OpenSHORE SMRL Tutorial

Writing SMRL Specifications for Your Documents

Helge Schulz

Version: 1.0
State: Released
Modified: 2006-06-29

Summary

This document is a tutorial for writing specifications in the "Semantic Markup Rule Language" (SMRL). SMRL is an XML based specification language to mark or extract semantic meaningful sections and relations in/from structured human readable documents. This tutorial shows the usage of each SMRL XML element and lists all possible attributes with their meaning. The appendix contains technical instructions how to use Eclipse WST as SMRL editor and how to call the SMRL Metaparser (a XSLT implementation of SMRL).

Version History

Version	Date	State	Author	Changes
0.1	2006-05-26	Work in progress	Helge Schulz	First outline
0.2	2006-05-20	Work in progress	Helge Schulz	Added contents of section A.1
0.3	2006-06-03	Work in progress	Helge Schulz	Added introduction
0.4	2006-06-04	Work in progress	Helge Schulz	Finished basic matching and appendix
0.5	2006-06-05	Work in progress	Helge Schulz	Finished naming, references, relations
0.6	2006-06-06	Work in progress	Helge Schulz	Added example document frame
0.7	2006-06-07	Ready for review	Helge Schulz	Finished example and last section
0.8	2006-06-17	Ready for review	Helge Schulz	Incorporated enhancements from David Jenkins until section 3
1.0	2006-06-29	Released	Helge Schulz	Incorporated final enhancements

Copyright © 2006 Helge Schulz

OpenSHORE is Open Source Software; you can redistribute it and/or modify it under the sd&m Common Public License. You should have received a copy of this License along with OpenSHORE (see sdm-cpl-v10.html).

OpenSHORE is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the sd&m Common Public License for more details.

Table of Contents

1 Introduction **4**
2 Example **4**
3 Basic Matching **8**
3.1 Document Type 8
3.2 Marking Objects 11
3.2.1 Sections 11
3.2.2 List Items 12
3.2.3 Tables 13
3.2.4 Text Items 14
4 Building Names **15**
5 Referencing Matches **16**
5.1 Default Visibility 17
5.2 Explicit Visibility 18
5.3 Defining Dependencies 18
5.4 Implicit Namespaces 19
5.5 Explicit Namespaces 20
6 Marking Relations **20**
6.1 Regular Relations 21
6.2 Composite Relations 22
7 Combining rules **23**
A Appendix **24**
A.1 Using Eclipse WST as SMRL Editor 24
A.2 Running the SMRL Metaparser 26
A.2.1 Apache Ant 27
A.2.2 OpenOffice Filter 28
A.2.3 MS Word 28
Bibliography **29**
Alphabetical Index **30**

1 Introduction

This document is a tutorial for writing specifications in the "Semantic Markup Rule Language" (SMRL). SMRL is a specification language to mark or extract semantic meaningful sections and relations in/from structured human readable documents. It is independent from the concrete input document format and output result format. Therefore such specifications contain pattern definitions for elements which exist in every structured document, such as sections, lists and tables. See OpenSHORE.org for applications, further general information and Open-Source implementations of SMRL processors.

This document is **not** a reference or definition for SMRL. SMRL is an XML dialect and is therefore defined in an XML schema file (**`smrl.xsd`**, [Vlist2002]). This file contains documentation for every XML element and attribute. Please use the schema file as reference. You can use it together with schema aware XML editors to get context sensitive help and input completion while writing SMRL files. One example of a such an editor is the Open-Source XML editor of the WST Eclipse plugin. Please see appendix A.1 for screenshot examples of input help and how to configure Eclipse WST to get SMRL editing support.

This tutorial consists of five parts: Section 1 is this introduction; Section 2 contains a complete example of an input document and a corresponding SMRL specification. Section 3 explains basic matching and introduces all SMRL XML elements. Section 4 describes how to reference other matches to specify relations and build up name(s) (-spaces). The appendix contains technical instructions on how to use Eclipse WST as an SMRL editor and how to call the SMRL Metaparser (an XSLT implementation of SMRL).

2 Example

The following example shows how SMRL can be used to mark and extract semantic information from a rich text document. A simple structured cookbook is used as input and with help of SMRL a database of all contained recipes, related ingredients and instruction steps should be created. For this imagine that we are now in the year 2050 and a world wide semantic web has been implemented, containing all public documents that are available electronically. Query engines crawling this web can answer nearly all logical questions to information in these documents. So for example you can advise your computer to reserve a table in a restaurant with a good customer rating, not further than 1000 meters from your flat, for a free evening in your appointment book. To solve this task, there must be some agreed definitions for terms like "rating", "distance", "opening hours", "free evening" and web sites must mark this semantic information on their sites accordingly.

Now imagine you find your grandma's cookbook stored on a good old DVD. Your grandma wrote the book with a pre-XML rich text editor. Fortunately you find a conversion service which has specialized in reading legacy media like DVDs and converting legacy documents into XML. But the service can not add semantic markup to the resulting document and you cannot advise your computer to automatically order all needed ingredients for a specific recipe or to read instruction steps aloud while cooking as usual. In this situation you can use SMRL to add semantic markup to the

book.

Your grandma's cookbook has the following structure, which is also represented in XML elements for sections, items and tables:

My Grandma's Cookbook

...

1. Ingredients

1.1 Flour

...

1.2 Eggs

...

...

2. Recipes

2.1 Pancakes

Ingredients:

Quantity	Ingredient
2 cups	Flour
2	Eggs
...	...

Directions:

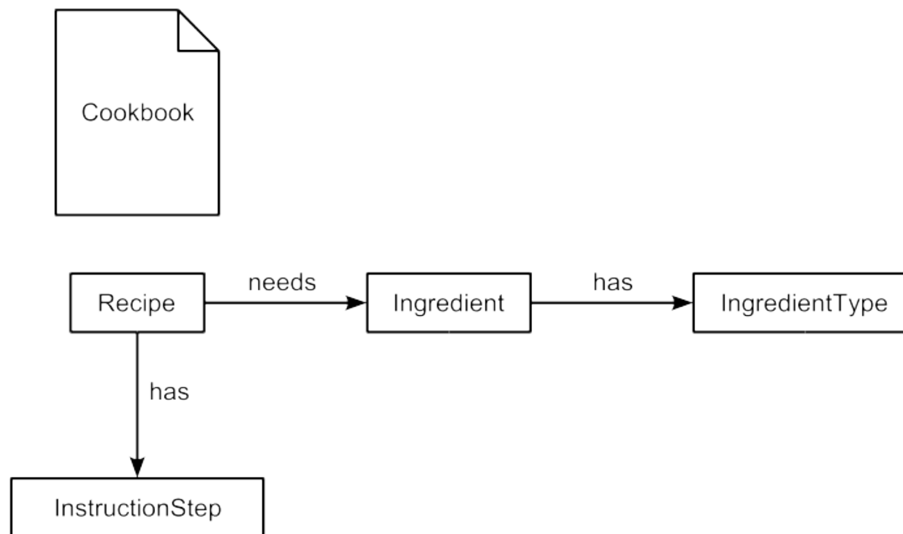
1. ...
2. ...
3. ...

