Minimize Mark-Up ! Natural Writing Should Guide the Design of Textual Modeling Frontends

Markus Lepper, Baltasar Trancón y Widemann, Jacob Wieland

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

Abstract Designing and implementing modeling frontends for domains in which *text* is predominant (it may be informal, semi-formal or formal) can and should benefit from using the evolving standard mark-up languages (SGML and XML), since standardization of interfaces, transmission and storage protocols as well as many valuable tools „come for free".
But the idiosyncratics of the existing mark-up concepts neither provide a structure clean enough to serve as foundation for syntax and semantics of exact modeling frontends, nor do they offer an *input format* feasible for text-based data maintanance.
Direct Document Denotation (DDD) as presented in this paper tries to remedy these defects: (1) it abstracts from the rough edges of XML, (2) it realizes a practical frontend processor for denotation of structured documents with special considerations to disabled users and voice controlled input, – and (3) is described completely and mathematically precise as a small system of transformation relations.
The theoretical basics and practical issues of DDD are discussed and a case study is reported.

Keywords: Data Acquisition, Semi-Formal Data, Accesibility, *inter language working*, XML, *Document Object Model*, DOM, *data binding*, SCHEMA

1 Introduction and Related Work

In the field of design and implemention of modeling tools there are three rôles in which XML will become more and more important:

– As underlying data exchange medium:
„Invisibly" to the user XML will be a coding standard for interchanging structured information in a relocatable format between different tools and different locations.
– As input format :
Using *screen mask editors* which are configured by DTD or SCHEMA, XML structured data can be directly acquisited and maintained by domain experts or less qualified personal.
– As mark-up language :
For shorter texts (e.g. configuration files) XML frontend syntax can be typed directly.

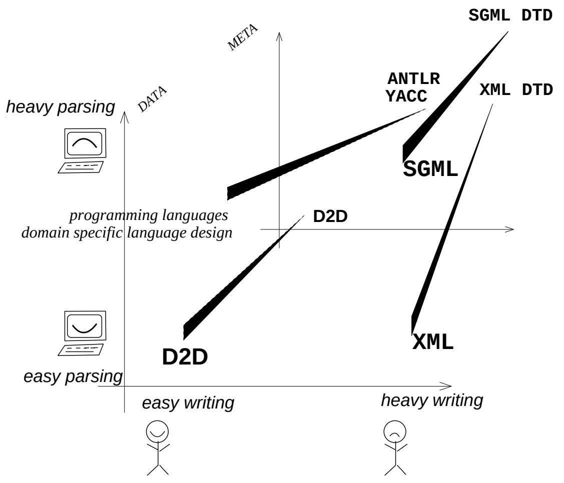

Figure1. Efforts needed for authoring vs. for parsing

But the last kind of application guarantees a maximum of inconvenience, even if supported by a syntax controled editor. This is somehow amazing, since the original SGML is intended to be a mark-up language for authoring purposes. The downgrading derivation of XML but aimed solely at simplificating the *computer's* task of parsing, not the human task of writing (see figure 1). While it was easily possible to write SGML-DTDs which were *not parsable at all*, XML takes the other extreme making the instances of all structure definitions parsable in LL(1), with the intention to enable the silliest browsers to display cool web pages with minimum knowledge of modern parsing techniques.

This requirement must lead to the opposite of an elaborated user-friendly textual input frontend.

The approach presented in this paper is called *Direct Document Denotation* (DDD) and tries to close this gap, thus realizing a fourth rôle of XML:

– XML is used as underlying Structure Definition generating a convenient *textual* input frontend automatically.

DDD is intendend to be useful in at least two cases: (1) There exists legacy data which is semi-formally textually coded and shall now be lifted to an exact and computer processible format with minimum effort, and (2) a textual modeling frontend for a given domain problem has to be created from scratch, and the designers and developers want to concentrate on semantic issues and get an input frontend for free.

We estimate DDD to be superiour to mask editors in those cases where (1) the kind of text data requires more flexibility, e.g. if recursive structure definitions are dominant, and (2) where the domain experts are strongly used to express themselves by authoring and are used to the comforts of „their“ text editing tool.

In both these cases a *minimal* mark-up syntax is beneficial, which follows the style of *natural writing* as established by the text gender or domain experts' tradition as close as possible.

DDD comes with a small definition language which *in parallel* defines content models and input syntax. Thus being a further schema definition language –

which would be among the things needed least at the moment, cf. [6] – the focus of its design and of this paper lies on the frontend generating aspect.

As an authoring tool the design of the DDD is inspired by the frontend behavior of TEX[5], `m4` [1], `curl` [7] and Lout [3], [4], concerning the means of recognizing the structure of input, e.g. the borders between parameter values, with minimal typing efforts and minimal „visual noise“[1].

It is an interesting fact that the most flexible frontend feature of TEX which allows the free definiton of LL(1) parsers for macro parameters is only used by very experienced „TEXhackers“ and hardly by „users“, even not by computer experts which do extensive programming on LATEX-level. So we decided to limit the degree of freedom for defining the input parsers, intending to make the definition of input patterns as well as their usage more transparent.

As a schema definition language DDD is restricted to a required minimum. It will be part of future work (see section 3 below) to correlate its expressiveness with other approaches. This should be done using a formal framework as in [8]. That paper of MAKOTO MURATA et.al. is as far as we know the first approach applying formal methods (i.e. mathematics) to XML. Since it is mainly about *validation* of given XML structures it cannot be compared with our work, because the parser approach of DDD only deals with documents which are „correct by construction“. Nevertheless [8] is an important and inspiring paper in our context.

