences both kinds of DTDs provide the same set of definitions. DTDs are already known to SGML [SGML86] and HTML documents. They do not provide significant type information or any aspect of object orientation. To overcome these insufficiencies XML schema has been introduced by the W3C. Schema introduces some basic type information and a limited support of object orientation. We will start with DTDs and demonstrate afterwards the opportunities of extended type information.

A DTD for the above sample document may look like this

```
<!ELEMENT university (library, department*, #PCDATA)>
<!ATTLIST university name CDATA #REQUIRED)>
<!ELEMENT library (book)+>
<!ELEMENT department EMPTY>
<!ATTLIST department name CDATA #REQUIRED)>
<!ELEMENT book (author+, title, abstract?)>
<!ATTLIST book isbn CDATA #REQUIRED)>
<!ELEMENT author #PCDATA>
<!ELEMENT title #PCDATA>
```

Focusing on the structure of a document, we will not use any information about the data of the document that is described. So #PCDATA, CDATA and so on will not be in the scope of this paper. Crucial to our target of optimization and completion of path expressions are the definitions of the child elements and attributes of elements. All elements mentioned in the definition of an element are child elements. The sequence operator ',' may be used to establish a sibling relation among the child elements. In this example author, title and abstract are childs of book. isbn is an attribute of book. author is a sibling of author and title and so on. Because we limit our presentation to the abbreviated syntax for XPath the sibling relations will be omitted, although these relations between elements are covered by our approach.



Different from the tree like structure of the XML documents themself, the description may contain cycles. A wellknown example of a cycle ist the HTML-table. The TD-element is a child of an TR-element, that is a child of the TABLE-element.

Because TABLE is part of arbitrary HTML-content, TABLE is a possible child of TD. The reasoning capabilities used are robust against these cyclical definitions.

Even in our quite small example DTD a cycle can be found. The element author is a sibling of itself.

Picking the Parts

Common to almost all processing of documents is the adressing of parts of it. The basic idea of all adressing schemes is a path expression, that specifies the navigation through the document. These path expressions may be rooted or relative to an existing position in the document tree. Severeal notational variants have been developed. We will use the abbreviated XPath 2.0 notation, that is covered by an W3C-working-draft [XP02]. Path expressions following these recommendations are incorporated in XSLT, Xquery, CSS2 and other standards.

Path expressions are explained by the following rules taken from [XP02]:

path ::= '//' relativePath | '/' relativePath | relativePath
relativePath ::= stepExpr ('/' | '//') stepExpr
stepExpr ::= '.' | '@' nameTest | '..' | nodeTest

The leading '/' and '//' construct a path starting at the document out of an relative path. '//' will expand to a path of zero or more steps. Step expressions access the current context node, it's attributes by a preceding '@' the parent of an element or perform node tests, ranging from simple element names to more complex expressions. Especially a wildcard '*' will match all child elements. A detailed description can be found in [XP02]. Because we restrict our path expressions to abbreviated syntax, the names of the axis are not mentioned, but inherently used. '.' uses the self axis. '..' uses the parent axis. The child axis will be used in nodeTest, as we will demonstrate with some examples:

/university/library will access the library element
//department will access the two department elements

several ways to access the book elements:
/university/*/* will access all grandchildren of the root
/university/*/book and /university/library/book will do the same
//book will also access the two book elements
//*[@isbn] this expression accesses all elements that have an isbn attribute
//*[author AND title] access all elements that have author and title child elements

The wildcard '*' and the universal path fragment '//' are a huge gain in comfort. A user may specify correct path expression even if she does not know the structure in detail, as can be seen in the expression //book. Regardless of the structure all book elements will be matched. This will happen regardless how many totally different paths exist in the document. As a first approach a XML processor may traverse the document and evaluate all constraints that a path expression carries. The larger the documents will be the worse this approach may become. We will provide decisions, that guide the traversal into those subtrees that may yield results with respect to the path expression. These decisions will be made based on the DTD of the document, using a description logics representation of the DTD.

The reasoning will also provide informations about empty result sets. If you specify a path like //book[@isbn=”987001001”] the XML processor will compare the isbn attribute of all book elements with the given string. In the above sample document no book element will fulfil this condition, and the result will be empty. But other documents according to the DTD may return results. This decision will be made upon comparison of all book elements.

An additional empty path expression will be //book/library. Different from the first empty path expression this expression will never return any book elements, because the structure in question, library as a child of book, is a contradiction to the DTD. Again the reasoning capabilities in the logic representation state, that this will always be empty. Any traversal of the document may be omitted.

