A Constraint System for Web Documents

Dániel Szegő
Department of Measurement and Information Systems
Budapest University of Technology and Economics
H-1117 Budapest, Magyar tudósok körútja 2.
Hungary
szegod@mit.bme.hu

Abstract – **During the last few years several different techniques for analyzing and validating Web documents have been developed including document checking tools, document querying and transformation frameworks or search engines. These techniques are usually based on different theoretical approaches, hence some of them are lack of simple formal semantics or efficient algorithms. This paper investigates the possibility of realizing a constraint system for Web documents in the context of description logics providing a formal method to separate valid and non valid documents. The paper compares the proposed solution with other industrial approaches.**

I. INTRODUCTION

During the last ten years, the success of World Wide Web was increasing, and it has become part of our daily life. Due to this enormous success, several techniques for processing, transforming or searching Web documents, like XML or HTML, have been developed. Unfortunately, these techniques are usually based on different theoretical approaches, no uniform representation is known. Primary consequence of different theoretical frameworks is that several parts of them are reinvented and re-implemented at each of the techniques. Hence, some of these frameworks are lack of simple formal semantics or efficient algorithms.

Description logics are simple logical formalisms which primarily focus on describing terminologies and graph style knowledge [1]. For example, Entity-Relationship diagrams or semantic networks can easily be translated to description logical formulas, having a more precise representation than the original ones [2]. Hence, these logics have pretty good computational properties, like EXPTIME satisfaction algorithms. Therefore, description logics seem to be adequate basis for developing a common computational environment for several Web document processing tasks. Although there are some approaches which use description logics in the context of Web, but these approaches rather concentrate on the semantics of Web than processing searching or transforming of simple documents or constructing constraint systems [3,4].

This paper investigates the possibility of realizing a constraint system for Web documents in the context of description logics providing a simple but powerful method to separate valid and non valid documents. The term 'constraint' is not the same as in the traditional constraint satisfaction systems (e.g. a hyper graph with a set of nodes, domains and constraints), rather constraints are represented as set of logical expressions similarly to the logical constraints in prolog programs or deductive databases. The paper also analyses the similarities and differences between the proposed approach and other industrial solutions, like DTD [5] or Xschema [6].

The origin of this work was motivated by a Web filter project. Several elements of the project and logic presented in this paper were previously published in [7,8]. However, until this point no constraint system based on the logic has been considered.

The remaining part of this paper is organized as follows. Section 2. introduces the basic architecture behind the logic. The logical approach covering the document model the syntax the semantics and some application examples is discussed in Section 3. Constraint satisfaction system based on the logical approach and comparison with industrial approaches are covered in section 4. Finally, Section 5 draws some conclusions.

II. BASIC ARCHITECTURE

The basic architecture behind the logical method can be seen on Figure 1. An experimental implementation of the architecture has been developed in C#. Web documents, like XML or HTML documents are firstly analyzed by parsers. The output of parsing is the document model. From a practical point of view, the document model is the

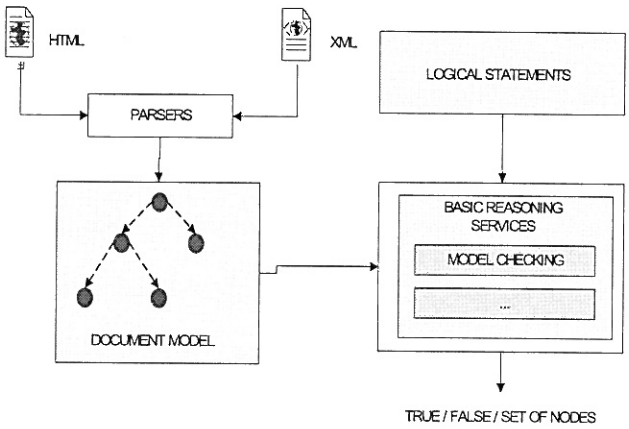

Fig. 1. Basic Architecture

mathematical formalization of a Web document, which is basically a relational structure (several basic sets and relations or maps over these sets). From a theoretical point of view, the document model is the model of a logic (like the model of the first order logic).

