aTool — Creating Validated XML Documents on the Fly Using MS Word

Oliver Meyer
RWTH Aachen
Department of Computer Science 3
Ahornstr. 55
52074 Aachen, GERMANY
omeyer@i3.informatik.rwth-aachen.de

ABSTRACT

This paper describes aTool, an extension to Microsoft's Word to create XML documents. aTool has been developed in a joint project of the publisher Springer Verlag, Technical University of Munich (TUM), and Technical University of Aachen (RWTH). It has been developed to provide Springer Verlag with uniform XML documents from its authors and has become a generic XML creation tool that can be adapted to different document structures.

For an author, aTool derives XML structures from MS Word editing commands, while he creates his text. For a technical manager, aTool can be parameterized in flexible, yet simple ways to suit the needs of a specific XML application. For a programmer, the paper describes the main implementation details of aTool. The paper closes with a short comparison to other approaches and a summary of the benefits and shortcomings of aTool.

Categories and Subject Descriptors

I.7.4 [**Computing Methodologies**]: Electronic Publishing; I.7.2 [**Computing Methodologies**]: Document and text processing—*Format and notation, Markup languages*; H.4.1 [**Information Systems Application**]: Office Automation—*Word processing*; K.8.1 [**Personal Computing**]: Application Packages—*Text processing, MS Word*; D.2.12 [**Software Engineering**]: Interoperability—*Data Mapping*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Abstracting methods*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation*

General Terms

Languages, Design, Documentation

Keywords

XML, DOM, Microsoft Word, Microsoft Office, Microsoft Word Add-In, A-Posteriori Integration, Character Formatting

1. INTRODUCTION

In our project's scenario, information needs to be exchanged between authors and their publisher. As a project partner Springer Verlag determines the requirements. For their scientific journals, Springer Verlag receives mainly MS Word documents as submissions that are manually tranformed into SGML and then processed further. The transformation process is slow, expensive, and introduces errors because of misunderstandings between author and publisher. The publisher wishes to process documents created by the authors directly with very little manual interaction. Therefore, we needed to equip the authors with a tool that creates structured text documents directly.

Hierarchical structures are a well-established model for texts. SGML has long been used in professional text processing and publishing. Yet, its complexity made processors and parsers expensive and error-prone. XML was designed to alleviate these problems. It uses the same document model (a hierarchy of elements with possibly text as its leaves), but an easier, more restrictive syntax. These restrictions with very few special cases make processing of XML documents comparatively simple.

Several XML parsers are widely available. XML processors use the transformation language XSLT to transform an XML document into another XML or text document according to predefined rules. Their use inside of web servers and clients makes XML a very important language in today's computer applications.

The main advantage of XML is that it provides standardized means to define the syntax of documents. If a community needs to exchange data, they must decide on a uniform format for their exchange documents. That format is defined by its syntax and semantics. The syntax defines the language, and the semantics define the meaning of documents in that language.

For our scenario, syntax and semantics are defined by the publisher. The authors should directly create XML documents and check these XML documents for conformance to the predefined syntax. We expect authors to have a moderate MS Word knowledge and no knowledge of XML. For the authors, the XML editor must be free and easy to use. To use this knowledge we allow the authors to continue using MS Word to write their texts and extended that application. This extension of MS Word is called *aTool* (for *a*uthoring *tool*).

The next subsection recapitulates roughly the document model of XML and the expressive power of its syntax constraints. To understand the problems aTool solves, a basic understanding of that model is necessary. After that, the basic ideas behind our tool aTool are explained.

The following sections describe aTool from a user's perspective. It follows the order in which commands are activated and describes user interface elements. Section 3 describes how aTool can be customized to serve in a different scenario. Section 4 gives some implementation details. Section 5 describes other products that are used to create XML documents. The last section summarizes in which scenarios using aTool might be helpful.

1.1 XML

The left side of Fig. 1 shows a cutout of an XML document. It is a serialization of the tree structure shown on the right.

Each box in the tree represents an *element*. Each element carries a *type* which is written inside the box and optionally *attributes* which are shown in ellipses besides the elements. Boxes are *ordered* from top to bottom. Lines represent *contains* relations between elements or between elements and text. Therefore, the document element contains the title, author, and contents elements in that order. The title element contains the title of the document as text.

In the textual representation the order is given by the order of the text. Elements are delimited by a *start-tag* (<*type*>) and an *end-tag* (</*type*>). All elements and text between these tags are contained in that element. Attributes are expressed as key/value pairs in the start-tag.

```
<document language="english">
   <title>aTool</title>
   <author>O. Meyer</author>
   <contents>
      <section>
         <heading>
            Introduction</heading>
         <paragraph>
            ... </paragraph>
         ...
      </section>
      ...
   </contents>
</document>
```

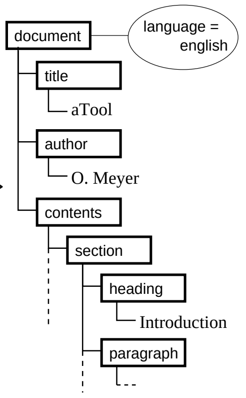


Figure 1: Basic XML document and structure

Every document written in XML consists of such a tree of nested, attributed elements and text. XML does not, per se, enforce specific element types or nesting structures. To be useful in data exchange, the syntax of the XML document must be further constrained.

