XML-based Acquisition of Fine-Granular Structured Data by Grammar Morphisms

Markus Lepper, Baltasar Trancón y Widemann, Jacob Wieland

Technische Universität Berlin, Fakultät IV, Institut für Softwaretechnik und Theoretische Informatik, ÜBB, Sekr. FR 5–13, Franklinstr. 28/29, D–10587 Berlin, E-mail: {lepper,bt,ugh}@cs.tu-berlin.de

Abstract The DDD language and tool is a textual front-end for comfortable authoring XML documents and acquisition of structured but flexible data, esp. suited for legacy problems.
DDD has a two-layered architecture: The upper layer contains explicit mark-up for ergonomic data input, characterized by single-character-escaping and tag inference. In contrast the lower layer does not use mark-up, but parser definitions by extended regular expressions, which implicitly construct the data model from every-days textual representation of structured entities like addresses, names, domain specific notions etc.
The algorithms for transforming the front-end grammar defining terms into abstract data definitions and then into XML DTD definitions are given, as well as the algorithm for the parser. All algorithms are based on term rewriting.

Keywords: Data Acquisition, Semi-Formal Data, Accessibility, *inter language working*, XML, *Document Object Model*, DTD generation, SCHEMA

1 Goals, Contexts and Related Work

1.1 Goals of the DDD Research and Tool Project

The work presented herein is part of the DDD project [8]. DDD aims at making legacy text corpora as well as new out-comings of traditional text editing and authoring techniques accessible for automated text processing. This is achieved by adding the *minimum* of necessary mark-up, thereby presenting the text as „semi-structured data“ to automated processing, while at the same time keeping it readable and writable by humans.

The upper layer of DDD uses *explicit* tagging, characterized by (1) one single escape character and (2) context dependent tag inference, — two rather simple principles yielding astonishingly complex mathematical consequences, cf. [8].

```
#adressbook
  #entry  ugh  #name Wieland, Jacob #phone +49 30 / 314 -23456
               #adress TU Berlin, FR 5-30
  #entry  bt ...
```

which is translated into a structure like

```
<adressbook:entry adressbook:id="ugh">
   <name>Wieland, Jacob</name>
   <phone>+49 30 / 413 -23456 /phone>
   <adress> TU Berlin, FR 5-30</adress>
</adressbook:entry>
<adressbook:entry> ...
```

The definition of the structures are given as terms of a small declaration language (synchronously generating the inference rules for parsing *and* a data model), which then is mapped to an XML DTD in a canonical way, but behaves itself somehow more orthogonal than the latter.

So, w.r.t. *data acquisition* and derivation of *data-base* meta-models, only one single *text*-meta-model (given anyhow, as DDD schema or DTD or XML schema) is given, and there is ongoing research by others dealing with transformations, consistency conditions, derivations etc. between these two, cf. [**?**],[**?**].

1.2 The Parser Layer of DDD

The parser from foregoing example, using only explicit tagging, delivered the „phone number" just as an unstructured string of characters. This can hardly be called „data acquisition". On the other hand we do not want to tag such fine-granular data explicitly. It seems neither economic nor convenient to be forced to write this like

```
#phone #country 49 #city 30 #main 314 #extension 45678
```

Fine granular data in semi-formal documents is traditionally represented by relatively small text fragments which follow a certain grammar, like personal names, postal addresses, calendar dates, citations, domain specific abbreviations etc. Our proposal is, to re-model the *grammar* of the existing every-day's encoding as a formal language, — also to make legacy documents treatable with least effort[1].

The syntax of these constructs is in most cases *regular*, that is neither context-dependent nor even requiring recursive definitions, — nevertheless we provide means for recursive definition, which indeed will require most of the calculation effort.

Using standard regular expressions (plus a set of arbitrary given „Scanners") we can write a definition like ...

```
phonenumber =  ("+" #decDigit #decDigit ) (#decDigit +)
                "/" (#decDigit +)  ("-" (#decDigit +) )?
```

[1] These grammars are in many cases highly dependent of the *cultural* context of the data's origin. One hypothesis back-grounding the DDD project is that the re-modeling of these traditionally grown grammars as exact expressions of a formal language maybe serve as analyzation technique w.r.t. the underlying structure of the denotated data itself, which may also be culturally determined.

The resulting parser will match the phone number from the example above.

One central design goal of DDD is to avoid the necessity of redundant definitions, and concentrate all necessary declarations as much as possible. So we extend the front-end grammar defining regular expressions by a means for specifying *where to put* the matched character data for a given sub-expression.

The top-level data model is that of objects (or „Elements“), to which „some data content“ is linked to *associations*, which are indexed by names. This names are called „Tags“ in the following, — and will mostly indeed be mapped to XML „Tags“.

The type of the data delivered by applying a tag name to the association belonging to some source object is defined with the definition of this „source“ object’s type.

This data type can be just another element type, i.e. a singleton set containing one of the elements of this element type, or may be a *bag* of such objects, or a *sequence*, or even a sequence containing objects of *different* types.

Following the principle of „least points of declaration“, we enhance the notation for defining the front-end syntax by a means for prescribing the *tag*, the corresponding data structure to which will receive the matched content.

In our example we can prescribe these target tags of the matched character data by writing . . .

```
phonenumber = ("+" [Country #decDigit #decDigit ])? [Area #decDigit +]
              "/" [Main #decDigit +]  ("-" [Extension  #decDigit +] )?
```

What this really means, „putting“ the parsed character data into a data object identified by the given tags, depends on the data type of the element/tag combination, the inference of which is the main topic of this paper.

1.3 Data Model Derivation

But now the picture w.r.t. data modeling is more complex than in the tagged layer described above, because now *three* declarations have to be synchronized: (1) The grammar of the front-end, (2) the mathematical data model, and (3) the XML encoding.

It has to be analyzed, how the explicitely given tag indications together with the implicit behavior of the grammar (which cannot be easily overlooked even by a skilled user) can be mapped to a single mathematical data model.

This model has to be as fine-granular as necessary, but also as abstract as possible. The latter, because it is mostly not feasible to store all information of the front-end representation in a data base system efficiently, — generally it is not even desirable to do so, because the meant data content shall of course be realized independently from the accidentally chosen front-end representation.