Logical statements represent the knowledge base which is necessary for a given document processing task. For example, if a search or query process is supported by the framework, logical statements describe the search conditions or the queries. The most important part of the architecture is the implementation of the basic reasoning services. It implements among others the model checking

algorithm that identifies if a given statement is true for a document model or not. The output of the framework is simply a true or false value (properties), which might be interpreted for individual documents for sets of documents or for subparts of documents. For realizing a constraint system true or false values must be interpreted for individual documents, indicating the valid and non-valid Web documents. However other application areas are also possible. For example, in a document query task several tags of the Web document, which is represented by several nodes of the document model, is presented as output.

III. A LOGICAL APPROACH FOR WEB DOCUMENTS

This section briefly introduces a fragment of SHIQ description logic, which fragment has primary importance in Web document processing. On the one hand, the term 'fragment' means that the syntax of SHIQ is simplified by eliminating several syntactical elements, like role constructors. On the other hand, semantics of SHIQ is also reduced from general relational structures to tree style models.

A logic for Web documents, as any other logics, requires three basic elements: a model, a syntax, and the connection between the syntax and model usually called as semantics or interpretation. First, the document model is introduced.

The ***document model*** is relational structure defined by the following sets maps and relations

<V,AP, AN, Name ,Type,Val,top,*c*,*ap*,*n*,*a*,*an*,*at*,*av*>.

- V is a set of nodes of the graph.
- AP is a set of atomic predicate, top∈ V is the root.
- AN is a set of attribute nodes, Name is a set of attribute names, T is a set of type names and Val is a set of possible values.
- $c \subseteq V \times V$ is a binary relation associating each node with its children nodes.
- $ap: V \rightarrow 2^{AP}$ is a partial map associating each node with a set of atomic predicates.
- $n: V \rightarrow V$ is a partial map associating each node with its following (next) node.
- $a \subseteq V \times AN$ is a binary relation associating each node with its attribute nodes.
- $an: AN \rightarrow Name$ is a partial map associating each attribute node with a name
- $at: AN \rightarrow 2^{Type}$ is a partial map associating each attribute node with a set of type names.
- $av: AN \rightarrow Val$ is a partial map associating each attribute nodes with a value.

Naming conventions:

- Edges of the graph are represented as <x,y> elements of c or n ($<x,y> \in c$ or $<x,y> \in n$)
- Paths of the graph are represented by $<v_1,v_2, v_3,\ldots v_{N-1},v_N>$ sequences, where $v_i \in V$, $<v_i, v_{i+1}> \in c$.

Axioms of the structure are the followings:

1. Each path of the graph must be acyclic, each maximal long path has to start from the t root, and each node of the graph must be reached from the root through one of the paths.
2. Following nodes must have the same parent : $n(v_i) = v_j$ implies that there exists a $v_k \in V$, for which $<v_k,v_i> \in c$ and $<v_k,v_i> \in c$.
3. Two children nodes of a common parent must be linked by next maps: $<v_k,v_i> \in c$ and $<v_k,v_j> \in c$ implies that there exists a sequence of nodes $<v_1,v_2,v_3,\ldots v_{N-1},v_N>$ such that $n(v_p)= v_{p+1}$ and $v_1=v_i$, $v_N=v_j$ or $v_1=v_j$, $v_N=v_i$.

This definition seems natural for an XML document. For example tags can be translated to nodes and embedding of tags represents the children relation. It is less trivial for an HTML document, consequently pre-transformations and pre-filters need to be applied. These transformations attempt to capture the necessary parts of an HTML document for a given task. Document model will be set up with these pre-filtered parts. The transformations are strongly task dependent and consist of plenty ad-hoc mechanisms. Attributes of tags of XML or HTML pages are described by special attribute nodes. Each attribute node is connected to a simple node by the '*a*' binary relation. Each attribute node is associated with three further elements: an attribute name, an attribute type and an attribute value. For representing types, attributes are linked with types symbols, whilst types of tags can be expressed by atomic predicates.