The XML standard distinguishes, therefore, between *well-formed* documents, that are written in generic XML, and *valid* documents that obey further constraints. These constraints are defined in a document type definition (DTD). A DTD declares all element types and their attributes. It also defines allowed ways to nest elements and texts.

Fig. 2 shows an example DTD for the document in Fig. 1. The first line declares the document element type and defines its content model: Each element of that type must contain an element of type title, followed by one or more elements (+) of type author, and an element of type contents. The second and third line declare the language attribute and define its allowed values: "english" or "german". Each document element must explicitly carry this attribute as it is declared as #REQUIRED. Elements of type title must contain only text (#PCDATA) and no other elements, while paragraph elements may contain emphasize elements between text fragments.

```
<!ELEMENT document  (title, author+, contents)>
<!ATTLIST document
    language ( english | german ) #REQUIRED>

<!ELEMENT title     (#PCDATA)>
<!ELEMENT author    (#PCDATA)>
<!ELEMENT contents  (section+)>
<!ELEMENT section   (heading, paragraph+)>
<!ELEMENT heading   (#PCDATA | emphasize)*>
<!ELEMENT paragraph (#PCDATA | emphasize)*>
<!ELEMENT emphasize (#PCDATA)>
```

Figure 2: DTD for document in Fig. 1

For a document to be valid, it must be well-formed and reference a DTD. Each used element type must be declared in the DTD and strictly follow the constraints defined there. As the syntax of the document is defined in the DTD and provided as data, the validating XML parser itself can be made generic. It does not contain any code specific to a single syntax.

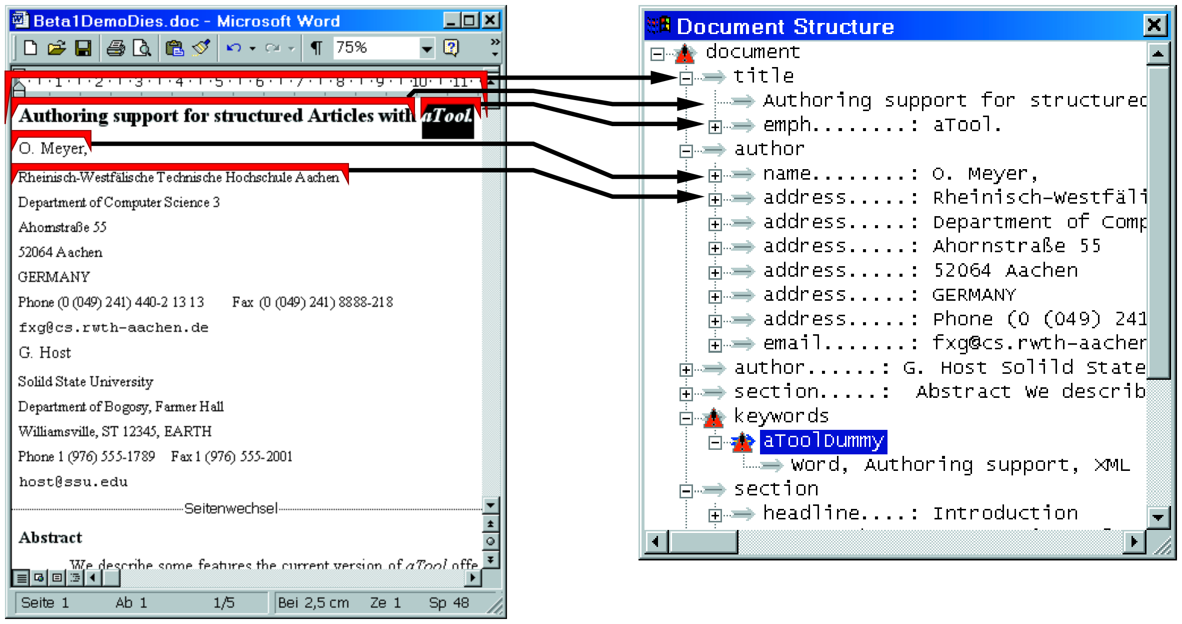
To allow data exchange, a common DTD is defined for all exchange partners. Every partner then writes and reads its exchange data in that format. With the use of XML, no parsers need to be specifically written for any of the partners, and defining the syntax has become much easier. What still needs to be defined in some other, informal way, are the semantics of the exchange documents.

1.2 Basic ideas of aTool

To cope with the different scientific journals published by Springer Verlag and to be usable for other publishers or in different contexts, we keep the flexibility of XML and designed our tool to be parameterized by a DTD. To make use of the MS Word text in creating the tree structure of the XML document, we integrate aTool with MS Word tightly. It is implemented as an AddIn loaded at MS Word startup-time.