Description logic for the Representation of DTDs

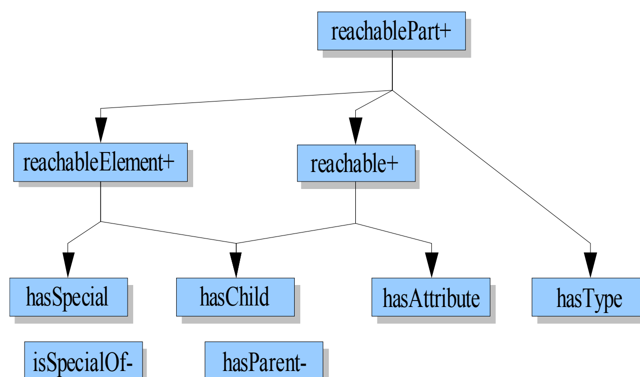
The target of our translation is a description logic (DL) that provides inverse and transitive roles and role hierarchies. Examples have been translated into input files of the RACER [HM01] that offers services of a SHIQ-reasoner. The second target was krhyper, a Tableaux-prover, including it's preprocessing tool for DL expressions.

Number restrictions have not been used so far, although DTDs provide information about singular or multiple occurrences of child elements. These informations are dropped right now, because we didn't expect significant enhancements towards optimization of path expressions.

The TBox

The Tbox contains few concepts that correspond to the distinct parts of a DTD. The concepts of our translation will be the building blocks of the DTD like element, attribute and type. These concepts will be populated by the individual elements of the specific DTD during the translation process.

The Type-Concept is trivial for DTDs, but allows the integration of the enhanced typing capabilities of XML-schema. We give details later on.



An important part of the Tbox are the definitions of roles and their attributes. We introduce roles to be instantiated in the following translation step as well as roles that are superroles of these. The following figure depicts the role hierarchy.

Roles with an appended – are inverse roles like hasParent is the inverse of hasChild. reacheable is a superrole of hasChild and hasAttribute. The appended + indicates that this role is transitive.

Translation of DTDs

The following translations demonstrate how we setup the Abox of our knowledge base from the DTD. The root element of the dociument and the correcponding DTD are mentioned in the prologue of the XML-file.

```
<!DOCTYPE rootelem Pointer_to_DTD>
```

This line leads to the Abox entry (root,rootelem):hasChild. root is an artificial individual that has the the document's rootelement as a child. For further translation we analyse the pointed DTD.

The description of the document structure contains information about the relationship of elements.

```
<!ELEMENT parent child1 child2 ...>
```

is translated to (parent, child1):hasChild, (parent, child2):hasChild and so on. (related parent child1 hasChild) in the Syntax of KRSS [PSS93], that is used by RACER. In addition to these role assertions we have to declare parent, child1 and child2 as individuals of the element concept (e.g. parent:element.) A concept definition like (define-concept element (some hasChild top)) and (define-concept element (some hasParent element)) will do that automatically.

The special empty element

```
<!ELEMENT elem EMPTY>
```

will lead to the simple assertion elem?element, that reads (instance elem element) in the KRSS syntax.

Translation of lists of attributes is straigt forward as well.

```
<!ATTLIST elem att1 type1 att2 type2 ...>
```

leads to the assertions (elem, att1):hasAttrinute, (elem, att2):hasAttrinute, (att1,type1):hasType, and (att2,type2):hasType. Again the definition (define-concept attribute (some hasType type)) will insure that att1 and att2 are elements of the concept attribute. Because of the limited number of types in DTDs, all types may be declared independent of the document: PCDATA, CDATA...:type.

Querying into the Knowledge Base

In our above example library is a role successor of university for role hasChild. book is a role successor to library in hasChild. These relationships are valid for role reacheable too because reacheable is a superrole of hasChild. Due to the transitive attribute of reacheable book is a role successor of university.

The DL reasoner provides queries like (individuals-related? university book reacheable) to ask, if the above mentioned relation is satisfied, i.e. book is a role successor of university for the role reacheable. This query will be written as (university,book):reacheable as in the usual notation of DL.

(individual-fillers book hasParent) reveals all possible parents of book this will be written as the concept $\exists$hasChild.{book} in the notation of DL.

Further reasoning support is provided to check the instances of concepts and the subsumption of concepts.

Empty Result Sets

With the Knowledge Base (KB) consisting of the Tbox and the Abox we are already able to decide, if a XPATH expression will lead to an empty result set. There are two reasons, why a XPATH expression will be empty. First of all the expression will locate optional elements that are not in the specific document. Similar to this an expression will return an emtpy result, if the expression incorporates comparisons to attributevalues that are not in the document. The second way to receive empty results are expressions that try to locate elements in an structurally impossible way. With our reasoning capabilities we are able to find most of these necessarily empty expressions. Overall the indication of an empty result due to the knowledge base is a partial detection of empty results.

As an example we consider the XPATH expression /book and our example from above. To judge the expression we have to check if book is an individual of the concept element, i.e. book:element? If this is true we have to check if book is a role successor of hasChild to root. Obviously this is not the case, so the expression /book will be empty for all documents according to the DTD.