2 Description and Specification of DDD

2.1 DDD Principles

DDD is a meta-language for defining and performing transformations of a document given as sequence of characters into a tree-like „Document Object Model“ (= DOM).

DDD stands for *D*irect *D*ocument *D*enotation, as the user can directly describe the tree structure of the document with a minimal count of keystrokes. It closely follows some fundamental paradigms found in SGML/XML, so that the resulting DOM is processable by eg. XSLT or other tools built on these standards, but DDD tries to abstract from all idiosyncratic distinctions made there. The tree constructed by DDD consists of *nodes*, which (under certain conditions of usage, see 2.6) can represent XML „`ELEMENT`s“ as well as their character data contents as well as „`ATTRIBUTE`“ values.

A concrete DDD system is a collection of parametrized definition modules[2]. A module primely defines node classes by declaring their structure (attribute list and content model) *together* with a convenient input representation. In addition a module can provide auxiliary objects like enumerations (used for `ATTRIBUTE`s of the appropriate type), macros, character sets etc. Each such definition is referred to by an identifier which is unique in the scope of the module.

[1] The generated output of these tools are `DVI`, Postscript, `HTML` or pure ASCII text resp., therefore the backends are not comparable.

[2] The current implementation does not yet support parametrization

property	EL	ATT
Can contain structure.	YES	no
Can be permutated arbitrarily in frontend syntax	no	YES
Can be assigned attribute values	YES	no
Are **implicitly** typed by their „name“; these types are totally unrelated	YES	no
Can be typed explicitely	no	YES
Must be typed explicitely	no	YES
Can have a default value	no	YES
Class names are unique w.r.t. a „module“	YES	no

Table1. Properties of ELEMENTs and ATTRIBUTEs in SGML/XML

A module can be defined by referring to other modules via `import` statements; node classes are defined by recurring to other (maybe imported) node definitions or to predefined parser types.

The basic operation of a DDD system is to apply the definition of one single „toplevel“ node to a text file. If the text file conforms to this definition, a Document Object Model is created, which can then be further processed, either by accessing it directly via an API or by writing it into a standard XML file[3].

2.2 Central Features of DDD

The basic principles of DDD seem to be rather trivial and aim solely at increasing the orthogonality of XML based object structures, trying to make exact reasoning and mathematical analysis feasible. But since the definitions try to cover all of the variants in XML usage, the consequences are not trivial at all.

These principles can be summarized:

– Unification of ELEMENT and ATTRIBUTE.

  We assume that the dichotomy ELEMENT and ATTRIBUTE come from an original intention to represent „object level“ text data and „meta level“ annotations. The design flaw is that belonging to the object- or the meta-level is decided when *interpreting* data structures, and should not happen on representation level: E.g. a table definition in – say – microsoft „ACCESS“ is „meta“ w.r.t. one window and at the same time „object“ in another window.

  This dichotomy led to a somehow „random“ conjunction of properties which from the abstract point of view should be freely combinable, see table 1.

  The (uncessarily) arising complexity is depicted nicely by figures 4 and 5 of [2], which still are simplified as in native XML „cardinality“ is not a property of an element type, but of the referring context.

– Inference of almost all *closing* tags.

  The original SGML approach of declaring tags as omittable has turned out to be unfeasible since the construction of parsers and the parsing itself can easily become NP-complete. The flaw was that „omissibility“ is a property of the *context* and can in general not be verified statically.

– Syntax directed automized tagging.

[3] In the current prototypical implementation a Xerces DOM is constructed[9], supporting the W3C DOM API.

– Partial parsing of incomplete, intermediate text versions.

When constructing a text in the „flow of authoring“, i.e. writing from the beginning to the end, many people prefer to work with *partially incomplete* texts, which are completed in later iterations. The DDD semantics support this method, – text structures not conformant to their corresponding structure definition, but to a *prefix* thereof, are accepted and tagged accordingly, either im- or explicitly.

Basically the DDD approach realizes a *two-staged* transformation :

On **top level**, with coarse granularity, there is an *explicit* tagging. For the sake of **accessibility** by the handicapped user – as one of the authors is – this level (1) uses only *one single* escape character, which (2) can be *redefined* for sake of maximal convenience in typing, and (3) can easily be mapped to *voice input*. The character data between (or „below“) these top-level tags will mostly be copied *opaquely* into the contents of a leaf node of the node tree under construction.

Alternatively there can be a further, „finer“ processing by a simple user-defined, token-based parser , generating at the **lower level** an *implicit*, syntax directed tagging.

The parser definition facility is intentionally rather limited. Its purpose is to define the analysing of *small* syntactic phaenomena appearing in everyday applications, as calendar dates, personal names and adresses, bible citations, short formulae etc. If parsing is required in a larger scale a different approach as in [11] is appropriate.

This two-staged approach leads to the fact, that in practice much expressive power is reached by the cooperation of *rather small* structures from both levels. This yields several advantages:

- Implementation can be done straight-forward. Since the structures tend to be rather small, all non-determinism/backtracking is strictly localized, thus performance is not really an issue.
- All interpretation of user input is exactly localized; error diagnosis and recovery is much less difficult to automize.
- Research on compositionality, reusability, refinement and extendability becomes feasible in practice.

In the following we give (1) an operational description of the semantics of the tagging level and (2) a function based denotation of the semantics of both levels in DDD. The former is more suited to the user, the latter allowing reasoning and supporting correct implementation.

