d2d — A ROBUST FRONT-END FOR PROTOTYPING, AUTHORING AND MAINTAINING XML ENCODED DOCUMENTS BY DOMAIN EXPERTS

Markus Lepper[1] and Baltasar Trancón y Widemann[1,2]
[1] *<semantics/> GmbH, Berlin, DE*
[2] *Universität Bayreuth, DE*
post@markuslepper.eu, Baltasar.Trancon@uni-bayreuth.de

Keywords: Knowledge Acquisition, Semi-Formal Documents, Domain-Specific Languages

Abstract: In many cases, domain experts are used to write down their knowledge in contiguous texts. A standard way to facilitate the automated processing of such texts is to add mark-up, for which the family of XML-based standards is current best practice. But the default textual appearance of XML mark-up is not suited to be typed, read and edited by humans. The authors' d2d notation provides an alternative which uses only one single escape character. Its documents can be fluently typed, understood and edited by humans almost in the same way as non-tagged text. In the last years, the d2d language underwent a development guided by practical experiences. In practice, *robustness* turned out to be highly desirable: This lead to revised semantics and a new algorithm which realizes a *total* translation function, This article gives the complete operational semantics of this algorithm after a short sketch of its context.

1 The d2d Approach

1.1 Design Principles

Modeling knowledge by XML-encoded documents is a rapidly expanding practice. XML seems to be esp. useful for semi-structured documents, but also as a representation of formal structures like object-oriented databases, technical configuration data, etc. New document types, suited for special needs, can easily be defined, importing and combining existing standards. However, the standard appearance of XML with its large number of reserved characters and complicated lexical rules is not suited to be directly typed, read and edited by domain experts, esp. when they are used to or dependent upon legacy production environments.

The authors' d2d XML notation provides an alternative front-end representation which uses only one single escape character. Documents can fluently be typed, read and edited by humans almost in the same way as non-tagged text. At the same time they represent an exactly defined XML document model, processable by computers. The first versions of d2d had been developed in 2001, (Lepper et al., 2001) and it has been successfully employed in widely varying applications. The complete documentation of the current version is in (Trancón y Widemann and Lepper, 2010). In d2d-encoded text there is explicit tagging, where tags are marked with a single user-definable character, and closing tags are inferred wherever possible.

One of the practical projects was the technical documentation for a mid-scale software project. Table 1 shows the number of single unicode characters required in `.d2d` and standard `.xml` encoding. While the 29% of key presses could of course also be eliminated by a syntax-controlled editor, the redundancy remains distracting when *reading*.

A document type definition is required to rule the parsing process as well as the structure of the generated XML output. The current tool implementation understands the W3C XML DTD (Bray et al., 2006), and some dedicated data format languages. Additionally, d2d has its own definition format "`.ddf`"with full

Table 1: Characters in xml and d2d encoded texts.

lines	words	characters	
12477	61086	456711	total "`*.d2d`"
16273	61183	589377	total "`*.xml`"
1.30	1.00	1,29	factor

control over character-level parsing. The following text works with an abstract representation of the document type definition, independent of its origin. The first lines in the fragments from Figure 6 can give an impression of the d2d input and the resulting XML document.

1.2 Incremental Specifications, Incomplete Documents and Robustness

In practice, two features turned out to be of high value and have been considered in the latest revision of d2d:

Firstly, the syntax of d2d has been carefully revised for a more convenient support of *incremental refinement* of document type definitions. Introducing new child elements, changing an element's content from empty to optional children, from optional to required or vice versa; all these operations require a certain robustness and redundancy, esp. in *tokenization*.

Secondly, the treatment of *incomplete* documents turned out to be highly desirable: The new semantics and the new algorithm realize a *total* translation function, which recovers from ilelgal input as early as possible. This allows convenient diagnostics in case of *erroneously* incomplete documents, as well as a well-defined way of creating incomplete documents *intentionally* as preliminary versions.

To his end, the user may simply leave out required child elements of a content model, or mark elements as semantically incomplete by a special "brute-force" closing tag, even if they are syntactically complete. Both cases result in special meta-tags which are inserted in the generated XML model and should affect further processing. In case of error, input may be not only be missing, but also the opposite case, superfluous tags or character data which cannot be parsed according to the current content model. This kind of input is transferred *verbatim* to the generated output, wrapped in a meta-element.