...

Listing 1: Cookbook example document

Before you can extract semantic information, you must have data model for the information you want to extract. This model is often called meta model or an ontology, if terms are ordered in

inheritance trees. Here is a simple meta model for the document above:



Graphic 1: Cookbook meta model

The model defines the document type “Cookbook”, 4 object types (text parts with semantic meaning) and 3 relation types. The SMRL specification can now derived from the model and the document structure:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<smrl xmlns="http://OpenSHORE.org/schemas/smrl/1.0/">
   <document title-contains="Cookbook" type="Cookbook">                  ❶
      <section title-contains="Ingredients">
         <section type="IngredientType" to-lower-case="true"/>            ❷
      </section>
      <section title-starts-with="Recipes">
         <section id="RecipeName" type="Recipe">                          ❸
            <table>
               <row gt="1">
                  <col eq="1"                                             ❹
                     type="Recipe_needs_Ingredient"
                     source-type="Recipe"
                     source-ref="RecipeName"
                     target-type="Ingredient"
                     target-ref="IngredientName"
                  />
                  <col eq="2" id="IngredientName"                         ❺
                     type="Ingredient"
                     to-lower-case="true"
                     namespaces="RecipeName"
                     composite-rel-type="Ingredient_has_IngredientType"
                     composite-source-type="Ingredient"
                     composite-source-ref="IngredientName"
                     composite-target-type="IngredientType"
                     composite-target-ref="IngredientTypeName"
                  />
                  <col eq="2" id="IngredientTypeName"/>
               </row>
            </table>
            <item id="DirectionStepName" type="DirectionStep"             ❻
               name-by-position="true"
               namespaces="RecipeName"
               composite-rel-type="Recipe_has_DirectionStep"
               composite-source-type="Recipe"
               composite-source-ref="RecipeName"
               composite-target-type="DirectionStep"
               composite-target-ref="DirectionStepName"
            />
         </section>
      </section>
   </document>
</smrl>
```

Listing 2: SMRL specification for “Grandmas Cookbook”

❶ The document tag marks the document as cookbook, if the title contains the word “Cookbook”.

❷ Subsections of the section titled with 'Ingredients' are marked as ingredients and the heading converted to lower case is used as the name.

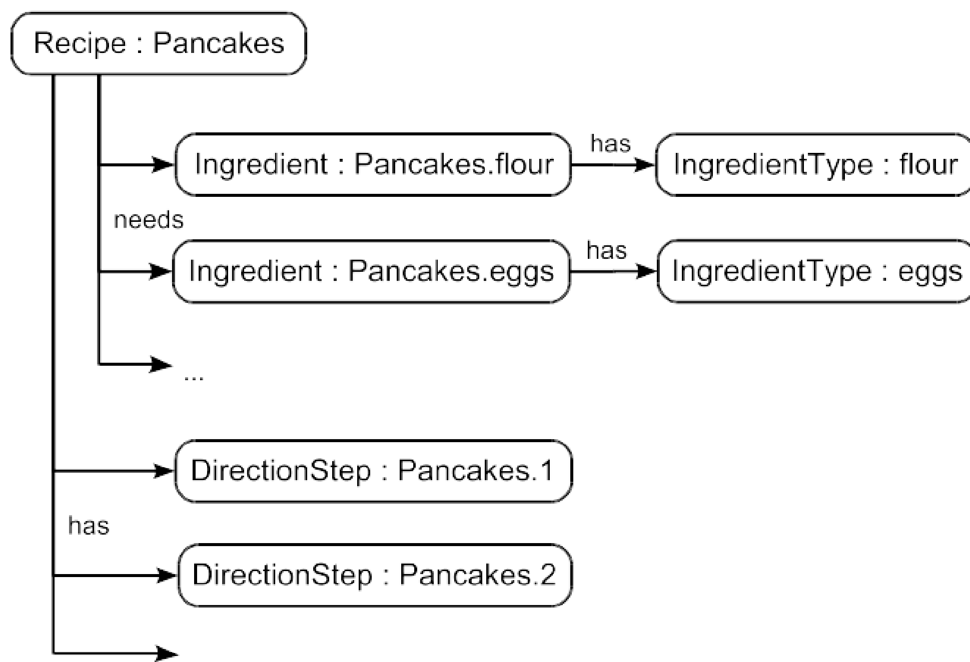
❸ Subsections of the section titled with 'Recipes' are marked as recipe and the heading is used as the name.

❹ Table cells of the first column after the first row contain the ingredient quantity and are marked as the relation 'Recipe_needs_Ingredient'.

❺ Table cells of the second column after the first row contain the ingredient name and are marked as such. An additional composite relation connects them with their types.

❻ List items inside recipes are marked as direction steps. They are numbered and their name is prefixed with the enclosing recipe name (namespace attribute). An additional composite relations connects them with the enclosing recipe.

SMRL processors may create very different output formats. They all have in common that the result document represents a semantic graph of the content of the input document. The semantic graph of the example document from listing 1 above is shown in graphic 2 below. Example result file formats are SHORE XML, “XML Topic Maps” (XTM, [Mück2002] chapter 11), “Graph eXchange Language” (GXL) and “Resource Description Format” (RDF). From all these formats a database can be filled with the semantic graph. This database can queried afterwards to find all ingredients and instruction steps to a specific recipe.



Graphic 2: Semantic graph derived from Cookbook example

3 Basic Matching

This section introduces all XML elements used by SMRL specifications and explain basic matching of input document structures. Higher level concepts like namespaces and references to matches are implemented as additional XML attributes explained in later sections.

3.1 Document Type

An SMRL specification file can contain rules for several document types. A document type defines common characteristics of a set of documents. Typical document types for a software project are for example “Requirement Specification”, “Use Case Specification”, “Data Model”, “Test Case De-

scriptions" etc. So the first task in writing a new SMRL specification is to define and match document types. This is done by using the "**`document`**" XML element and the document type is defined by the contained "**`type`**" XML attribute. This element is the only top level element inside the "**`smrl`**" document element. The "**`type`**" attribute is used whenever document content is associated with a semantic meaning and to declare it to be an instance specific type of semantic statement.

Matching is done by predicates defined as XML attributes of the document element. SMRL provides the two attributes "**`title-contains`**" and "**`description-contains`**" as predicates for document type matching. This attributes match if either the document title or the document description contains the given string value of the attribute. Document title and description are extracted from the document meta data and **not** from document text content, which might be displayed or styled as title or document description. Meta data title and description are usually set via a special document property dialog of the rich text editor. This is also the reason why SMRL has only limited features to match document types, because title and description properties can be set inside a document template file, which is then used for all documents of this type.

Hence the template for the top level elements of a SMRL file looks like this:

```
<?xml version="1.0"?>
<smrl xmlns="http://OpenSHORE.org/schemas/smrl/1.0/">

   <document
      type="[document type name]"
      [document predicates]
   >
      [pattern rules of document type]
   </document>

   [...]

