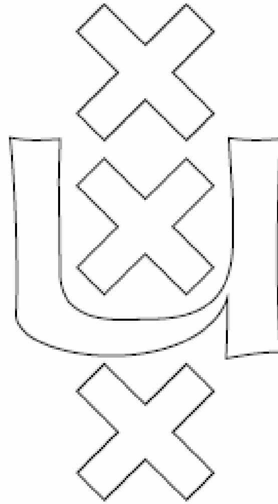


XML document transformation processes using ASF+SDF

G.G. Stap
geraldst@science.uva.nl

August 2007

Supervisor: prof. dr. P. Klint

Acknowledgements

Since I was ten years old I have been playing around with computers primarily as a substitution for Lego. Now, roughly 20 years later, the quest for information continues and the curiosity still grows. I'd like to thank Prof. Dr. Paul Klint for introducing me to the world of grammars, formal languages, Toolbus, and best of all Tcl/TK. I'd also like to thank my mom for allowing me to play around with my computer so much and my girlfriend Karin and my two daughters Esmée and Lotte for sticking by me and showing patience when daddy was grumpy because of yet another deadline. Further more I'd have to thank my managers at department Strategische Informatievoorziening of de Hogeschool van Amsterdam for having some time off so I could dedicate myself to my studies and Jurgen Vinju for the fun we had running our own company. Last but not least I'd like to thank Remco van de Woestijne for sharing his skills as a reverse-engineer and for playing board games and drinking tea with me.

Abstract

This thesis describes the processes involved with transformation of an XML document to other formats using ASF+SDF and the ASF+SDF Meta-Environment. We will discuss several approaches to the transformation process using ASF+SDF and compare these with the commonly used XSLT approach. As an example we will trace the steps involved in transforming an XML document.

Contents

1 Introduction 5
1.1 The ASF+SDF Meta-Environment 5
1.2 SDF 6
1.2.1 SGLR Parsing 7
1.2.2 Grammars 7
1.2.3 Context-Free grammars 8
1.2.4 BNF and EBNF 8
1.3 ASF+SDF 9
1.3.1 Structure of an ASF+SDF module 9
1.3.2 ATerms 10
1.3.3 Term rewriting 10
1.3.4 Traversal functions 11
1.4 XML and SGML 11
1.5 XML Schema Languages 12
1.5.1 XML DTD 12
1.5.2 W3C XML Schema 13
1.5.3 RELAX NG 14
1.5.4 Other schema languages 15
1.6 XML Document Formats 15
1.6.1 DocBook 15
1.7 XSLT, XSL and XPATH 16
1.7.1 XSL and XSLT 16
1.7.2 XPATH 17
1.8 Processes 17
1.8.1 Conventions for modeling processes and activity diagrams 17
2 Transformation of XML documents 20
2.1 Generic Approach 20
2.1.1 Determine the schema 21
2.1.2 Validate the schema 21
2.1.3 Validate XML Document 22
2.1.4 Deduce XML patterns 22
2.1.5 Define transformation rules 23
2.1.6 Transform XML document 23
3 The transformation process of transforming XML documents using ASF+SDF 25
3.1 Dedicated parser approach 25
3.2 Generic XML parser approach 25
4 Transformation process using a dedicated parser 27
4.1.1 Convert Schema Specification 28
4.1.2 Develop ASF+SDF RelaxNGc Schema Validator 32
4.1.3 Validate schema 33
4.1.4 Develop an ASF+SDF XML Parser Generator 34
4.1.5 Generate SDF XML Parser 35
4.1.6 Validate XML document 36
4.1.7 Develop ASF+SDF XML Transformer 36

4.1.8 Transform XML document ... 38
5 Transformation process using a generic XML parser in ASF+SDF ... 39
5.1.1 Convert Schema Specification ... 40
5.1.2 Develop ASF+SDF RelaxNGc Schema Validator ... 40
5.1.3 Validate Schema ... 41
5.1.4 Develop ASF+SDF XML Document Validator ... 41
5.1.5 Validate XML Document ... 42
5.1.6 Develop ASF+SDF XML Transformer ... 42
5.1.7 Transform XML document ... 43
6 Transformation process using XSLT ... 44
6.1.1 Validation ... 45
6.1.2 Create XSL Stylesheet ... 48
6.1.3 Develop XSL Templates ... 49
6.1.4 Link XML Document to XSL Stylesheet ... 51
6.1.5 Transform XML Document ... 52
7 Process Comparison ... 54
7.1 Process Comparison Table ... 54
7.2 Process Comparison ... 55
8 Implementation ... 56
8.1 Tooling ... 56
8.2 Implementation issues ... 57
9 Conclusions ... 59
10 Topics for further study ... 60
10.1 Improvements for the SDF2 SGLR parser ... 60
10.2 Improvements for ASF+SDF ... 60
10.2.1 Extending ASF+SDF with an include mechanism ... 60
10.2.2 Extending ASF+SDF with URI support ... 60
10.3 Improvements for the ASF+SDF Meta-Environment ... 61
10.3.1 Displaying Partial Parse Trees ... 61
10.4 Generating ASF+SDF XML parse tree transformer ... 61
11 Bibliography ... 62

1 Introduction

XML is a widely used SGML based markup language. Many formal languages and data specifications are described in XML. The tools to transform XML documents are dominated by technologies like XSLT, XPATH and XQUERY. But what if we were to transform XML documents with a term rewrite system? Would this have significant advantages over the use of XSLT? This thesis is about transformation processes of XML documents with ASF+SDF using the ASF+SDF Meta-Environment.

The main questions we will try to answer is:

- *How can XML documents be transformed using ASF+SDF and the Meta-Environment? What does the transformation process look like?*
- *What are the advantages and disadvantages of using the ASF+SDF transformation processes compared to the process of using XSL with an XSLT engine?*

To answer these questions we will describe and compare two different ASF+SDF transformation processes and a common XSLT transformation process for transforming XML documents.

Typical readers of this paper should have basic knowledge of parsers, parse techniques and familiarity of basic XML and XSLT concepts.

The next sections of the introduction will explain in brief the building blocks for our processes.

1.1 The ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment is an Interactive Development Environment (IDE) for development and testing of SDF and ASF+SDF specifications as described in **[META05].** The IDE supports functionality for visualizing SDF parse trees, a syntax directed editor for writing SDF and ASF+SDF specifications, an editor for editing, selecting, displaying en pretty printing terms. A navigation window shows you the currently loaded ASF+SDF modules. An import graph displays the dependencies between the modules.

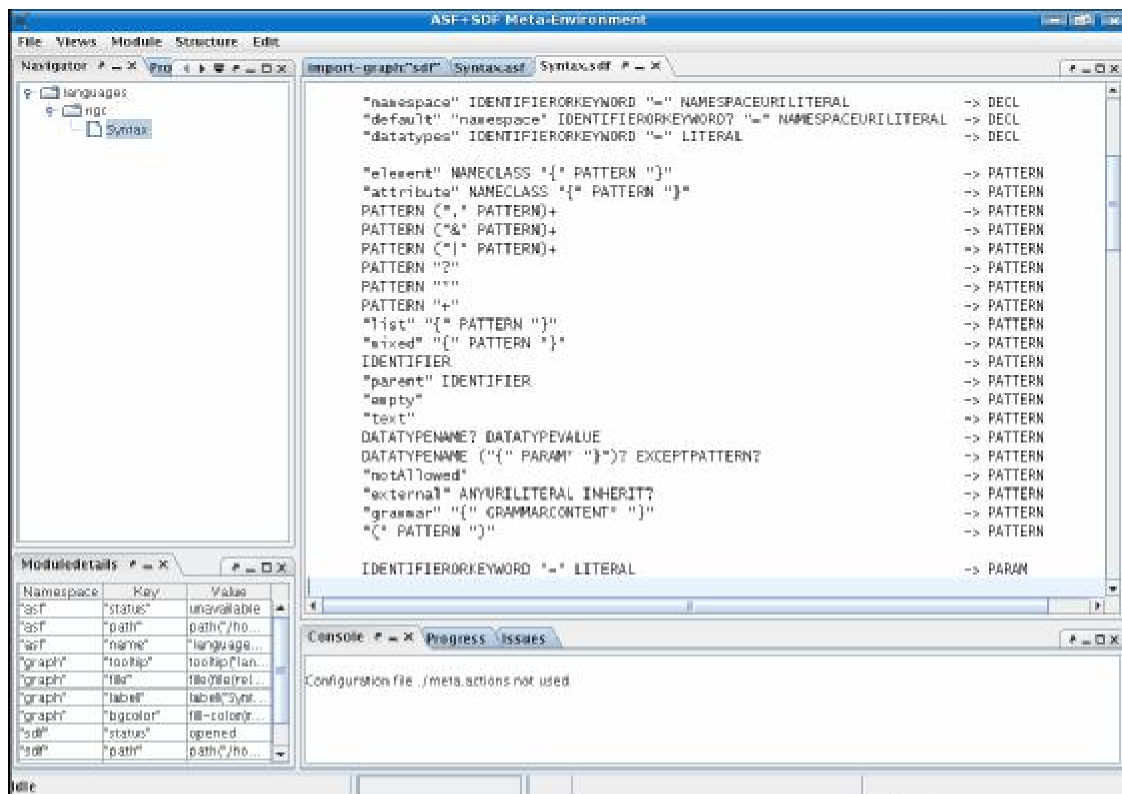
Note that the classic abbreviation IDE stands for Integrated Development Environment, not Interactive Development Environment. The ASF+SDF Interactive Development Environment differs from the classic Integrated Development environment in that it has interactive support for writing formal specifications, offers incremental compilation and testing of specifications, compilation of ASF+SDF specifications into dedicated interactive stand-alone environments and user defined extensions of the default user interface **[COM01]**.

The Meta-Environment is extendible. Its extension points are documented in **[EXT07].** It is possible to extend the Meta-Environment with user-defined menu options, or extend

the Meta-Environment with language independent or language specific tools. Types of extensions include new extensions, configuration scripts, Toolbus scripts, any new tools that provide their own toolbus adapter, MetaStudio plugins written in Java and TIDE adapters.

[ARCH06] gives a detailed overview of the architecture of the Meta-Environment. This document describes the identified stakeholders and views for the ASF+SDF Meta-Environment. Stakeholders are categorized in end-users, power users and system developers. The defined views are the end-user view, the decomposition view, the coordination view, the data layer view, the package view, the package dependency view and the power user view. Each view's element catalog, context diagram, variability guide and architectural background is explained.

Figure 1.1: Screenshot of the ASF+SDF Meta-Environment.



1.2 SDF

The Syntax Definition Formalism (SDF) is a notation for describing the high-level description of grammars for programming languages, application languages, domain-specific languages, data formats and other computer-based formal languages. Its primary purpose is the definition of syntax. SDF's focus is on the formal specification of context-

free grammars.**[SDFXX]** describes the syntax of SDF and its basic semantics in the form of a reference manual. It is distributed with the daily build of the SDF software.

SDF is implemented as a combination of an SLR(1) parse table generator and a scannerless generalized LR parser. SDF main features are that is has no separate scanner thus avoiding lexical ambiguity, it accepts all context-free syntaxes including ambiguous syntaxes, it generates all ambiguous derivations thus avoiding implicit choices. SDF constructs parse trees automatically.

1.2.1 SGLR Parsing

A parser is a tool that takes a string that represents a program as input and outputs a parse tree, which represents the same program but in a more structured form. An SGLR parser is a scannerless generalized LR parser, an implementation of Tomita's Generalized LR algorithm with extensions for scannerless parsing. An LR parser is a parser for context-free grammars that reads input from Left to right and produces a rightmost derivation.

As stated in **[SDF00]** an SGLR parser does not need a separate lexical scanning tool to tokenize the input stream before parsing. Therefore lexical and context-free parsing are handled by a single tool.

In generalized parsing, ambiguities are allowed in the grammar. The parsing process can produce several equally correct results. The SGLR parser interprets parse tables generated by the parse table generator from an SDF2 syntax definition and outputs parse trees or parse forests (multiple parse trees) if the parsed input is ambiguous. In generalized parsing non-ambiguous subclasses of the context-free grammars are allowed. An important property of generalized parsing is that it allows for modularity. Grammars from the ambiguous grammar class can be merged.