In the example above, one can infer that there are tags `Country`, `Area`, etc., which all point to one single „character data“ object each. Consider e.g. one wants to encode more than one extension, which can easily achieved by just replacing a `?` by an `*` :

```
phonenumber = ("+" [Country #decDigit #decDigit ])? [Area #decDigit +]
               "/" [Main #decDigit +]  ("-" [Extension #decDigit +] )*
```

The mathematical model inferable from this definition is now just a *bag* of numbers, — but if the *order* of these numbers shall carry semantics, we have to derive a *sequences*. This fact must of course be indicated by the author of the data model, using a further keyword:

```
phonenumber = ("+" [Country #decDigit #decDigit ])? [Area #decDigit +]
               "/" [Main  #decDigit +]
                                  (#ORD "-" [Extension  #decDigit +] )*)
```

This declaration will generate an *ordered* list of extension numbers.

More complex declarations can be written using non-terminals, or defining tag contents explicitly. Both methods can be mixed freely, as shown in ...

```
phonenumber = country ? Area  "/" main   (#ORD extension *)

country     = "+" [Country #decDigit #decDigit ]
[Area]      = #decDigit +
main      = [Main #decDigit +]
extension     = "-" [Extension  #decDigit +]
```

2 Construction of Meta-Models and Transformations

2.1 Grammar Definition

The starting definitions of our front-end representation shall be given as a *module* $\mathcal{M}$.

First we construct a set of identifiers T which will be used as „Tag“ values, which may occur in the derived data model, and a set N of names of „non-terminals“ used for defining the front-end grammar. Further shall there be a set of predefined (or maybe user-defined) „scanners“, which later will be used to consume the front-end character data, thus representing the „terminals“ of our grammars[2].

All sets shall be disjoint[3], and no identifier of the former sets may start with some given „reserved“ character, for which we chose the „#“.

$$
\begin{array}{|l}
T : \mathbb{F}\ \mathit{Ident} \\
N : \mathbb{F}\ \mathit{Ident} \\
\mathit{Scan} : \mathbb{F}\ \mathit{Ident} \\
\hline
\mathsf{disjoint}\ \langle N, T, \mathit{Scan} \rangle \\
\forall\, t \in (N \cup T) \bullet t.1 \neq \texttt{\#}
\end{array}
$$

[2] The topic of user-defined scanner generation, and the related problems of white-space-preservation, mixed content generation etc. are not treated in this paper.

[3] We will use Z notation in the following, slightly enhanced (e.g. by type sums) and slightly weakened for sake of readability.

The set of „*Extended Regular Expressions*" is now given by the free type definition:

$$\begin{array}{ll} \mathcal{X} & ::= \mathcal{X} :: \mathcal{X} \mid \mathcal{X}\texttt{|}\mathcal{X} \mid \mathcal{X}\texttt{*} \mid \mathcal{X}\texttt{+} \mid \mathcal{X}\texttt{?} \mid \texttt{\#Perm(}\mathbb{N}\mathbb{N}\mathcal{X} \ldots \mathbb{N}\mathbb{N}\mathcal{X}\texttt{)} \\ & \mid \texttt{[}T\ \mathcal{X}\texttt{]} \mid \texttt{\#ORD}\,\mathcal{X} \mid \mathit{Scan} \end{array}$$

$$\begin{array}{ll} \mathit{Scan} == & \texttt{\#Ident} \mid \texttt{\#ident} \mid \texttt{\#Int} \mid \texttt{\#Float} \mid \texttt{"}\mathit{stringConstant}\texttt{"} \\ & \mid \mathit{Enum} \mid \ldots \end{array}$$

Our „extended" RegExps firstly behave just normal regular expressions: In the first line :: means sequencing (indicated as mere juxtaposition in standard Regexp, or e.g. as a comma-operator „`,`" in XML-DTD), `|` means alternative, and also `*`,`+`,`?` have their standard meaning[4].

Please note that our XRegExps need *not* to be „un-ambiguous" nor their languages „LL(1)". The semantics we give to XRegExps (cf. the implementation of the parser in section 3) resolve ambiguity by applying the principle of longest prefix match w.r.t. the *whole* XRegExp. For instance, when `p` has the following definition

```
 p =  (  a a (a|b) (a|b) | a a ) ( b b b b b )?
//                         a a     b b b b b      ccc
//       a a    b     b                           bbbccc
```

then matching of `aabbbbbbccc` against `p` will chose the *shorter* prefix for matching the first „`()`", as indicated in the first comment line, because the *overall* match will be longer.

Additionally we have three „interfaces" for controlling the data-flow, two „upward" and one „from below": The notation `[` t x`]` means that the content which (when parsing some text input) afterwards will be recognized as an expansion of α, shall be assigned as content to an element with the tag value t.

The central feature (`#ORD` x) is used to indicate explicitely that the *order* of the content carries some semantics.

Finally the call of a scanning function $s \in \mathit{Scan}$ is used to control the final parsing process of the input character data.

Then a definition module is given as a function

$$\mathcal{M} \ : \ \mathsf{Module} \ = \ (T \cup N) \ \rightarrow \ \mathcal{X}$$

2.2 Term Transformation

For manipulating the definition module we first define a basic rewriting function for XRegexp, which is given in figure 1. This function can be used for actually rewriting a term, or just for collecting information. As given in the figure, visit

[4] The `#Perm` operator is intended to make some evaluation structures directly usable in the front-end declaration language and is not treated in this paper.