MS Word offers different programmatic views on its documents. As none already provides a hierarchy, we use a stream of paragraphs as a model. Each paragraph consists of text sequences of equally formatted texts.[1] We call such a text sequence up to the end of a paragraph a *text span.*

[1]Tables consist of a range of paragraphs. Formulas and inline images can be seen as special characters and are thus text sequences of their own. We cannot handle floating images yet.



```
Authoring support for structured Articles with aTool.
O. Meyer,
Rheinisch-Westfälische Technische Hochschule Aachen
Department of Computer Science 3
Ahornstraße 55
52064 Aachen
GERMANY
Phone (0 (049) 241) 440-2 13 13     Fax (0 (049) 241) 8888-218
fxg@cs.rwth-aachen.de
G. Host
Solild State University
Department of Bogosy, Farmer Hall
Williamsville, ST 12345, EARTH
Phone 1 (976) 555-1789    Fax 1 (976) 555-2001
host@ssu.edu
-----------------Seitenwechsel-----------------
Abstract
    We describe some features the current version of aTool offe

Seite 1    Ab 1    1/5    Bei 2,5 cm    Ze 1    Sp 48
```

```
document
  title
    Authoring support for structured
    emph........: aTool.
  author
    name........: O. Meyer,
    address.....: Rheinisch-Westfäli
    address.....: Department of Comp
    address.....: Ahornstraße 55
    address.....: 52064 Aachen
    address.....: GERMANY
    address.....: Phone (0 (049) 241
    email.......: fxg@cs.rwth-aacher
  author......: G. Host Solild State
  section.....:  Abstract We describ
  keywords
    aToolDummy
      Word, Authoring support, XML
  section
    headline....: Introduction
```

Figure 3: Text ↔ Element-Association

A paragraph of text without a format change consists of a single text span, while a paragraph with an emphasized word might consist of two or three text spans. The structure aTool creates and maintains on the other hand is that of an XML tree as shown in Fig. 1.

The main task of aTool is to integrate these two data structures. The basic idea behind this integration is shown in Fig. 3. On the left side, an MS Word document is shown, while on the right side, aTool's structure view shows a representation of the associated XML tree. The brackets show some of the text sequences relevant to aTool; arrows indicate their association to elements and text in the tree.

The first element contains the text "Authoring support ..." and the element containing the text "aTool". This element has been created, because "aTool" is formatted in italic type. Assuming that words, with a different format, play different roles in a text, "aTool" has become an element of its own. Embedded objects like figures, formulas, or tables become elements of their own, too. Each formatting change thus always results in an element border. aTool initially creates and then incrementally maintains this text↔element-association.

All characters in a single element have the same format. This format information is also used to conclude the type of the corresponding element. For example, authors use boldfacing and increased font size to mark the title of a document. aTool uses this information to determine element types. The first element is therefore typed title, the contained element is typed emphasize. Which formats indicate which types, is initially defined in a parameterization step, but aTool also extends this mapping at runtime, if requested.

To parameterize aTool, the publisher creates a so-called *aToolKit*. It contains at least the DTD and an initial map of formats to element types. The aToolKit reflects the special requirements of the journal the author writes for or, more general, of the integration scenario aTool is used in. Its contents is explained in section 3 in detail.

2. A USER'S PERSPECTIVE

From a user's perspective, aTool presents itself as follows. Let us assume, part of the document has already been created in MS Word and is open (you can also start with an empty document). The author then starts aTool from within MS Word and chooses the appropriate aToolKit. For a journal article the publisher will provide the aToolKit, for a manual a DocBook adaption might be used.

The author is given a structure view that contains at first a list of paragraphs with some encapsulated elements. It reflects the simple structure of the MS word document. Each paragraph has become an element of its own and each text span also. We call this process of structure derivation *parsing*.

All elements without a specific type get the type aToolDummy as these elements are just placeholders for correctly typed elements. To create a *meaningful structure* for the already existing text, types must be set for each of the elements. aTool offers tool support for that batch-oriented step also. The command "Apply Roles" iterates over all elements of type aToolDummy and tries to find a type that matches the formatting of the corresponding text. If a single element type is associated with the formatting, its type is automatically applied to the element. For elements with ambiguous formatting, the context is considered as well. If only one matching element type is valid at the specified position, it is also applied automatically. If the element type remains ambiguous, or if requested explicitly by the user, the user is queried with the Mapping Dialog (see below).

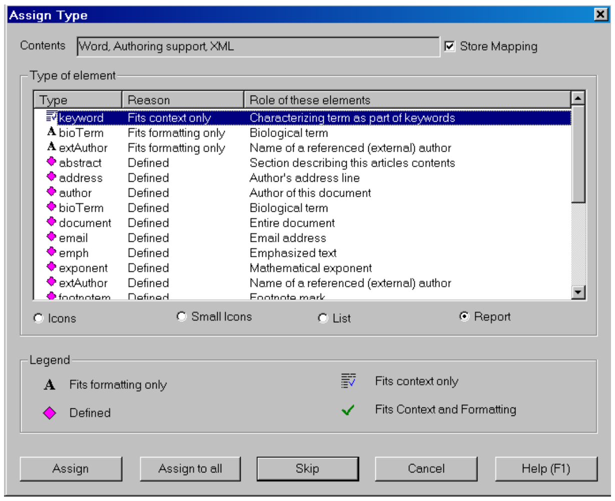
Along with assigning element types, aTool creates encapsulating elements. In Fig. 3 an author element is shown that contains multiple paragraphs with types name, address, and email. When an element gets the type address it is automatically encapsulated by an author element. An email element, created as a sibling of the author element, is automatically moved inside the author element, becoming the sibling of the address elements.