In all these cases, the parsing algorithm tries to resume work as soon as possible, maximizing the diagnostic output in one single run. The input syntax and the complete operational semantics of this new, robust parsing algorithm are subject of this paper.

2 The Parsing Algorithm

There are three distinct layers of parsing in the d2d architecture: Tokenization, tag-based parsing, and direct character-based parsing for user-defined embedded syntax. This text focusses on the second layer.

Explicit tagging and the LL(1) criterion for grammar determinism can easily be taught to domain experts without training in formal language theory. Since the design of the formal and semi-formal document structures for a new project or a certain production context will happen in mixed teams of computer scientists and domain experts, this is the adequate level for communication.

2.1 Tokenization

As preprocessing to tag-based parsing, tokenization effectively is the identification of the different kinds of tags and of character data. Tokenization is described informally in Figure 2. Its result is an instance of the data type D from Figure 1.

Basically, all tags start with the command lead-in character. This can be re-defined by the user, and defaults to "#". It is followed by an identifier. Like in XML, a leading slash "/" indicates a closing tag, a trailing slash an empty element. For situations in which a closing tag cannot be inferred, d2d supports a short-hand notation, similar to the "\verb$\kappa$...κ" construct known from LaTeX, abbreviating the closing tag to the single character κ. Additionally there are forms with triple slashes, which are "brute-force" closing and empty tags. Using them indicates that the contents of the element are *intentionally* left incomplete.

2.2 Content Model Declarations

For the purpose of this article we simply put all tag identifiers into one global name space. Then every element is typed by a simple identifier as its tag. Each such identifier used as a tag is mapped to one content model. A content is an *extended regular expression* T from Figure 1.

The meaning is fairly standard: Any *ident* stands for an element with that identifier as its tag. `#empty` stands for empty content. `#chars` stands for character data. In contrast to W§C DTD and other formats, we have full compositionality of all operators. The three unary operators "?", "+" and "*" stand for optional, repeated and optional-repeated occurrence, as usual. The three binary operators ",", "|", "&" mean sequence, alternative and permutation. Note that, in contrast to Relax-NG (Clark and Murata, 2008), the operator "&" stands for permutation of its contiguous sub-terms, *not* for interleaving.

As mentioned above, the following description works on an instance of the abstract data type T_D of

Extended regular expressions for content model declaration:

$$T ::= \mathit{ident} \mid \texttt{\#empty} \mid \texttt{\#chars} \mid T\texttt{,}T \mid T\texttt{|}T \mid T\texttt{\&}T \mid T\texttt{?} \mid T\texttt{+} \mid T\texttt{*}$$

Document type definition:

$$T_D ::= \mathit{ident} \nrightarrow T$$

Input data after tokenization:

$$\mathsf{Token} ::= \texttt{chars}(\alpha) \mid \texttt{OPEN}_{\mathit{ident}} \mid \texttt{CLOSE}_{\mathit{ident}} \mid \texttt{CLOSE}^{F}_{\mathit{ident}} \mid \texttt{EMPTY}_{\mathit{ident}} \mid \texttt{EMPTY}^{F}_{\mathit{ident}}$$

$$D ::= \mathsf{Token}^{*} \frown \texttt{\#eof}$$

Tree of nodes, generated as output:

$$\begin{aligned} N \quad ::= \quad & \texttt{chars}(\alpha) \mid \texttt{node}(\mathit{ident}, N^{*}) \mid \texttt{perm}(T \times (T \nrightarrow N^{*})\ ) \\ & \mid \texttt{missing}(T) \mid \texttt{skipped}(\mathsf{Token}) \end{aligned}$$

Figure 1: Basic Data Types

Assume

- the tag lead-in character remains set to “#”
- ␣ stands for any whitespace character (blank, tab, newline, etc)
- π for an opening parenthesis from the list
  “(”, “<”, “[”, “!”, etc.,
  and π' for the corresponding closing parenthesis from the list
  “)”, “>”, “]”, “!”, etc.,
- and κ for any input character which neither # nor π'

Then the following transformations describe the tokenization process in a semi-formal way, if successively applied to the head of the character input stream, with *decreasing priority*:

#(# | ␣)*	$\rightsquigarrow$	#
#TAG␣	$\rightsquigarrow$	$\texttt{OPEN}_{TAG}$
#TAGπ	$\rightsquigarrow$	$\texttt{OPEN}_{TAG}$
		Additionally, $\pi' \rightsquigarrow \texttt{CLOSE}_{TAG}$ is pushed to the parentheses context.
π‘	$\rightsquigarrow$	$\texttt{CLOSE}_{TAG}$
		when $\pi' \rightsquigarrow \texttt{CLOSE}_{TAG}$ is currently in the parentheses context.
		Additionally, this assignment is popped off the context stack.
π‘	$\rightsquigarrow$	$\texttt{chars}(\pi')$
		when no assignment $\pi' \rightsquigarrow$ _ is in the parentheses context
#TAG	$\rightsquigarrow$	$\texttt{OPEN}_{TAG}$
#/␣	$\rightsquigarrow$	$\texttt{CLOSE}_{i}$
		where $\texttt{OPEN}_{i}$ is the most recently recognized open tag.
#/TAG	$\rightsquigarrow$	$\texttt{CLOSE}_{TAG}$
#///TAG	$\rightsquigarrow$	$\texttt{CLOSE}^{F}_{TAG}$
#TAG///	$\rightsquigarrow$	$\texttt{EMPTY}^{F}_{TAG}$
#TAG/	$\rightsquigarrow$	$\texttt{EMPTY}_{TAG}$
κ^{*}	$\rightsquigarrow$	$\texttt{chars}(\kappa^{*})$

Figure 2: Tokenization and Tag Recognition

$$\mathsf{epsilon} : T \rightarrow \{\mathsf{false}, \mathsf{true}\}$$
$$ident^C = ident \cup \{\texttt{\#chars}\}$$
$$\mathsf{first} : T \rightarrow ident^C$$

$$\mathsf{epsilon}(i : ident) = \mathsf{false}$$
$$\mathsf{epsilon}(\texttt{\#chars}) = \mathsf{epsilon}(\texttt{\#empty})$$
$$= \mathsf{epsilon}(T?) = \mathsf{epsilon}(T*) = \mathsf{true}$$
$$\mathsf{epsilon}(T+) = \mathsf{epsilon}(T)$$
$$\mathsf{epsilon}(T_1, T_2) = \mathsf{epsilon}(T_1 \,\&\, T_2)$$
$$= \mathsf{epsilon}(T_1) \wedge \mathsf{epsilon}(T_2)$$
$$\mathsf{epsilon}(T_1 \mid T_2) = \mathsf{epsilon}(T_1) \vee \mathsf{epsilon}(T_2)$$

$$\mathsf{first}(x : ident^C) = x$$
$$\mathsf{first}(\texttt{\#empty}) = \{\}$$
$$\mathsf{first}(T?) = \mathsf{first}(T+) = \mathsf{first}(T*) = \mathsf{first}(T)$$
$$\mathsf{first}(T_1, T_2) = \begin{cases} \mathsf{first}(T_1) \cup \mathsf{first}(T_2) & \text{if } \mathsf{epsilon}(T_1) \\ \mathsf{first}(T_1) & \text{otherwise} \end{cases}$$
$$\mathsf{first}(T_1 \mid T_2) = \mathsf{first}(T_1 \,\&\, T_2) = \mathsf{first}(T_1) \cup \mathsf{first}(T_2)$$

Figure 3: Auxiliary Functions

document type definitions. In the concrete implementation, this can result directly from a module of d2d's own type definition format `.ddf`. But when using a W3C DTD, it is the result of a transformation: For instance,

```
<!ELEMENT e  (#PCDATA | c)*>
<!ATTLIST e  a1 NMTOKEN #REQUIRED
             a2 NMTOKEN #IMPLIED   >
```

will be translated to

```
e = (a1 & a2?), (#chars | c)*
```

2.3 Parsing Process

After tokenization, the input to the parsing process is of type D from Figure 1. All character data is treated as if tagged with a dedicated, reserved and invisible tag `#chars`. $ident^C$ is the set containing this tag and all explicit, visible tags.