Changing the expression to //book will have an impact on the second condition. We will query if book is a role successor of reacheable to root. This is true with respect to our DTD.

To judge the expressions empty is quite useful when expressions have to be evaluated, because these empty result sets are provided without any search in the document itself. Furthermore the presence of permanently empty expressions in a software may be an indication that the expression is not useful or contains an error.

For more complex XPATH expression we can combine the parts of the expression to an conjunctive condition. //book/*@name is possibly not empty if the condition for //book are satisfied and book has a child that has an attribute 'name'.

We will show a neat way to combine these fragments in a single query.

Optimizing the search in XML documents

Beyond the ability to decide necessarily empty resultsets we are able to predict in which parts of then document tree further search may be successful. Thus we avoid traversal of those parts of the documents, that will not lead to results. In our simple expression //book the traversal of department elements cannot lead to any book elements. The DTD does not allow for it, and the knowledge base does not contain the pair (department, book) in reacheableElement. On the traversal of the document tree

the search can be limited to those elements (here library), that contain the requested structure made up of elements and attributes.

Our approach evaluates a concept expression to obtain all possible child-elements of the current node of the traversal. If this concept expression return an empty set, no further traversal of the subtree is required. Consequently no traversable childs of 'root' indicate an empty result of the XPATH expression. This is equivalent to an unsatisfiable concept in terms of the used reasoners.

Additionally the traversal of subtrees stops as soon as possible. If for example the structure is extended in a way that library may contain book elements as well as journal elements, the raversal will not deepen the search into the journal elements, even if there are no book elements present.

The extension of the concept expressions contains a maximum number of possible elements that have to be cut to the elements found in the document. Thus a query may be empty in a document even if the concept expression is not empty, because the structure allows for appropriate documents.

How to construct the concept expressions

In order to use the complete information provided by the XPATH expression we choose a 'bottom-up' approach to contruct the DL concept expression adopteed from the XPath syntax. Some detailed patterns for this are shown in the following section. In a path like //library/*[@isbn] we will start at the end to construcht a concept term for *[@isbn] this searches for some element, name unknown, that has an attribute named 'isbn'. The corresponding concenpt term is

∃hasAttribute.{isbn}

explicit values in an XPATH expression like @isbn="987123123" are discarded because we focus on the structural decision. To step back to library we add

{library} ∩ ∃**hasChild.**∃hasAttribute.{isbn}

where the latter part contains the parents of the previous term that is intersected with the individual concept, that is mentioned in the XPATH expression. In case of an arbitrary element '*' no intersection or specialization will occur. To complete the task we add the '//' part, that is formed by the child elements of 'root' and the parents of the existing expression. These parents are either immediate or transitive parents.

∃**hasParent.{root}** ∩ ∃**reacheable**.({library}∩ ∃hasChild.∃hasAttribute.{isbn})

The resulting concept contains all child elements of the root that may leed to the elements in question. To guide the search of the XML processor one has to iterate the query after each step.

Regarding our sample DTD and the corresponding KB, this expression would yield {university}. We conclude from this, that the expression in question is structurally possible. If the resulting concept is empty no search in the document will be needed.

Disjunctive or konjunctive connections of XPATH expressions are easily tranferred into union or intersection of the concept expressions.

Finite model

While the number of possible path is unlimited in a cyclical structure, the possible child elements are finite. In fact every concept expression is a subset of the concept element.

Limitations

A strict limitation to all structural analysis are the 'pointing' elements IDREF. Because no information is provided where they point to, the optimization is limited to the path from the root to the IDREF. At the referenced element an optimization or prediction may start with the remaining XPATH expression.
The completion of the path through a XML document is perfomed on a step by step basis. The iteration of the completion is performed by the document processor. While this puts some tasks into the XML processor, the optimization is robust against cyclical structures.
Further improvements in the decision about possible elements, that lead to non empty results might be derived from number restriction. A structure with two Y-childs in an X-element will not satisfy an expression X/Y[3], because there is no third Y-child in this structure. We have dropped these numberrestrictions because we found no Xpath expression in a number of sample code, that addressed childs this way while the structure limited the number of childs in an appropriate way.