Syntax and semantics of the logic are based on roles and concepts. Concepts, denoted by 'C', represent subsets of nodes of the document model whilst roles, denoted by 'R', are binary relations between nodes. Syntax of the logic can easily be described by the following rules. For better readability basic elements of the syntax are written by bold characters.

R ::= **child** | **next** | **attribute-of** | **value-of** | **inverse** R | **infinite** R

C ::= {a}| **value** {value}| **type** {type}| **attribute-name** {name}| **every** | **none** | C **and** C | C **or** C | **not** C| **all** R.C | **some** R.C | =N R.C

There are four kinds of simple role, 'child' and 'next', describing children relation and next partial map of the document model. There are also two role constructors, 'inverse' and 'infinite' which represent inverse and transitive closure relations. There are three basic concept constructors. {a} does not represent a syntactic form, but the abbreviation of one piece of atomic concept which must equal with an atomic predicate of the document model. Similarly, the term 'type' or 'attribute-name' must be followed by a concrete type or attribute name, which must be found in the document model. The third simple role ,'attribute-of', associates the structure of the document with the attributes, whilst 'value-of' represents the connection between attributes and values. 'every' and 'none' are the universal and bottom concepts of the logic, 'and' 'or' 'not' are the basic logical operators whilst 'all' and 'some' denote universal and existential quantification over a role. '= N' is a special quantification meaning that there are exactly 'N' pieces of elements which are in 'C'.

In order to define a formal semantics of the syntax, an I interpretation function is considered, which assigns to every concept a set of nodes of a given 'd' document model and to every role a binary relation over $\{V \cup AN \cup Val\} \times \{V \cup AN \cup Val\}$. ('#' means cardinality of a set)

$a^I = \{ v \in V \mid a \in ap(v)\}$

attribute-name $name^I = \{ n \in AN \mid name=an(n)\}$

type type_name^I = { n∈AN | type_name∈at(n)}
value value_name^I = { n∈AN | value_name=av(n)}
child^I = c
next^I = {<x,y> | y = n(x)}
attribute-of^I = a
value-of ^{I} = {<a,v> | v ∈ at(a)}
(**inverse** R)^I = {<w,v>∈ V×V | <v,w>∈ R^I}
(**infinite** R)^I = $\cup_{j>=1}(R^I)^j$, transitive closure of R^I
every^I = {V∪AN∪Val}
none^I = ∅
(**not** C)^I = V \ C^I
(C$_1$ **and** C$_2$)^I = C$_1^I$ ∩ C$_2^I$
(C$_1$ **or** C$_2$)^I = C$_1^I$ ∪ C$_2^I$
(=N R.C)^I = { v∈{V∪AN∪Val} |
#{∃w. <v,w>∈ R^I and w∈ C^I}=N}
(**all** R.C)^I = {v∈{V∪AN∪Val} |
∀w. <v,w>∈ R^I implies w∈ C^I}
(**some** R.C)^I = {v∈{V∪AN∪Val} |
∃w. <v,w>∈ R^I and w∈ C^I}

As a simple example, we can imagine a company whose confidential Web documents are marked by special confidentiality notes. There are two kind of notes, an image and a natural language text, which indicate that the given document is confidential. Assume that we would like to develop a tool which identifies secret documents. This identification could be the basis of a document checking task, like documents which has no highlighted privacy

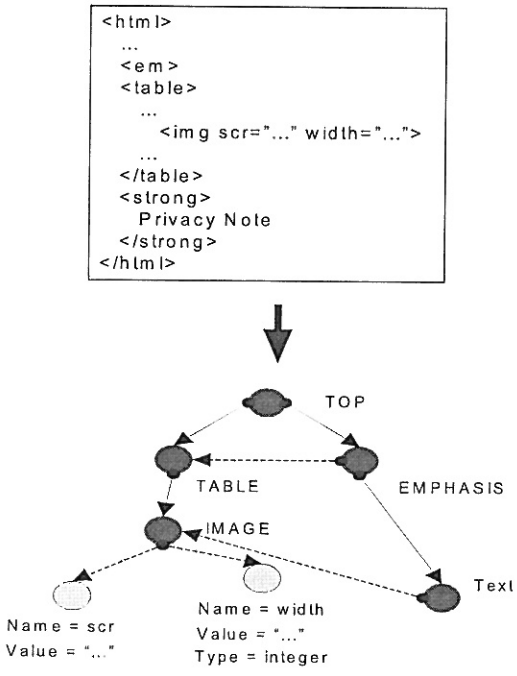

Fig. 2. Example

notes should be considered as non valid. Similarly, different category of the confidentiality could also be identified.