1.2.2 Grammars

Chomsky defined four families of grammars which together form the Chomsky hierarchy. The four families are:

Grammar	Languages
Type 0	Unrestricted
Type 1	Context-sensitive
Type 2	Context-free
Type 3	Regular

Type-0 grammars include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine.

Type-1 grammars generate the context-sensitive languages. The languages described by these grammars are exactly all languages that can be recognized by linear bounded automata.

Type-2 grammars generate the context-free languages. The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic pushdown automaton

Type-3 grammars generate the regular languages. These languages are exactly all languages that can be decided by deterministic and nondeterministic finite state automaton.

In this paper we will focus on Type-2 grammars.

1.2.3 Context-Free grammars

In **[LANG88]** a language is defined as a set of strings over an alphabet. Syntactically correct strings in a computer language are called programs. The alphabet of the language consists of indecomposable elements from which the strings can be constructed. We call these elements terminal symbols or tokens. Intermediate symbols, variables and non-terminals are used to enforce syntactic restrictions on the language. A context free grammar is a formal system used to generate the strings of a language.

[LANG88] defines a context-free grammar as a quadruple (V, Σ, P, S) where V is a finite set of variables, Σ (the alphabet) is a finite set of terminal symbols, S is a distinguished element of V called the start symbol, and P is a finite set of rules. The sets V and Σ are assumed to be disjoint. When $v \in (V \cup \Sigma)^*$ the set of strings derivable from v can be defined recursively.

1.2.4 BNF and EBNF

There are several meta-syntaxes available to describe context-free grammars. The most common are BNF and EBNF. BNF stands for Backus-Naur Form. John Backus and Peter Naur used a system of rules to define the programming language ALGOL 60. EBNF is the extended version of BNF and defines extra constructs for optional and repetitive symbols.

A BNF specification is a set of derivation rules. Each rule is written as:

```
<token> ::= <expression>
```

The symbol <token> is a non-terminal symbol. The expression may contain tokens or a sequence of tokens or sequences separated by the | operator indicating choice. The right hand side of the derivation rule is a possible substitution for the symbol on the left.

Any grammar represented in EBNF can also be represented in BNF. But this will probably result in a much larger specification.

1.3 ASF+SDF

SDF can be used in conjunction with ASF, the Algebraic Specification Formalism. The combination is usually referred to as ASF+SDF. ASF+SDF is a modular specification formalism for the integrated definition of a (programming) language.

1.3.1 Structure of an ASF+SDF module

An ASF+SDF specification has the structure as described in **[META05]** Modules and Modular Structure. Because we will be using ASF+SDF extensively we will summarize the building blocks of an ASF+SDF module.

The structure of a module has the form:

```
module <ModuleName>

  <ImportSection>*

  <ExportOrHidddenSection>*

equations
   <CondtionalEquation>*
```

The import sections contain references to other modules that have to be imported. Imported modules contain grammar elements that are needed within the current module. . The export section is used to specify the grammar elements that are available for use in other modules. Grammar elements specified in the hidden section are only visible to the current module. A sort is specified in a production rule and is declared by listing its name in the sorts section. Production rules are specified in context-free or lexical syntax sections. The context-free start-symbols section contains the sorts that can serve as a start-symbol when parsing terms. The aliases section may be used to specify an alias for complicated symbols that occur repeatedly in the specification. Variables are declared in the variables section of a module. Declared variables can be used in equations.

The equations section contains specifications of conditional or unconditional equations. With equations semantics may be added to functions declared in the lexical or context-free syntax sections. Equations can be executed as rewrite rules to reduce some initial closed term to its normal form. A default equation can be specified for cases where no other rewrite rule matches.

1.3.2 ATerms

ATerms are Annotated Terms. An annotated term is the data format used for the internal representation of all data in ASF+SDF. The ATerm data type is a powerfull mechanism for representing parse trees and abstract syntax trees. It is used in the ASF+SDF dataformat *AsFix* as a data format to represent parse trees.

ATerms are language independent and can be processed by programs in any language. They can be annotated with auxilary information that does not affect the tree structure. ATerms preserve maximal subterm sharing meaning that common parts of the data are not duplicated but shared. This leads to size reduction of the data.

[ATRM00] describes the abstract data type of ATerms and discusses their design, implementation and application.

1.3.3 Term rewriting

[TERM06] explains the principles of term rewriting. Term rewriting is computational paradigm that is based on the repeated application of simplification rules and is particularly suited for tasks like symbolic computation, program analysis and program transformation.

Three basic concepts are essential to term rewriting: terms, substitution and matching.

Terms are defined in a strict prefix format.

- A single variable is a term.
- A function is a term

A substitution is an association between variables and terms. Substitution can be used to create new terms from old ones by replacing variables according to a mapping.

Matching is the process of determining whether a substitution exists that can make two different terms identical.

The term rewriting algorithm uses a left-most innermost reduction strategy for traversing the term. Given a set of rewrite rules and an initial term T, the rewriting algorithm is applied and will yield a simplified (or normalized) term T' as answer.

Term rewriting has been extended with user-defined syntax, conditional rules, default rules, lists and list matching and traversal functions.

1.3.4 Traversal functions

ASF+SDF supports the use of traversal functions as described in **[TRAV04]**. The ASF+SDF traversal functions are used for automating recursion. They traverse the tree recursively and allow the accumulation of values or transformation of the tree. ASF+SDF allows for three kinds of traversal:

1. The accumulator: traversal(accu)
2. The transformer: traversal(trafo)
3. Accumulating transformer: traversal(accu,trafo).

The order of the traversal can be specified as being either top-down or bottom up. With top-down traversal each of the nodes of the tree is visited recursively in parent/child order from the root of the tree. With bottom-up traversal the traversal is started at the leaves of the tree. We can also specify if the traversal function has to break or continue after visiting a node.

Traversal functions are best used when the tree has to be traversed recursively and only performs actions on a few nodes.

1.4 XML and SGML

In the Extensible Markup Language (XML) specification **[XML06]** XML is described as a subset of SGML. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

Standard Generalized Markup Language (SGML) is a meta-language for defining markup languages. Authors mark up their documents by representing structural, presentational, and semantic information alongside content.

The XML language can be extended allowing users to define their own tags. Many application languages and data formats are implemented using XML. Examples are IMS, XHTML, RSS, MathML.

The building blocks of a XML documents are elements, attributes and entities. Elements have either the form:

```
<name attribute="value">content</name>
```

or

```
<name attribute="value"/>
```

Nesting is allowed thus content can contain other nested elements.

Entities have the form `&entityname;` and are mostly used for defining special characters like `€` for the euro symbol.

Figure 1.4: XML Example of an address book

```
001 <addressBook>
002   <card>
003     <name>John Smith</name>
004     <email>js@domain.nl</email>
005   </card>
006   <card>
007     <name>Joe Jones</name>
008     <email>jj@domain.nl</email>
009   </card>
010 </addressBook>
```

1.5 XML Schema Languages

A schema defines a class of XML documents. An XML schema is a set of rules which defines the validity of an XML parse tree. An XML validator can verify the validity of an XML document by parsing the document and comparing the resulting parse tree against the schema.

There are a lot of XML schema languages available. The following paragraphs will summarize several popular schema languages which are supported by major XML authoring tools.

1.5.1 XML DTD

The Document Type Definition (DTD) is one of the first XML schema languages and was inherited from SGML. It is also referred to as a DOCTYPE. Popular formats like XHTML, WML are described with an XML DTD. Although using DTD has its disadvantages it is still widely used because it is part of the XML 1.0 specification **[XML06]**.

Disadvantages of using DTDs are that they have no support for the namespaces features of XML, can not express certain formal aspects of an XML document and that the DTD itself is specified using a non-XML syntax.

XML documents containing references to an external DTD do so in the document type declaration in the first line of the XML document.

Figure 1.5.1: XML DTD Example for addressBook

```
001 <!DOCTYPE addressBook [
002 <!ELEMENT addressBook (card*)>
003 <!ELEMENT card (name, email)>
004 <!ELEMENT name (#PCDATA)>
005 <!ELEMENT email (#PCDATA)>
006 ]>
```

1.5.2 W3C XML Schema

XML Schema is the W3C XML Schema Language. It is often referred to as XSD. The W3C XML schema specification is maintained by the World Wide Web Consortium. Its specification version 1.0 is available in parts. **[XM0S04]** contains the primer, a non-normative document intended to provide an easily readable description of the XML Schema facilities. **[XMS1S04]** describes the W3C XML Schema structures. **[XMSD04]** describes the W3C XML Schema data types. For XML Schema version 1.1 a working draft is available.

W3C Schema basic concepts consist of complex type definitions, element and attribute declarations, simple types, anonymous type definitions, element content, annotations, content models, attribute groups and nil values.

Advanced concepts comprise namespaces, schemas and qualification, target namespaces and unqualified locals, qualified locals, global versus local declarations and undeclared target namespaces.

Figure 1.5.2: W3C XML Schema Example

```
001 <?xml version="1.0"?>
002 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
003 targetNamespace="http://www.example.com"
004 xmlns="http://www.example.com"
005 elementFormDefault="qualified">
006
007   <xs:element name="note">
008       <xs:complexType>
009         <xs:sequence>
010               <xs:element name="to" type="xs:string"/>
011               <xs:element name="from" type="xs:string"/>
012               <xs:element name="heading" type="xs:string"/>
013               <xs:element name="body" type="xs:string"/>
014         </xs:sequence>
015       </xs:complexType>
016   </xs:element>
017 </xs:schema>
```

1.5.3 RELAX NG

RELAX NG is a schema language for XML. Its specification is maintained by the Organization for the Advancement of Structured Information Standards (OASIS). The name RELAX NG is pronounced as relaxing. The key features of RELAX NG are that it is simple, easy to learn, has both a XML syntax and a compact non-XML syntax and does not change the information set of an XML document. On the contrary to XML DTD it does have support for XML namespaces. Other features of RELAX NG are that it treats attributes uniformly with elements as far as possible and has unrestricted support for mixed content. RELAX NG is derived from schema languages Relax Core and TREX and has a solid theoretical basis. RELAX NG can use external W3C data types as described in **[XMSD01]** and **[XMSD04].**

The RELAX NG schema comes in two flavors: The XML syntax as specified in **[RNG01]** which we will refer to as RelaxNG XML syntax (RelaxNG) and the compact non-XML syntax as specified in **[RNC02]** which we will refer to as RelaxNG Compact Syntax (RelaxNGc) throughout this document.

RELAX NG contains several constructs for defining XML patterns like choice, attributes, named patterns, data typing, enumerations, lists and interleaving. RELAX NG supports modularity by referencing external patterns, combining definitions, merging grammars and replacing definitions. It also supports name classes, internationalization, annotations, and nested grammars.

Figure 1.5.3-1: RELAX NG XML syntax schema example

```
001 <element name="addressBook"
002 xmlns="http://relaxng.org/ns/structure/1.0">
003   <zeroOrMore>
004     <element name="card">
005       <element name="name">
006         <text/>
007       </element>
008       <element name="email">
009         <text/>
010       </element>
011     </element>
012   </zeroOrMore>
013 </element>
```

Figure 1.5.3-2: RELAX NG compact syntax schema example

```
001 element addressBook {
002   element card {
003     element name { text },
004     element email { text }
005   }*
006 }
```

1.5.4 Other schema languages

Except for the XML DTD, W3C Schema and RelaxNG there are many more schema languages. To name but a few DSD, DSDL, Schematron, XDR, SOX, TREX, DDML. However, covering these schema languages is beyond the scope of this paper.

1.6 XML Document Formats

1.6.1 DocBook

DocBook is an XML document format with a schema available in several languages including RelaxNG, RelaxNG Compact Syntax, SGML, XML DTD and W3C XML Schema. The standard is maintained by the DocBook Technical Committee of OASIS. DocBook was designed for describing books and papers about computer hardware and software.