With these simple steps a typed, order tree of XML elements has been derived from the already existing MS Word document.

The author may use these elements to manipulate text parts as a unit. To change the order of the authors, he just needs to drag the author element in the structure view down. All contained elements follow and the MS Word document is changed as well.

The structure view also informs the author about *validation results*. The triangle[2] beside the keywords element in Fig. 3 marks it as invalid. The author may use an *element inspector* to get elaborated error messages. He is informed that a keywords element must contain other elements and may not contain text directly. A click on the error message brings him to the troubling part of the MS Word document. As some element is required here, he nests the text with the keywords into another element. The author is then queried for an element type.

The *Mapping Dialog*, shown in Fig. 4, offers all defined types for the user to choose from. It first lists all types that match the formatting of the text and that are valid at the position of the element in the structure. Then the elements that belong only in one of these groups are listed and at the end all defined elements are shown in alphabetical order. For the newly created element bioTerm and extAuthor match the formatting, while keyword is the only type allowed as child of keywords. The author chooses keyword as the correct type and also selects "Store Mapping". The formatting of the keyword is, therefore, mapped to the keyword type and will later on be applied automatically.



Contents: Word, Authoring support, XML [✓ Store Mapping]

Type	Reason	Role of these elements
keyword	Fits context only	Characterizing term as part of keywords
A bioTerm	Fits formatting only	Biological term
A extAuthor	Fits formatting only	Name of a referenced (external) author
◆ abstract	Defined	Section describing this articles contents
◆ address	Defined	Author's address line
◆ author	Defined	Author of this document
◆ bioTerm	Defined	Biological term
◆ document	Defined	Entire document
◆ email	Defined	Email address
◆ emph	Defined	Emphasized text
◆ exponent	Defined	Mathematical exponent
◆ extAuthor	Defined	Name of a referenced (external) author
◆ footnotem	Defined	Footnote mark

Icons Small Icons List [✓ Report]

Legend: A Fits formatting only; ◆ Defined; Fits context only; ✓ Fits Context and Formatting

Figure 4: Mapping Dialog

The author concludes that this aToolKit requires each keyword to be marked explicitly, so he splits the element between the keywords using the textual MS Word cursor to indicate the splitting position. As the newly created elements inherit their types from the split element, no types need to be applied manually.

At any time, the author may save the structured document together with the MS Word document. Saving also saves the possibly modified mapping table. The added structure information is stored in a separate file besides the MS Word document. It is currently up to the author to keep these files together. When loading an aTool document, the corresponding aToolKit and MS Word document are loaded automatically. While the MS Word document is a regular file, that could be edited with any unextended MS Word application, this would lead to inconsistencies when opening it as an aTool document again.

After the already-written part of the text is associated with a corresponding structure, the author continues to write in MS Word. Whenever the cursor leaves the paragraph he just wrote (e. g. by hitting return or by repositioning the cursor), it gets reparsed and is thus restructured as in the beginning. According to the formatting, elements are created, nested, and possibly typed. If no unique type can be determined aToolDummy is applied. No annoying pop-up windows interrupt the author in writing. Only the red error marks might show that correct types still need to be applied. At any time, the author can switch his focus from the text to the structure. Using a context menu he can manually (re-)type single elements, subtrees, or the complete document, always being supported by context checks and format-to-type mapping.

When editing existing text, aTool tries to keep the structure already created. If the author, for example, adds another keyword and then leaves the paragraph, the manually split elements remain split and keep their type. The new keyword itself is added to one of the existing elements that then needs to be split as before.

When the author has finished his document, he will export it as XML. This creates a text-only document in XML syntax. If the root element has been valid in aTool, that document is valid with respect to the DTD provided with the aToolKit. It depends on the concrete scenario whether an XML export should be allowed only if the complete document is valid. Currently, the XML document can be created at any time. It might also still contain aToolDummy elements.

3. PARAMETERIZATION

aTool offers flexible, yet simple ways to parameterize its behavior. In this section I will describe the contents of an aToolKit (technically a directory with several files) in more detail, to give the reader an idea how aTool can be adapted to help him in his own project.

3.1 DTD

The most important part of an aToolKit is the *DTD*. It defines the element types offered to the author, their required nesting structure, and possibly their attributes. As it strictly follows the XML syntax definition, any existing DTD can be used. aTool's validation will check, whether the documents created later are valid with respect to this DTD.

Although aTool can use any XML DTD for validation, it might be advised to change the DTD slightly to better support document creation. aTool creates a separate element for each paragraph, so elements that contain multiple paragraphs must contain other elements, not just text. MS Word displays only the textual content of elements and not their attributes, so data modeled as attributes might better be modeled as elements or vice versa. Using unique type names for deeply nested elements and more rigid structure

[2]It resembles a German traffic sign.