Figure1 Visitor Function for XRegexp

RewriteXRegexp$[\tau]$

$$\begin{array}{l}
\mathit{visit} : \tau \times \mathcal{X} \rightarrow \tau \times \mathcal{X} \\
\textit{// should be:}\ \ \mathit{visit}[\beta]\ :\ \tau \times \beta \times \mathcal{X}\ \rightarrow\ \tau \times \beta \times \mathcal{X} \\
\textit{//}\qquad\qquad\qquad \mathit{visit}\ =\ \mathit{visit}[] \\
\\
x : \mathcal{X}, \pi : \tau, n : N, t : T, l : (N \cup T) \nrightarrow \mathbb{N} \times \mathbb{N}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{visit}_\pi\ t & = & \pi, t \\
\mathsf{visit}_\pi\ n & = & \pi, n \\
\mathsf{visit}_\pi\ \texttt{[}\ t\ x\ \texttt{]} & = & (\mathsf{visit}_\pi\ x).1,\ \ \texttt{[}\ t\ (\mathsf{visit}_\pi\ x).2\ \texttt{]} \\
\mathsf{visit}_\pi\ \texttt{\#ORD}\ x & = & (\mathsf{visit}_\pi\ x).1,\ \ \texttt{\#ORD}\ (\mathsf{visit}_\pi\ x).2 \\
\mathsf{visit}_\pi\ x\ \texttt{*} & = & (\mathsf{visit}_\pi\ x).1,\ (\mathsf{visit}_\pi\ x).2\ \texttt{*} \\
\mathsf{visit}_\pi\ x\ \texttt{+} & = & (\mathsf{visit}_\pi\ x).1,\ (\mathsf{visit}_\pi\ x).2\ \texttt{+} \\
\mathsf{visit}_\pi\ x\ \texttt{?} & = & (\mathsf{visit}_\pi\ x).1,\ (\mathsf{visit}_\pi\ x).2\ \texttt{?} \\
\mathsf{visit}_\pi\ x_1\ x_2 & = & (\mathsf{visit}_{\pi'} x_2).1,\ (\mathsf{visit}_\pi\ x_1).2\ ::\ (\mathsf{visit}_{\pi'}\ x_2).2 \\
\mathsf{visit}_\pi\ x_1\ \texttt{|}\ x_2 & = & (\mathsf{visit}_{\pi'} x_2).1,\ (\mathsf{visit}_\pi\ x_1).2\ \texttt{|}\ (\mathsf{visit}_{\pi'}\ x_2).2 \\
\quad \textbf{where}\ \pi' & == & (\mathsf{visit}_\pi\ x_1).1
\end{array}$$

is just a complicated way of writing the *identity* function. Derivations from this visitor are made by instantiating it with the types of the accumulator parameters, and overriding the function definitions, where all non-overridden methods are „inherited“ from figure 1, e.g. the cases of the function definition are treated like „methods“ in an OO-context[5].

E.g. the function „flat“ collects all values occurring at the leaves of an XRegexp and shows the typical and convenient way of working with „functionally encapsulated pseudo side-effects“:

$$\begin{array}{l}
\mathsf{flat}\ :\ \mathcal{X}\ \rightarrow\ \mathbb{P}(N \cup T) \\
\mathsf{flat}\,(x)\ =\ \mathsf{RewriteXRegexp}\,[\mathbb{P}(N \cup T)].\mathsf{visit} \\
\qquad\qquad \bigoplus\ \big(\mathsf{visit}_S(t : T)\ =\ S \cup \{t\} \\
\qquad\qquad\qquad \mathsf{visit}_S(n : N)\ =\ S \cup \{n\} \\
\qquad\qquad\qquad \big)\ \ (\{\}, x)\ \ .1
\end{array}$$

2.3 Normalization of Order-Respecting Directives

First step of the transformation of a given module $m \in \mathcal{M}$ is to remove all #ORD constructors, keeping in each branch only the top-level ones. This is done

[5] For convenience we only give one single definition of a multi-purpose visitor. When either the accumulator parameters or the rewriting facility is not used in the following, we simply omit the corresponding parameters and results from the notation informally.

by applying $\mathsf{normOrder}$ to all those $x \in \mathcal{X}$ which appear on the right side of definitions in m.

Since the #ORD constructor creates an order-respecting area which shall distribute even over the *contents* of all enclosed non-terminals, we capture this fact by doubling the set of non-terminals and their definitions, to be used in ordered and un-ordered contexts resp.

```
$N_{\mathsf{O}} \;=\; \{ n \in N \bullet n_{\mathsf{O}} \}$
$\mathcal{M}_1 = \mathcal{M} \cup \{ n \mapsto x \in \mathcal{M} \bullet n_{\mathsf{O}} \mapsto \mathsf{elimOrd}(x) \}$
$\mathsf{normOrder}, \mathsf{elimOrder} \;:\; \mathcal{X} \to \mathcal{X}$
$\mathsf{normOrder} \;=\; \mathsf{RewriteXRegexp}\,[].\mathsf{visit}$
 $\bigoplus$ $\big($$\mathsf{visit}\,(\texttt{\#ORD}\,x) \;=\; \texttt{\#ORD}(\mathsf{elimOrder}(x))$
      $\big)$

$\mathsf{elimOrder} \;=\; \mathsf{RewriteXRegexp}\,[].\mathsf{visit}$
 $\bigoplus$ $\big($$\mathsf{visit}\,\texttt{\#ORD}\,x \;=\; x$
      $\mathsf{visit}\,(n : N) \;=\; n_{\mathsf{O}}$
      $\mathsf{visit}\;\texttt{[}t\ \alpha\ \texttt{]} \;\mid\; \alpha \notin \mathsf{ran}\,\texttt{\#ORD} \;=\; \texttt{[}t\ (\texttt{\#ORD}\,\mathsf{visit}(\alpha))\ \texttt{]}$
      $\big)$
```

From here on we simply write N, meaning $N \cup N_{\mathsf{O}}$.

2.4 Extraction of In-Line Content Models

Now, as the „ordered“ information is pushed down into the XRegExp, we can extract and collect all definitions for element content, which are written „in-line“ inside an XRegExp:

```
$\texttt{exCD1} \;=\; \mathsf{RewriteXRegexp}\,[T \times \mathcal{X}].\mathsf{visit}$
        $\bigoplus$ $\big($$\mathsf{visit}_{\perp,\perp}\;\texttt{[}t\ \alpha\ \texttt{]} \;=\; t, \alpha, \texttt{CALL}(t)$
           $\mathsf{visit}_{t,\alpha}\; x \;=\; t, \alpha, x$
           $\big)$
```

$\texttt{exCD2} \;:\; \mathsf{Module} \to \mathsf{Module}$

$$\texttt{exCD2}(M) \;=\; \left\{ \begin{array}{l} M \oplus (t_i \mapsto c_i) \oplus (t_o \mapsto c_o) \\ \qquad \textbf{if}\ \exists\, t_o \in (N \cup T) \\ \qquad\qquad \bullet\ \texttt{exCD1}(\perp, \perp, M[t_o]) = (t_i, c_i, c_o) \wedge t_i \neq \perp \\ M\ \ \textbf{otherwise} \end{array} \right\}$$

$\mathsf{extractContentDefs} \;:\; \mathsf{Module} \to \mathsf{Module}$

$\mathsf{extractContentDefs} = \;\mathbf{Y}\ \texttt{exCD2}$

From here on we assume that $\mathcal{M}_2 \;=\; \mathsf{extractContentDefs}(\mathcal{M}_1)$, and simply write $\mathcal{M}$ for $\mathcal{M}_2$.