The output of a parsing process is a *finite tree* according to the type definition of N from Figure 1. A term of type $\texttt{chars}(\alpha)$ is a leaf node carrying a contiguous sequence of character data, a term $\texttt{node}(ident, N^*)$ represents an element of the generated XML model with its name and its child nodes, and the terms of type $\texttt{perm}(t, \{\ldots\})$ are required because the child nodes corresponding to a permutation expression t shall later be shipped out in the normalized, sequential order of declaration.

A term of type `missing` is attributed with an expression from T. The node must be replaced with some input corresponding to the annotation in order complete the document. A node of type `skipped` contains tags which could not be accepted, together with the immediately following character data. If no errors have been encountered, the result does not contain nodes of these both kinds.

Content model declarations in d2d follow the LL(1) discipline, in a more strict sense than standard XML DTDs. Therefore the parsing process is easily directed by "first sets", which are well-known in parser construction (Aho et al., 1986) and calculated as in Figure 3.

The parsing process is specified by the *total* function translate from Figure 5. It operates on the input D and a stack of frames F. Every stack frame from F represents a future choice point, and holds both the regular expression it is parsing and the intermediate, accumulated parsing result. For readability we assume that the document type definition $dt : T_D$ is globally accessible. The top level conversion function text2tree is invoked with the tokenized input stream and the tag of the root element.

Due to the LL(1) discipline, a step of the parsing algorithm is determined completely by the head of the tokenized input stream and the state of the stack:

descend is called when an *open* tag is to be consumed, and this tag is contained in the set first of the currently parsed expression. The stack will grow by descending into this expression.

ascend_o is called for an open tag *not* contained in first of the current expression. The stack is unwound up to the innermost frame where the tag can be consumed instead.

ascend_c is called for an explicit close tag. The stack is unwound in a similar manner. In the latter two functions, `missing` tree nodes are generated for tags which *should* be present for a valid document.

skip If, in all these cases, no continuation can be found, then the current tag (together with immediately following character data) is rejected and the corresponding error elements inserted into the generated output.

Since one of these transformations will be possible for any input situation, the top-level functions text2tree and translate are always *total* functions. It depends on the concrete tool implementation what to do with the embedded, error-indicating meta-elements. Esp. the presence of "brute-force" closing tags should affect diagnostics, forcing tools into "incomplete" mode, even though they act as ordinary closing tags from the parser's perspective.

$F : T \times N^*$

$\mathsf{ascend\_o}, \mathsf{ascend\_c} : ident^C \times F^* \times N^* \to F^*$

// calculate a new stack, truncated as far as required, for consuming an open/close tag

// $\nu : N^*$ are the result nodes accumulated in previous translation steps.

// $\tau : N^*$ are the result nodes collected and created during ascend.

// $\phi : F^*$ is the upper part of the stack, i.e. all frames created earlier when descending.

$$\mathsf{ascend\_o}\ (i, (j : ident, \nu) \frown \phi, \tau) = \mathsf{ascend\_o}\ (i, \phi, \langle \mathtt{node}(j, \nu \frown \tau) \rangle)$$

$$\mathsf{ascend\_o}\ (i, (r = t^{*/+}, \nu) \frown \phi, \tau) = \begin{cases} (r, \nu \frown \tau) \frown \phi & \text{if } i \in \mathsf{first}(t) \\ \mathsf{ascend\_o}\ (i, \phi, \nu \frown \tau) & \text{otherwise} \end{cases}$$

$$\mathsf{ascend\_o}\ (i,\ ((t_1, t_2, \ldots, t_n), \nu) \frown \phi, \tau)$$
$$= \begin{cases} ((t_2, \ldots, t_n), \nu \frown \tau) \frown \phi & \text{if } i \in \mathsf{first}(t_1) \\ \mathsf{ascend\_o}\ (i,\ ((t_2, \ldots, t_n), \nu) \frown \phi, \tau') & \text{if } i \notin \mathsf{first}(t_1) \wedge n > 1 \\ \mathsf{ascend\_o}\ (i, \phi, \nu \frown \tau') & \text{if } i \notin \mathsf{first}(t_1) \wedge n = 1 \end{cases}$$
$$\text{where } \tau' = \begin{cases} \tau & \text{if } \mathsf{epsilon}(t_1) \\ \tau \frown \mathtt{missing}(t_1) & \text{otherwise} \end{cases}$$