Figure 2. demonstrates an HTML document fragment and the corresponding document model. Black circles of Figure 1. represent the standard nodes of the document model (V set), whilst grey nodes are the attribute nodes. For the sake of simplicity, name value ant type elements of an attribute node are represented as simple list of values, so '*an*', '*at*' and '*av*' maps do not appear explicitly. '*c*' and '*n*' relations appear as links between the standard nodes and '*a*' binary relation is represented by the links between simple and attribute nodes.

Different properties of Web documents, containing privacy text or image can easily be analyzed by logical expressions:

- 'Text' expression is true for those tags of a document which contain text.
- 'Text **or** Table' expression is true for those tags of a document which is either marked by Text or Table label. This expression can be regarded as a very simple query expression.
- 'Emphasis **and some child** Text' is true for those tags of a document model which contain text and this text is emphasized (more specifically the expression is true for the emphasis tag of the document model).
- '**type** integer' is true for those attribute nodes of the document model which type is integer.
- 'Table **and some child**.(image **and some attribute-of** (**type** integer))' is true for those tags of the document model which are labeled by 'Table' and have a children called 'image' which has an integer attribute.
- 'image **and all attribute-of** (**type** integer)' is true for those image tags of a document for which all attributes have integer type. Such expressions can typically be used as document checking expressions..

In this example, the architecture of Figure 2. has extended with two additional document processing elements: a natural language processing and an image processing service for identifying if a text or image is a privacy note.

Using a logic in real life applications requires the existence of several basic reasoning services and efficient algorithms for computing these services. One of the most important and most efficient service is model checking and different variations of model checking.

- Model checking problem: Given a 'd' document model and an 'exp' expression. The question is whether the 'exp' expression true for 'd' or not.
- Querying in model checking: Given a 'd' document model and an 'exp' expression. The result of querying is the elements of the document model for which 'exp' is true. Querying can easily be interpreted in a modal or description logical framework, whilst it is more complicated in first order formalism.
- Satisfaction: Given an 'exp' expression. The question is if there is a 'd' document for which 'exp' expression is true.
- Subsumption or implication: An 'exp_1' expression implying 'exp_2' means that, whenever 'exp_1' is true 'exp_2' is also true. Implication can be regarded globally or locally. Global interpretation means that it holds for all possible document models, whilst local implication holds only for a given 'd' document model (it can also be called as model checking style of implication).
- Equivalence: An 'exp_1' equivalent with 'exp_2' means that, 'exp_1' is true if and only if 'exp_2' is also true. Similarly to implication , equivalence can also be interpreted both globally and locally.

Of course the most important question is efficiency, which highly influences the industrial applicability of every theory. Unfortunately, satisfaction is a hard problem for all logical formalisms (e.g. NP hard for proposition logic, undecidable for first order logic). Hence, model checking problem is NP complete for second order logic, even if the model is constrained to graphs [9]. However, it is possible to build efficient applications if modal or description languages are used, because model checking can be solved by very efficient polynomial algorithms [1,2]. In our case, the limited syntax and the special semantics causes that model checking and different variations of model checking, like local subsumprion or equivalence, can be solved with linear time and space complexity [8].

IV. CONSTRAINT SYSTEM

The previously introduced architecture and logical formalism can be easily used as a constraint system. In this work, the term 'constraint' is not the same as in the traditional constraint satisfaction systems (e.g. a hyper graph with a set of nodes, domains and constraints), rather it is similar to the logical constraints in prolog programs or deductive databases.