2.4.1 Example With the last both steps a definition module like ...

```
[C]  =  b  b  [C #ORD b  (#int | (#ORD b*) ) ]
b    = #int [C #int] | (#ORD #ident *)
```

. . . is transformed to . . .

```
[C]  =  b  b  [C #ORD bo  (#int | (     b*) ) ]
b    = #int [C #int] | (#ORD #ident *)
bo   = #int [C #int] | (     #ident *)
```

. . . and then the content definition for `C` are extracted .

```
[C]  =  b  b  C
     | #ORD b  (#int | (     bo*) )
     | #int
b    = #int C | (#ORD #ident *)
bo   = #int C | (     #ident *)
```

2.5 Abstracting Grammars to Cardinalities

The next steps do the abstraction from regular expressions to just counting the possible occurrences of Tags and Nonterminals: What we do is building a kind of „disjunctive normal forms“: For each given XRegExp (which is bound to the declaration of an tag or nonterminal) we consider all possible „paths“.

Since only net „effects“ are considered, and order must be discharged, we switch now to modeling these normal forms as „sets“[6].

The algorithm is given in figures 2 and 3. We model every „path“ through an XRegExp, i.e. every term of our virtual disjunctive normal form, as a pair of two sets, representing the ordered and unordered occurrences, resp., each given the minimal and the maximal count.

In a first step the constructors `?`, `+` and `*` are resolved. Then all paths in which recursion occurs are lifted to „infinity“, and all XRegExp where a „power recursion“ occurs, (or at least two distinct simple recursions in one same path!), are lifted to „infinity“ for their *whole* content.

Then all nonterminals are replaced by there summarized net content recursively, and at last all paths for a given Nonterminal/Tag are summarized.

2.6 Possible Data-Base Content Models

Now we have indeed a function

$$\mathsf{SumFin}\ :\ (T \cup N)\ \rightarrow\ (\ ((T \cup \mathit{Parse})\ \rightarrow\ (\mathbb{N} \times \mathbb{N})) \times ((T \cup \mathit{Parse})\ \rightarrow\ (\mathbb{N} \times \mathbb{N}))\ )$$

giving for any Tag value ($t \in T$) or any Non-Terminal ($n \in N$) the minimal and maximal multiplicity, in which a given parser or tag reference may occur in valid contents of n or t. This is given for „ordered“ and „unordered“ occurrences separately.

[6] Please note that using „sets“ is just a simplified way of writing term transformations, e.g. modulo a certain permutation order. All data („sets“ and „relations“) are given by construction, and all evaluation can always be done symbolically by transformations.

Figure2 Collecting net effects per path

$$
\begin{array}{ll}
\mathsf{effect} & = T \text{⇸} (\mathbb{N} \times \mathbb{N}) \\
\mathsf{effects} & = \mathsf{effect} \times \mathsf{effect}
\end{array}
$$

// repr. ORDered and UNORDered .

$$
\begin{array}{ll}
\mathit{combine} & : \mathsf{effects} \times \mathsf{effects} \times (\mathsf{effects} \to \mathsf{effects})^2 \;\to\; \mathsf{effects} \\
\mathit{combine}(e_1, e_2, \phi_1, \phi_2) & \\
 & = \mathsf{dom}\, e_2 \lhd e_1 \;\cup\; \mathsf{dom}\, e_1 \lhd e_2 \\
 & \quad \cup\, (\lambda\, x \bullet x \mapsto ((\phi_1(e_1(x).1, e_2(x).1), \phi_2(e_1(x).2, e_2(x).2)) \\
 & \qquad \llparenthesis\, \mathsf{dom}\, e_1 \cap \mathsf{dom}\, e_2 \,\rrparenthesis \\
\uplus, \diamond & : \mathsf{effects} \times \mathsf{effects} \to \mathsf{effects} \\
\uparrow, \downarrow & : \mathsf{effects} \to \mathsf{effects} \\
\hat{*}, \downarrow & : \mathsf{effects} \times \mathbb{N} \times \mathbb{N} \to \mathsf{effects} \\
e_1 \uplus e_2 & = \mathit{combine}(e_1, e_2, (+), (+)) \\
e_1 \diamond e_2 & = \mathit{combine}(e_1, e_2, (\mathit{min}), (\mathit{max})) \\
e \uparrow & = (\lambda\, x \bullet x \mapsto (e(x).1, \infty)) \;\llparenthesis\, \mathsf{dom}\, e \,\rrparenthesis \\
e \downarrow & = (\lambda\, x \bullet x \mapsto (0, e(x).2)) \;\llparenthesis\, \mathsf{dom}\, e \,\rrparenthesis \\
e \hat{*} (a, b) & = (\lambda\, x \bullet x \mapsto (e(x).1 * a, e(x).2 * b)) \;\llparenthesis\, \mathsf{dom}\, e \,\rrparenthesis
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{effs}, \mathsf{recEffs}, \mathsf{finEffs} & : \mathcal{X} \to \mathbb{P}\, \mathsf{effects} \\
\mathsf{effs}\, (n : N) & = \{(\{\}, \{n \mapsto (1,1)\})\} \\
\mathsf{effs}\, (t : T) & = \{(\{\}, \{t \mapsto (1,1)\})\} \\
\mathsf{effs}\, (x \,|\, y) & = \lambda\, a, b \bullet ((a.1 \diamond b.1), (a.2 \diamond b.2)) \;\llparenthesis \mathsf{effs}\, x \times \mathsf{effs}\, y \,\rrparenthesis \\
\mathsf{effs}\, (x :: y) & = \lambda\, a, b \bullet ((a.1 \uplus b.1), (a.2 \uplus b.2)) \;\llparenthesis \mathsf{effs}\, x \times \mathsf{effs}\, y \,\rrparenthesis \\
\mathsf{effs}\, (x?) & = \_\!\downarrow \;\llparenthesis \mathsf{effs}\, x \,\rrparenthesis \\
\mathsf{effs}\, (x+) & = \_\!\uparrow \;\llparenthesis \mathsf{effs}\, x \,\rrparenthesis \\
\mathsf{effs}\, (x*) & = \_\!\downarrow \;\llparenthesis\, \_\!\uparrow \;\llparenthesis\, \mathsf{effs}\, x \,\rrparenthesis \,\rrparenthesis \;=\; \mathsf{effs}((x+)?) \\
\mathsf{effs}\, (\mathtt{\#ORD}x) & = (\lambda\, s \bullet (s.2, \{\}) \;\llparenthesis\, \mathsf{effs}\, x \,\rrparenthesis
\end{array}
$$