$$\mathsf{ascend\_o}\ (i,\ (t_x, \mathtt{perm}(t = (t_1 \,\&\, \ldots \,\&\, t_n), M = \{t_{k_1} \mapsto \nu_{k_1}, \ldots, t_{k_m} \mapsto \nu_{k_m}\})) \frown \phi, \tau)$$
$$= \begin{cases} (t_y, \mathtt{perm}(t, M')) \frown \phi & \text{if } \exists y \bullet i \in \mathsf{first}(t_y) \wedge t_y \in \{t_1, \ldots, t_n\} \wedge t_y \notin \{t_{k_1}, \ldots, t_{k_m}\} \\ \mathsf{ascend\_o}\ (i, \phi, \mathtt{perm}(t, M'')) & \text{otherwise} \end{cases}$$
$$\text{where } M' = M \cup \{t_x \mapsto \tau\} \quad M'' = M \cup \{t_x \mapsto \tau \frown \mu\}$$
$$\mu = \langle \mathtt{missing}(t_z) \mid t_z \in \{t_1, \ldots, t_n\} \wedge t_z \notin \{t_{k_1}, \ldots, t_{k_m}\} \wedge \neg \mathsf{epsilon}(t_z) \rangle$$

$$\mathsf{ascend\_c}\ (i, (j : ident, \nu) \frown \phi, \tau) = \begin{cases} \mathsf{ascend\_c}(i, \phi, \langle \mathtt{node}(j, \nu \frown \tau) \rangle) & \text{if } i \neq j \\ (t_x, \mu \frown \mathtt{node}(j, \nu \frown \tau)) \frown \phi` & \text{otherwise} \end{cases}$$
$$\text{where } \phi = (t_x, \mu) \frown \phi'$$

$$\mathsf{ascend\_c}\ (i, (r = t^{*/+}, \nu) \frown \phi, \tau) = \mathsf{ascend\_c}\ (i, \phi, \nu \frown \tau)$$

$$\mathsf{ascend\_c}\ (i, ((t_1, t_2, \ldots, t_n), \nu) \frown \phi, \tau)$$
$$= \mathsf{ascend\_c}\ (i, \phi, \nu \frown \tau \frown \langle \mathtt{missing}(t_z) \mid t_z \in \{t_1, \ldots, t_n\} \wedge \neg \mathsf{epsilon}(t_z) \rangle)$$

$$\mathsf{ascend\_c}\ (i, (t_x, \mathtt{perm}(t = (t_1 \,\&\, \ldots \,\&\, t_n), M = \{t_{k_1} \mapsto \nu_{k_1}, \ldots, t_{k_m} \mapsto \nu_{k_m}\})) \frown \phi, \tau)$$
$$= \mathsf{ascend\_c}\ (i, \phi, \langle \mathtt{perm}(t, M'') \rangle)$$
$$\text{where } M'' = M \cup \{t_x \mapsto \tau \frown \mu\}$$
$$\text{where } \mu = \langle \mathtt{missing}(t_z) \mid t_z \in \{t_1, \ldots, t_n\} \wedge t_z \notin \{t_{k_1}, \ldots, t_{k_m}\} \wedge \neg \mathsf{epsilon}(t_z) \rangle$$

$$\mathsf{ascend\_o}\ (\_, \langle\rangle, \_) = \mathsf{ascend\_c}\ (\_, \langle\rangle, \_) = \langle\rangle$$

$\mathsf{descend} : T \times ident^C \to F^*$

// Precondition: $\mathsf{descend}(t, i, \_)$ is only called when $i \in \mathsf{first}(t)$

$$\mathsf{descend}((t_1, t_2, \ldots, t_n), i) = \begin{cases} \mathsf{descend}(t_1, i) \frown (\,(t_2, \ldots t_n), \langle\rangle) & \text{if } i \in \mathsf{first}(t_1) \\ \mathsf{descend}((t_2, \ldots t_n), i) & \text{otherwise} \end{cases}$$

$$\mathsf{descend}((t_1 \mid \ldots \mid t_n), i) = \mathsf{descend}(t_k, i)$$
$$\text{where } 1 \leq k \leq n \ \wedge\ i \in \mathsf{first}(t_k)$$