2.3 DDD Case Study

A first DDD case study was performed in 2000 ([12]). Exctracts from the sources are given in appendix A to illustrate the following descriptions. They construct an XML page as service for protestant sacral musicians, listing all sundays and observances of the year 2000 of the German Protestant Church (= EKD) together with the related bible texts, which again are related to cantatas and motets, which on their part have been set to notes by different composers, requiring different ensembles of choir and solo voices[4]. The reader is asked to refer to these examples when reading the following specifications. We assume these sources – while based on German terms – nevertheless to be self-explanatory.

The data type definition modules (last example in appendix A) had of course been created by the developers, together with one prototypical data file for each toplevel node.

But the completion and maintenance of all DDD data file sources had successfully been done by total *laymen/laywomen* in the field of programming or even data acquisition.

2.4 Operational Semantics of the Explicit Tagging Level

When explaining the semantics of DDD to such a layman, who is e.g. a typist used to construct textual documents „from left to right“, an operational model of semantics seems adequate, which analyzes the text in the same direction. In future such an interpreter could be implemented as integral part of a syntax driven editor.

Roughly spoken: Whenever this interpreter starts its basic evaluation cylce, the name of a (currently visible) node class must be found at its current input pointer[5]. Then a *new node of the given class* is constructed and inserted into the tree built so far at the *lowest possible position* (as seen from the point of the last preceding insertion). Only if no such position exists, an error is thrown and the input is rejected.

This implements the feature of „inference of closing tags“, which is central for supporting Natural Writing.

All currently growing nodes the contents of which are „complete“ w.r.t. their contents' spefication are closed implicitly, – all those with *incomplete* content are marked as such and closed as well, iff the interpreter runs in a mode permitting incomplete documents. Otherwise an error is thrown.

This behavior allows handling of temporarily incomplete documents, and is also of main importance for Natural Writing.

If the contents' definition of the new node requests *text data*, then all input characters up to (but excluding) the next *escape character* are copied into the contents of this node. This text is typed as normal plain text, as the person doing the acquisition is used to do. After discarding the escape character the basic cycle is entered again.

Furthermore explicit closing tags are supported, too, in which case the automized closing inference is carried out up to the lowest node of the class requested for closing. A variant of the closing tag called „break tag“ permits the user to mark a node's content as incomplete explicitly.

[4] These special pages have by now disappeared from the webside for business reasons, but all other informative pages there are instances of DDD as well.

[5] Additionally there are a few built-in commands for controlling the interpreters behavior, which can also appear at these positions and can be considered as removed from the text after their successful execution.

$$
\begin{aligned}
\mathsf{Node} &: (\mathsf{Ident} \times \mathsf{seq\ Nodes}) \cup (\{\mathit{\$text}\} \times \mathsf{seq\ CHAR}) \;\rightarrow\; \mathsf{Nodes} \\
P &\in \mathsf{Parsers} \\
i, i_N, i_P, i_E &\in \mathsf{Ident} \\
\mathsf{UPCDATA} &= \mathsf{seq\ \ CHAR} \\
\kappa &\in \mathsf{UPCDATA} \\
j, k, m &\in \mathcal{N} \\
e_N &= i_N \mid i_P \\
&\mid e_N\ e_N \mid e_N \texttt{"|"}\, e_N \mid e_N\ \texttt{"+"} \mid e_N\ \texttt{"*"} \mid e_N\ j\ \texttt{"..*"} \mid e_N\ j\ \texttt{".."}k \mid \texttt{"("}\ e_N\ \texttt{")"} \\
e_P &= \texttt{"\#ident"} \mid \texttt{"\#numeric"} \mid \texttt{"\#string"}\ c \mid \ldots \mid \texttt{\#like}\ i_P \mid i_P \mid \texttt{"["}\ i\ e_P\ \texttt{"]"} \\
&\mid e_P\ e_P \mid e_P \texttt{"|"}\, e_P \mid e_P\ \texttt{"+"} \mid e_P\ \texttt{"*"} \mid e_P\ j\ \texttt{"..*"} \mid e_P\ j\ \texttt{".."}k \mid \texttt{"("}\ e_P\ \texttt{")"} \\
\llbracket \_ \rrbracket^{N}, \llbracket \_ \rrbracket^{N+}, \llbracket \_ \rrbracket^{N-} &: e_N \;\rightarrow\; (\ \mathsf{UPCDATA} \;\rightarrow\; \mathsf{seq\ Nodes}\ ) \\
\llbracket \_ \rrbracket^{P} &: e_P \;\rightarrow\; (\ \mathsf{UPCDATA} \;\rightarrow\; \mathsf{seq}\ (\mathsf{Nodes} \cup \mathsf{CHAR})\ ) \\
\llbracket \_ \rrbracket^{X} &: (e_P \cup e_N) \;\rightarrow\; (\ \mathsf{UPCDATA} \;\rightarrow\; \mathsf{seq\ Nodes}\ )
\end{aligned}
$$