DocBook is a large and robust schema containing main structures that correspond to the notion of what constitutes a book. Its structure contains elements like titles, chapters, sections, paragraphs and so forth. Many commercial tools support DocBook out of the box.

Figure 1.6.1: Example of a typical DocBook structure

```
001 <!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V3.1//EN">
002 <book>
003 <bookinfo>
004   <title>My First Book</title>
005   <author><firstname>Jane</firstname><surname>Doe</surname></author>
006   <copyright><year>1998</year><holder>Jane Doe</holder></copyright>
007 </bookinfo>
008 <preface><title>Foreword</title> ... </preface>
009 <chapter> ... </chapter>
010 <chapter> ... </chapter>
011 <chapter> ... </chapter>
012 <appendix> ... </appendix>
013 <appendix> ... </appendix>
014 <index> ... </index>
015 </book>
```

In this paper we will work towards parsing DocBook documents as input for transformations. Because of its popularity many organizations are considering DocBook as a standard for their documentation and help files.

1.7 XSLT, XSL and XPATH

1.7.1 XSL and XSLT

XSLT is al language for transforming XML documents into other XML documents. XSLT stands for XSL Transformations. Its specification version 1.0 **[XSLT99]** is maintained by the World Wide Web Consortium. XSLT is designed for use as a part of XSL **[XSL06]**. There is also a version 2.0 of the XSLT specification **[XSLT07]** which is a revised version of the 1.0 specification. XSLT 2.0 is used in conjunction with XPATH version 2.0 **[XPAT07]**.

XSLT uses an XSLT processor for its transformations. An XSLT processor is the software responsible for transforming source trees into result trees using an XSL stylesheet.

A transformation expressed in XSLT describes rules for transforming a source tree into a result tree by associating patterns with templates. A pattern is matched against elements in the source tree. A stylesheet is a transformation expressed in XSLT.

A stylesheet contains a set of template rules. A template rule has two parts: a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

A template can also contain elements from the XSLT namespace that are instructions for creating result tree fragments. When a template is instantiated, each instruction is executed and replaced by the result tree fragment that it creates

When processing the XSL stylesheet a list of source nodes is processed to create a result tree fragment. The result tree is constructed by processing a list containing just the root node. A list of source nodes is processed by appending the result tree structure created by processing each of the members of the list in order. A node is processed by finding all the template rules with patterns that match the node, and choosing the best amongst them; the chosen rule's template is then instantiated with the node as the current node and with the list of source nodes as the current node list. A template typically contains instructions that select an additional list of source nodes for processing. The process of matching, instantiation and selection is continued recursively until no new source nodes are selected for processing.

With XSLT it is possible to combine multiple stylesheets by importing, including or embedding other stylesheets. The output of an XSL transformation can be XML, HTML or text.

1.7.2 XPATH

XPATH **[XPAT99]** is a language for addressing parts of an XML document, designed to be used by XSLT.

XSLT uses the expression language defined by XPATH. Expressions are used in XSLT for selecting nodes for processing, specifying conditions for different ways of processing a node and generating text to be inserted in the result tree.

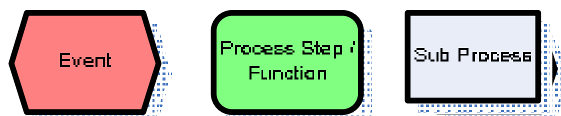
1.8 Processes

1.8.1 Conventions for modeling processes and activity diagrams

Throughout this paper we will frequently use diagrams to indicate process flow. The technique used for describing a process is based on Even Driven Process Chains (EPC) **[EPC06]** and Activity Diagrams. EPCs are widely used by the Business Process Management community. In many cases they are used to model data flow of complex information systems.

Event driven process chains consist of the symbols as displayed in figures 1.8.1-1 and 1.8.1-2.

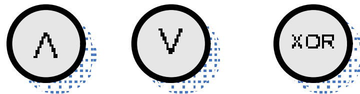
Figure 1.8.1-1: EPC Symbols



Events are used to trigger a process, function (process step) or a sub-process. All EPC process diagrams start with an event and end with an event. In sequential process flow events are alternated by functions. Before and after every function there must be an event in the flow. However, in many cases this constraint leads to the addition of useless or dummy events with names like *before X* or *after Y*. We will omit such dummy events using an informal notation of EPC allowing sequences of functions without events in between functions. A process or part of a process can be substituted by the sub-process symbol allowing abstraction of process details and viewing the process on a higher abstraction level. It also helps in maintaining readability. Many of our processes will be modeled as sequential processes without branches even though every function can fail or succeed. Again this is to allow for readability. Branching and feedback loops can easily be modeled using operators but they make our process diagrams more difficult to read. If

possible we will omit branches and assume that a function will always succeed unless it is import for our process to show otherwise.

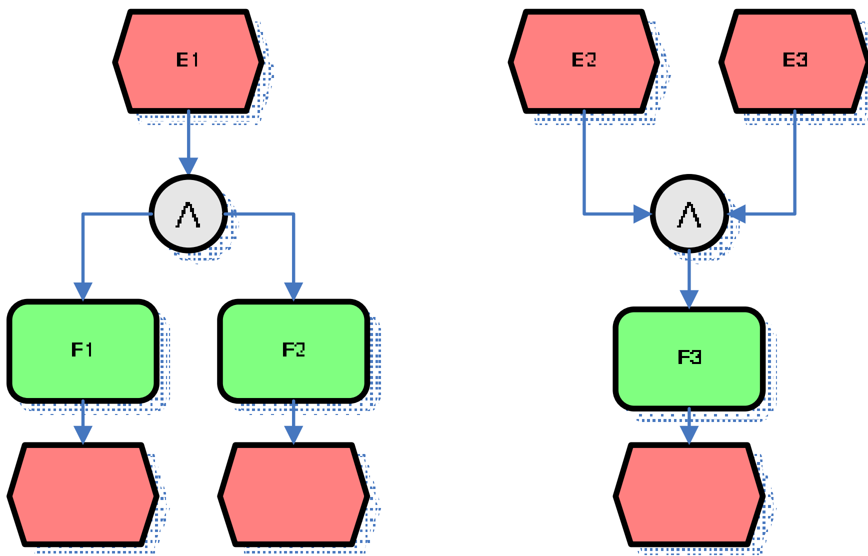
Figure 1.8.1-2: Operators, AND, OR, XOR



EPC Diagrams are very useful for visualizing parallelism in processes. This can be accomplished using operators.

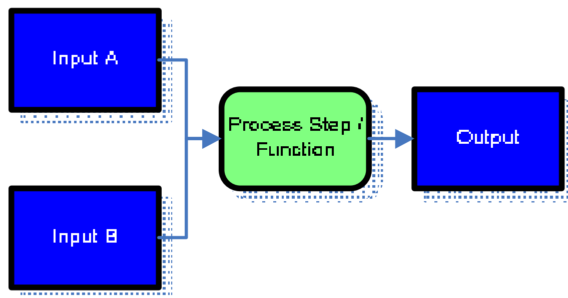
In figure 1.8.1-3 two processes are depicted. When event E1 occurs process paths containing F1 and F2 are executed in parallel. When event E2 and E3 occur simultaneously function F3 is executed. Note that EPC operators can either have one input and multiple outputs or one output and multiple inputs as shown in the example.

Figure 1.8.1-3: Example of visualizing parallelism using the AND operator



Activity Diagrams are used to model the input and output of a single function. The inputs are modeled on the left hand side of the function while the output is modeled on the right hand side. Note that we have chosen to allow only a single output on the right hand side of any function in an activity diagram.

Figure 1.8.1-4.: Activity diagram example



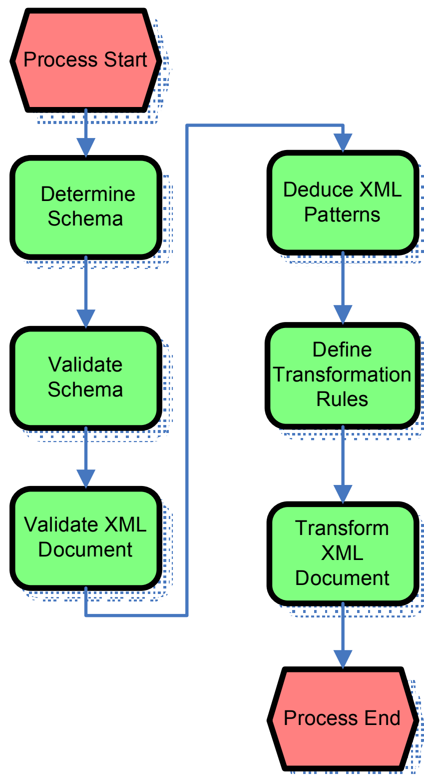
A function can either be executed manually or by a supporting system (like an SDF parser). What method of execution is used will become clear in the explaining text for each function. A function or process step can consist of a number of activities to transform the inputs to the desired output.

2 Transformation of XML documents

2.1 Generic Approach

The process of transforming XML documents generally follows the process steps given in the EPC diagram in figure 2.1-1.

Figure 2.1-1 Process – Generic Approach to transformation of XML documents.



Now let's look at these process steps in more detail by regarding the process step's inputs and output.

2.1.1 Determine the schema

Before we even consider transformation of an XML document the first question we have to ask ourselves is if the XML document in question is a valid XML document? It could be that the XML document has a structure that is not suitable for our intended transformations. Therefore we first have to establish that this XML document indeed has a valid structure. Having the right structure means that the document is part of a certain document class. A document class is defined by a schema. So the first thing to do is to determine the schema belonging to this XML document.

Figure 2.1.1: Activity Diagram - Determine Schema

Given an XML Document as input this process step determines the schema belonging to the XML document. The output is the known schema for the XML document.

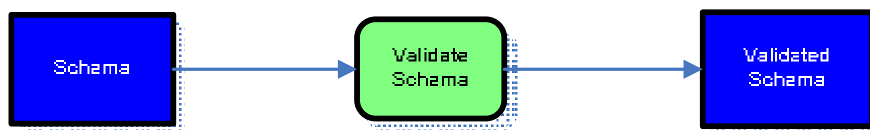
Note that XML documents can contain references to a schema. In these cases determining the schema can be automated. If there is no schema reference a schema must be provided or in some cases derived form the XML document.

Input: XML Document
Output: Known Schema

2.1.2 Validate the schema

In cases where we know the schema or are able to retrieve the schema we can validate the schema itself. Schemas are defined in a schema language. A validator for that language can tell us if the schema is syntactically correct. In other cases validation can go even further using additional validation rules as constraints on the schema. In some cases the language used for defining the schema is an XML language.

Figure 2.1.2: Activity Diagram - Validate Schema



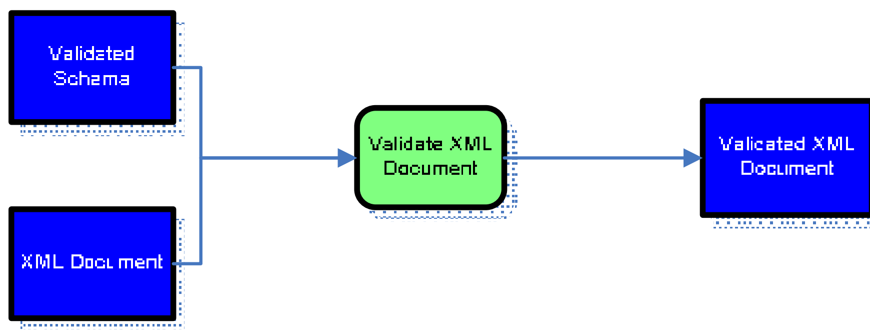
Given a schema as input this process step validates the schema using a validation system and a set of validation rules built into this system. The output is a validated schema that adheres to the validation rules.

Input: Schema
Output: Validated schema

2.1.3 Validate XML Document

Now that the schema is validated the XML document can be validated using the schema. For this we need a validating XML parser. A validating XML parser has the ability to interpret the schema, deduce patterns form the schema, and determine if the patterns in the XML document match the deduced patterns.