A constraint is a logical expression, which is true for some Web documents and false for the others. It simply validates or invalidates a document. Those documents for which all logical expressions are true, in other words all constraints are satisfied, are the valid documents and the others are the invalid ones. Since expressions must be evaluated over documents, more specifically over document models, therefore model checking and different variations of model checking (e.g. local interpretation of subsumption and equivalence) can be used as a mechanism which evaluates constraints over document models. Instead of constraint system, we could also speak about model checking of Web documents.

Certainly, there are industrial techniques which also validates or invalidates Web documents. The most important ones DTD and Xshema. In the followings similarities and differences between these industrial techniques and our approach will be discussed in details.

A. Defining Pure Structures

The easiest way of defining a constraint by the logical approach is to define a direct structural dependency. Almost every kind of dependency can be defined by statements. Actually, there is no other definition in DTD than structural constraints. For example, the following statement is true for those documents in which every slideshow tag contains one or more slide tags.

<!ELEMENT slideshow (slide+)>
slideshow ⇒ **some child**.slide

B. Basic Attribute Types

The situation gets a little more complicated if type system should also be expressed. Surprisingly, the type system can be expressed by constraints. First of all, attributes and tags of a document have to be separated. Both elements can have a type, which is manifested as a type label at attribute types, and appear as an atomic predicate at tag types. This separation is quite similar at Xshema (DTD does not have type system at all), simple types describe the individual attributes and complex types describe the type system of the tags.

Unfortunately, the validity of basic attribute types, like boolean, integer, string, number, can not be checked by a pure logical formalism. They require knowledge on the deep level representation of these types, e.g. a string contains more than one Unicode characters, an integer can range from 0 to 65535, and so on. This kind of deep level representation should be supported by an additional non-logical constraint system.

However there are some properties of basic attribute types which can be represented. On the one hand type hierarchy or type combinations between basic attribute types can be handled by the formalism (see section E). On the other hand optional or required value of an attribute can be described (similarly as NOT NULL in database systems).

- Required value (any kind of value):
  attribute_type ⇒ **some value-of.(every)**
- Required value which has some special property:
  attribute_type ⇒ **some value-of.**('propety description')
- Default value:
  attribute_type **and not some value-of.(every)** 'trigger an outer modification which transforms the document model'

C. Basic Tag Types

Basic tag types can be described by the formalism much better than basic attribute types. A basic tag type constraint means that the tag has some attributes with attribute types.

- Simple tag type definition: In simple definition, the tag consists of attribute nodes with names and types:
  <attribute1 attribute_type1>,
  <attribute2 attribute_type2>,
  ...,
  <attributeN attribute_typeN>
  and no other attribute node is possible. It is quite similar to an object definition in an object oriented environment.

  tag_type ⇒ **all attribute-of.** ((**attribute-name** attribute1 **and type** attribute_type1) **or**...**or** (**attribute-name** attributeN **and type** attribute_typeN))
  tag_ type ⇒ **some attribute-of.** (**attribute-name** attribute1 **and type** attribute_type1)
  tag_ type ⇒ **some attribute-of.** (**attribute-name** attribute2 **and type** attribute_type2)
  ...
  tag_type ⇒ **some attribute-of.** (**attribute-name** attributeN **and type** attribute_typeN)
  tag_type ⇒ =N **attribute-of.** (**every**)

  First expression describes that all attribute nodes must have one of the attribute name and type. The middle expressions describe the existence of the nodes, whilst the last one states that no other attribute node is possible.
- Definition with variations: The previous constraint can easily be extended with variations. For example the second attribute can be from attribute_type1 or

attribute_type2.

tag_type ⇒ **some attribute-of. (attribute-name** attribute2 **and (type** attribute_type1 or type attribute_type2))

Other parts of the definition remain unchanged.

- Half definition: If we miss the last statement of the simple tag type definition, than other attributes than attribute1…attributeN can also appear with any types.
- Two direction definition: If equivalence is used instead of subsumption, we can get two direction definitions. For example, the following statement means that all tag_type must have an attribute with the name attribute1 and type attribute_type1. Hence every tag which has an attribute with the name attribute1 and type attribute_type1 must be in tag_type.

tag_type ⇔ **some attribute-of. (attribute-name** attribute1 **and type** attribute_type1)

DTD does not really handle types. In Xshema types are special simple tag type definitions, which might contain children node definitions as well (see complex tag types).