// bec. of normalization there is only UNORDered content below
// — shift it to ordered content !

$$
\begin{array}{ll}
\mathit{usesMulti}, \mathit{usesOnce}, \mathit{uses} & : \mathsf{effect} \leftrightarrow N \\
\mathit{usesMulti} & = \{(e,t) \;|\; (e.1)(b).2 > 1 \;\vee\; (e.2)(b).2 > 1\,\} \\
\mathit{usesOnce} & = \{(e,t) \;|\; (e.1)(b).2 = 1 \;\vee\; (e.2)(b).2 = 1 \\
 & \qquad\qquad \wedge\; \neg\; \mathit{usesMulti}(a,b)\} \\
\mathit{uses} & = \mathit{usesOnce} \;\cup\; \mathit{usesMulti}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{calls}, \mathit{callsDirectMulti}, \mathit{callsDirectOnce}, \mathit{inCircleWith}} \\
 & : \; N \leftrightarrow N \\
\mathit{callsDirect} & = \{(a,b) \;|\; \exists\, e \in \mathsf{effs}\, \mathcal{M}[\mathsf{a}] \bullet \mathit{uses}(e,b)\} \\
\mathit{calls} & = (\mathit{callsDirect})^* \\
\mathit{callsDirectMultiple} & = \{(a,b) \;|\; \exists\, e \in \mathsf{effs}\, \mathcal{M}[\mathsf{a}] \\
 & \quad \bullet \mathit{usesMulti}(e,b) \\
 & \quad \vee\; (\mathit{uses}(e,b) \wedge \exists\, m \in N \,|\, m \neq b \bullet e(m).1.2 + e(m).2.2 > 0 \wedge\; m \;\mathit{calls}\; a) \\
 & \;\}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{callsDirectSingle} & = \mathit{callsDirect} \setminus \mathit{callsDirectMultiple} \\
\mathit{callsMultiple} & = \{(a,d) \;|\; \exists\, b, c : N \bullet \quad (a \;\mathit{calls}\; b \;\vee\; a = b) \\
 & \qquad\qquad\qquad\qquad \wedge\, (b \;\mathit{callsDirectMultiple}\; c) \\
 & \qquad\qquad\qquad\qquad \wedge\, (c \;\mathit{calls}\; d \;\vee\; c = d)
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{isMultiRec} & : \mathbb{P}\, N \\
 & = \{n : N \;|\; n \;\mathit{callsMultiple}\; n\,\}
\end{array}
$$

Figure3 Collecting net effects per path *(continued)*

$$\begin{array}{lcl}
\mathsf{recEffs}(n) & = & \left\{ \begin{array}{ll} (\mathsf{effs}\ \ n) \uparrow & \textbf{if}\ \mathsf{isMultiRec}(n) \\ (\lambda\, x \bullet \mathit{calcRec}(n, x, \mathcal{M})\,)\ (\!|\ \mathsf{effs}(\mathcal{M}[n])\ |\!)\ & \textbf{otherwise} \end{array} \right\} \\
\mathit{calcRec} & : & N \times \mathsf{effect} \times \mathsf{Module}\ \ \rightarrow \mathsf{effect} \\
\mathit{calcRec}(n, e, \mathcal{M}) & = & \left\{ \begin{array}{ll} e \uparrow & \textbf{if}\ \exists\ \ n' \in \mathcal{M} \bullet e\ \mathit{uses}\ n'\ \ \wedge\ \ n'\ \mathit{calls}\ \ n \\ e & \textbf{otherwise} \end{array} \right\}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{sumEff}, \mathsf{sumFin} & : & N\ \longrightarrow\ \mathsf{effect} \\
\mathsf{sumEff}(n) & = & \mathit{collEff}(\mathsf{recEffs}\ \mathcal{M}[n])
\end{array}$$

// accumulate effect of the alternatives !

$$\begin{array}{lcl}
\mathit{collEff} & : & \mathsf{effects}\ \longrightarrow\ \mathsf{effect} \\
\mathit{collEff}\ (\{e_0\} \cup e) & = & e_0\ \diamond\ \mathit{collEffs}(e) \\
\mathit{collEff}\ (\{\}) & = & (\{\}, \{\})
\end{array}$$

$$\mathsf{sumFin}(n) = \mathbf{Y}\ \mathit{collFin}(\{n\}, \mathsf{sumEff}\ \mathcal{M}[n])$$

// replace nonterminals by terminal content.

$$\begin{array}{lcl}
\mathit{collFin} & : & \mathsf{effect} \times \mathbb{P}\, N\ \longrightarrow\ \mathsf{effect} \\
\mathit{collFin}(e, S) & = & \left\{ \begin{array}{l} (\{m\} \lhd e) \uplus\ (\ \mathit{collFin}(\mathcal{M}[m], S \cup \{m\})\ \hat{*}(a.b)) \\ \qquad \textbf{if} \exists\, m : N \bullet\ m \mapsto (a, b)\ \in e\ \wedge\ \ m \notin S \\ e\ \ \textbf{otherwise} \end{array} \right\}
\end{array}$$

From this data we can derive the abstract definitions of the data model, which is appropriate for storing the parsed data according to the user's prescriptions:

$\mathcal{O}$		$\mathcal{U}$			
0	0	1	1	$b:\ \Delta \rightarrow \mathcal{T}$	$—\Diamond^{1}—\mathcal{T}$ **maybeXML – ATT**
0	0	0	1	$n:\ \Delta \rightarrow$ `OPT` $\mathcal{T}$	$—\Diamond^{0..1}—\mathcal{T}$**maybeXML – ATT**
0	0	a	$b > 1$	$n:\ \Delta \rightarrow$ `BAG` $\mathcal{T}$	$\mathbb{N}—\Diamond^{1}—\mathcal{T}$
_	$= 1$	_	1		
_	> 1	_	1	*// May be discussed.*	
_	$a > 1$	_	$b > 1$	`Error` *(„Automated Derivation Fails“)*	
_	≥ 1	0	0		
				let *ordererSubs* $=$ dom (sumFin(n).1)	
				$m\ =\ \sharp\ \mathit{ordererSubs}$	
	$m = 1$			$n:\ \Delta \rightarrow$ `SEQ` $\mathcal{T}$	$\mathbb{N}—^{1}\Diamond—\mathcal{T}$
	$m > 1$			$n:\ \Delta \rightarrow$ `SEQ` (`UNION`(*orderedSubs*))	$\mathbb{N}—^{1}\Diamond—(\mathbb{N} \times (\mathcal{T}_1 \uplus \mathcal{T}_2 \uplus \ldots)\ )$

The situation that there is one single „unordered“ and more than one „ordered“ occurences of the same sub-grammar may be very common. Consider e.g. the following parser, representing two possible ways of writing *personal names* taken from the cultural sphere of German language:

```
PersName = [Vorname #Ident][Nachname #ORD #Ident (#ident|#Ident)*]
         | [Nachname #ORD #Ident (#ident|#Ident)*]
              "," [Vorname #ORD #Ident +]
```

This imposes no real problems, as a *single* occurence can alwas be considered to be „ordered“.

2.7 XML Content Models

For automated transformation we define an XML DTD to be represented as a term of the grammar given by module_X:

$$\begin{array}{lcl} \mathsf{module}_X & ::= & \mathit{XmlDef} \ \mid \ \mathsf{module}_X \ :: \ \mathit{XmlDef} \\ \mathit{XmlDef} & ::= & \mathtt{EL}_X(T, \mathit{Xgr}) \ \mid \ \mathtt{ATT}_X(T, \mathit{Ident}, \{\mathtt{REQUIRED}, \mathtt{IMPLIED}\}) \\ \mathit{Xgr} & ::= & T \ \mid \ \mathit{Xgr} \text{","} \mathit{Xgr} \ \mid \ \mathit{Xgr} \ \mathtt{|} \ \mathit{Xgr} \ \mid \ \mathit{Xgr}\ \mathtt{?} \ \mid \ \mathit{Xgr}\ \mathtt{+} \ \mid \ \mathit{Xgr}\ \mathtt{*} \ \mid \ \varepsilon_X \end{array}$$

ε_X is needed for technical reasons and is neutral w.r.t. sequencing, e.g.

$$x\text{","}\varepsilon_X = \varepsilon_X\text{","}x = x$$

. The transformation from effects to module_X is given in figure 4.

3 The Parsing Process

Parsing is realized as parallel breadth-first match, implemented again by transformation of terms, or set of terms, resp.:

The following formulae expect some globally given character data, „visible“ only on Scanner level. Requests to and results from such scanners are represented by only the „position information“ $p \in \mathbb{Z}$.

The function parseResult, when called with a XRegExp from our normalized module (together with the starting position), will transform the definition term into the solution term. This term is element of the language given by

$$\mathit{matched} \ ::= \ \mathtt{[}t \ \mathit{matched}\mathtt{]} \ \mid \ S_{\mathbb{N}\times\mathbb{N}} \ \mid \ \mathit{matched} \ :: \ \mathit{matched}$$

it contains only references to scanners for which matching character data has been found (which is annotated as two position values), and tag constructors, which group the contained data as representing one element.

This structure can one-to-one be translated to an XML document, and fulfills all properties of the mathematical data model derived before.

3.1 A Simple Meta-Model for XML

Now we can give an exact definition of a well-formed XML *term*, (which can represent a well-formed document or fragment)[4]. Our definition has the additional property of disjointness of the both sets of names used for attributes („Attribute name“) and for elements („Tags“) resp.. We indicate this by choosing the name „**x***Wellformed*“

Figure4 Deriving XML DTD content models

$$
\begin{array}{ll}
[\![\_]\!]^X & : \ \mathsf{module} \ \rightarrow \ \mathsf{module}_X \\
\mathit{collXml} & : \ \mathbb{P}\ T \ \rightarrow \ \mathsf{module}_X \\
\mathit{collXD} & : \ T \ \rightarrow \ \mathsf{module}_X \\
\mathit{collCounter} & : \ T \times \mathbb{N} \times \mathbb{N}_\infty \ \rightarrow \ \mathsf{module}_X
\end{array}
$$

$$
\begin{array}{ll}
[\![\,\mathcal{M}\,]\!]^X & = \mathit{collXml}(\mathsf{dom}\,\mathcal{M} \ \cap \ T) \\
\mathit{collXml}\,(t \cup t_R) & = \mathit{collXD}(t, \mathsf{sumFin}(n), \varepsilon_X, \varepsilon_X, \varepsilon_X) \ :: \ \mathit{collXml}(t_R) \\
\mathit{collXml}\,(\{\}) & = \varepsilon_X
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{collXD}(t, (S_O, \{\}, \{\}, \alpha, \beta, \gamma)} \\
 & = \mathtt{EL}_X(t, (\beta\ \text{","}\ \mathtt{(}\gamma\mathtt{)*}\,)\ ) \ :: \ \alpha \\
\multicolumn{2}{l}{\mathit{collXD}(t, (S_O, \{a \mapsto (1,1)\} \cup S_U, \alpha, \beta, \gamma)} \\
 & = \left\{ \begin{array}{l} \mathit{collXD}(S_O, S_U, \alpha :: \mathtt{ATT}_X(n, a, \mathtt{REQUIRED}), \beta, \gamma) \\ \qquad \mathbf{if}(\mathsf{mayBeAtt}\ a) \wedge (\mathsf{userWantsAtt}\ a) \\ \mathit{collXD}(S_O, S_U, \alpha, \beta\ \text{","}\ a, \gamma) \\ \qquad \mathbf{otherwise} \end{array} \right\} \\
\multicolumn{2}{l}{\mathit{collXD}(t, (S_O, \{a \mapsto (0,1)\} \cup S_U, \alpha, \beta, \gamma)} \\
 & = \left\{ \begin{array}{l} \mathit{collXD}(S_O, S_U, \alpha :: \mathtt{ATT}_X(n, a, \mathtt{IMPLIED}), \beta, \gamma) \\ \qquad \mathbf{if}(\mathsf{mayBeAtt}\ a) \wedge (\mathsf{userWantsAtt}\ a) \\ \mathit{collXD}(S_O, S_U, \alpha, \beta\ \text{","}\ (a\mathtt{?}), \gamma) \\ \qquad \mathbf{otherwise} \end{array} \right\}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{collXD}(t, (S_O, \{n \mapsto (a, b > 1)\} \cup S_U, \alpha, \beta, \gamma)} \\
 & = \mathit{collXD}(t, S_O, S_U, \alpha, \beta\text{","}\,(\mathit{collCounter}(n, a, b)), \gamma\ )
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{collXD}(t, (\{a \mapsto (\_, n > 1)\} \cup S_O, S_U, \alpha, \beta, \gamma)} \\
 & = \mathit{collXD}(t, S_O, S_U, \alpha, \beta, (\gamma\text{","}\,a)\ )
\end{array}
$$