Figure 2.1.3: Activity Diagram - Validate XML Document



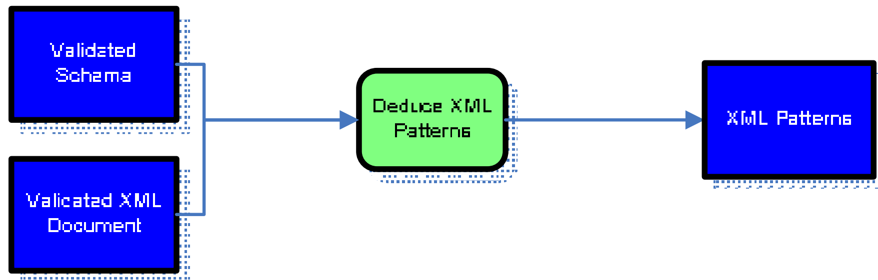
Given an XML document and an associated validated schema as input this process step validates the XML document. The output is a validated XML document meaning that the XML document is either found to be valid or invalid.

Input: Validated schema
Input: XML document
Output: Validated XML document

2.1.4 Deduce XML patterns

We know by validation that the XML document consists of the patterns described by the schema. So if we interpret the schema correctly we can deduce what XML patterns to expect in the XML file. This can help us when define our transformations. We need to know where to transform from. Another approach is to inspect the XML file and see what patterns are in the file.

Figure 2.1.4: Activity Diagram - Deduce XML Patterns



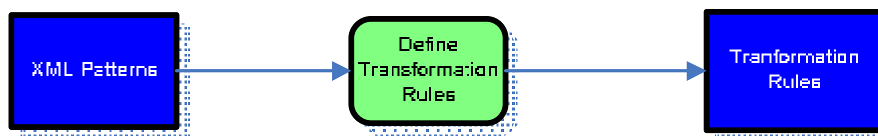
Given a validated schema or a validated XML document as input this process step deduces XML patterns. The outputs of this process step are the available XML patterns for transformation.

Input: Validated schema
Input: Validated XML document
Output: Available XML patterns for transformation

2.1.5 Define transformation rules

Now that we have information about the XML patterns in the XML document we are able to define transformations on those patterns. For this transformation rules have to be defined that can be subsequently interpreted and executed.

Figure 2.1.5: Activity Diagram – Define Transformation Rules



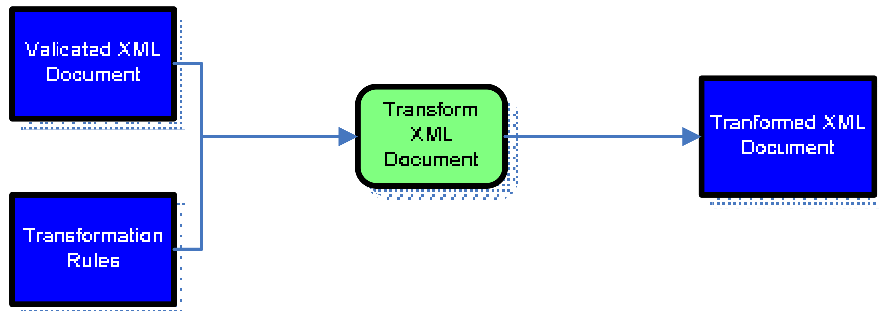
Given a known set of XML patterns as input this process step defines transformation rules on a set or subset of patterns. The output is a set of transformation rules that can be interpreted and executed by a system.

Input: XML patterns
Output: Transformation rules

2.1.6 Transform XML document

When we have defined our transformation with transformation rules and the XML document has been validated we are finally ready for the actual transformation of the XML document.

Figure 2.1.6: Activity Diagram – Transform XML Document



Given a valid validated XML document and a set of transformation rules as input, a transformation system can transform an XML document. The output is the transformed XML document. The output does not have to be an XML document. It can be any structured document. It is possible to transform to non XML document formats.

Note that it is possible to omit process steps involving validation. However, doing so could result in transformations failing when unexpected patterns are encountered.

In the next chapter we will try to map the generic process to specific implementations in ASF+SDF.

3 The transformation process of transforming XML documents using ASF+SDF

In the previous chapter a generic process for transforming XML documents was described. With knowledge of these specific process steps we can explore different approaches for transformation of XML documents with ASF+SDF. Next we can compare how these ASF+SDF transformation processes differ from the generic approach.

3.1 Dedicated parser approach

Assuming a generic XML parser for ASF+SDF isn't available and given the fact that an XML schema defines an XML document class, could a dedicated XML parser for the document class be generated from the schema? After all, the schema defines valid patterns within the XML document. So in theory it should be possible to use these patterns to generate a parser for a single XML document class.

Constraints on the schema in question would be that it is defined in a language that is not an XML language. Without this constraint we would still need an XML-parser to parse the schema. Another constraint would be that the parser or validator for this schema could be implemented in ASF+SDF which would be easiest when a formal description of the grammar of the schema language were available in e.g. BNF or EBNF.

We will refer to this concept as the dedicated parser approach since this parser is not a generic XML parser but restricted or dedicated to a single XML document class. The dedicated parser can be used as a basis for an XML-document validator or an XML-document transformer in ASF+SDF.

3.2 Generic XML parser approach

Assuming that a generic XML parser for ASF+SDF is available then this parser can be used for parsing the XML document. The result is a parse tree. Validation of the schema can be accomplished using a dedicated parser and derived validator for the schema language in question. If the schema language is an XML language we could also extend the generic XML parser for validation of the schema.

Validation of the XML document can be implemented by comparing the parse tree structure with the patterns in the schema. Once the parse tree is declared valid transformations can be performed. Nodes in the tree can be substituted using ASF+SDF traversal functions or the specification can be rewritten using term rewrite rules or a combination of both. We will refer to this concept as the generic XML parser approach.

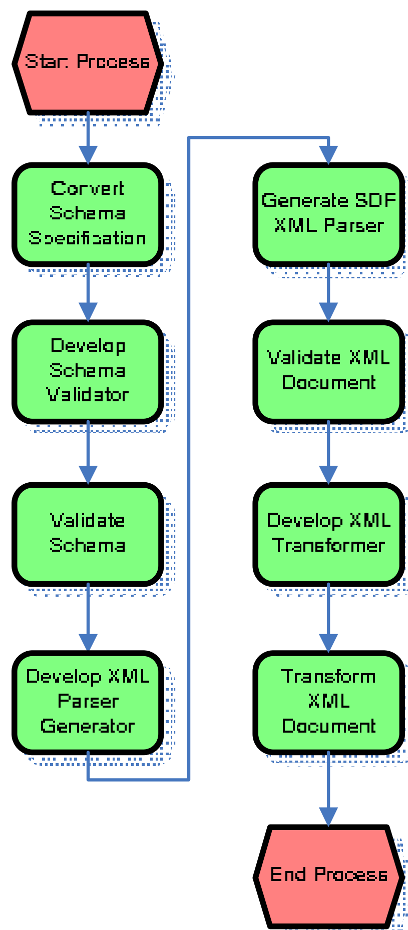
In the next section we will look more closely at the processes involved with implementation of the dedicated and generic parser concepts as they form the basis for our transformation.

4 Transformation process using a dedicated parser

As previously stated the dedicated parser concept involves generating an XML parser for an XML document class given a schema in a language for which a grammar is available. In this section we will step through the implementation process using an XML document defined by a RelaxNG compact syntax schema.

In general the process will look as described in figure 4.1.

Figure 4.1: Process – Dedicated Parser Approach

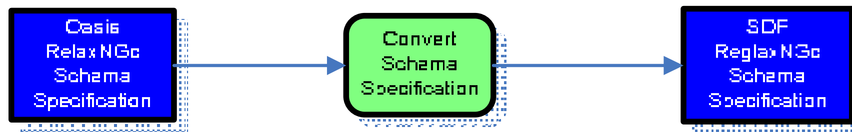


Suppose we want to convert an XML document using this process. What would an implementation look like?

4.1.1 Convert Schema Specification

In order to validate the RelaxNG compact syntax schema we need to implement a RelaxNGc validator. The first step in the process consists of converting the specification of the RelaxNG compact syntax schema language to an SDF specification which we can use as the basis for our validator.

Figure 4.1.1-1: Activity Diagram – Convert Schema Specification



Input: OASIS RelaxNGc Schema Specification
Output: SDF RelaxNGc Schema Specification

The RelaxNG compact syntax specification is available in either a non-formal EBNF syntax or a formal BNF syntax. The EBNF version is more human readable but is only used for gaining insight in the specification. It can not be used for conversion to SDF because it lacks certain details like handling comments and annotations. Also several restrictions regarding operator precedence were left out. Converting the EBNF specification would only allow us to parse a small subset of the RelaxNG schema documents. If we want a shot at a working SDF specification we will have to use the BNF specification as input for conversion. Conversion is done manually using procedure 4.1.1-2.

Procedure 4.1.1-2: Conversion procedure for RelaxNGc BNF to SDF

1. For every rule in the specification convert the lowercase Terminal and Non-Terminal token names to an SDF sort name with the first letter being uppercase.

2. Add the context-free start symbol to the SDF file under section context-free start-symbols

```
context-free start-symbols
  TopLevel
```

3. Transform all rules from the BNF grammar in the form

```
LHS ::= RHS
```

to

```
RHS -> LHS
```

thereby omitting return values, the returns statement, arguments and constraints. LHS is the left hand side of the rule. RHS the right hand side.

An example:

topLevel **returns** Element ::=
  preamble$_e$ *topLevelBody*(**environment** := *e*)$_x$
    { *x* }

becomes the SDF production rule:

```
PreAmble TopLevelBody -> TopLevel
```

If the right hand side of an BNF rule contains vertical bars "|" we add multiple rules to the SDF specification.

for

```
token1 ::= token2 | token3 | token4
```

we add the SDF production rules:

```
Token2 -> Token1
Token3 -> Token1
Token4 -> Token1
```

4. For every token on RHS of an SDF production rule add a sort to the sorts section;

```
sorts TopLevel
```

5. Determine if production rules are part of context-free or lexical syntax and move them to the context-free or lexical syntax sections accordingly.

The difference between context-free and lexical syntax sections in SDF is that context-free grammar assumes that there can be optional LAYOUT, like whitespace, carriage return characters etc. between tokens. Also character classes can only be used in the lexical syntax section.

6. Rewrite rules for lexical-syntax containing terms of the form:

```
[characterclass] – [characterclass]
```

to

```
[characterclas] :/ [characterclass]
```

7. Rewrite rules of the form

```
tokenC ::= tokenA – tokenB
```

SDF has got no operators to simulate such rules. Thus we split the rule into two separate rules. Next we tell the SDF parser to reject a part of the parse tree generated by a rule by adding {reject} to the rule.

```
SortA -> SortC
SortB -> SortC {reject}
```

8. Identify external sorts used in the grammar which are defined in or as part of other grammars.

We can either add these sorts to the current SDF specification or to their own SDF file. SDF supports modularity in the form of SDF modules. If we put the external specification in its own module we can import it if the sorts in that module are made available through the exports section.

The RelaxNGc specification contains references to parts of other specifications. After conversion the token *NCNames* will be unresolved. There is no production rule for *NCName* but the token is used in the specification. According to **[RNG01]** NCName is described in **[XMLN06]** the XML 1.0. namespaces specification. This specification uses references to **[XML06]**, the XML 1.0 W3C recommendation. Both **[XMLN06]** and **[XML06]** specifications are not available in BNF but in EBNF. Conversion of EBNF to SDF is somewhat similar to procedure 4.1.1-2. but differs on several points.

In EBNF the syntax [Symbol], [Symbol]* and [Symbol]+ are used for optional and repeating symbols.

In SDF an optional part in the syntax rule is described by the postfix option operator ?.

```
Symbol?
```

The repetition operator * repeats a symbol at least zero times.

```
Symbol*
```

The repetition operator + repeats a symbol at least one time:

```
Symbol+
```

4.1.1.1 Conversion Issues

During conversion of BNF or EBNF to SDF several problems can arise that need to be handled. This section will discuss in brief the findings during conversion.