D. Complex Tag Types

A complex tag type definition contains not only the definition of the attributes but the definition of children nodes. Theoretically it could also contain the definition of parent nodes, but it is not very common design practice.

- Strict tag type definition: In strict tag type definition, the tag consists of children tag nodes:

tag_name1
tag_name2
…
tag_nameN

and no other children node is possible.

tag_type ⇒ **all child.** (tag_name1 **or** tag_name2 **or** … tag_nameN)
tag_type ⇒ **some child.** (tag_name1)
…
tag_type ⇒ **some child.** (tag_nameN)
tag_type ⇒ =N **child. (every)**

Certainly, a strict tag type definition might contain attribute definitions as well. In this case the statements presented at basic tag type definition should be added to the previous statements. A children node is not necessarily defined by names, but by type or other structural definition (like name and type pairs…). If so, tag_name must be replaced by the children node definition in the previous constraint. For example, the following constraint states that nodes which are typed by tag_type must have a children node which has an integer attribute. Name or type of the children node can be anything.

tag_type ⇒ **some child. (some attribute-of. (type** integer))

Complex type definitions of XSchema can be expressed by the previous statements. The only difference is that further restrictions can also be added to the occurrence of the children nodes. One of these restrictions is that children nodes can appear just in a well defined order.

- Definition with further restrictions (SEQUENCE): SEQUENCE (tag_name1, tag_name2) means that tag_name2 must be directly preceded by tag_name1.

tag_type ⇒ **some child.** (tag_name1 **and some next.** (tag_name2))

- Definition with further restrictions (CHOICE): CHOICE (tag_name1, tag_name2), CHOICE (tag_name3, tag_name4) means that the tag has two attributes, the first one is either tag_name1 or tag_name2, the second one is either tag_name3 or tag_name4.

tag_type ⇒ **some child.** (tag_name1 **or** tag_name2)
tag_type ⇒ **some child.** (tag_name3 **or** tag_name4)
tag_type ⇒ all child (tag_name1 **or** tag_name2 or tag_name3 **or** tag_name4)
tag_type ⇒ =1 **child.** (tag_name1 **or** tag_name2)
tag_type ⇒ =1 **child.** (tag_name3 **or** tag_name4)

- Definition with further restrictions (ALL): ALL restriction is pretty much the same as the strict tag type definition. The only difference is that not all child nodes must be presented.

Similarly to basic tag types, half definition or two direction definition could also be presented. Structure of these statements are pretty much the same as at basic tag type definitions. The only difference is that tag names should be used instead of attribute names.

E. Type Hierarchy

There are three structures which can be used to define a hierarchy between types. Simple and multiply inheritance are commonly used in object oriented environments, whilst type combination is a special structure which takes benefit of the pure logical approach.

- Simple type hierarchy: In simple type hierarchy we have two types, a base and a derived type. The derived one inherits and overwrites some of the properties of the base type. In object oriented environments this inheritance is limited to the attributes and methods of the objects. Unfortunately, in case of documents other constraints might also exist, which must be inherited or overwritten. Therefore we separate between permanent and over-writable statements, with separating each type to permanent and redefinable parts. In the following example tag_type1 is the base type, tag_type2 is the derived type.

permanent_baseType ⇒ 'DEFINITION'
redefinable_tag_type1 ⇒ 'DEFINITION'
tag_type1 ⇒ permanent_tag_type1 **and** redefinable_tag_type1
redefined_tag_type2 ⇒ 'DEFINITION'
extension_tag_type2 ⇒ 'DEFINITION'
type_def2 ⇒ permanent_tag_type1 **and** redefined_tag_type2 **and** extension_tag_type2

From an object oriented point of view, permanent and redefinable statements are similar as private or public (if exists protected) attributes of an object. As a simple

example, consider the following object oriented type declarations and the corresponding constraints.