</smrl>
```

XML Attribute	Description
`title-contains`	(Meta data) title contains string
`description-contains`	(Meta data) description contains string

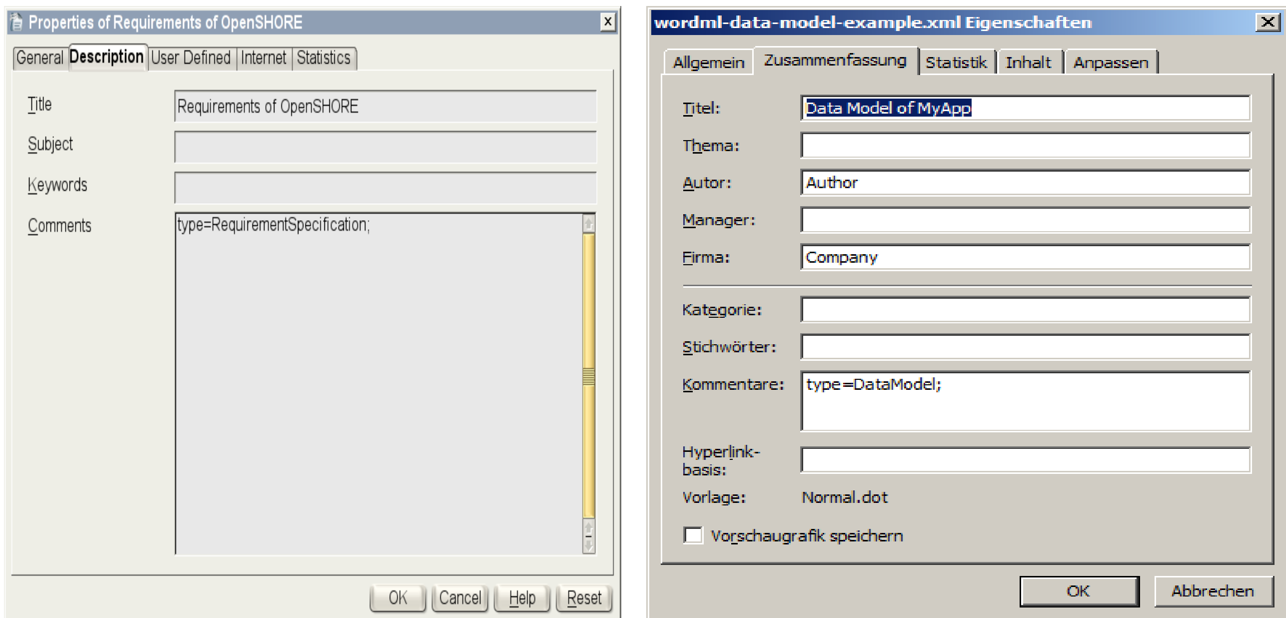
Table 1: Document predicate attributes

If you are writing SMRL specification for all documents of a software project, the skeleton of your SMRL file might look like this:

```
<?xml version="1.0"?>
<smrl xmlns="http://OpenSHORE.org/schemas/smrl/1.0/">

   <document
      type="RequirementSpecification"
```

```
        title-contains="Requirements of"
    >
        [pattern rules for requirements]
    </document>

    <document
        type="UseCaseSpecification"
        description-contains="type=UseCaseSpecification;"
    >
        [pattern rules for use cases]
    </document>

    <document
        type="TestCaseDescription"
        title-contains="Test Case"
    >
        [pattern rules for a test case]
    </document>

</smrl>
```

To change the meta data title and description select menu “File → Properties” in OpenOffice and MS Word. Screenshot 1 below shows the meta data dialog of both applications.



Screenshot 1: Setting title and description in OpenOffice and MS Word

3.2 Marking Objects

Objects in the sense of SMRL are text sections inside a document which define or describe a semantically meaningful object of the system which is described by the document. Each such object has a type and a name. The name should be unique for a type, if all documents are in a consistent state and might be composed from several name spaces or numbering parts (see section 4 on page 15 for further details). The following sections describe SMRL pattern elements to mark/extract document sections as such objects.

3.2.1 Sections

The "**section**" XML elements marks/extracts a whole document section as an object. Such a document section is derived from the logical hierarchical (outline) section structure of a document with a composite tree of high level parts (chapter or section) and composite elements (subsection, subsubsection etc.). So if a top level section is marked, all contained sub- and subsubsections are part of the marked region.

The type of a section object is defined by the "**type**" XML attribute. The name of the object defaults to the section heading and can be tailored with several attributes described in section 4. The matching of a section rule depends on title and content predicates listed in table 2 and 3. If several predicates are used, all must be true for matching (logical "and"). So the template for a section object rule looks like this:

```
<section
    type="[object type name]"
    [title and content predicates]
    [name tailoring attributes]
>
    [object and relation rules for a matched section]
</section>
```

XML Attribute	Description
title-contains	Section heading contains the given string
title-not-contains	Section heading does not contain the given string
title-starts-with	Section heading starts with the given string
title-ends-with	Section heading ends with the given string

Table 2: Title predicate attributes

XML Attribute	Description
contains	Section contains the given string
not-contains	Section does not contain the given string
not-equal	Section text is equal to the given string
pred-contains	A predecessor (preceding sibling) section contains the given string
succ-contains	A successor (following sibling) section contains the given string
is-number	The whole section text is a number

Table 3: Content predicate attributes

If you want mark for examples all sections inside an use case description document and section titles of uses cases starts with “Use Case:”, your SMRL section rule might look like this:

```
<section
   type="UseCase"
   title-starts-with="Use Case:"
   name-after=": "
>
   [object and relation rules for an use case]
</section>
```

3.2.2 List Items

The “**item**” XML element marks/extract a list item as an object. This element matches items of ordered (with numbers) and unordered lists (with bullets) on any indentation level. This lists might also be enclosed in tables.

The type of the item object is defined by the “**type**” XML attribute. The name of the object defaults to the item content text and can be tailored with several attributes described in section 4. The matching of an item rule depends on content predicates as listed in table 3 above. If several predicates are used, all must be true for matching (logical “and”). An item rule is usually combined with a section rule to match only items of lists in a specific section of the document. So the template for an item object rule looks like this:

```
<item
   type="[object type name]"
   [content predicates]
   [name tailoring attributes]
>
   [object and relation rules for a matching item]
</item>
```

For example, if you want to mark all possible use case states (e.g. “work in progress” to “accepted”) and they are defined as unordered list items in a section titled “Use Case States”, your SMRL rule

might look like this:

```
<section title-contains="Use Case States">
   <item
      type="UseCaseState"
      name-before=":"
   />
</section>
```

3.2.3 Tables

The “**table**” XML element with enclosed “**row**” and “**col**” elements for table rows and columns provides rules to match table cells. The table element contains row elements and row elements contain column elements. This is similar to the table model of HTML and DocBook.

The type of the table cell object is defined by the “**type**” XML attribute. The name of the object defaults to the cell content text and can be tailored with several attributes described in section 4. The matching of table elements depends on title predicates applied to the table caption (see table 2 above). The matching of row and column elements depends on content predicates as listed in table 3 above and position attributes described in table 4 below. The row and column numbering starts with one. If several predicates are used for one element, all must be true for matching (logical “and”). So the template for a table rule looks like this:

```
<table
   [title predicates]
>
   <row
      [content predicates]
      [position predicates]
   >
     <col
        type="[object type name]"
        [content predicates]
        [position predicates]
     >
        [object and relation rules for matching cell]
     </col>
     [column elements for same row]
   </row>
   [row elements for same table]
</table>
```

XML Attribute	Description
eq	Position is equal to given value
lt	Position is less then given value
gt	Position is greater then given value

Table 4: Position predicate attributes

For example, if you want to mark all possible use case states (e.g. “work in progress” to “accepted”) and they are defined in the first column of a table titled “Use Case States”, your SMRL rule might look like this:

```
<table title-contains="Use Case States">
   <row pred-contains="State">
      <col
         type="UseCaseState"
         eq="1"
      />
   </row>
</table>
```

3.2.4 Text Items

The “**text**” elements provides rules for matching single words inside text as objects (text items). There are two alternatives forms for matching text:

1. The matched words are positioned between two text markers (also words or special characters separated by whitespace). Several words are separated by whitespace or punctuation characters.
2. The matched words starts with a common prefix, which identifies their type (i.e. “**REQ_**” for references to a requirement).

The type of the text item is defined by the “**type**” XML attribute. The name of the object defaults to the word itself and can be tailored with several attributes described in section 4. A text item rule can be declared as global, if it is inserted as child of the “**document**” element and the SML attribute “**global**” is set to “**true**”. Such a rule is applied to all text paragraphs which are not subject to any other rule. Possible XML attributes of a text rules are listed in table 5 below. So the templates of both forms look like this:

```
<text
   type="[object type name]"
   marker-starts-with="[marker start word]"
   marker-ends-with="[marker end word]"
   [optional global attribute]
/>
```

```
<text
    type="[object type name]"
    word-starts-with="[word prefix]"
    [optional global attribute]
/>
```

XML Attribute	Description
global	Text rule is marked as global
marker-starts-with	Start marker for words as text items
marker-ends-with	End marker for words as text items
word-starts-with	Prefix for searched words

Table 5: Text item attributes

If you want mark for examples all words between “(possible use case states:” and “)” as use case states, your rule might look like this:

```
<text
    type="UseCaseState"
    marker-starts-with="(possible use case states:"
    marker-ends-with=")"
/>
```

Another possibility is to prefix all possible states with “UCS_” in a defining section:

```
<section title-contains="Use Case States">
    <text
        type="UseCaseState"
        word-starts-with="UCS_"
    />
</section>
```

4 Building Names

All pattern rules described in the previous section mark a named object, if they are match. Naming of these objects is existential, because relation definitions use these names to specify their end points and therefore a name should be unique between all instances of an object type. Applications reading output files of SMRL processor will also present object names to users, so users should recognize these names and they should not deviate from usual naming schemes. SMRL therefore provides several XML attributes to tailor names. Such attributes can be applied to all pattern rule elements.

One special situation is the case of elements containing a list of names. In such cases every word separated by whitespace or punctuation characters represents an object or relation of its own. For these cases SMRL provides the "**`word-list`**" attribute, which switches to word handling instead of using the whole heading or content as one name.

Name tailoring attributes act either directly on the current element or they reference to other matches to composite a name of several elements. Table 6 lists all attributes acting on the current element. Attributes with references ("**`name-ref`**" and "**`namespaces`**") are explained in the following section.

XML Attribute	Description
`name-after`	Use characters after the given string as name
`name-before`	Use characters before the given string as name
`name-prefix`	Add the given string as name prefix
`name-postfix`	Add the given string as name postfix
`name-by-position`	Use the relative position number (sibling number) as name
`to-lower-case`	Convert the name to lower case characters
`to-upper-case`	Convert the name to upper case characters
`number-format`	If the name is a number, format it by given template (containing "`.,%#0`")
`decimal-format`	Reference to XSLT decimal format for numbers
`word-list`	Use every word separated by whitespace/punctuation characters as name

Table 6: Name building XML attributes

5 Referencing Matches

Identification and naming of objects is only one part of semantic markup. The second part is detection and marking of relations between objects inside and outside the document. Objects are identified by their type and name. Consequently the task of relation definition is to find and select type and name of the relation source and target objects. Therefore it is not sufficient to extract a name from a current element, because this name can either be the source or the target name. Sometimes neither name is part of the matching element and you must reference other matches to select them.

SMRL provides an identification and reference system for matches to select relation end names, define dependencies and namespaces. Every pattern rule may contain the "**`id`**" XML attribute to associate a match with a given identifier. These identifiers can then be referenced at other places. Such identifiers have a defined visibility, which is related to the structure of the input document (not of the SMRL document). It defines where other matches can reference a match to extract its name. The default visibility enclose all sibling elements (elements on same level) and all their children (elements on enclosed sublevels). This default can be changed by the "**`visibility`**" attribute explained in next sections.

Table 7 lists all XML attributes related to identifiers and references without relation definition. Relations are explained in section 6. The following sections describe the usage of references.

XML Attribute	Description
`id`	Associate a match with the given identifier
`visibility`	Change visibility of a match from default to named behavior
`depends`	Link matching to other matches
`namespaces`	Use names of outer matches listed by identifiers as namespace
`name-ref`	Compose name from names of other matches listed by identifiers

Table 7: Identification and reference XML attributes

5.1 Default Visibility

The default visibility encloses all sibling elements (elements on same level) and all their children (elements on enclosed sublevels). This region is broadened to all nephews at the same position, if a rule contains no "**`type`**" attribute. These function is very practical for matching matrix oriented structures such as tables. Nephews at same position means that they match in the same column.

For example, if you want to mark database table names in a document table containing a table space name in the first column and a table name in the second column and you want prefix the table name by the corresponding table space name, your SMRL fragment might look like this:

```
<table title-contains="Tables">
   <row gt="1">
     <col
        eq="1"
        id="TableSpaceName"
        type="DatabaseTableSpace"
     />
     <col
        eq="2"
        id="TableName"
        type="DatabaseTable"
        name-ref="TableSpaceName TableName"
     />
   <row>
</table>
```

Because all column elements are siblings, the "**`TableSpaceName`**" and "**`TableName`**" identifiers are visible in the whole row and all sub elements. The "**`name-ref`**" attribute composes the table name of both names and concatenates them with a dot ("`.`"). This is also called explicit name space handling (see section 5.5). If you want to use table column headers to select columns instead of using hard coded numbers, you can rely on the "nephews at same position" rule to do this:

```
<table title-contains="Tables">
   <row eq="1">
      <col contains="Table Space" id="TableSpaceColumn"/>
      <col contains="Table" id="TableColumn"/>
   </row>
   <row gt="1">
     <col
        depends="TableSpaceColumn"
        id="TableSpaceName"
        type="DatabaseTableSpace"
     />
     <col
        depends="TableColumn"
        id="TableName"
        type="DatabaseTable"
        name-ref="TableSpaceName TableName"
     />
   <row>
</table>
```

Because the "**col**" elements for matching table headers have no "**type**" attribute, the used identifiers are visible in the whole table column and can be used to declare their dependencies.

5.2 Explicit Visibility

You can change the default visibility by setting the "**visibility**" XML attribute for an element with an identifier. Table 8 lists all legal values, their meaning and for which input format they can be used. If more then one match has a visibility on the same level, only the first match will be found. So please use caution when defining rules with broader visibility.

Visibility	Rich Text	Spreadsheet	Description
default	√	√	Siblings elements and all children elements
global	√	√	Whole document
section	√		Section containing match
table	√	√	Table containing match
indent		√	Following rows with next one lower indent level

Table 8: Possible visibility values

5.3 Defining Dependencies

SMRL provides the "**depends**" XML attribute to define explicit dependencies between matches. This attribute contains a space separated list of identifiers from other matches, which must all be found to result in a matching of the rule. Dependencies are not resolved recursively, so if the refer-

enced match also has dependencies, they are **not** checked (only primary predicate attributes are checked). Implicit dependencies also exists as defined by the relations attributes explained in section 6. SMRL processors do not issue a warning about missing dependencies. This is because you can define several rules at the same level, which are then selected by dependencies (logical "or"). It would be annoying to get several messages for every input element evaluated by such rules. If you want to get these messages, you must set/increase the debugging level of the SMRL processor used.

Please use and define dependencies with care, because evaluation of dependencies is more complex than checking primary predicate attributes. SMRL processors might use an algorithm with quadratic or worser complexity to evaluate dependencies. If you process a large spreadsheet with more then 1000 rows and 500 columns, the processing time might rise over half an hour, if you define a dependency for every cell ((1000 * 500) ^ 2 = 250 billion steps).

5.4 Implicit Namespaces

Names of objects must be unique between all instances of an object type to distinguish them from each other and establish proper relations. But often identical names are used in a large system in different contexts and this context is not only related to the object type. Many systems defines a system of namespaces to avoid name clashes in such cases. These namespaces must be transformed into SMRL to avoid name clashes their too. SMRL provides two forms of namespace building to resolve this. The more convenient form are implicit namespaces. But it can not used in all situations, however, so an explicit namespace handling is also provided, described in section 5.5.

You can use implicit namespaces if the structure of your input document corresponds to the structure of namespaces. This means that every element of a namespace must be enclosed in an outer element defining the namespace. An example for such a structure would be sections defining software components, where all interfaces of the component are defined in subsections.

SMRL provides the "**`namespaces`**" XML attribute to use implicit namespaces. This attribute contains a list of identifiers referencing outer matches. The names of these outer matches are used as prefix to the current name separated by dots ("."). If your input document defines software components in a document section, for example, your SMRL rules might look like this:

```
<section
   title-contains="Component:"
   id="ComponentName"
   type="Component"
   name-after=": "
>
   <section
      title-contains="Interface:"
      type="Interface"
      namespaces="ComponentName"
      name-after=": "
   />
/>
```

In this example the interface name is prefixed with the component name. The advantage of implicit namespaces is that all references to the inner matches are automatically use the fully qualified composed name.

5.5 Explicit Namespaces

If your the structure of your documents does not corresponds to the namespace structure, you must use explicit namespaces. This is done by using the "**`name-ref`**" XML attribute. This attribute contains a list of identifiers referencing other matches. The name is composed by concatenating all names of referenced matches by a dot ("**`.`**"). Consequently the example of the previous section can be written with explicit namespaces as follows:

```
<section
   title-contains="Component:"
   id="ComponentName"
   type="Component"
   name-after=": "
>
   <section
      title-contains="Interface:"
      id="InterfaceName"
      type="Interface"
      name-ref="ComponentName InterfaceName"
      name-after=": "
   />
/>
```

The disadvantage of explicit namespaces is that you must repeat all identifier names in all references that refer to the composed name. In the example above you must always use "**`ComponentName InterfaceName`**" to obtain the full interface name.

6 Marking Relations

Relations marked by an SMRL processor inside documents have a relation type, a source object and a target object. Objects are identified by type and name. They might be defined inside or outside of the processed document. The SMRL processor does not resolve references to objects. This job is left to a semantic repository like the OpenSHORE server or other servers, which are read SMRL processing output files of related documents and build up a semantic network inside a database. But the SMRL processor must determine names and types of both relation ends.

Regular relations are user visible inside the input document by using names as references to other objects. Non user visible relations implicitly defined by the document structure are handled as composite relations.

6.1 Regular Relations

Rules for regular relations are defined with the same XML elements as rules for objects. You can use every XML element described in section 3 (Basic Matching) to define a rule resulting in a relation, if it matches. The difference between an object matching rule and a relation rule is that the "`type`" attribute contains a relation type name and that the first **four** XML attributes of table 9 must be defined. These attributes define type and name of the source and target objects. Attributes "`source-type`" and "`target-type`" are simple names as used for the "`type`" attribute of objects. Attributes "`source-ref`" and "`target-ref`" are space separated lists of identifiers of other matches. The source and target name used are composed from these matches.

XML Attribute	Description
`source-type`	Type name of source object
`source-ref`	Identifier(s) of matches containing the source object name
`target-type`	Type name of target object
`target-ref`	Identifier(s) of matches containing the target object name
`direction`	Navigation direction ("`source`" or "`target`", defaults to "`target`")

Table 9: Relation XML attributes

We can distinguish the following three types of relation rules:

1. The target name is contained inside the matched element (usual case for references)

   → "`source-ref`" refers to an outer object which references the target
   → "`target-ref`" refers to the identifier itself

2. The source name is contained inside the matched element (inverted relation definition)

   → "`source-ref`" refers to the identifier itself
   → "`target-ref`" refers to an outer object which is the relation target
   → "`direction`" might be set to "`source`" to invert navigation direction

3. Neither source nor target name are contained inside the matched element

   → "`source-ref`" refers to another match containing the source name
   → "`target-ref`" refers to another match containing the target name

For example, in a requirement specification, if you want to mark all occurrences of words starting with "REQ_" as references to another requirement, your SMRL fragment might look like this:

```
<section
   title-starts-with="REQ_"
   id="RequirementName"
   type="Requirement"
```

```
>
   <text
      word-starts-with="REQ_"
      id="ReferencedName"
      type="Requirement_refers_Requirement"
      source-type="Requirement"
      source-ref="RequirementName"
      target-type="Requirement"
      target-ref="ReferencedName"
   />
</section>
```

6.2 Composite Relations

Composite relations are relations which are implicitly defined by the document structure and are not directly visible to the user. Typical examples are “part of” relations which are often implicit defined by the document structure. Such relations do not need XML rule elements of their own. They are defined instead as a set of additional attributes, which are in turn defined with another object or relation rule. Table 10 lists these set of additional XML attributes. Their meaning is exactly the same as the corresponding attributes of regular relations (without “**composite-**”).

XML Attribute	Description
composite-rel-type	Relation type name
composite-source-type	Type name of source object
composite-source-ref	Identifier(s) of matches containing the source object name
composite-target-type	Type name of target object
composite-target-ref	Identifier(s) of matches containing the target object name

Table 10: Composite relation XML attributes

If you want for example to parse a software specification document with component and enclosed interface definitions, your SMRL fragment might look like this:

```
<section
   title-contains="Component:"
   id="ComponentName"
   type="Component"
   name-after=": "
>
   <section
      title-contains="Interface:"
      id="InterfaceName"
      type="Interface"
      namespaces="ComponentName"
      name-after=": "
```

```
        composite-rel-type="Component_implements_Interface"
        composite-source-type="Component"
        composite-source-ref="ComponentName"
        composite-target-type="Interface"
        composite-target-ref="InterfaceName"
    />
/>
```

7 Combining rules

Pattern XML elements inside a “document” element can be combined by placing them on the same level or by enclosing them in on another. Elements on the same level are evaluated in document order and set up a logical “or” between the predicates used. Enclosed elements set up a logical “and” between between inner and all outer predicates. Declared dependencies defined by the “depends” XML attribute set up also a logical “and”. You can combine declared dependencies with several elements on the same level (“or”) to construct all possible logical expressions between defined predicates. The following listing shows these possibilities:

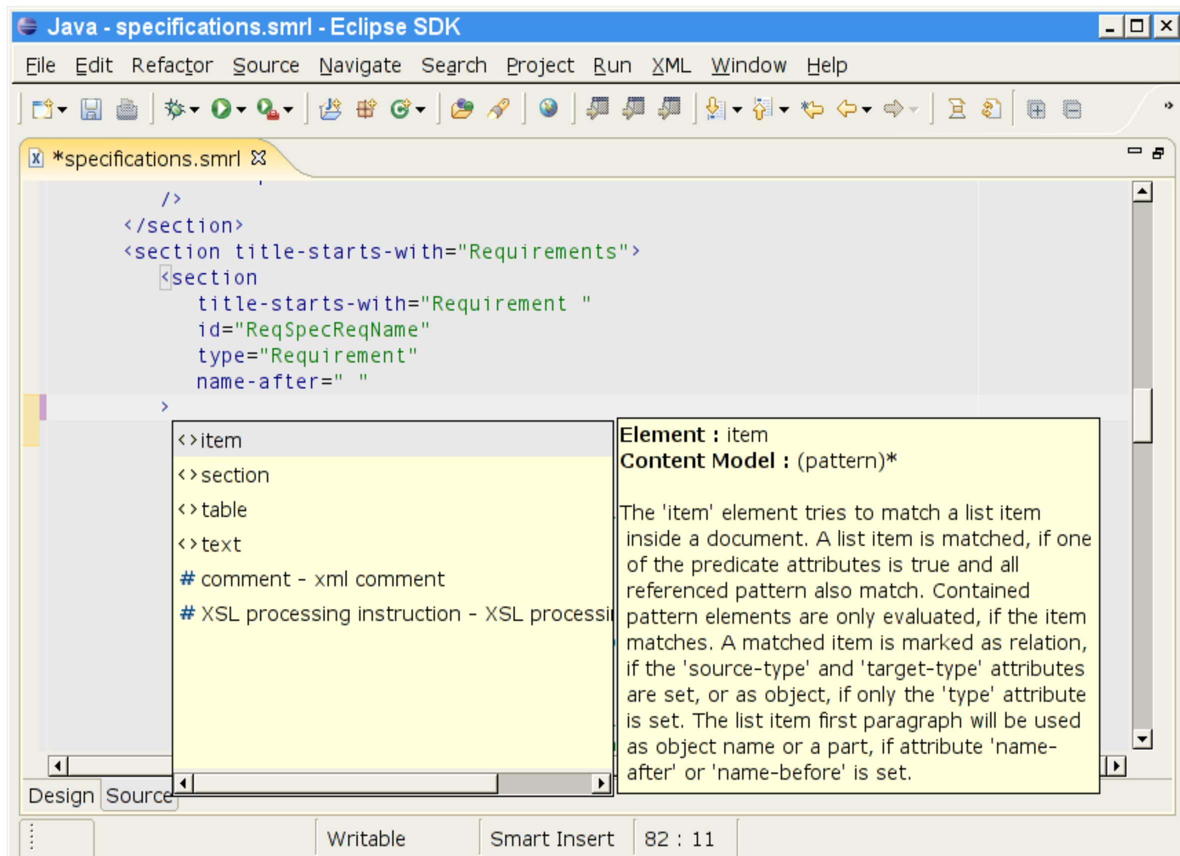
```
<document ...>

   <section ❶> ... </section> <section ❷> ... </section>    ❶ ∨ ❷

   <section ❸>
      <section ❹> ... </section>                               ❸ ∧ ❹
      <item ❺> ... </item>                                     ❸ ∧ ❺
   </section>

   <section id="s1" ❻/>
   <section depends="s1" ❼> ... </section>                     ❻ ∧ ❼

   <section id="s2" ❽/> <section id="s2" ❾/>
   <section depends="s2" Ⓐ> ... </section>              (❽ ∨ ❾) ∧ Ⓐ