$$\mathsf{descend}(t = (t_1 \,\&\, \ldots \,\&\, t_n), i) = \mathsf{descend}(t_k, i) \frown (t_k, \mathtt{perm}(t, \{\}))$$
$$\text{where } 1 \leq k \leq n \ \wedge\ i \in \mathsf{first}(t_k)$$

$$\mathsf{descend}(t?, i) = \mathsf{descend}(t, i)$$

$$\mathsf{descend}(r = t^{*/+}) = \mathsf{descend}(t, i) \frown \langle (r, \langle\rangle) \rangle$$

$$\mathsf{descend}(i, i) = \langle (dt(i), \langle\rangle) \rangle \quad \text{if } i \neq \mathtt{\#chars}$$

$$\mathsf{descend}(\mathtt{\#chars}, \mathtt{\#chars}) = \langle\rangle$$

Figure 4: Extending and Truncating the Stack during Tag Parsing

$$\mathsf{skip}, \mathsf{skip}^C : D \times F^* \to D \times F^*$$

$$\mathsf{skip}^C(d \frown \delta, (\_, \tau) \frown \phi) == \begin{cases} \mathsf{skip}^C(\delta, (\_, \tau \frown \texttt{skipped}(d)) \frown \phi) & \text{if } d = \texttt{chars}(\_) \\ (d \frown \delta, (\_, \tau) \frown \phi) & \text{otherwise} \end{cases}$$

$$\mathsf{skip}(d \frown \delta, (\_, \tau) \frown \phi) == \mathsf{skip}^C(\delta, (\_, \tau \frown \texttt{skipped}(d)) \frown \phi)$$

// as an alternative call "id()" instead of $\mathsf{skip}^C()$, see section 3

$$\mathsf{translate} : D \times F^* \to D \times F^*$$

$$\mathsf{translate}(\texttt{CLOSE}_j \frown \delta, \phi) = \begin{cases} \mathsf{translate}(\delta, \phi') & \text{if } \phi' \neq \langle\rangle \\ \mathsf{translate}(\mathsf{skip}(\texttt{CLOSE}_j \frown \delta, \phi)) & \text{otherwise} \end{cases}$$

where $\phi' = \mathsf{ascend\_c}\,(j, \phi, \langle\rangle)$

$$\mathsf{translate}(\texttt{OPEN}_j \frown \delta, \phi) = \begin{cases} \mathsf{translate}(\delta, \mathsf{descend}(t, j) \frown \phi) & \text{if } j \in \mathsf{first}(t) \\ \mathsf{translate}(\delta, \mathsf{descend}(t, j) \frown \phi') & \text{if } j \notin \mathsf{first}(t) \wedge \phi' \neq \langle\rangle \\ \mathsf{translate}(\mathsf{skip}(\texttt{OPEN}_j \frown \delta, \phi)) & \text{otherwise} \end{cases}$$

where $\phi = (t, \_) \frown \_$

where $\phi' = \mathsf{ascend\_o}(j, \phi, \langle\rangle)$

$$\mathsf{translate}(\Delta = \texttt{chars}(\alpha) \frown \delta, \Phi = (t, \nu) \frown \phi)$$

$$= \begin{cases} \mathsf{translate}(\delta, (t, \nu \frown \texttt{chars}(\alpha)) \frown \phi) & \text{if } \texttt{\#chars} \in \mathsf{first}(t) \\ \mathsf{translate}(\Delta, \phi') & \text{if } \texttt{\#chars} \notin \mathsf{first}(t) \wedge \phi' \neq \langle\rangle \\ \mathsf{translate}(\mathsf{skip}^C(\Delta, \Phi)) & \text{otherwise} \end{cases}$$

where $\phi' = \mathsf{ascend\_o}(\texttt{\#chars}, \Phi, \langle\rangle)$

$$\mathsf{text2tree} : D \times \mathit{ident} \to N$$

$$\mathsf{text2tree}(d, i) = \texttt{node}(i, \nu)$$

where $\mathsf{translate}(d, \langle(dt(i), \langle\rangle)\rangle) = (\langle\texttt{\#eof}\rangle, \langle(\_, \nu)\rangle)$

Figure 5: Algorithm for Tag Parsing

2.4 Shipping out the Node Tree as XML Structure

The ship-out of the internal model "*N*" to standard XML textual representation requires further, but minor, transformations: All nodes represented as XML attributes are treated separately, and all nodes which have been matched by a content model of permutation type are written out in the sequential order of that definition. All other nodes are converted to XML elements or text nodes in the order they appear in the d2d input document.