Text Format (cutout)

Attribute	*Type*	*Description*
Bold	boolean	Is boldfaced?
Italic	boolean	Is italicized?
Superscript	boolean	An exponent?
Subscript	boolean	An index?
Font	string	Name of the font
Size	float	Size of the font in points
Underline	enum	How the text is underlined
Style	string	Base character style name

Paragraph Format

Attribute	*Type*	*Description*
Alignment	enum	Alignment of paragraph (block, left, right, center)
FirstLineIndent	float	First line's indention in point
LeftIndent	float	Left indention in point
RightIndent	float	Right indention in point
LineSpacing	float	Line spacing in point
ReferenceFormat	element	Base text format
Style	string	Base paragraph style name

Table Format

Attribute	*Type*	*Description*
Columns	long	Number of columns
Rows	long	Number of rows

Inline Format

Attribute	*Type*	*Description*
Type	string	Kind of inline element (formula, figure, etc.)

Table 1: Format information considered in mapping

constraints better supports aTool's automatic creation of encapsulating elements.

3.2 Mapping

To derive element types from formatting, a *Mapping Table* must be defined. This table maps formats to element types. Each rule consists of a *Format Pattern* that describes the typographic format and an element type to map text with that format to. The mapping table is stored as an XML file. There are different types of patterns for different objects in a MS Word document.

Tab. 1 lists all pattern types and a few of their properties. For text, for example, the font and size of the text can be queried. For a paragraph its alignment and indentation can be used to determine an element type. Nearly all properties MS Word uses to format text, paragraphs, etc. are represented and can be used to distinguish text fragments. This information is also used to determine element borders when parsing. Whenever text differs in any of the defined properties, an element border is created.

Fig. 5 shows two mapping rules. The first uses a *text format pattern*. It describes the character format of a text sequence. In the example, the sequence must be in italic type to be mapped to the emphasize type. The font, its size, its boldness etc. do not matter for this mapping; their settings are simply left out of the format pattern.

The second rule uses a *paragraph format pattern*. As paragraphs have more properties than a text sequence (like indention, line spacing etc.) a different pattern is used. Paragraphs also contains text and that text has the same properties as any other text sequence. Therefore, a paragraph format pattern contains a text format pattern. In the example, a paragraph without a first-line indent and with most characters in "Times New Roman", 14pt, bold font is considered to be the title of the document.

The rules are merely suggestions. Whenever the format of a text matches the given pattern, the mapped element type is considered as type for the associated element. Therefore, the same format can be mapped to different element types and different formats can be mapped to the same element.

The mapping table may be designed differently for specific

```
<MappingRule>
  <FormatPattern_Text Italic="true" />
  <MapToElemType ElementName="emphasize"/>
</MappingRule>

<MappingRule>
  <FormatPattern_Paragraph FirstLineIndent="0">
    <FormatPattern_Text Font="Times New Roman"
                        Size="14.0" Bold="true"/>
 </FormatPattern_Paragraph>
 <MapToElemType ElementName="title"/>
</MappingRule>
```

Figure 5: Two sample mapping rules

groups of authors. In this project, we mainly target novice MS Word users that do not make use of styles but use direct instance-based formatting. In the mapping table we therefore use only visible properties of text and paragraphs. If the author does use character or paragraph styles, only the resulting visible format change is considered when determining element types. An experienced MS Word user might use different styles with equal formatting to distinguish different types of text in MS word already. Using the style property of text and paragraph patterns that difference can be used to reduce manual interaction when applying element types.

3.3 Annotations

As mentioned in section 2, aTool automatically creates higher-level elements like author or section. Depending on the DTD, not all elements have an unique parent and are suited for such an automation. A name element might be used in very different contexts. To add one of these context elements around any name element, would lead to errors the author would have to fix manually. Therefore, an additional *annotation document* exists in an aToolKit. It annotates the element definitions from the DTD in various ways.

The *Auto-Insert* property causes an element to be created automatically, when one of its children is created in the source structure. Of all possible parent element types the one with this property is chosen. If the choice is ambiguous, no element is inserted. In this way the aToolKit creator separates the different contexts and decides for one most probable.

The *Auto-Generate* property of an element type causes aTool to generate a subtree of elements when an element with that property is inserted. In the annotation document the value of that property is the complete tree to be inserted. This is very helpful for element types that model data objects rather than text. For example, the administrative data for a manual fragment, containing version information, formal conditions for parameterized manuals, and fine-grained references to specific functions in the source code, falls into this category. Such elements often contain lots of structure but little text. For the user, it is impossible to remember the various subelements and their required order. Auto-Generate creates empty child elements in the correct order and nesting for the author to fill in.

```
<!ELEMENT FragmentHeader (Version, Cond, Refs?)>
<!ELEMENT Version (#PCDATA)>
<!ELEMENT Cond (#PCDATA | OR | AND | NOT | TERM)*>
<!ELEMENT TERM (#PCDATA | OR | AND | NOT | TERM)*>
<!ELEMENT Refs    (Source+)>
<!ELEMENT Source  (Class, Method?)>
<!ELEMENT Class   (#PCDATA)>
...

<FragmentHeader>
   <Version></Version>
   <Cond></Cond>
   <Refs>
      <Source>
         <Class></Class> <Method></Method>
      </Source>
   </Refs>
</FragmentHeader>
```

Figure 6: Example for automatic generation

Fig. 6 shows this example. In the upper half a cutout of the DTD for a manual fragment is shown. Each fragment must be preceeded by a header containing the mentioned information. If the annotation document sets the AutoGenerate property for FragmentHeader, the author is presented with the XML structure shown in the lower half of the figure. That FragmentHeader element is a template of a correct FragmentHeader element to fill out. It contains not only the obligate parts of a FragmentHeader element, which are determined by the DTD, but also usual parts.