First of all, the RelaxNG compact specification supports an include mechanism. This include mechanism allows for inclusion of other RelaxNGc code snippets. SDF however hasn't got a preprocessing function to expand a file using a include pattern. If this were the case one could use a single RelaxNGc document containing includes as input, scan for include statements and replace the include statement with the contents of the include file. Since SDF lacks this functionality the user has to be aware of any include statements residing within the input document. The included files have to be syntax checked

manually. The include file itself can also contain include statements, so we have a recursive problem. Not only do the input files have to be scanned for include statements, also the included files themselves need to be scanned.

Second, RelaxNGc supports several character set encodings. SDF however supports only the Latin-1 character set. This poses a problem when parsing documents containing e.g. Chinese, Korean or Japanese or other special characters. For our implementation the character recognition in the specification had to be altered because it supports Unicode which contains a far broader range of characters encoded in two bytes.

Third, the RelaxNGc specification supports the use of W3C data types. The include problem becomes even worse if we want to parse documents on the fly, so without first having to gather dependent documents. Inclusion of W3C data types generally doesn't mean including a local file (e.g. from disk) but from an URI any resource on the internet. This doesn't pose a problem during parsing but it does during validation. The URI could be a web location in which case we would want ASF+SDF to do a HTTP GET before including the file.

Fourth, W3C data types are not defined in RelaxNGc but in XML. This means that although we can implement a RelaxNG schema validator in SDF by converting the BNF grammar to SDF, we still need an XML-parser or a dedicated parser which parses the XML data types. This means that the user should have familiarity with XML-schema languages and the available XML data types. Further more this induces a readability problem because the schema documents contain two types of notations. Notations used like in programming languages like C, using accolades for statement blocks, and XML-language like notations using opening and closing tags and attributes.

4.1.1.2 Fixing ambiguities

When our specification is converted to SDF we are able to test our specification using input terms. The input terms must be valid RelaxNGc schema samples. We can for instance extract our test cases from the RelaxNGc tutorial. To test the specification we created a large number of SDF .trm files containing valid RelaxNGc samples by selecting the ASF+SDF new term option in the Meta-Environment. Saving the term results in the term being parsed by the SDF2 parser.

Now one of three things can happen:

1. The term parses correctly. We can view the tree to see if it has the expected structure. If so we can skip to the next term for testing.

2. The term can not be parsed correctly and contains a parse error. This could either be due to a syntax error (e.g. an error introduced when copy/pasting our example to the .trm file or an error in the example itself) or the term cannot be parsed because there is an error in our specification.

3, The term is parsed but the Meta-Environment generates warnings that there is an ambiguity in the term and shows the number of alternatives for a token when parsing.

In the last two cases the specification has to be fixed before we can move on. Fixing large specifications can be a cumbersome and time consuming task. The type of example used and the warning generated by ASF+SDF should give us some idea where to look for the error.

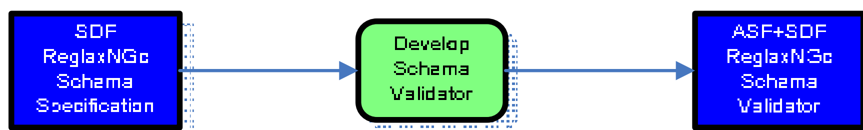
An approach for removing ambiguities is testing smaller parts of the specification to see if those contain the errors. For example, if ASF+SDF doesn't parse an identifier correctly we could copy the rules for parsing an identifier to a separate SDF module and try to parse only a term containing the troublesome identifier. Since this is a smaller problem it is generally easier to fix. If the problem isn't the identifier we extend the part to include other production rules and test again. The downside to this approach is that it only works if the person debugging has affinity with the specification. Before creating test terms for the smaller problem we first need to analyze the specification to determine what examples to create for testing.

When the resulting SDF specification is free of ambiguities it can already be used to check the syntax of RelaxNGc schemas. When RelaxNGc schema examples are used as input terms for SDF then parsing these terms will either succeed, indicating that the syntax of the input term is correct, or it will fail, indicating that the term contains a syntax error.

4.1.2 Develop ASF+SDF RelaxNGc Schema Validator

In the ASF+SDF Meta-Environment we can use the SDF RelaxNGc schema specification for syntax checking. Unfortunately this does not mean that the RelaxNGc schema is a valid schema. For a RelaxNGc schema to be valid is has to comply with a set of constraints. To test for these constraints the SDF RelaxNGc schema specification needs to be extended in ASF+SDF with tests to determine if the constraints hold.

Figure 4.1.2: Activity Diagram – Develop Schema Validator



The RelaxNGc specification defines 13 constraints that have to hold. These constraints have to be implemented in ASF+SDF thus we need to manually extend the SDF specification with an ASF part. The result of our enhancement is an informal ASF+SDF RelaxNGc Schema Validator.

Examples of constraints:

- Constraint: valid prefix:
  It is an error if the value of a *namespacePrefix* is xlmns.
- Constraint: xml prefix:
  It is an error if the value o *namespacePrefix* is xml and the value of *namespaceURILiteral* is not http://www.w3.org/XML/1998/namespace
- Constraint: xml namespace URI:
  It is an error if the value of the *namespaceURILiteral* is http://www/w3/org/XML/1998/namspace and the value of the namespace is not xml.
- …..

According to **[RNC02]** a textual object is a correct RELAX NG Compact Syntax schema if:

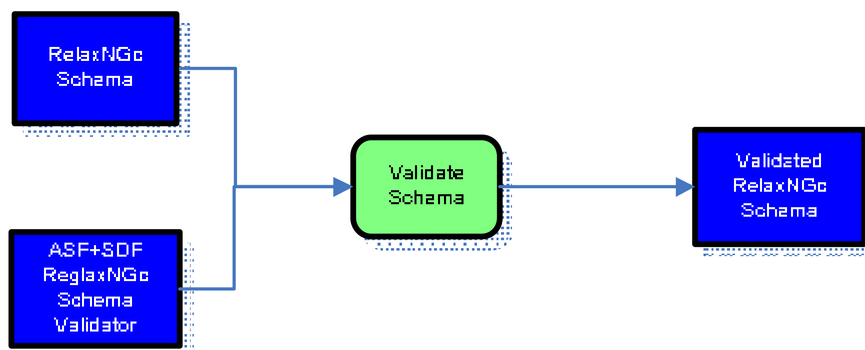
- it matches the formal BNF grammar
- it satisfies the specified constraints
- the result of the translation is a correct RELAX NG schema

Input: SDF OASIS RelaxNGc schema specification
Output: ASF+SDF RelaxNGc schema validator

4.1.3 Validate schema

When our ASF+SDF validator is realized we can use it to validate RelaxNGc schemas. Given a schema in the form of a .trm file and an ASF+SDF specification of the validator we can determine in the Meta-Environment if the term is correct RELAX NG Compact Syntax schema by rewriting the term to either true (a valid RelaxNGc schema) or false (an invalid RelaxNGc schema).

Figure 4.1.3: Activity Diagram – Validate Schema



Input: RelaxNGc schema
Input: ASF+SDF RelaxNGc schema validator
Output: Validated RelaxNGc schema

4.1.4 Develop an ASF+SDF XML Parser Generator

We now know that our input term is a valid RelaxNGc schema. If we were to write an XML document conforming to this schema the next we step taken would be to determine if our XML file conforms to the RelaxNGc schema. We need some way of checking if the XML document is in the document class defined by the RelaxNGc schema. Since we assume we do not have a generic XML parser for ASF+SDF a possible solution is to generate a SDF parser for the RelaxNGc document class using an ASF+SDF program, the parser generator.

When we use the following code as input:

```
001 element addressBook {
002   element card {
003     element name { text },
004     element email { text }
005   }*
006 }
```

The parser generator should produce an SDF parser that accepts XML documents in the document class and rejects everything else.

In this case the following term should be accepted.

```
001 <addressBook>
002   <card>
003     <name>Hannibal Smith</name>
004     <email>hannibal.smith@a-team.com</email>
005   </card>
006 </addressBook>
```

Since the card element can occur zero or more times the following term should also be accepted:

```
001 <addressBook>
002   <card>
003     <name>Hannibal Smith</name>
004     <email>hannibal.smith@a-team.com</email>
005   </card>
006   <card>
007     <name>B.A. Baracus</name>
008     <email>b.a.baracus@a-team.com</email>
009   </card>
010 </addressBook>
```

And so would this term:

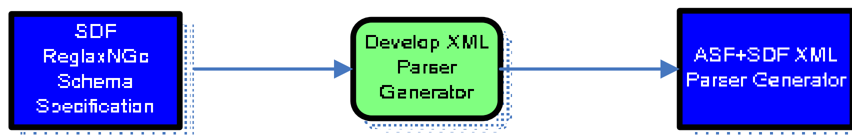
```
001 <addressBook></addressBook>
```

But this term file should be rejected because the use of vcard tags is not allowed according to our RelaxNGc schema.

```
001 <addressBook>
002   <vcard>
003     <name>Hannibal Smith</name>
004     <email>Hannibal.smith@a-team.com</email>
005   </vcard>
006 </addressBook>
```

The output of this process step should be a XML parser generator written manually in ASF+SDF by extending the SDF RelaxNGc schema specification with an ASF part. The ASF part contains rewrite rules used to rewrite the RelaxNGc SDF specification to an SDF XML parser that accepts only XML documents in the schema's document class. For every construct in the SDF schema parser we need to define parser generation functions that output SDF production rules that parse the constructs XML counterpart.

Figure 4.1.4: Activity Diagram – Develop XML Parser Generator

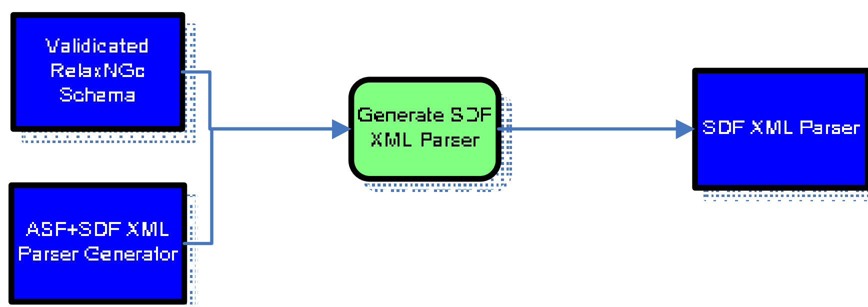


Input: SDF RelaxNGc schema specification
Output: ASF+SDF XML Parser Generator

4.1.5 Generate SDF XML Parser

With the ASF+SDF XML Parser Generator we can generate an SDF XML parser given a valid RelaxNGc schema as input.

Figure 4.1.5: Activity Diagram – Generate SDF XML Parser



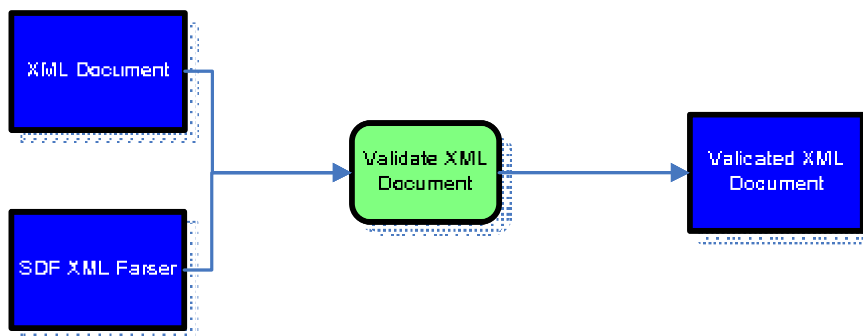
The ASF+SDF parser generator takes the validated RelaxNGc schema as input in the form of a .trm file and the ASF+SDF XML parser generator specification and generates a term containing an SDF XML parser as output.

Input:	Validated RelaxNGc schema
Input:	ASF+SDF XML Parser Generator
Output:	SDF XML parser

4.1.6 Validate XML document

With the SDF XML parser we can determine the validity of the XML document by using it as input term for the SDF parser and parsing it in the ASF+SDF Meta-Environment. If the document parses correctly it is a syntactic correct and well formed XML document in the document class defined by the RelaxNGc specification. Our output is a validated XML document. Note that no type checking is performed.