2.5 Error Messages after XHTML Transformation

The result of translating an erroneous input is a tree containing error nodes, i.e. `missing` and/or `skipped` nodes. When shipping out, these are translated into XML elements with dedicated tags from a reserved name space.

The d2d library of pre-defined general-purpose document types comes with a collection of translation rules from XML into various back-ends (currently: XHTML 1.0 and partly LaTeX), realized as XSLT 1.0 templates. In this context also the error meta-elements are translated, i.e. "rendered". An eye-catching style has been chosen for them, similar to the "raspberry red" for parenthesis mismatch used in XEmacs. Figure 6 demonstrates all these formats for a fragment containing correct and erroneous input.

3 Future Work

The treatment of errors is, of course, heuristic. In our practical experience, In 80 percent of all cases a sensible continuation is found. But of course variants and extensions are possible:

1. In the current implementation, after a spurious tag has been skipped, all subsequent character data is also discarded. The alternative is indicated in the comment to the skip function in Figure 5.

Document Type Definition, in `.ddf` format:

```
module structure
  ...
  tags h1 = title, label?, (p | px | P | DOMAIN_SPECIFIC_VERTICAL_ELEMENTS)*, h2*
  tags h2 = title, label?, (p | px | P | DOMAIN_SPECIFIC_VERTICAL_ELEMENTS)*, h3*
  tags title = (#chars | DOMAIN_SPECIFIC_HORIZONTAL_ELEMENTS)*
  tags DOMAIN_SPECIFIC_VERTICAL_ELEMENTS,
       DOMAIN_SPECIFIC_HORIZONTAL_ELEMENTS  = #generic
  ...
end module
```

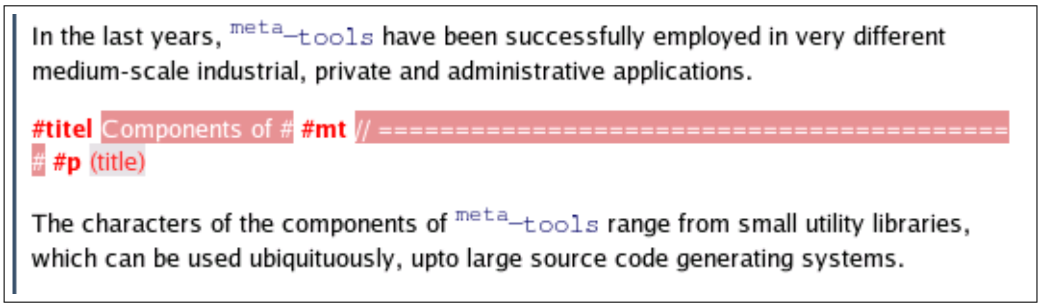
Erroneous Input:

```
#p In the last years, #mt  have been successfully employed in very different
medium-scale industrial, private and administrative applications.

// =========================================
#h1 #titel Components of #mt
// =========================================
#p
The characters of the components of #mt  range from small utility libraries,
which can be used ubiquituously, upto large source code generating systems.
#p
```

XML Parsing Result:

```
<p>In the last years, <metatools/> have been successfully employed in very different
medium-scale industrial, private and administrative applications.</p>

<h1>
  <d2d:parsingError kind="open" location="file.d2d:46:0->56:11" tag="titel"/>
    <d2d:skipped>Components of #</d2d:skipped>
  </d2d:parsingError>
  <d2d:parsingError d2d:kind=’open’ location=’..’  tag=’mt’>
    <d2d:skipped>// =========================================#</d2d:skipped>
  </d2d:parsingError>
  <d2d:parsingError d2d:kind=’open’ location=’..’  tag=’p’>
    <d2d:expected>(title)</d2d:expected>
  </d2d:parsingError>
      <p>The characters of the components of <mt /> range from small utility libraries,
  which can be used ubiquituously, upto large source code generating systems.
  </p>
```

XHTML Rendering:



In the last years, meta–tools have been successfully employed in very different medium-scale industrial, private and administrative applications.