$$
\begin{array}{ll}
\mathit{collCounter}\,(n, a > 0, b) & = n\text{","}\,(\mathit{collCounter}(n, a-1, b-1)) \\
\mathit{collCounter}\,(n, 0, b \neq \infty) & = \mathtt{(}\ n\text{","}\,(\mathit{collCounter}(n, 0, b-1)\ \ \mathtt{)?} \\
\mathit{collCounter}\,(n, 0, 0) & = \varepsilon_X \\
\mathit{collCounter}\,(n, 0, \infty) & = n\ \mathtt{*}
\end{array}
$$

Xml_xWellformed_Term$[T_E, T_A, C]$

$$
\begin{array}{l}
\mathtt{\#self} \ \stackrel{def}{=} \ \mathit{Xml\_xWellformed\_Term}[T_E, T_A, C] \\
\mathit{tag} \ : \ T_E \\
\mathit{atts} \ : \ T_A \nrightarrow C \\
\mathit{cont} \ : \ \mathrm{seq}\,(\mathtt{\#self} \cup C) \\
\hline
T_E \cup T_A \ \subset \ T \\
T_E \cap T_A \ = \ \varnothing \\
\text{.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-} \\
\mathit{sups} : \ \mathbb{F}\ \mathtt{\#self} \\
\qquad = \ \{ e : \mathtt{\#self} \ \mid \exists\ n : \mathbb{N} \bullet e.\mathit{cont}.n \ = \ \theta\mathtt{\#self}\ \} \\
\qquad\quad // \ \mathit{weaker:} \ \mid \theta\mathtt{\#self} \in \mathsf{ran}\,(e.\mathit{cont})
\end{array}
$$

Figure5 The Parsing Algorithm

$$
\begin{array}{lcl}
\mathbb{N}_\bot & = & \mathbb{N} \cup \{\bot\} \\
\mathsf{scan} & : & \mathit{Scan} \times \mathbb{N} \;\rightarrow \mathbb{N}_\bot \times \mathbb{N} \\
\hat{\mathcal{X}} & ::= & \mathcal{X} \;\mid\; s_{\mathbb{N}\times\mathbb{N}} \;\mid\; \varepsilon \\
 & & \textbf{where } \varepsilon :: \alpha \;=\; \alpha :: \varepsilon \;=\; \alpha \\
\mathsf{parse1} & : & \mathcal{X} \times \mathbb{N} \;\rightarrow\; \mathbb{P}(\hat{\mathcal{X}} \times \mathbb{N}) \\
\mathsf{parse1}(s : \mathit{Scan}) & = & \left\{ \begin{array}{ll} \{\, s_{a,b} \} & \textbf{if } a \;\neq\; \bot \\ \{\} & \textbf{otherwise} \end{array} \right\} \\
 & & \textbf{where } (a, b) = \mathsf{scan}(s, p) \\
\mathsf{parse1}(x \mid y), p & = & \mathsf{parse1}(x, p) \;\cup\; \mathsf{parse1}(y, p) \\
\mathsf{parse1}(x :: \alpha), p & = & (\lambda\; \mu, \nu \bullet \mu :: \mathsf{parse1}(\nu, \alpha)\;)\;\;(\!|\mathsf{parse1}(x, p)|\!) \\
\mathsf{parse1}(x?) & = & \mathsf{parse1}(x, p) \;\;\cup\;\; \{(\varepsilon, p)\} \\
\mathsf{parse1}(x\texttt{+}) & = & \mathsf{parse1}(x :: (x\texttt{*}), p) \\
\mathsf{parse1}(x\texttt{*}) & = & \mathsf{parse1}((x\texttt{+})\texttt{?}, p) \\
\mathsf{parse1}(n : N, p) & = & \mathsf{parse1}(\mathcal{M}[n], p) \\
\mathsf{parse1}(t : T, p) & = & \texttt{[}t\; \mathsf{parse1}(\mathcal{M}[t], p)\;\texttt{]} \\
 & & \\
\mathsf{maxParse}(x, p) & = & \{(a, b) \in \mathsf{parse1}(x, p) \;\mid\; \neg\; \exists (a', b') \in \mathsf{parse1} \bullet b' > b \;\} \\
\mathsf{parseResult}(x, p) & = & \texttt{RANDOMCHOICE}\; \mathsf{maxParse}(x, p)
\end{array}
$$

Standard XML only deals with finite terms[7] , given like . . .