A FragmentHeader element with the above structure is valid according to the DTD, because it is valid according to the standard. Yet, it contains no information at all. Therefore, the structural constrains in the DTD can be further restricted in the annotation document. The *Min-Length* and *Max-Length* properties require the textual content length of the element to lie in specific borders. For our FragmentHeader example we would require a Min-Length of 1 for each element, making empty elements invalid. The length of the elements can be given in characters or words. The Max-Length property is used for example for abstract elements, that may contain at most 200 words.

To provide the novice user with additional information when choosing the appropriate type for an element, the annotation document may contain a description of each element type. That description consists of a more elaborate name and a text describing the usage of such elements. It is presented to the user in the mapping dialog.

3.4 Consistency Rules

When creating an aToolKit, consistency rules must be obeyed. For example: (1) The DTD should not contain element definitions that are not used in the complete expansion of the root element, as all defined elements are presented to the user in the mapping dialog. (2) For Auto-Insert to be helpful, at most one parent element type for each child type may have that property set or no unique selection for the parent can be made and elements of that type will never be inserted automatically. (3) The template subtree of the Auto-Generate property should follow the model defined for elements of that type or invalid documents are created automatically. (4) The element types used in the mapping table must be defined in the DTD. Currently only (4) is checked by aTool. We expect the creator of an aToolKit to be an experienced XML and MS Word user.

3.5 Applications

In manual creation, multiple authors write different parts of a manual or documentation and all these parts must later be put together in a post-processing step. To automate that process and to derive documents in other formats, the various parts should follow a predefined structure.

The scenario is different from the publishing scenario originally addressed. The writers are no longer novice authors that want to use their own layout. We can expect technical writers to have at least a rudimentary knowledge of XML and its implied hierarchical text structure. They will know how to use MS Word styles and can cope with a more complex DTD with a larger number of different element types.

When using aTool, the aToolKit for such a scenario is different. The various element types might be related to MS Word styles and the mapping table can be reduced to map the usage of these styles to the various elements. Specific Auto-Insert and Auto-Generate settings might create deeper nested elements. Section 5.2 describes the benefits of aTool compared to a pure MS Word approach.

4. BEHIND THE SCENES

aTool is implemented in C++ as a COM-AddIn to MS Word. Its code is stored in a dynamic link library (DLL) and loaded at MS Word startup-time. Access to MS Word is through MS Word's COM interface that allows complete programmatic access to the data structures accessible by an author and also allows to activate any command a user could activate.

Fig. 7 shows aTool's overall architecture. The `AToolDocumentPkg` package contains the main data structures used in aTool. It maintains and manipulates the element tree and its association with the MS Word document. Within that package the `AToolDocumentIntegrator` propagates changes in the structure into the MS Word document and vice versa.

At startup, aTool extends the menus of MS Word and adds a command bar with callbacks into the DLL code. All commands from these menus are processed by a single module, `ToolControl`, and dispatched to various tools that call

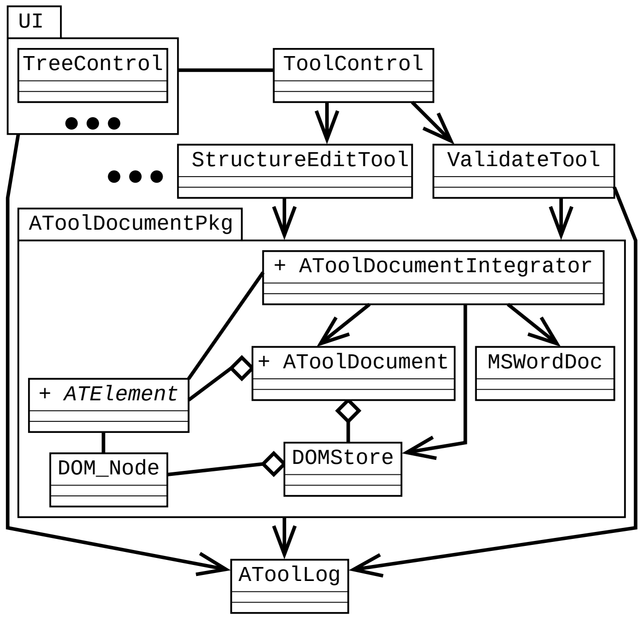


Figure 7: Cutout of the architecture of aTool

functions in the base layers of the architecture. Changes in the base data structures are logged and that log is then processed by the validation machinery and the UI components.

`ATElement` provides the interface for all XML element access within aTool. Objects of this class represent elements and text spans. We use Xerces-C [1] and its DOM implementation [2] to store the element tree. Its `DOM_Nodes` are used as the realization of the `ATElement` class.[3] They store the pure XML data, like element types, attributes, and the structure information. Each `DOM_Node` also stores a pointer to an `ATElement`.

`ATElements` extend the pure XML data by storing some internal administrative data, a reference into the MS Word document and the format information of the text spans the element is associated with. The latter is used for parsing and element typing, which is described in [3] in detail.