Figure2. Basic Types and Notations

$$
\begin{aligned}
\llbracket\ \texttt{'"'}\ \mathit{const}\ \texttt{'"'}\ \rrbracket^{P} &= \{\langle\texttt{"const"}\rangle \mapsto \langle\texttt{"const"}\rangle\} \\
\llbracket \texttt{\#numeric} \rrbracket^{P} &= \{\texttt{"0"} \mapsto \texttt{"0"}, \texttt{"1"} \mapsto \texttt{"1"}, \ldots\} \\
\llbracket \texttt{\#ident} \rrbracket^{P} &= \{\texttt{"a"} \mapsto \texttt{"a"}, \texttt{"aa"} \mapsto \texttt{"aa"}, \ldots\} \\
\llbracket \texttt{\#string}\ \ c \rrbracket^{P} &= \{\ \kappa \frown \langle c\rangle \mapsto \kappa \mid \kappa\ \in \mathsf{seq}\ (\mathsf{CHAR} \setminus \{c\})\ \ \} \\
\llbracket \texttt{\#escape} \rrbracket^{P} &= \{\texttt{"\#"} \mapsto \texttt{"\#"}\} \\
\llbracket \texttt{\#select\ \ from}\ \ i_E \rrbracket^{P} &= \{\mathit{ident0a} \mapsto \texttt{"0"}, \mathit{ident0b} \mapsto \texttt{"0"}, \mathit{ident1a} \mapsto \texttt{"1"}, \ldots\} \\
\ldots
\end{aligned}
$$

$$
\begin{aligned}
\square &\in \{N, P, X\} \\
\llbracket\ e_1\ \ e_2\ \rrbracket^{\square} &= \llbracket e_1 \rrbracket^{\square}\ \ \widehat{\times}\ \ \llbracket e_2 \rrbracket^{\square} \\
A\ \widehat{\times}\ B \quad &= \quad \lambda((a,b),(c,d)) \bullet a^\frown c \mapsto b^\frown d \quad (\!| A \times B |\!) \\
\llbracket\ e_1\ \texttt{"|"}e_2\ \rrbracket^{\square} &= \llbracket e_1 \rrbracket^{\square} \ \ \cup\ \ \llbracket e_2 \rrbracket^{\square} \\
\llbracket\ e\ \texttt{"?"} \rrbracket^{\square} &= \llbracket e \rrbracket^{\square} \ \ \cup\ \ \{\langle\rangle \mapsto \langle\rangle\} \\
\llbracket\ e\ \texttt{"*"} \rrbracket^{\square} &= \llbracket e\ \ \texttt{0..*} \rrbracket^{\square} \\
\llbracket\ e\ \texttt{"+"} \rrbracket^{\square} &= \llbracket e\ \ \texttt{1..*} \rrbracket^{\square} \\
\llbracket\ e\ \ \ j\ \texttt{".."}\ \texttt{"*"} \rrbracket^{\square} &= \llbracket \_ \rrbracket^{\square} (\!| \{\langle e_1, \ldots, e_m\rangle \mid j \leq m, e_\_ = e\} |\!) \\
\llbracket\ e\ \ \ j\ \texttt{".."}\ \ k\ \ \rrbracket^{\square} &= \llbracket \_ \rrbracket^{\square} (\!| \{\langle e_1, \ldots, e_m\rangle \mid j \leq m \leq k, e_\_ = e\} |\!)
\end{aligned}
$$

$$
\llbracket \texttt{"("}\ e_1\ \texttt{"|"}\ \ \ldots\ \ \texttt{"|"}\ e_m\ \texttt{")"}\ \ \texttt{"!"}\ \ \rrbracket^{N} = \llbracket \_ \rrbracket^{N} (\!| \mathsf{permutations}\ \ \{e_1, \ldots, e_m\} |\!)
$$

Figure3. Extended Regular Expressions, generic for both Levels of Parsing

2.5 Denotational Semantics

The semantics of a mark-up language like DDD can be seen as a *syntactic transformation*. For this it is not too hard to give an exact specification, which can be found in figures 2 to 7. As notation we use a weak variant of a small subset of the Z notation, since the Z toolkit [10] provides exact definitions and handy conventions for a kind of „common sense“ set based mathematics, which in most

$$
\begin{array}{l}
\quad\llbracket\ \texttt{"}\sim\texttt{"}\ \rrbracket^{N} \ = \ \langle\rangle \ \mapsto \ \langle\rangle \\
\texttt{def node } i \texttt{ as parser : } e_P \ \texttt{"."} \\
\Longrightarrow \llbracket i \rrbracket^{N} \ = \ \lambda(a,b) \bullet \langle i\rangle ^\frown a \ \mapsto (\mathsf{Node}(i, \mathsf{filter}\ b)) \ \llparenthesis \llbracket\ e_P\ \rrbracket^{P} \rrparenthesis \\
\ \wedge \ \llbracket i \rrbracket^{P} \ = \llbracket \texttt{[}\ i\ e\ \texttt{]} \rrbracket^{P} \\
\ \wedge \ \llbracket \texttt{\#like } i \rrbracket^{P} \ = \llbracket\ e\ \rrbracket^{P} \\
\quad \llbracket \texttt{"["}\ i\ e\ \texttt{"]"} \rrbracket^{P} \ = \ \lambda(a,b) \bullet a \mapsto (\mathsf{Node}(i, \mathsf{filter}\ b)) \ \llparenthesis \llbracket\ e\ \rrbracket^{P} \rrparenthesis \\
\quad \mathsf{filter}\ \alpha \ = \begin{cases} \alpha & \textit{if}\ \ \mathsf{ran}\ \alpha\ \cap\ \mathsf{Nodes}\ = \{\} \\ \mathsf{squash}\ (\alpha \rhd \mathsf{Nodes}) & \textit{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\texttt{def node } i \texttt{ as empty } \texttt{"."} \\
\Longrightarrow \llbracket i \rrbracket^{N} \ = \ \langle i \rangle \mapsto \langle \mathsf{Node}(i, \langle\rangle)\rangle \\
\texttt{def node } i \texttt{ as plain text } \texttt{"."} \\
\Longrightarrow \llbracket i \rrbracket^{N} \ = \ \lambda(a,b) \bullet \langle i\rangle ^\frown a \ \mapsto \langle \mathsf{Node}(i, b)\rangle \ \llparenthesis \llbracket\ \texttt{\#string}\ \ \texttt{"\#"}\ \rrbracket^{P} \rrparenthesis \\
\texttt{def node } i \texttt{ as grammar : } e_N \ \texttt{"."} \\
\Longrightarrow \llbracket i \rrbracket^{N} \ = \ \lambda(a,b) \bullet \langle i\rangle ^\frown a \ \mapsto \langle \mathsf{Node}(i, b)\rangle \ \llparenthesis \llbracket\ e_N\ \rrbracket^{N} \rrparenthesis \\
\texttt{def node } i \texttt{ as mixed : } i_1, \ldots, i_k \ \ \texttt{"."} \\
\Longrightarrow \llbracket i \rrbracket^{N} \ = \ \lambda(a,b) \bullet \langle i\rangle ^\frown a \ \mapsto \langle \mathsf{Node}(i, b)\rangle \\
\qquad \llparenthesis\ \llbracket\ [\mathit{\$text}\ \texttt{\#string}\ \ \texttt{"\#"}]\ \ (\ \texttt{"}\sim\texttt{"}\ [\mathit{\$text}\ \texttt{\#string}\ \ \texttt{"\#"}] \mid i_1 \mid\ \ldots\ \mid i_k\ \ ) *\ \rrbracket^{X} \rrparenthesis
\end{array}
$$