type myType {
private attr1 string
protected attr2 int
}
type childType extends myType{
private override attr2 string
private attr3 string
}

permanent_myType ⇒ **some attribute-of. (attribute-name** attr1 **and type** string)
redefinable_myType ⇒ **some attribute-of. (attribute-name** attr2 **and type** int)
redefinable_myType ⇒ **all attribute-of. ((attribute-name** attr1 **and type** string) or (**attribute-name** attr2 **and type** int))
redefinable_myType ⇒ =2 **attribute-of. (every)**
myType ⇒ permanent_myType **and** redefinable_myType
extension_childType ⇒ **some attribute-of. (attribute-name** attr2 **and type** string)
extension_childType ⇒ **some attribute-of. (attribute-name** attr3 **and type** string)
redefined_childType ⇒ **all attribute-of. ((attribute-name** attr1 **and type** string) or (**attribute-name** attr2 **and type** string) or (**attribute-name** attr3 **and type** string))
redefinable_childType ⇒ =3 **attribute-of. (every)**
childType ⇒ permanent_myType **and** extension_childType **and** redefined_childType

It might seem to be a lot of statements at first sight, but it is constant in the size of objects connected by inheritance. Consequently, efficiency is not really effected by the inheritance. On the other hand, it is not necessary to write all statements by hand, an expression generating algorithm can easily generate the statements for the most common inheritance structures.

- Type hierarchy with multiply inheritance: In case of multiply inheritance, redefinable parts of the two base types must be overwritten and extended in the derived type.

childType ⇒ permanent_baseType1 **and** permanent_baseType2 **and** redefined_childType **and** extended_childType

- Type combinations: Type combination means another way, beside of inheritance, to construct new types from existing ones. It can be easily realized by the basic logical operators.

type ⇒ type1 **and** type2 (type restriction)
type ⇒ type1 **or** type2 (type extension)
type ⇒ type1 **or** type2 **and not** type3 (general combination)

In Xschema, a limited type hierarchy with simple inheritance is used, in which only new elements can be added to the derived types.

V. CONCLUSION

This paper has investigated the possibilities of realizing a constraint system for Web documents with the help of a domain specific description logic. The paper has introduced the architecture and the logic in details, and demonstrated through examples the application possibilities of the constraint system. We mainly focused our attention to express types and type hierarchies, since expressing simple structural properties with the help of a description logic is quite trivial and widely published.

In comparison with industrial technologies of the field, a logical approach has several benefits. It has well defined syntax, semantics and computational elements (basic reasoning services). Industrial techniques are usually lack of formal semantics or well-characterized algorithmic elements. Surprisingly, even expressive power of the logic is better at some points, than industrial technologies have.

VI. REFERENCES

[1] F. Baader, W. Nutt. Basic Description Logics, *In the Description Logic Handbook,* edited by F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider, Cambridge University Press, 2002. pp. 47-100.

[2] A. Borgida, R. J. Brachman. Conceptual Modeling with Description Logics *In the Description Logic Handbook*, edited by F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, P.F. Patel-Schneider, Cambridge University Press, 2002. pp. 359-381.

[3] Dieter Fensel, Ian Horrocks, Frank van Harmelen, Stefan Decker, Michael Erdmann, Michel C. A. Klein, OIL in a Nutshell, *Knowledge Acquisition, Modeling and Management*, 2000 pp. 1-16.

[4] Stefan Decker, Dieter Fensel, Frank van Harmelen, Ian Horrocks, Sergey Melnik, Michel C. A. Klein, Jeen Broekstra, Knowledge Representation on the Web, *Description Logics*, pp. 89-97, 2000.

[5] XML Base, *W3C Recommendation*, http://www.w3.org/TR/xmlbase/, 2001.

[6] XML Schema Part 0: Primer, *W3C Recommendation*, http://www.w3.org/TR/xmlschema-0/, 2001.

[7] D. Szegő, A Logical Framework for Analyzing Properties of Multimedia Web Documents, *Workshop on Multimedia Discovery and Mining, ECML/PKDD-2003*, pp. 19-30.

[8] D. Szegő, Using Description Logics in Web Document Processing, *SOFSEM*, vol. 2, 2004 pp.256-263.

[9] C.H. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.