</document>
```

If an SMRL pattern element is enclosed in another element, then SMRL processors “jumps” over non-matched elements of the input document inside the enclosed element to find the inner element. If you do not want this behavior, you can set the “**`strict-enclosing`**” XML attribute to “**`true`**” in the outer element to switch it off.

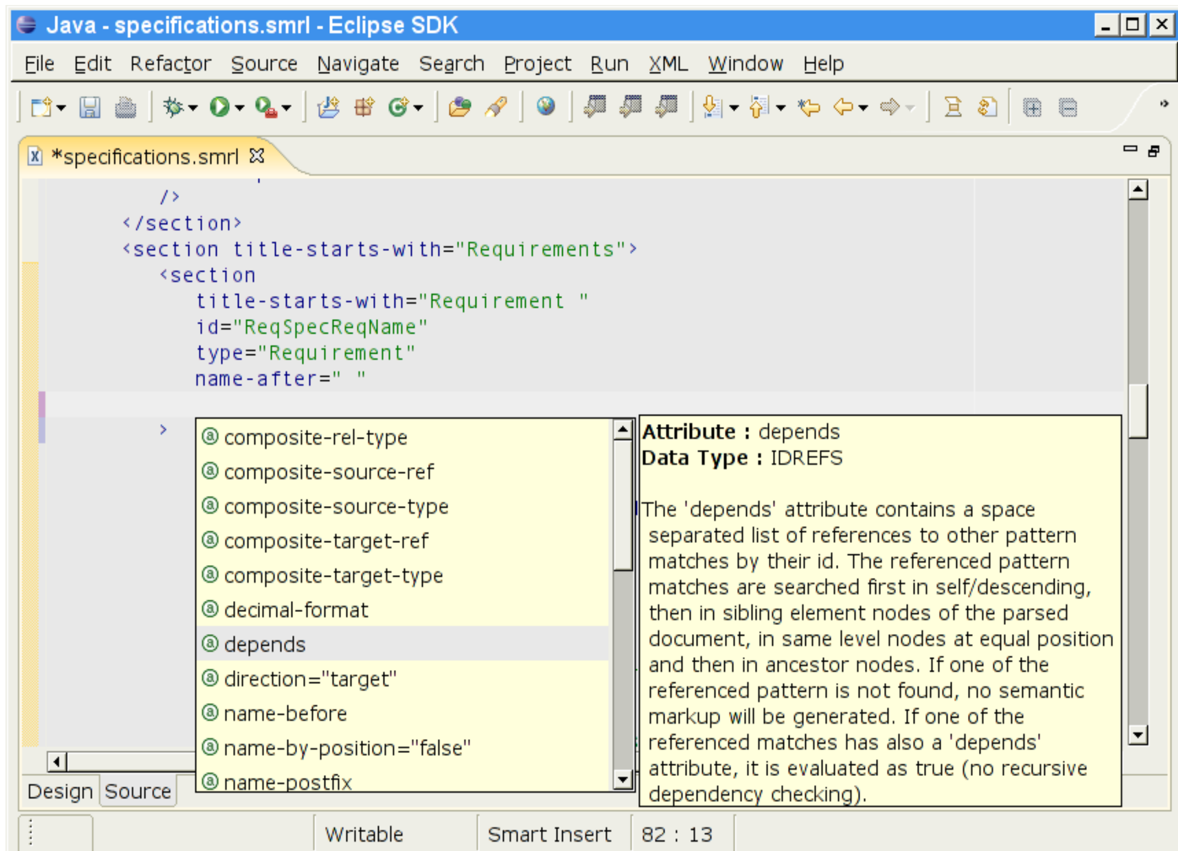
A Appendix

A.1 Using Eclipse WST as SMRL Editor

The open source development environment Eclipse 3.1 (or newer) with its XML editor from the "Web Standard Tools" (WST) plugin provides excellent support for editing SMRL specifications. You can add the SMRL XML name space URL (**http://OpenSHORE.org/schemas/smrl/1.0/**) with a reference to the SMRL schema file (**smrl.xsd**) to the Eclipse XML catalog. Eclipse provides afterwards context sensitive help with lists of insert able elements and attributes and descriptions from the schema documentation tags (see screenshot 2 and 3 on page 25).



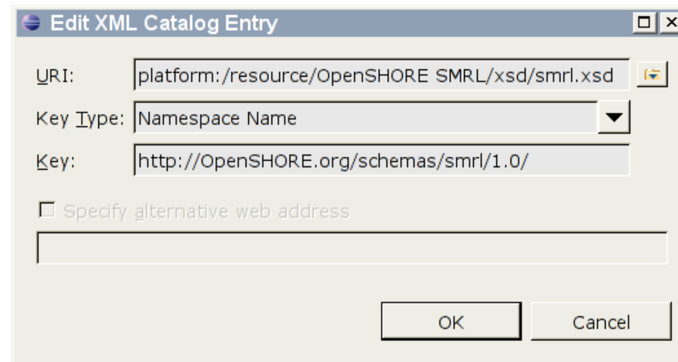
Screenshot 2: Eclipse context help for SMRL elements



Screenshot 3: Eclipse context help for SMRL attributes

Please follow these steps for adding the SMRL schema file to the Eclipse XML catalog:

1. Select “Window → Preferences → Web and XML → XML Catalog → User Specified Entries → Add”
2. Press the folder button in the right corner and select the SMRL schema file (**`smrl.xsd`**) from the current workspace (Eclipse will store a relative path with the project name as root) or from the file system (Eclipse will store an absolute path)
3. Select “Namespace Name” as key type
4. Enter “**`http://OpenSHORE.org/schemas/smrl/1.0/`**“ as key and press the “Ok” button (see screenshot 4 below)



Screenshot 4: SMRL XML catalog entry

You must associate the SMRL extension (`*.smrl`) with the Eclipse WST XML editor (with menu "Window → Preferences → General → Editor → File Associations") and use the correct name space in your SMRL file to get context sensitive help from Eclipse. Your file could look like the following template (using SMRL as default namespace):

```
<?xml version="1.0" encoding="..."?>
<smrl xmlns="http://OpenSHORE.org/schemas/smrl/1.0/">
   <document
      ...
   >
      ...
   </document>
   ...
</smrl>
```

A.2 Running the SMRL Metaparser

The SMRL Metaparser is an implementation of a SMRL processor in standard XSLT and can be downloaded from the OpenSHORE project pages as Open Source. The Metaparser main XSLT driver scripts are named by the following schema:

```
smrl4<input format>2<output format>.xslt
```

This can be read as "SMRL **for** <input format> **to** <output format>". You can run these scripts with your favorite XSLT processor, but the recommended way to run the SMRL Metaparser is to use of Apache ANT or Maven. This Java build tools are platform independent and provides all necessary abstractions for collecting files and paths to apply the parser in a well defined manner on your documents. For convenience and testing purposes you can also run the parser directly from inside OpenOffice 2.0 and MS Word 2003. The following sections describe all these alternatives.

A.2.1 Apache Ant

You can directly use the Ant “style” task to run the SMRL parsers. You can set up the path of the SMRL specification file used and the debugging mode over XSLT parameters. Here is an excerpt of an ANT build file creating SHORE XML from WordML XML files:

```
<target name="parse">
   <style
      basedir="${input.dir}"
      includes="*.xml"
      destdir="${output.dir}"
      extension=".xml"
      style="${smrl.dir}/xslt/smrl4wordml2shore.xslt"
   >
      <param
         name="smrl.file"
         expression="${basedir}/MySpec.smrl"
      />
      <param name="debug.level" expression="0"/>
   </style>
</target>
```

If the input documents contain images in WMF/EMF format and the parsers should convert them into PNG, you must add the path of the XSLT extension and several OpenOffice JAR files to the XSLT processors classpath. If you set up the environment variable **`OFFICE_HOME`** to your OpenOffice installation path, you can do this with the following Ant definitions for example:

```
<property environment="env"/>

<path id="image2shore.classpath">
   <path location="/${env.OFFICE_HOME}/program"/>
   <fileset dir="/${env.OFFICE_HOME}/program/classes">
      <include name="juh.jar"/>
      <include name="jurt.jar"/>
      <include name="ridl.jar"/>
      <include name="unoil.jar"/>
   </fileset>
   <fileset dir="${smrl.dir}/lib" includes="image2shore-*.jar"/>
</path>

<target name="parse">
   <style
      ...
   >
      <classpath refid="image2shore.classpath"/>
      ...
   </style>
</target>
```

The OpenOffice program directory must be included in the classpath, because the Java interface bootstrap code is searching the OpenOffice binary inside the classpath. If the XSLT extension does not find OpenOffice classes, it tries to call an installed ImageMagick binary for transformation. This only works on MS Windows, though, because ImageMagick do not implement WMF/EMF conversion on other platforms.

A.2.2 OpenOffice Filter

To run SMRL parsers from inside OpenOffice install the SMRL filter package by follow these steps:

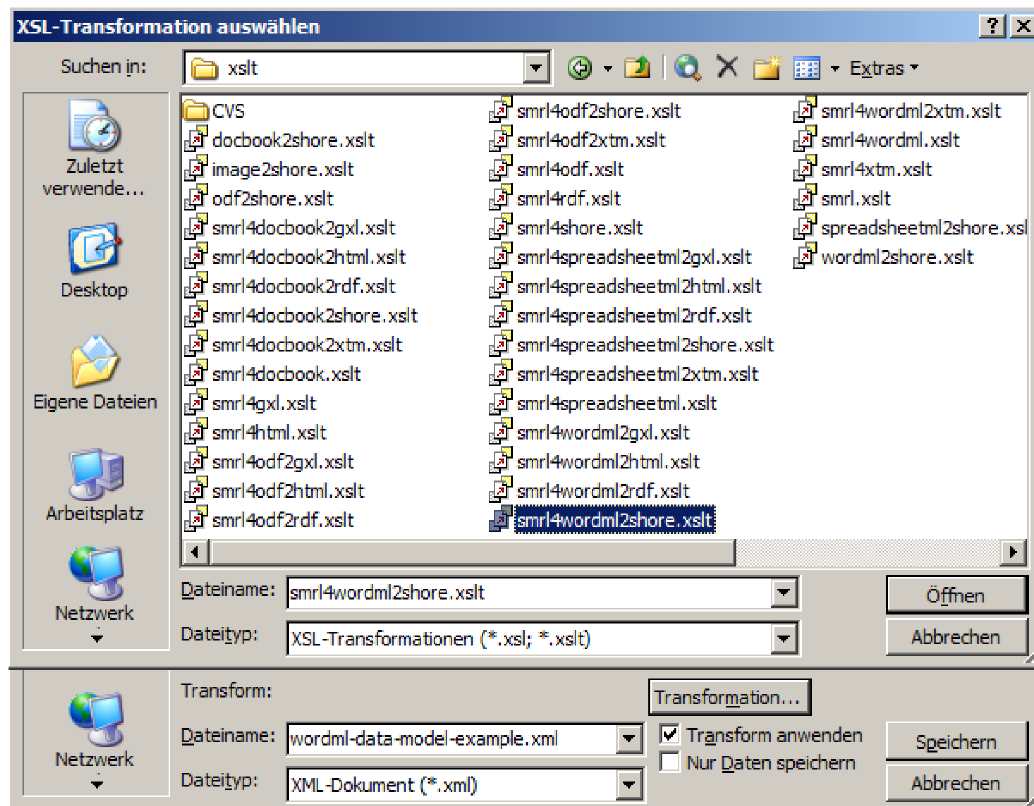
1. Startup OpenOffice
2. Open a new text document (File → New → Text Document)
3. Select menu Tools → XML Filter Settings
4. Click on button "Open Package"
5. Choose the OpenOffice filter package file inside the **`filter`** directory

Afterwards you will find new output formats inside the OpenOffice export file dialog. To transform a document, select "File → Export → File format → SMRL * → Export". All these export filters are using the default SMRL specification file (**`documents.smrl`**) inside the user specific OpenOffice configuration directory (**`.../user/xslt/OpenSHORE...`**). The SMRL filter package also contains an input/output filter to store documents inside a flat XML file without compression and splitting into several XML parts. Such files also contain all images and embedded objects as base 64 coded data URLs. The flat XML format can be used for batch transformations without using OpenOffice. This file format can be selected inside the "save as" and "open" dialogs ("Flat XML (**`*.xml`**)").

A.2.3 MS Word

Because the SMRL Metaparser is compatible with the MS XSLT processors from MSXML 3.0 and 4.0, you can run SMRL parsers directly from inside MS Word 2003. To do this, open the input document in MS Word and follow these steps:

1. Select "File → Save as"
2. Check option "Apply transform"
3. Press button "Transform..."
4. Select the desired SMRL parser (i.e. **`smrl4wordml2xtm.xslt`**) and press the "Open" button (see screenshot 5 below)



Screenshot 5: Running SMRL from MS Word 2003

5. Press the “Save” button

Bibliography

[Mück2002] Richard Widhalm, Thomas Mück. Topic Maps. Springer, ISBN 3-540-41719-2, Berlin, Germany, 2002.

[Vlist2002] Eric van der Vlist. XML Schema. O'Reilly, ISBN 0-596-00252-1, Sebastopol, USA, 2002.

Alphabetical Index

A

Ant 27

E

Eclipse 24

  WST 24

M

MS Word 2003 28

O

OpenOffice 28

S

SMRL 4

X

XML attribute ....

  composite-rel-type 22

  composite-source-ref 22

  composite-source-type 22

  composite-target-ref 22

  composite-target-type 22

  contains **12**

  decimal-format 16

  depends 17

  description-contains 9

  direction 21

  eq 14

  global .... 14p.

  gt 14

  id 17

  is-number 12

  lt 14

  marker-ends-with 15

  marker-starts-with 15

  name-after 16

  name-before 16

  name-by-position 16

  name-postfix 16

  name-prefix 16

  name-ref 17

  namespaces 17

  not-contains 12

  not-equal 12

  number-format 16

  pred-contains 12

  source-ref 21

  source-type 21

  strict-enclosing 23

  succ-contains 12

  target-ref 21

  target-type 21

  title-contains 9, 11

  title-ends-with 11

  title-not-contains 11

  title-starts-with 11

  to-lower-case 16

  to-upper-case 16

  type 9

  visibility 17

  word-list 16

  word-starts-with 15

XML element ....

  col 13

  document 9

  item 12

  row 13

  section 11

  table 13

  text 14