Extensions

So far we have given patterns for the XPATH expressions used in the abbreviated syntax. These expressions incorporate the child and parent axis. The transitive reacheable role and it's inverse cover the ancestor and descendant axis. To extend the KB towards other axis we would simply have to introduce a role for the axis and integrate these roles into the translation of the DTD.
An important extension is the integration of a type system, that is introduced by XML schema. The type information is available on two levels. There are more (44) types for attributes, this can easily be integrated by additional individuals in the type concept. A type hierarchie is implemented by some concept subsumption. These types offer a further refinement of the concept expressions. Similar to the expression ∃**hasAttribute.{isbn}** we could easily use the hierarchie of types to query with concept expressions like ∃**hasAttribute.Number**. Using the subsumption of types, as in NaturalNumber ⊆ Number, this query will provide all elements that have attributes of type Number or NaturalNumber. On a second level XML schema introduces some very basic kind of extension and restriction mechanism of elements. We expect this feature to develop in an object oriented fashion, that would lead to a subsumption hierarchy of these typed elements. Currently we can deal with this typing mechanism through the roles hasSpecial and the inverse, or the above introduced subsumption.
A fully object oriented typing might introduce the needs for nonmonotonic reasoning, beyond the capabilities of a standard DL system. Experiments with the first order prover krhyper [B98, W03, DUN01] indicate the ability to add the hopefully needed features.

Translation to logic programs

To make use of the model generating proover krhyper, we used Peter Baumgartners dl2lp translation tool, that converts a DL Tbox and Abox into a stratified logic program. Replacing the DL reasoner with this combination dl2lp and krhyper improved the performance significantly.
Opposed to DL the queries into the KB are performed under the closed world assumption. Concerning the queries that are imposed by our approach, there will be no differences, because all concept expressions within the queries are bound by the concept 'element'.

Conclusion

With the evolving use of DL reasoners in the field of the semantic web the question arose if the optimization of the document processing has to use different reasoners or graph oriented tools. We have shown so far, that a DL reasoner or a compatible reasoner is suitable for the task of predicting an empty result and the optimization of the document processing itself.

With the ability to act as a reasoner about the semantic annotation of documents and the documents and their structure themselv a single reasoning component can be used to perform both tasks. This single tool concept may reduce overhead in system load and maintenance. The reduced learning effort, to understand the reasoning component, should help to promote the use of reasoning systems.

The use of a DL-representation for semistructuresd data is not new at all. In [Calv99, Calv99b] a thorough translation is presented that aims at a Tbox representing the structure and an Abox containing the content of such data. Our focus was the ability to guide an existing XPath processor through the document tree, not the processing of data ourselves. Furthermore the ocurrence of cyclical structures introduces some challenges for this approach. In contrast to [BS02] we don't want to introduce an enhanced language for queries.

Our focus has led to the use of an Abox reflecting the structural properties of documents. Using the krhyper we could get the results of 20 queries into a document structure within less than 0.04 seconds on an ordinary PC including the setup of the knowledge base and the queries.

In contrast to a merely relational calculus, we did not want to sacrifice the ease of quering for a possible increase in speed, that is still in doubt.

Finally we have to admit that the reasoner has to catch up with the increasing complexity of the document’s description and the reasoning task. The foreseable task to integrate some kind of OO type system will lead to some needed improvements in the reasoner. The enhancements of the reasoning component may offer additional reasoning capabilities to all clients of the service. Focussing on a single system may speed up the process of improvement compared to the development of multiple services.

References

[XML00] Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, 6 October 2000, http://www.w3.org/TR/REC-xml

[XP02] XML Path Language (XPath) 2.0, W3C Working Draft 15 November 2002, Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, Jérôme Siméons, http://www.w3.org/TR/xpath20

[XS01] XML Schema Part 0: Primer, W3C Recommendation, David C. Fallside, 2 May 2001, http://www.w3c.org/TR/xmlschema-0

[SGML86] ISO8879:1986, Information processing -- Text and office systems -- Standard Generalized Markup Language (SGML)

[HM01] RACER System Description, Volker Haarslev and Ralf Möller, LNCS vol. 2083, pages 701ff, 2001

[PSS93] P.F. Patel-Schneider, B. Swartout 'Description Logic Knowledge Representation System Speci cation from the KRSS Group of the ARPA Knowledge Sharing Effort', 1993, http.//www-db.research.bell-labs.com/iser/pfps/papers/krss-spec.ps

[W03] KRHYPER System Description, Ch. Wernhard, 2003

[B98] Hyper Tableaux – The Next Generation, H. de Swaart, editor, 'Automated Reasoning with Analytic Tableaux and Related Methods', vol. 1397 LNAI, pages 60-76, 1998

[DUN01] J. Dix, U. Furbach, I. Niemelä, 'Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations', A. Voronkov A. Robinson editors, Handbook of Automated Reasoning, pages 1121-1234, Elsevie-Science-Press, 2001.

[Calv99] D. Calvanese, G. De Giacomo, M. Lenzerini: Representing and Reasoning on XML Documents: A Description Logic Approach. Journal of Logic and Computation 9(3): 295-318 (1999)

[Calv99b] D. Calvanese, G. De Giacomo, M. Lenzerini: Queries and Constraints on Semi-structured Data. CAiSE 1999: 434-438

[BS02] F. Bry, S. Schaffert, 'Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification', in Proceedings of the ICLP02, 2002