Figure 4.1.6: Activity Diagram – Validate XML Document



Input:	XML document
Input:	SDF XML Parser
Output:	Validated XML document

4.1.7 Develop ASF+SDF XML Transformer

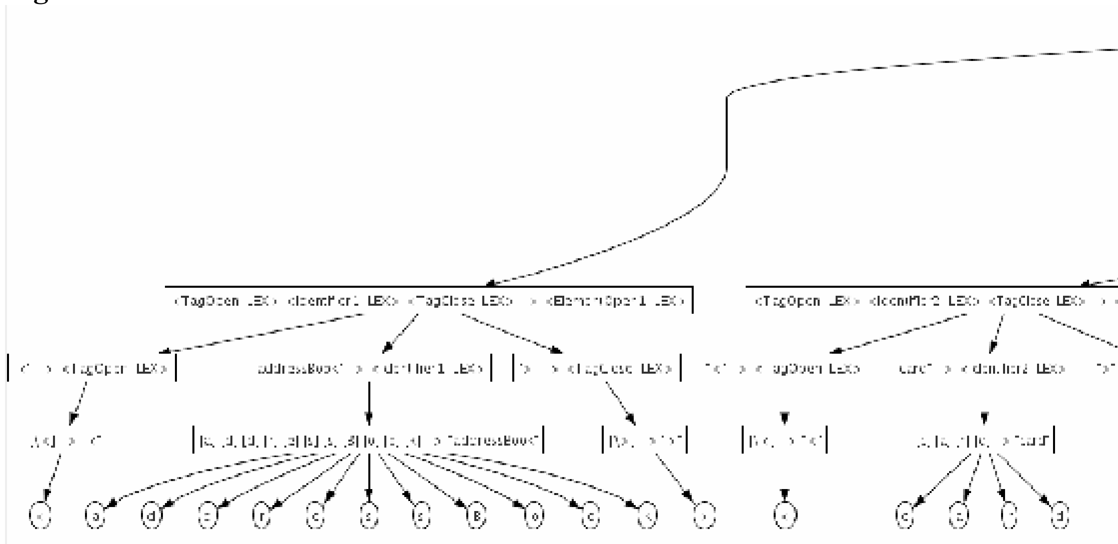
Finally our XML document can be parsed. So now we are curious about the resulting parse tree. The Meta-Environment has functionality for rendering the parse tree of a parsed term.

We can either view the term's structure as:

- A shared tree, visualizing the tree structure for a specific focus as a graph. Each node is maximally shared in this visualization, but not the information inside the nodes.
- A full tree without layout, visualizing the tree structure for a specific focus as a graph, leaving out LAYOUT, but including lexical syntax.

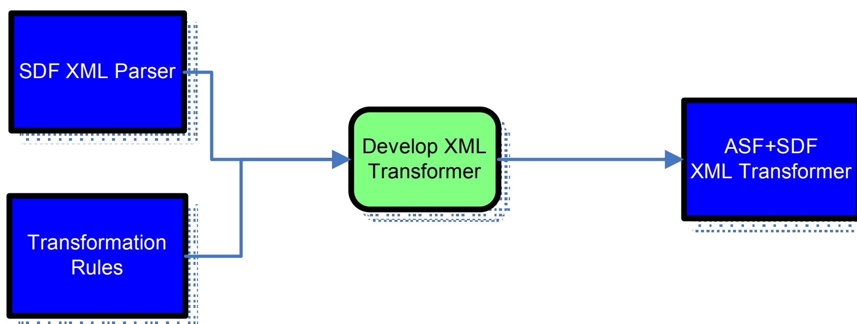
- A full tree, visualizing the tree structure for a specific focus as a graph, including all LAYOUT and lexical syntax.
- A tree, visualizing the tree structure for a specific focus as a graph, leaving out LAYOUT and lexical syntax.

Figure 4.1.7-1: Nodes of a full tree



Before we can do any transformations on the term we need to extend the SDF XML parser to an ASF+SDF program containing transformation rules. The transformations can best be handled with ASF+SDF traversal functions as described in **[TRAV04]** because using a recursive definition gives us a large number of production rules.

Figure 4.1.7-2: Activity Diagram – Develop XML Transformer

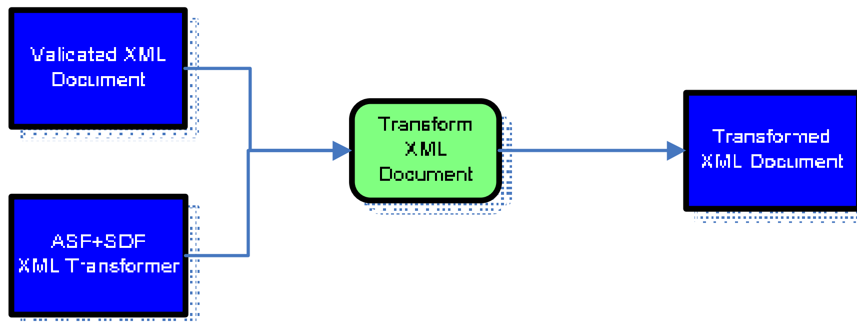


Input:	SDF XML parser
Input:	Transformation rules
Output:	ASF+SDF XML transformer

4.1.8 Transform XML document

The last step in our process is transforming the validated XML document using the ASF+SDF XML transformer containing our transformations. The input term contains the transformation function with its argument, the XML data for transformation. The result is an output term which contains the transformed XML data.

Figure 4.1.8: Activity Diagram – Transform XML Document



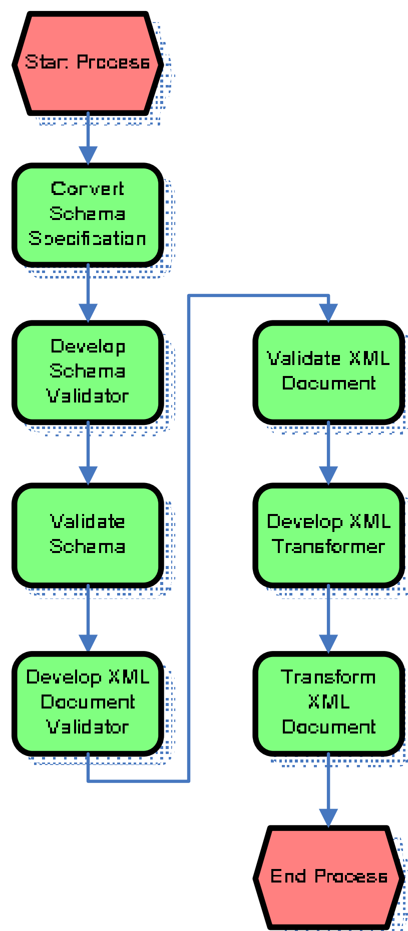
Input:	Validated XML document
Input:	ASF+SDF XML transformer
Output:	Transformed XML document

5 Transformation process using a generic XML parser in ASF+SDF

The generic XML parser approach uses an already implemented SDF XML parser for the parsing of XML documents. The XML parser specification was converted to SDF from the formal XML specification from **[XML06]**. Although the SDF parser specification still contains ambiguities it still might be useful to explore if the path reusing this specification differs from the dedicated parser process. Again we will be using a XML document from a XML document class specified by a RelaxNGc schema.

The process will look as described in figure 5.1.

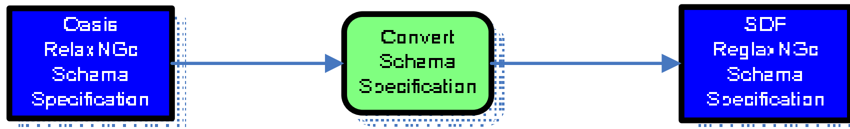
Figure 5.1: Process – Generic XML Parser Approach



5.1.1 Convert Schema Specification

This step is essentially the same as in our dedicated parser approach. Because we want to validate the RelaxNG compact syntax schema we have to convert the schema specification to an SDF specification. This is a one time occurrence. Once the BNF specification has been converted we can reuse it.

Figure 5.1.1: Activity Diagram – Convert Schema Specification



Input: OASIS RelaxNGc schema specification
Output: SDF RelaxNGc schema specification

5.1.2 Develop ASF+SDF RelaxNGc Schema Validator

This step's is the same as the dedicated parser approach in 4.1.2.

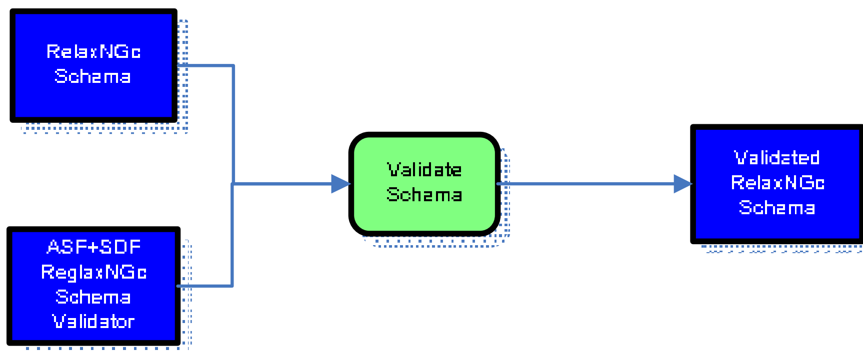
Figure 5.1.2: Activity Diagram – Develop RelaxNGc Schema Validator

Input: SDF RelaxNGc schema specification
Output: ASF+SDF RelaxNGc schema validator

5.1.3 Validate Schema

This step is the same as in the dedicated parser approach 4.1.3.

Figure 5.1.3: Activity Diagram – Validate Schema

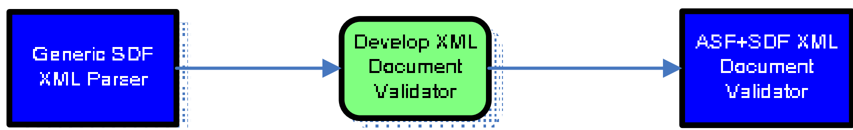


Input: RelaxNGc schema
Input: ASF+SDF RelaxNGc schema validator
Output: Validated RelaxNGc schema

5.1.4 Develop ASF+SDF XML Document Validator

In order to validate XML documents we need to be able to parse XML documents and determine that the document has got a valid structure containing only the patterns specified in the RelaxNGc schema. We extend the SDF generic XML parser with ASF adding recursive tree traversal rewrite rules. For every pattern in a relevant production rule from the RelaxNGc specification the XML syntax tree has to be traversed and inspected for valid or invalid sub trees.

Figure 5.1.4: Activity Diagram – Develop XML Document Validator

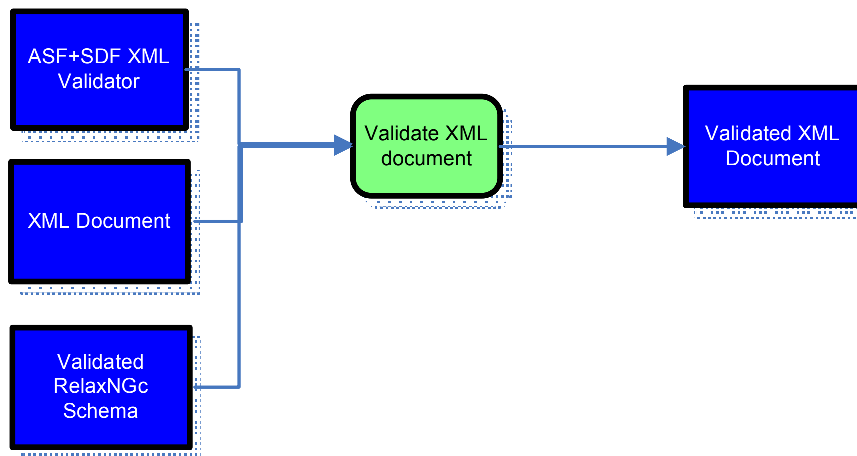


Input: Generic SDF XML parser
Output: ASF+SDF XML-document validator

5.1.5 Validate XML Document

To test if XML document has got a valid structure we need to validate the document using the ASF+SDF XML validator. The validator takes as input a XML document and determines if the document is valid by comparing elements and attributes from its syntax tree with the allowed elements and attributes from the RelaxNGc specification.