[7] By the way, a rather different thing is an XML document „*object*“ representation, which can be described as . . .

$$
\begin{array}{l}
\text{— } \mathit{Xml\_xWellformed\_Object}[T_E,\, T_A,\, C, \mathcal{I}] \text{ —} \\
\mathit{Xml\_xWellformed\_Finite\_Term}[T_E,\, T_A,\, C] \\
\texttt{\#self} \;\; \overset{def}{=} \;\; \mathit{Xml\_xWellformed\_Object}[T_E,\, T_A,\, C, \mathcal{I}] \\
\mathit{id} \;:\; \mathcal{I} \\
\hline
\forall\;\; o_1, o_2 \;:\; \texttt{\#self} \bullet o_1.\mathit{id} \;=\; o_2.\mathit{id} \;\;\Rightarrow\;\; o_1 \;=\; o_2 \\
\qquad\qquad\qquad \wedge\, (o_1.\mathit{cont} \,\cap\, o_2.\mathit{cont} \;\neq \varnothing) \;\;\Rightarrow\;\; o_1 \;=\; o_2 \\
\text{.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-} \\
\mathit{sup} :\; \texttt{\#self} \cup \{\texttt{\#topmost}\} \\
\qquad = \;\; \mu\; (\quad \{1 \mapsto \texttt{\#topmost}\;\} \\
\qquad\qquad\quad \oplus\, (\lambda\, e \bullet\;\; 1 \mapsto e\,)\; (\!|\mathit{sups}|\!) \;\;\; // = \{1\} \;\times\; \mathit{subs} \\
\qquad\qquad\quad)\;(1) \\
\theta\texttt{\#self} \;\;\notin\;\; \mathsf{ran}\; (\_.\mathit{sup})^{*}
\end{array}
$$

$Xml\_xWellformed\_Finite\_Term[T_E, T_A, C]$

$Xml\_xWellformed\_Term[T_E, T_A, C]$

// One possible rule for prohibiting circularity :

$\theta\texttt{\#self} \not\in \mathsf{ran}\ (\_.cont)^*$

.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-

$\theta\texttt{\#self} \not\in \mathsf{ran}\ (\_.sups)^*$

The transformation of the data acquired by the parsing process (given in the language „*matched*“) into such an XML term is straight-forward and mainly has to do the re-ordering of all „unordered“ elements according to the order in the derived DTD.

4 Related and Future Work

4.1 Grammars in Data Base research

The are numerous approaches in data base research dealing with grammars and grammar transformations. This is not astonishing, if one considers grammar transformations to be an abstractions of graph transformations, which is common place in data-base theory.

The work presented herein differs (to the best of our knowledge) from all existing work, as our focus is on *front-end* generation, integrated with schema definition, and on the problems which arise when *two* independently existing grammars (front-end and data definition) interfere.

The closest relation is to the works about automated derivations of schemas (conceptual or physical) from grammars (or attributed grammars, or XML DTDs), and vice versa, as in [15], [14], [5], [9], [7].

Only on the level of methodology there are (rather tight) relations to those approaches which use grammar morphism for transforming the *data* of semi-structured objects, as in [1] and [10], — for processing of queries, the usage of XML for presenting views and query results, and automated derivation of a corresponding DTD, as in [13], [12], — for model unification, as in [6], — and to all those approaches which propose a formal definition of SGML, XML (or semi-structured data in general) as a mathematical sound calculus, like [2] and [11].

An interesting survey on semi-structured data and XML is given in [16].

Furthermore there are common areas with *computer linguistics*, because the problem of different structural organization of front-end syntax and „data syntax“ is a central topic there, cf. e.g. [3].

4.2 Future Work

The principle of „single point of definition“, as applied in the architecture presented herein, suits well for local definitions of small grammars for „details“.

It does indeed conflict with principled of *compositionality*, e.g. when using one single front-end definition with different (e.g. legacy) data definitions.

While keeping the in-lined tag definitions as an option, we need some means for combining front-end grammars with separately existing data schemas *and/or* XML schemas.

The other possible kinds of derivation (front-end grammars from DTDs or data base schemas) may also turnout to be useful.

Further basic research in these both areas seems promising.

References

1. An Feng An and Toshiro Wakayama. SIMON: A grammar-based transformation system for structured documents. In *Fifth International Conference on Electronic Publishing*, pages 361–372, Darmstadt, Germany, 1994. Origination, Dissemination, and Design (EPODD) EP '94.
2. Chutiporn Anutariya, Vilas Wuwongse, Ekawit Nantajeewarawat, and Kiyoshi Akama. Towards a foundation for XML document databases. In *EC-Web*, pages 324–333, 2000.
3. Rens Bod and Ronald Kaplan. A data-oriented parsing model for lexical-functional grammar.
4. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation, `http://www.w3.org/TR/2000/REC-xml`.
5. Ronaldo dos Santos Mello and Carlos Alberto Heuser. A rule-based conversion of a dtd to a conceptual schema, November 2001.
6. Marc Gyssens, Jan Paredaens, and Dirk van Gucht. A grammar based approach towards unifying hierarchical data models. *SIAM Journal on Computing*, 23(6):1093–1137, 1994.
7. Gerti Kappel, Elisabeth Kapsammer, Stefan Rausch-Schott, and Werner Retschitzegger. X-ray – toward integrating xml and realational database systems. In *Conceptual Modeling – ER 2000.* Springer LNCS 1920, 2000.
8. Markus Lepper, Baltasar Trancon y Widemann, and Jacob Wieland. Minimze mark-up ! – natural writing should guide the design of textual modeling frontends. In *Conceptual Modeling — ER2001*, volume 2224 of *LNCS*. Springer, November 2001.
9. Dongwon Lee Murali Mani and Richard R. Muntz. Semantic data modeling using xml schemas, November 2001.
10. Makoto Murata. Transformation of documents and schemas by patterns and contextual conditions. In *PODP*, pages 153–169, 1996.
11. Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of xml schema languages using formal language theory. In *Extreme Markup Languages*, `http://www.cobase.cs.ucla.edu/tech-docs/dongwon/mura0619.ps`, august 2001.
12. Frank Neven and Jan Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *PODS '98. Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 11–17. ACM press, 1998.
13. Yannis Papakonstantinou and Victor Vianu. DTD inference for views of XML data. In *Symposium on Principles of Database Systems*, pages 35–46, 2000.
14. Patricia Rodriguez-Gianolli and John Mylopoulos. A semantic appproach to xml-based data integration, November 2001.
15. Airi Salminen and Frank Wm. Tompa. Grammars++ for modelling information in text. *Information Systems*, 24(1):1–24, 1999.
16. Dan Suciu. Semistructured data and XML. In *FODO*, pages 0–, 1998.