$$
\begin{array}{l}
\mathsf{mixembed}(a,b,i) = \{a \mapsto b\} \cup \ \{\ a ^\frown \kappa ^\frown \langle \texttt{"\#"}\rangle \mapsto b ^\frown \langle \mathsf{Node}(\mathit{\$text}, \kappa)\rangle \\
\qquad\qquad\qquad\qquad\qquad\quad \mid \neg \exists \kappa', \kappa'' \bullet \kappa = \kappa' ^\frown \kappa'' \wedge \kappa ^\frown \kappa' \in \mathsf{dom} \llbracket i \rrbracket \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge \ \ a.(\#a) \neq \texttt{"\#"} \ \ \wedge \ \ \texttt{"\#"} \notin \kappa \ \ \} \\
\llbracket i \rrbracket^{X} \ = \ \lambda(a,b) \bullet \mathsf{mixembed}(a,b,i) \ \llparenthesis \llbracket i \rrbracket^{N} \cup \llbracket i \rrbracket^{P} \rrparenthesis
\end{array}
$$

$$
\llbracket i \rrbracket^{N+} \ = \ \llbracket i \rrbracket^{N} \ \cup \ \lambda((a,b),c,d) \bullet a ^\frown c ^\frown d \mapsto b \ \ \llparenthesis \llbracket i \rrbracket^{N} \times \{\langle \texttt{"/"}, i\rangle, \langle\rangle\} \times \mathsf{seq}\ \{\texttt{"\#"}\} \rrparenthesis
$$

Figure4. Semantics of the Node and Parser Definition Statements

$$
\begin{array}{l}
\quad \mathsf{parse}^{+/-} \ \ : \ \ \mathsf{Ident} \times \mathsf{UPCDATA} \ \ \longrightarrow \mathsf{Nodes} \\
\mathsf{parse}^{+/-}\ (i, \kappa) = \mu \ \ (\ (\langle i \rangle ^\frown \kappa) \ \ \lhd \llbracket i \rrbracket^{N+/-} \ \ )
\end{array}
$$

Figure5. Applying the top-level parsing function to a text

parts is readable without further explanation. Some notations special to Z are explained in appendix B[6].

The Document Object Model is defined as a Node, which is a free type either with an identifier as constructor and a sequence of Nodes as data, or with the special reserved identifier $\mathit{\$text}$ as constructor and a character sequence as data.

Basic paradigm is to define *parsing functions* of type seq CHAR $\rightarrow$ Node. The semantic function $\llbracket \_ \rrbracket^{P}$ assigns such a parsing function to the predefined primitive parsers as well as to the names of user defined parsers, $\llbracket \_ \rrbracket^{N+}$ and $\llbracket \_ \rrbracket^{N-}$ do the same on the upper level of node class names, the former forbidding, the latter allowing incomplete documents.

[6] Furthermore we use some self explaining slight extensions to the original Z notation, eg. using mere juxtaposition of elements for building singleton lists and concatenations : $\llbracket\ i\ \alpha\ x\ \rrbracket$ is an abbreviated notation for $\llbracket\ \langle i \rangle\ ^\frown\ \alpha\ ^\frown\ \langle x \rangle\ \rrbracket$.

$\kappa \in \mathsf{UPCDATA}$
$\texttt{def node } i_0 \texttt{ as plain text .}$
$\texttt{def parser } p \texttt{ ... .}$
$q = p \quad \vee \quad q = \langle \texttt{\#like},\ p \rangle$

$\texttt{def node } i \texttt{ as complex :}$
$\quad \texttt{plain prefix : } i_0 \texttt{ ";"}$
$\quad \texttt{grammar : } e_N \texttt{ ";"}$
$\Longrightarrow \llbracket i \rrbracket^N = \ \lambda(a,b) \bullet \langle i \rangle ^\frown \kappa ^\frown \langle \texttt{"\#"} \rangle ^\frown a \ \mapsto \ \mathsf{Node}(i, \mathsf{Node}(i_0, \kappa)^\frown b) \quad (\!| \llbracket e_N \rrbracket^N |\!)$