Figure 5.1.5: Activity Diagram – Validate XML Document



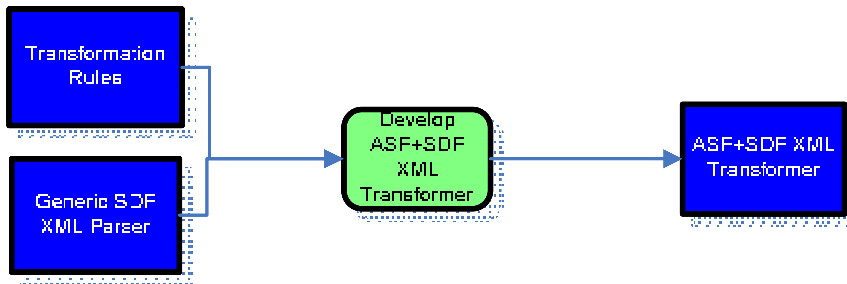
Input:	ASF+SDF XML validator specification
Input:	XML document
Input:	RelaxNGc schema
Output:	Validated XML document

5.1.6 Develop ASF+SDF XML Transformer

After validation the document is in the appropriate RelaxNGc document class. We now have to create an XML transformer that transforms XML documents. So again we need to extend the generic SDF XML parser specification with ASF.

In ASF we define our transformation rules on the syntax tree of the parsed XML document. In the SDF section we add variable definitions and specify prefix functions **[META05]**.

Figure 5.1.6: Activity Diagram – Develop XML Transformer

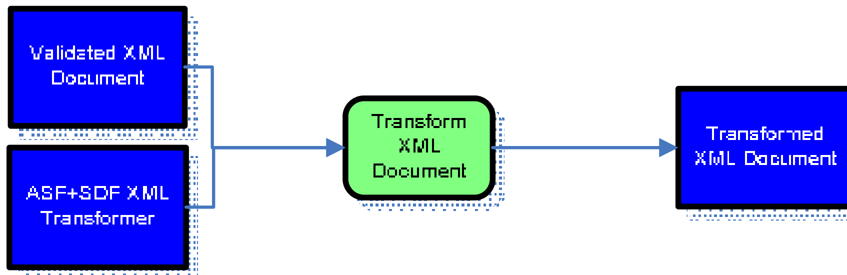


Input:	Transformation Rules/Mapping
Input:	Generic SDF XML parser
Output:	ASF+SDF XML Transformer

5.1.7 Transform XML document

To transform the validated XML document we use our validated XML document and the ASF+SDF XML transformer as input for the transformation. The result is a term containing the transformed XML document.

Figure 5.1.7: Activity Diagram – Transform XML document



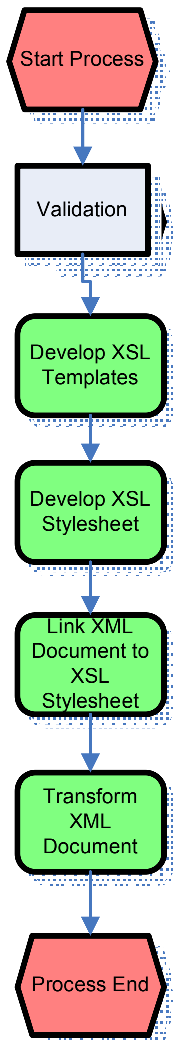
Input:	Validated XML document.
Input:	ASF+SDF XML Transformer
Output:	Transformed XML Document

6 Transformation process using XSLT

In many cases XML transformations are performed with the aid of XSLT. Commercial Integrated Development Environments like *XMLSpy* or *Stylus XML Studio* support the transformation process. There are also non commercial XSLT processors available without IDE like the *Gnome libxslt library*. Enterprise process integration tools like SAP Netweaver PI or the Peoplsoft Integration Engine and several BPEL engines have XSLT processors built in for realtime transformation of XML messages.

The general transformation process will look as follows. We will trace the steps of this process using a commercial tool.

Figure 6.1: Process - Transformations with XSLT



6.1.1 Validation

In the commercial tool validation of the XML document can take place after the XML document has been opened in the IDE. The option *Validate Document* will validate the opened XML file against the schema referenced in the XML file. The output is written to the output window.

As an example we have created a DocBook file using the XMLMind XML editor. Apparently it uses XML DTD for the DocBook schema.

Schema 6.1.1-1: Example of a malformed DocBook document.

```
001 <?xml version=”1.0” encoding=”UTF-8”?>
002 <!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.5//EN"
003 "http://www.OASIS-open.org/docbook/xml/4.5/docbookx.dtd">
004 <book
005   <title>A DocBook Example</title>
006
007   <chapter>
008     <title>The First Chapter</title>
009
010     <section>
011       <title>The First Section</title>
012
013       <para>The first paragraph of the first section.</para>
014     </section>
015
016     <section>
017       <title>The Second Section</title>
018
019       <para>The first paragraph of the second section.</para>
020     </section>
021   </chapter>
022 </book>
```

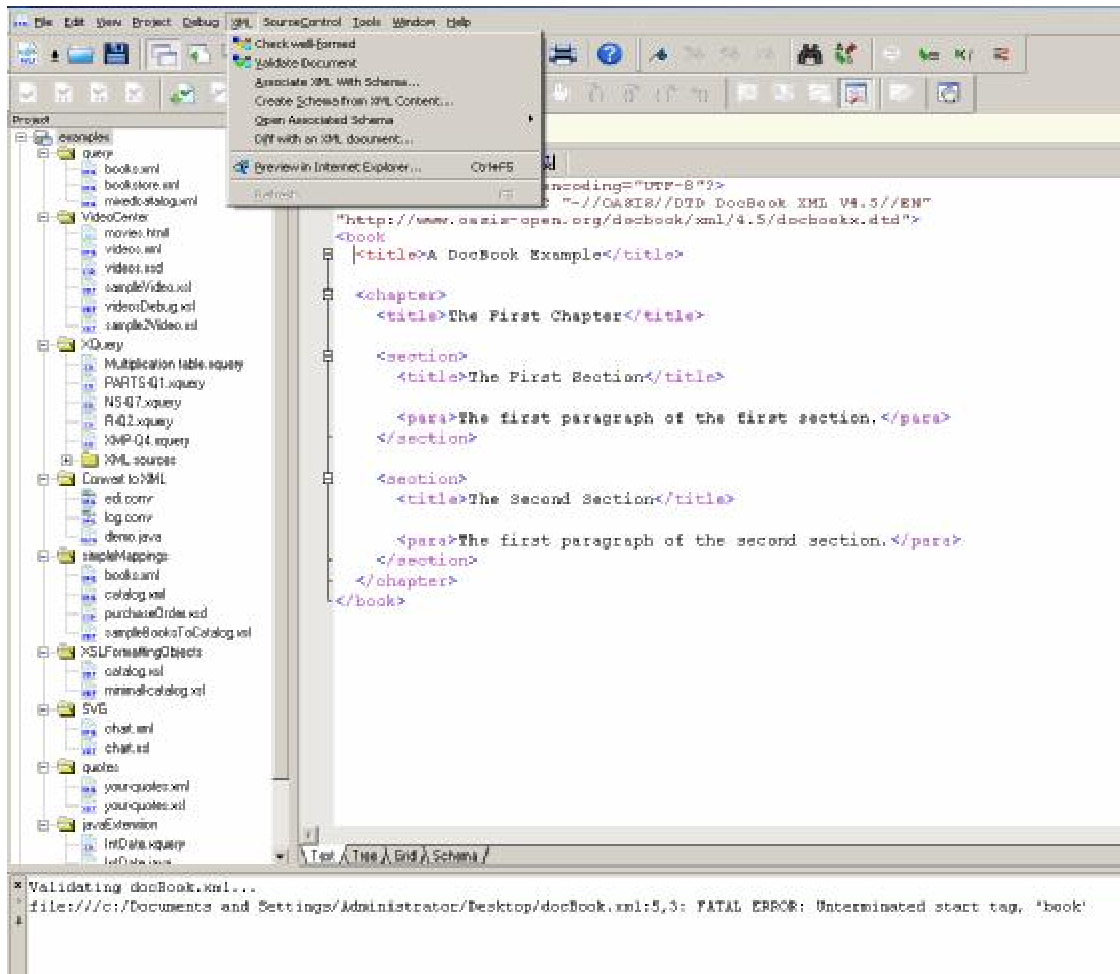
After validation the output window displays the error:

```
FATAL ERROR: Unterminated start tag, 'book'.
```

This is indeed correct since we purposely altered the book tag to force an error.
After correcting the error and validating again the output window displays:

```
The XML document docBook.xml is valid
```

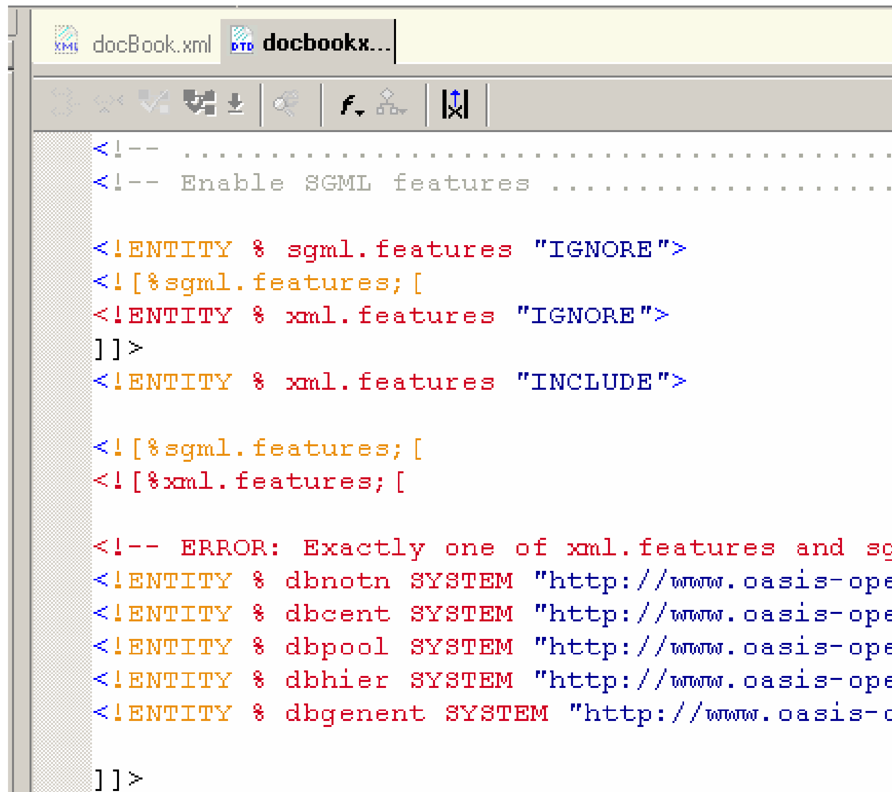
Figure 6.1.1-2: Validation options



We have other options as well. We can for instance open the associated schema which will be downloaded via the intranet form the URI in the DOCTYPE tag.
http://www.OASIS-open.org/docbook/xml/4.5/docbookx.dtd

We can view the DTD.

Figure 6.1.1-3: Schema associated with our DocBook XML document



And we can even create a new W3C XML schema from an existing XML file without a schema.