#titel Components of # #mt // ===
#p (title)

The characters of the components of meta–tools range from small utility libraries, which can be used ubiquituously, upto large source code generating systems.

Figure 6: Error Messaging in the XHTML Back-End

2. The current version of the algorithm only looks "forward" and assumes that required elements have been omitted as a whole. What the current algorithm does *not* do is to look "into depth" and test whether simply an opening *tag* has been forgotten. But this analysis is not trivial, because the LL(1) discipline no longer applies, and more look-ahead or back-tracking is necessary.
3. The point when the "skip" transformation occurs is exactly the place where *fuzzy matching* of tag identifiers could be useful for detecting small typos.
4. The source text position of an error is included in the generated error message in a standard way. It also could be included in the XHTML rendering, for instance as a "tool-tip text".
5. In certain production contexts it could be useful to generate an extended version of the original *d2d source document* which is enriched with eye-catching comments for skipped input as well as for missing elements. The latter could include the synthesized regular expression which describes the missing contents, or even hyperlinks to the online documentation, as a diagnostic aid.

4 Related Work

W.r.t. the declaration of content models, there are strong similarities with relax-ng (Clark and Murata, 2008). But in detail there are significant differences: their "`&`"-operator means interleaving, ours permutation; the mechanism for parameterization is different, and, last but not least, relax-ng is restricted to verification of existing documents and does not deal with human-friendly notation at all.

In the field of document construction tools we have found hardly any similar approach to d2d. Some similarities can be found to M4 (m4, 2000). But these are restricted to the simplicity of the textual representation in general, and rather different to our syntax in detail. M4 has, e.g., also a single-letter qualifier for macro names: It uses the open parenthesis as suffix, while we use a user-defined character as prefix. It would be a very different, but also very interesting project, trying to use M4 directly for authoring XML document generation, but this has not been undertaken as far as we know.

The principles of the "`\verb`*k*...*k*" construct and of the "`\begin{verbatim}`" construct are taken directly from LaTeX (Lamport, 1986). Since grammar and semantics are totally different from our approach, this is currently the only similarity. We are thinking about implementing a kind of "macro expansion" mechanism, allowing the user to defined small ad-hoc convenience abbreviations. In this context, TeX could perhaps again serve as a model.

A similar degree of neighborhood can be seen to Lout, (Kingston, 1992), (Kingston, 2000), but again only in the syntax, employing a single active character, here the "`@`". Lout is intended as a replacement for LaTeX, i.e. it ultimately acts as a type-setting system for directly generating postscript documents. A semantic text structure *can* be incorporated by the macro definition mechanism, but is not intended to be exported for knowledge representation.

REFERENCES

Aho, A., Sethi, R., and Ullman, J. (1986). *Compilers: Principles, Techniques, and Tools.* Pearson Education.

Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F., and Cowan, J. (2006). *Extensible Markup Language (XML) 1.1 (Second Edition).* W3C, `http://www.w3.org/TR/2006/REC-xml11-20060816/`.

Clark and Murata (2008). *Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG.* ISO/IEC, `http://standards.iso.org/ittf/PubliclyAvailableStandards/c052348_ISO_IEC_19757-2_2008(E).zip`.

Kingston, J. H. (1992). The design and implementation of the lout document formatting language. *Software—Practice & Experience*, 23 (9).

Kingston, J. H. (2000). *The Lout Homepage.* `url://savannah.nongnu.org/projects/lout`.

Lamport, L. (1986). *LaTeX User's Guide and Document Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Lepper, M., Trancón y Widemann, B., and Wieland, J. (2001). Minimze mark-up ! – Natural writing should guide the design of textual modeling frontends. In *Conceptual Modeling — ER2001*, volume 2224 of *LNCS*. Springer.

m4 (2000). `m4` *Manual.* Free Software Foundation, `http://www.seindal.dk/rene/gnu/man/`.

Trancón y Widemann, B. and Lepper, M. (2010). *The BandM Meta-Tools User Documentation.* `http://bandm.eu/metatools/docs/usage/index.html`.