$\texttt{def node } i \texttt{ as complex :}$
$\quad \texttt{plain prefix : \#content ";"}$
$\quad \texttt{grammar : } e_N \texttt{ ";"}$
$\Longrightarrow \llbracket i \rrbracket^N = \ \lambda(a,b) \bullet \langle i \rangle ^\frown \kappa ^\frown \langle \texttt{"\#"} \rangle ^\frown a \ \mapsto \ \mathsf{Node}(i, \mathsf{Node}(\mathit{\$text}, \kappa)^\frown b) \quad (\!| \llbracket e_N \rrbracket^N |\!)$

$\texttt{def node } i \texttt{ as complex :}$
$\quad \texttt{plain prefix : } q \texttt{ ";"}$
$\quad \texttt{grammar : } e_N \texttt{ ";"}$
$\Longrightarrow \llbracket i \rrbracket^N = \ \lambda((a,b),(c,d)) \bullet \langle i \rangle ^\frown a ^\frown c \ \mapsto \ \mathsf{Node}(i, b^\frown d) \quad (\!| \llbracket q \rrbracket^N \ \times \ \llbracket e_N \rrbracket |\!)$

$\texttt{def node } i \texttt{ as complex :}$
$\quad \texttt{grammar : } e_N \texttt{ ";"}$
$\quad \texttt{mixed with chars : } i_1 \texttt{ "," ... "," } i_k \texttt{ ";"}$
$\Longrightarrow \quad \llbracket i \rrbracket^N \ = \ \lambda((a,b),(c,d) \bullet \langle i \rangle ^\frown a ^\frown c \ \mapsto \langle \mathsf{Node}(i, b^\frown d) \rangle$
$\quad (\!| \llbracket e_N \rrbracket^N \times \llbracket \ [\mathit{\$text} \ \texttt{\#string "\#"}] \ (\texttt{"}\sim\texttt{"} \ [\mathit{\$text} \ \texttt{\#string "\#"}] \mid i_1 | \ldots | i_k) * \rrbracket^X \ |\!)$

$\texttt{def } i \texttt{ as complex :}$
$\quad \beta_1$
$\quad \texttt{once: } a_1 \texttt{ ,.., } a_k \texttt{;}$
$\quad \texttt{optional: } b_1 \texttt{ ,.., } b_l \texttt{;}$
$\quad \texttt{grammar: } \gamma \texttt{ ;}$
$\quad \beta_2$

$==$

$\texttt{def } i \texttt{ as complex :}$
$\quad \beta_1$
$\quad \texttt{grammar : (} a_1 | \ldots | a_k \mid b_1\texttt{?} | \ldots | b_l\texttt{?) ! } \gamma \texttt{ ;}$
$\quad \beta_2$

Figure6. Semantics of „Complex“ Node Definitions

The semi-formal notation in figure 4 using „$\Longrightarrow$“ shows how a syntactic construct in a definition module defining a new node or parser induces the semantic function $\llbracket \_ \rrbracket^{P/N+/N-}$ for the newly defined identifier by combining other parsing functions.

Aim of the game is to define a parsing function for the top-level node of the document, which is applied to the text as a whole as seen in figure 5[7].

The reader who considers these definitions somewhat complicated may be assured that the given formulae *completely* describe the DDD transformation framework, and that they can easily be lifted to a fully formal specification.

Please compare this *specification* to the 200 pages of plain english *description* of XML.

[7] Function application is written as finding the only element (by using „μ“) of a domain restriction.

$$\begin{aligned}
A_{i,\kappa} &= \{\alpha \mid \exists\kappa' \bullet [\![\, i \,]\!]^{N} \; (\langle i\rangle^\frown\kappa^\frown\kappa') \;=\; \alpha\} \\
\mathsf{pretrees} &: \mathsf{seq}\ \ \mathsf{Nodes} \;\longrightarrow\; \mathcal{P}\ \ \mathsf{seq}\ \ \mathsf{Nodes} \\
\mathsf{pretrees}(\mathsf{Node}(\$text,\kappa)) &= \{\ \mathsf{Node}(\$text,\kappa)\ \} \\
\mathsf{pretrees}(\mathsf{Node}(i,\alpha)) &= \{\ \mathsf{Node}(i,\alpha)\ \} \\
&\cup \{\ \mathsf{Node}(i,\alpha') \mid l \;=\; \#\,\alpha' \;\;\wedge\;\; \forall k \in 1 \ldots (l-1) \bullet \alpha.k \;=\; \alpha'.k \\
&\qquad\qquad\qquad\qquad\qquad \wedge\;\; \alpha'.l \;\in\; \mathsf{pretrees}\ \alpha.l\ \ \} \\
\mathsf{markinc} &: \mathsf{Nodes} \times \mathsf{Ident} \;\longrightarrow\; \mathsf{Nodes} \\
\mathsf{markinc}(\mathsf{Node}(i,\alpha^\frown\langle n\rangle),j) &= \mathsf{Node}(i,\alpha^\frown\langle\mathsf{markinc}(n,j)\rangle^\frown\langle\mathsf{Node}(\$incomplete,j)\rangle\ \ ) \\
\alpha_{i,\kappa} &= \mu\ (\ \ \bigcap \mathsf{pretrees}(\!|A_{i,\kappa}|\!)\ \ ) \\
[\![\ i\ ]\!]^{N-} = [\![\ i\ ]\!]^{N+} &\cup\ \ \{\forall\kappa | \kappa \notin [\![\, i \,]\!] \;\;\wedge\;\; A_{i,\kappa} \neq \{\} \bullet\;\; \langle i\rangle^\frown\kappa \;\;\mapsto\;\; \mathsf{markinc}(\alpha_{i,\kappa},i)\ \} \\
&\cup\ \ \{\forall\kappa | A_{i,\kappa} \neq \{\} \bullet\;\; \langle i\rangle^\frown\kappa^\frown\langle\texttt{"///"},i\rangle \;\;\mapsto\;\; \mathsf{markinc}(\alpha_{i,\kappa},i)\ \}
\end{aligned}$$