4.1 Usage of bookmarks

To reference a range of text, a bookmark is inserted in the MS Word document whenever a new `ATElement` is created. The name of that bookmark is stored in the `ATElement`.

The advantage of using bookmarks is, that MS Word often automatically updates them correctly when the MS Word document changes. If text is changed before the start of a bookmark, the bookmark changes to still contain the same text. If text is removed or inserted within a bookmark, the bookmark shrinks or extends to accommodate the change. If text that completely contains a bookmark is cut and pasted, that bookmark is cut and pasted as well. Therefore, we often do not have to change a lot of internal references, although the MS Word document changed.

A drawback of using bookmarks is, that they are visible to the user. He could manipulate them manually and get aTool into an inconsistent state. Also, as we do not have full control over bookmark maintenance we often had to examine through test runs how MS Word treats them in special cases, like text insertion right at the start or end of a bookmark, or at the start or end of the MS Word document. Often bookmarks need to be readjusted to correctly reflect the hierarchical structure of their corresponding `ATElements` after modifications.

[3]Within aTool, we do not follow the separation of `DOM_Element` and `DOM_Text`, but handle both uniformly as `ATElements`.

4.2 Dynamics

To give some insight into the dynamics of aTool, we describe moving an `ATElement` around in more detail. If the user drags an element in the structure view to a new location, the `TreeControl` in the view sends a "move subtree" command together with the necessary parameters: the element to move, a reference element, and an indicator whether to move the element before, after, or into the referenced element. `ToolControl` calls the appropriate method in the `StructureEditTool` that checks and transforms the parameters, and calls e. g. `InsertBefore` on the referenced `ATElement`.

`InsertBefore` calls the `AToolDocumentIntegrator` to cut out the text of the moving element and put it into the pasteboard. The `AToolDocumentIntegrator` knows exactly how MS Word handles bookmarks and takes care that only bookmarks of the elements in the subtree are removed from the MS Word document. Empty bookmarks at the start or end of the cut out element, caused by empty children, are removed explicitly and stored separately. After cutting out the text, `ATElement` changes the structure by calling the corresponding DOM method from Xerces on its `DOM_Node`. That change is written to the log. It then gives control to `AToolDocumentIntegrator`, which pastes the text from the pasteboard at the new location and queries the structure, manipulated by the `ATElement`, to fix the position of some of the affected bookmarks.

Control is then returned to the `StructureEditTool` that updates the structure selection. After that, `ToolControl` activates validation and then an UI update. The `ValidationTool` queries the log, to incrementally revalidate the changed elements only. The UI elements (element inspector and structure view) again process the log to update their representation data. Only now, the tree control shows the new structure and validation information.

4.3 Control Integration

All commands reaching `ToolControl` are processed this way. They are sent from the command bar or through the UI elements of aTool itself. While it is quite easy to react on these commands, it is very tricky to get informed about and to react to changes purely on the MS Word side. The simpler, coarse-grained change events include "new document created", "document opened", "document changed".

For these simpler events MS Word offers the interface `ApplicationEvents2`. It allows to register a listener for these events. aTool uses them to switch its context when the current document changes and to remove its data structures from memory when a document is closed. `ApplicationEvents2` also includes a `SelectionChanged` callback that is used to move the structure selection in accordance with the text selection.

To immediately create a corresponding structure through (re-)parsing and apply types to elements with an unique mapping, we need to get informed about fine-grained change events. As a compromise between technical possibilities and expected behavior, aTool (re-)parses a paragraph whenever the cursor leaves it.

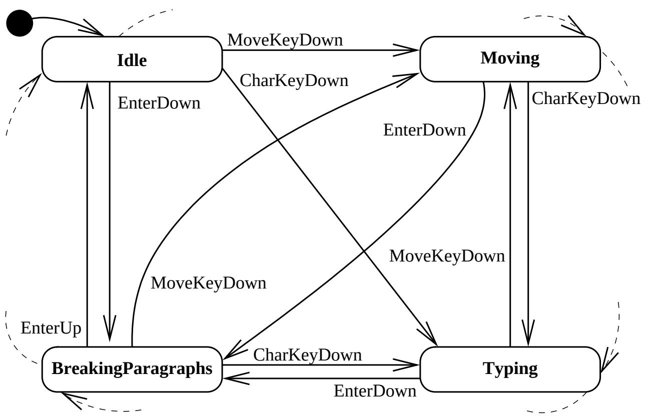


Figure 8: Cutout of the input automaton

For these updates, we listen to MS Windows events (mouse clicked, key pressed, etc.) before they reach MS Word. A finite automaton builds a model of the internal states within MS Word. Fig. 8 shows a cutout of that automaton. The **Idle**, **Typing**, **Moving**, and **BreakingParagraphs** states are shown. Initially, the automaton is in the **Idle** state. Transitions are annotated with the filtered, abstract events created from the MS Windows events. If the author presses the enter key (EnterDown) the state changes from **Idle** to **BreakingParagraphs**. If that key is released again (EnterUp), it changes back to **Idle**. If any of the cursor keys is pressed (**MoveKeyDown**), the state changes to **Moving**.