Figure7. Semantics of Incomplete Node Denotations

2.6 Additional Information in Definition Modules

The grammar of DDD definition modules is (of course `;-)` a self-application and provides firstly the basic defining statements for node classes, the semantics of which are depicted in figure 4 and have been discussed above. Furthermore there are additional *modifier*s which can be applied to each single node class definition[8].

The most important of these is the modification attribute `xmlrep`, which determines the encoding when writing out the internal document tree to an external XML file or when masquerading as W3C DOM. Hairy context conditions are checked by the existing implementation to ensure feasibility: a node declared as `ATTRIBUTE` must not appear more than once in the language of the grammar of *any* node, a node used for `plain prefix` in a complex definition must be of flavour `parser` or `plain text`, and so must be each node with `xmlrep` equal to `CDATA`, etc.

To minimize the count of required constructs there is no special means for defining abstract grammars. Instead there is a „`#like` *Ident*“ construct which makes the pure grammar definition of a node class accessible, – speaking with [8] it shortens a production in P to its corresponding content model, which is a production in P2.

2.7 Extendability and Refinement

The seperation into two layers of transformation yields localization and thereby supports stepwise refinement of transformation definitions.

In a project where a new text format is defined from scratch, new node classes can be defined and alternatives can *incrementally* be added to some contents grammar, as soon as necessity arises (see future work below, 3).

In a project of tagging existing legacy semi-formal text data in a first approach only the explicit, coarse tagging can be defined and inserted into the text.

[8] The `modifier` grammar is meant as user-extendable. There are modifiers foreseen to support connection to a type system, to configure a syntax controlled editor etc.

As soon as a more structured access turns out to be necessary, parser definitions can be added, requiring re-editing only of these parts which do not yet confirm to the new grammar.

In our case study (appendix A) there was no parser provided for the „plain text header“ of "psalm", "lesung" etc., which was stored as plain text in a node class element "quelle". To give *finer control* of correctness of the input we can add . . .

```
def public enum books mos = 0,   mosis = 0,
                      gen = 1,   genesis = 1,
                      ex  = 2,   exodus = 2,
                      lev = 3,   leviticus = 3,
                      ....
                      apoc = 666 .
def enum subverse  a,b,c,d,e,f,g,h .

def public parser cites    cite  (";" cite) * .
def public parser cite     #numeric ?   #select from books
                           pericope   (  "," pericope )*  .
def parser pericope        #numeric ?
                           #numeric (#select from subverse)?
                           ( "-" #numeric (#select from subverse)? )? .
```

"cites" now accepts input like

```
            2 mos 1 14- 16, 18, 2 1 ; apoc 1 12 - 14
```

which is linked to the existing definition by

```
            def quelle like cites .
```

"cites" delivers all correctly parsed character substrings as a whole simply as plain text. If one wants a structured node representation the names of the subnodes have to be provided in brackets "[]" :

```
def public parser cite  [booknumber #numeric]? [book #select from books]
                        pericope
                        (  "," pericope )*  .
def parser pericope     [chapter #numeric] ?
                        [firstverse #numeric (#select from subverse)?]
                ( "-" [lastverse #numeric (#select from subverse)?] )? .
```

3 Conclusion and Future Work

Direct Document Denotation, DDD, a modular system for generic definition of transformations from a slightly mark-uped text document into an XML conformant document object model has been presented. Its semantics are described formally, – with some weakening for sake of readability.

A prototypical implementation has been successfully applied to enable non-professionals doing data acquisition and maintanance of text oriented, semi-structured data.

While this seems a promising start (with much work still to do on the engineering side) there are still open questions on the research side: How can DDD *parser* definitions be converted into into sensible XML schema definitions automatically? – How far can a DDD frontend definition be derived from an XML schema automatically? – How can a given XML document be converted into DDD format? – Of what practical use would it be to introduce „offside rule“ in the input format? – And how can this be formalized? –

The most interesting research concerns „evolving schemata“: Since, as explained above, DDD is intended for gradual refinement of tagging and parsing of legacy data „on demand“, the rules ensuring compositionality of definition modules have to be explored, thereby bringing together established and new results in parser theory and data base theory.

4 Acknowledgements

We owe special thanks and respect to our dear colleagues Wolfgang Grieskamp (now at microsoft research), Michael Cebulla and Peter Pepper, the head of the ÜBB team. Without the intensive and exciting discussions on their results, our problems and the common visions this paper could not have been written.

References

1. Free Software Foundation, `http://www.seindal.dk/rene/gnu/man/`. `m4` *Manual*.
2. 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.
3. Jeffrey H. Kingston. *A New Approach in Document Formatting*. `http://snark.ptc.spbu.ru/~uwe/lout/overview.ps.gz`, 1992.
4. Jeffrey H. Kingston. *The* Lout *Homepage*. `http://snark.ptc.spbu.ru/~uwe/lout/lout.html`, 2000.
5. Donald E. Knuth. *The TEXbook*. Addison-Wesley, 1987.
6. Dongwon Lee and Wesley W.Chu. Comparative analysis of six xml schema languages. *ACM SIGMOD record*, 29(3), 2000.
7. MIT Laboratory for Computer Science, http://curl.lcs.mit.edu/curl/wwwpaper.html. *curl*.
8. 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.
9. Apache XML Project. *Xerces Java Parser*. Apache Software Foundation, `http://xml.apache.org/xerces-j`.
10. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
11. Baltasar Trancon y Wideman, Markus Lepper, Jacob Wieland, and Peter Pepper. Automized generation of abstract syntax trees represented as typed dom xml. In *Proceedings of the ICSE 2001 First International Workshop on XML Technologies and Software Engineering (XSE'01)*, 2001.
12. Zacharias Musikversand, `http://www.kirchennoten.de`. *DDD Generated Web Pages*, 2000.

A Examples Extracted From the Case Study Sources.

Source Text 1

```
#d2d text using musicaSacra : annusLiturgicus_EKD ;

   tag Karfreitag #datum 21. April 2000 #

       wochenspruch Joh 3,16 #~
          Also hat Gott ## die Welt geliebt, #nl
          daß er seinen eingeborenen Sohn gab, #nl
          damit alle, die an ihn glauben, nicht verloren werden, #nl
          sondern das ewige Leben haben.#

          werk  Motette "Also hat Gott die Welt geliebt"#
               vertonung Schütz      #vox 5 #
               vertonung Hufschmidt  #vox Soli S+T + 4 #

       psalm  LV I Phil 2, 8; LV II Phil 2, 10, 8b  #
          werk      Motette "Ecce homo" #
               vertonung Reda        #vox 4
                 #bemerkung Frühes Werk, nicht einfach,
                    #spitzenton T = a', #spitzenton S = a''
                 #
 eof
```

Ende of Source

Source Text 2

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<celebrationdays>
  <tag datum=" 21. April 2000 " name=" Karfreitag ">
      <wochenspruch quelle="  Joh 3,16">  Also hat Gott # die Welt
          geliebt, <nl/>   daß er seinen eingeborenen Sohn gab,<nl/>
          damit alle, die an ihn glauben, nicht verloren werden,<nl/>
          sondern das ewige Leben haben.
              <werk> Motette &quot;Also hat Gott die Welt geliebt&quot;
                 <vertonung composer=" Schütz      " voices=" 5 "/>
                 <vertonung composer=" Hufschmidt  " voices=" Soli S+T + 4 "/>
              </werk>
       </wochenspruch>
       <psalm>  LV I Phil 2, 8; LV II Phil 2, 10, 8b;
              <werk>   Motette &quot;Ecce homo&quot;
                  <vertonung composer=" Reda        " voices=" 4 ">
                       <bemerkung> Früher Reda, nicht einfach,
                             <spitzenton voice="T" pitch=" a' "/>,
                             <spitzenton voice="S" pitch=" a'' "/>
                       </bemerkung>
                 </vertonung>
            </werk></psalm></tag>
</celebrationdays>
```

Ende of Source

Source Text 3

```
#d2d module musicaSacra ; import  richtext ;
def public node annusLiturgicus_EKD  as list of tag+ ;  xmlrep = el celebrationDays .
def node tag  as complex :
             plain prefix : (def name;  xmlrep att) ;
             once :         (def datum; xmlrep att),
                                wochenspruch ;
             optional :   psalm, graduale, lesung, epistel, halleluja,
                                evangelium, predigt .
def node wochenspruch as complex :
             plain prefix     : quelle ;
             mixed with chars : #from richtext:rtf copy #all .
def node psalm as complex :
             plain prefix     : quelle ;
             grammar          : werk * .
use type of psalm for graduale, lesung, epistel, halleluja,
                      evangelium, predigt .
def node werk   as complex : plain prefix  : #content ;
                             grammar       : vertonung + .
def node vertonung as complex :
             plain prefix     : (def composer ; xmlrep = att) ;
             once             : (def vox ; xmlrep = att voices) ;
             optional         : (def bemerkung as mixed with chars :
                                   #from richtext:rtf copy #all ,
                                   spitzenton, (def schwer as empty) ).
def node spitzenton as parser  [voice #ident] "=" [pitch #ident] .
eof
```

Ende of Source

B Some Special Symbols and Constructs in Z

$R \ (\!\mid A \mid\!)$	= the set containing the results of applying relation R to all elements of the set A.
$A \lhd R$	= „domain restriction“ = the relation identical to R, but containing only those pairs the left element of which is in the set A.
$R \rhd B$	= „range restriction“ = the relation identical to R, but containing only those pairs the right element of which is in the set B.
$\mu\ A$	= the only element contained in the set A. If A does not contain exactly one element, this expression is *undefined*.
$\#\ A$	= the cardinality of any set A.
$a \mapsto b$	=just syntactic sugar for (a, b).
$\mathsf{seq}\ A$	= the set of all sequences of elements of A. Each $S \in \mathsf{seq} A$ is a finite mapping $\mathcal{N}^+ \rightarrow A$, where there are no „holes“ in the domain, i.e. S is defined for all n with $1 < n < \#\ S$.
$S_1 \frown S_2$	= the concatenation of two sequences.
$S.k$	= the kth element of S. Only defined if $1 < k < \#S$.
$\mathsf{squash}\ S$	= if S is of type $\mathcal{N}^+ \ \rightarrow \ A$, then $\mathsf{squash}\ A$ is the sequence made by „compactifying“ S by „shifting left“ all elements right to a „hole“.