Whenever the automaton enters one of the states **Moving** or **BreakingParagraphs**, a reference to the current paragraph is recorded. Then MS Word handles the key-press. When that state is left again, the position of the recorded paragraph is compared to the current paragraph and a range is calculated. That range is then reparsed.

To keep the structure information and type mapping, we compare the newly created structure with the stored structure. The bookmarks help us to associate possibly changed text spans in the MS Word document with the elements present in the XML structure.

5. RELATED WORK

This sections presents other approaches to create structured documents. It first sketches other XML editors and then compares the use of aTool with a pure MS Word approach.

5.1 XML Editors

Pure XML editors move the XML structure in the foreground. For example, SoftQuad's XMetaL [5] either works in *validating*- or *well-formed*-mode. In validating-mode it denies most user commands that would lead to invalid documents. A novice user will often not understand why certain commands do not work. Only in this mode may the user check his document for validity. The results of this batch oriented processing step are presented as a list of error messages that are linked to elements.

In well-formed-mode an initially provided DTD is used only to declare element type names. While the user is free to create any document structure he wishes, he may also define new element types and attributes. Once in well-formed-mode, he cannot validate his document.

Tagless Editor from i4i [4] also uses MS Word as its user interface, but conceals most of MS Word's functionality. Strict guidance by an Office assistant allows only to create valid documents.

5.2 MS Word

MS Word itself can be used to create structured text without using aTool. Applying MS Word styles to paragraphs and text sequences is much like assigning types to elements. Yet, there are differences that make aTool more powerful and better address the needs of authors.

According to Springer Verlag, authors do not use styles at all or correctly. We see mainly two reasons for this: (1) They do not know which MS Word styles they should use for their current article. (2) MS Word styles seem to only change the format of the text which can be achieved easier if the format is applied directly to the text. Authoring guidelines try to alleviate this, but fail as they contain to much information that is hard to comprehend. Very few authors read them at all.

The aToolKit contains this information in a formalized, compact way. aTool, not the author, checks for conformity. This way the first problem is solved as every author now gets informed when he does not use a required element type (or MS Word style in a scenario without aTool). Also, aTool offers only those element types the publisher intended. No "standard" styles from the user local configuration are offered that should not be used. The author cannot extend the list of element types by himself.

Because of the second reason, aTool rather accepts the fact that direct formatting is used and can derive element type information from direct formatting as well as from MS Word styles.

aTool supports a hierarchical document model which MS Word styles do only very limited: character styles can be nested in paragraph styles. aTool's model gives greater expressive power when formalizing the authoring guidelines. Hierarchical structure creation is supported on the author side, allowing more powerful editing commands.

6. SUMMARY

aTool is of value, if multiple authors need to write text documents with a predetermined structure. For these scenarios XML is a well established standard. The required structure is formally defined in the DTD and tools can automatically check the created documents for conformity.

aTool is designed as a hybrid solution that offers the benefits of MS Word with the costs of a little less XML support. aTool currently does not support ID/IDREF attributes very well. Their validity is checked, but no integration, e. g. with MS Word's cross reference mechanism is implemented. Use of entities is not supported in aTool. As implementing aTool took a year longer than expected no evaluation has been made yet. We are planning to do that in the coming year.

When using aTool, the costs for creating XML documents are low. MS Word is in most situations already available for every writer. aTool itself is free[4], so no additional software costs result. Training can be reduced to a minimum as the writers continue to use MS Word as before. If they are used to produce MS Word documents in a predetermined layout, aTool fully automatically creates the XML structure. With a well defined mapping table, applying an element type is done by applying a certain format within MS Word. The writers only need to learn how to react on invalid elements in their structure view.

Effort must be invested in the creation of an aToolKit. This might also mean creating a first DTD and changing existing layout requirements a bit. While aTool technically can work with any DTD, little simplifications will help with document creation later on. This only requires a single specialist and thus little training. The aToolKit provides enough adaption means to suite aTool for your specific needs. In the future we might provide an aToolKit editor or a consistency checker.

aTool was created as a research prototype and even after three years of development some parts of its implementation are incomplete. For example, aTool should integrate is data into the MS Word file when saving.

aTool's power is its integration with MS Word. If that is not required, because all authors are willing to switch to an XML editor, that is the way to go. If you are currently trying to produce consistent text documents in a group solely with MS Word, aTool might be worth a try.

7. ACKNOWLEDGMENTS

The work presented here has been supported by the German Ministry for education, research, and technology (BMBF) as part of the Global-Info project.

8. REFERENCES

[1] Apache Software Foundation. *Xerces C++ Parser.* `http://xml.apache.org/xerces-c`.

[2] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood, editors. *Document Object Model (DOM) Level 1 Specification.* W3C, Oct. 1998. Version 1.0.

[3] C. Fuß, F. Gatzemeier, M. Kirchhof, and O. Meyer. Inferring Structure Information from Typography. In *Digital Documents and Electronic Publishing (DDEP00)*, Lecture Notes in Computer Science. Springer, 2000.

[4] i4i. *Tagless Editor.* `http://www.i4i.com/product_TE.htm`.

[5] SoftQuad. *XMetaL.* `http://www.softquad/products/xmetal`.

[4]It can be downloaded at `http://www-i3.informatik.rwth-aachen.de/research/projects/atool`.