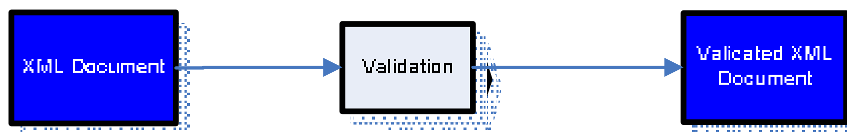
Figure 6.1.1-4: Schema derived from a DocBook XML file

```
001 <?xml version="1.0" encoding="UTF-8"?>
002 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
003 elementFormDefault="qualified">
004   <xs:element name="book">
005     <xs:complexType>
006       <xs:sequence>
007         <xs:element ref="title"/>
008         <xs:element ref="chapter"/>
009       </xs:sequence>
010     </xs:complexType>
011   </xs:element>
012   <xs:element name="chapter">
013     <xs:complexType>
014       <xs:sequence>
015         <xs:element ref="title"/>
016         <xs:element maxOccurs="unbounded" ref="section"/>
017       </xs:sequence>
018     </xs:complexType>
019   </xs:element>
020   <xs:element name="section">
```

```
021       <xs:complexType>
022         <xs:sequence>
023           <xs:element ref="title"/>
024           <xs:element ref="para"/>
025         </xs:sequence>
026       </xs:complexType>
027     </xs:element>
028     <xs:element name="para" type="xs:string"/>
029     <xs:element name="title" type="xs:string"/>
030   </xs:schema>
```

Or we can check the well formedness of our XML document.

Figure 6.1.1-5: Activity Diagram - Validation



In the activity diagram we have entered the validation process step as a sub process because several tools are available for:

- Validating XML DTD and W3C XML schema
- Inspecting the schema
- Showing the difference between the schema and the XML file

In the sub process we can use any of these tools to aid validation.

Input: XML document
Output: Validated XML document

6.1.2 Create XSL Stylesheet

Next we need to create a new XSL Template for our transformation. We can either do this by hand using a text editor, or we can use the IDE to generate an empty stylesheet. We choose the latter and select *File -> New -> XSLT Stylesheet* from the menu.

We can specify all kinds of parameters like the source XML file for transformation, the output XML file and even what XSLT processor to use.

Note that with this IDE the files do not even have to be on the local file system. They could as well reside somewhere on the internet. In this particular IDE the transformation process is referred to as a scenario.

Figure 6.1.2-1: Defining source and output of a transformation.

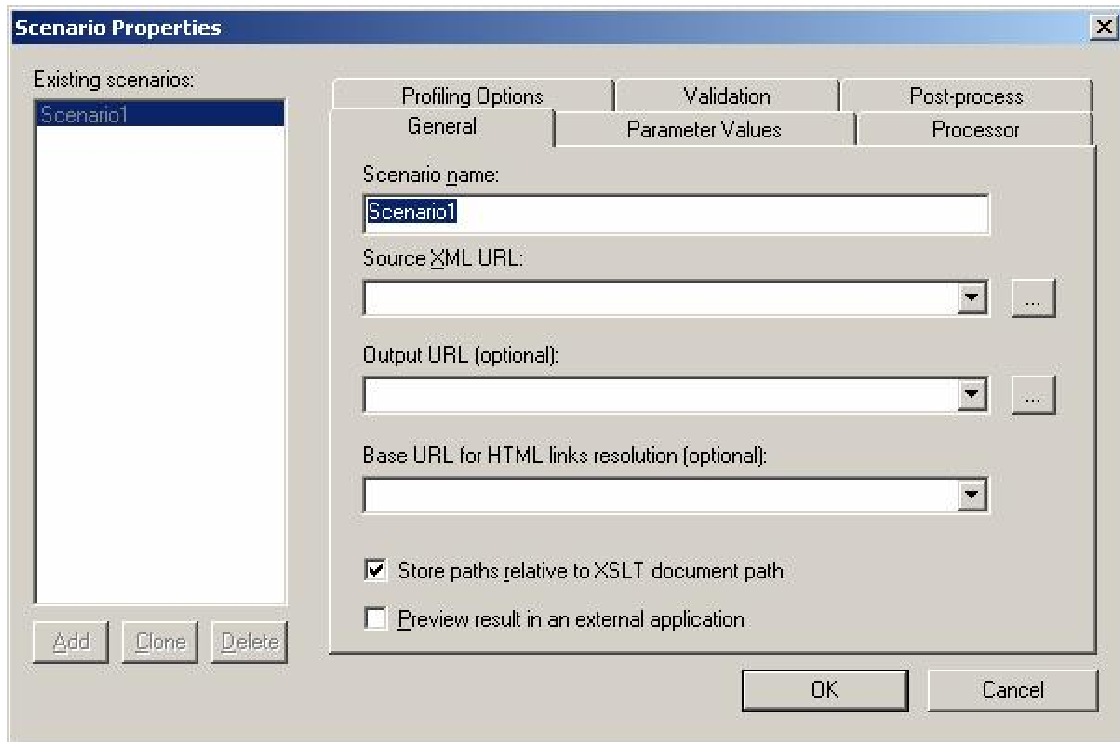


Figure 6.1.2-2: Activity Diagram - Develop XSL Stylesheet

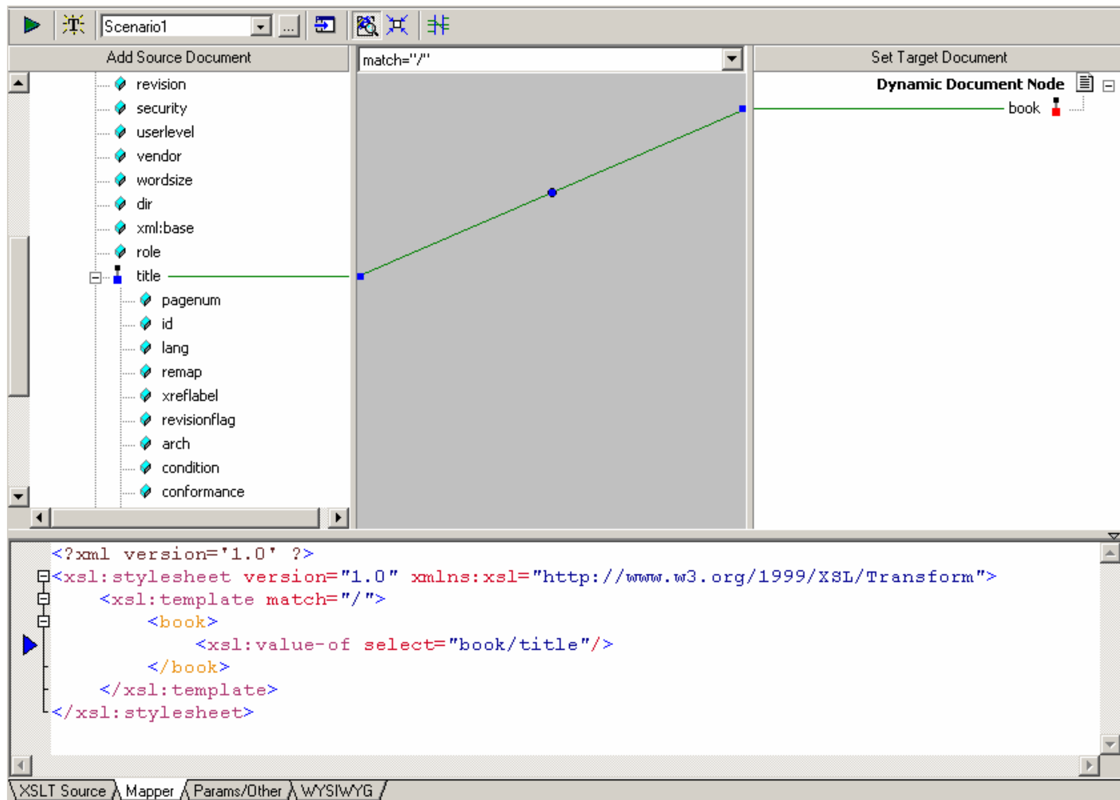
After creating the stylesheet a default template is created in the stylesheet.

Input: XSL Templates
Output: XSL Stylesheet

6.1.3 Develop XSL Templates

Using the mapper tool from the IDE we can add or modify XSL templates by connecting elements from the Source XML document and output document. But first we need to enter a root element for the destination document. Since we would like the result to also be a DocBook file we enter *book* as the root element. Next we connect the *title* element from the source to the newly created *book* element. The stylesheet template is automatically created.

Figure 6.1.3-1: The mapping tool



In this example we only selected one node for our transformation. However, defining more complex transformations is just as easy using XSLT functions and instructions which can be select from the function list and inserted at correct location in the XLS template using the mapping tool.

Figure 6.1.3-2: Available XSLT functions for mapping.

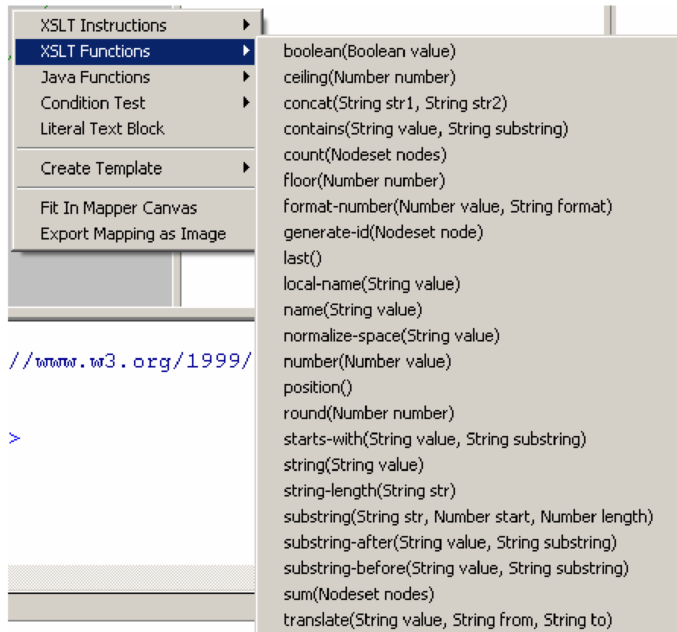
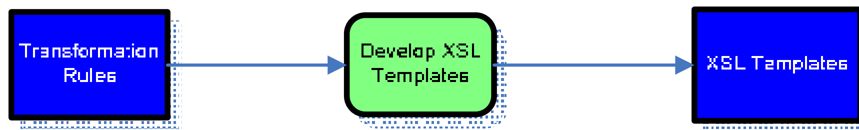


Figure 6.1.3-3: Activity Diagram - Develop XSL Templates

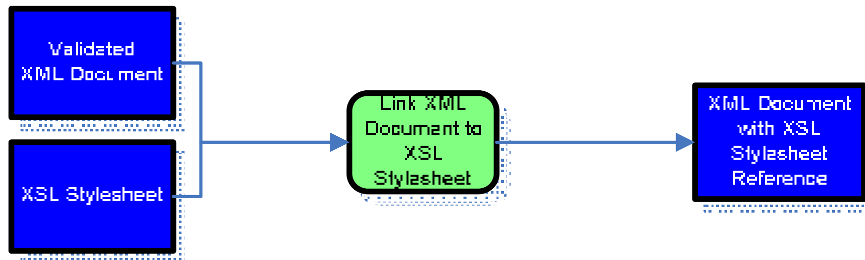


Input: Transformation rules
Output: XSL Templates (in a single stylesheet)

6.1.4 Link XML Document to XSL Stylesheet

When not using a WYSIWYG editor or a web browser as the XSLT processor the XSL stylesheet has to be created manually with a text or XML editor. Some XSLT processors need a reference to the stylesheet in the XML source document. Other processors, like the one we are using in this example, have separate input and output parameters for the source and destination XML files. Depending on the type of XSLT processor this step is optional.

Figure 6.1.4: Activity Diagram - Link XML Document to XSL Stylesheet



Input: Validate XML document
Output: XML document with XSL stylesheet reference

6.1.5 Transform XML Document

To transform our document we push the preview button. We can select the output window to either display a tree view, a browser window or plain text. Our output is the tree representation of the output.

Figure 6.1.5-1: Transformation output in tree view.

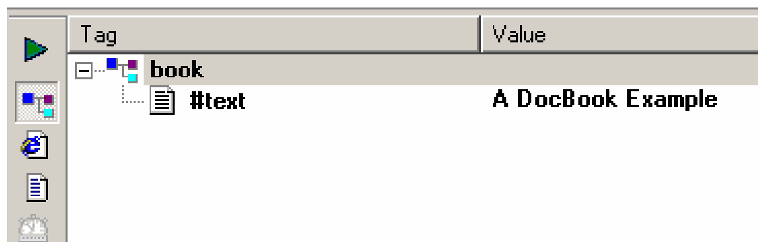
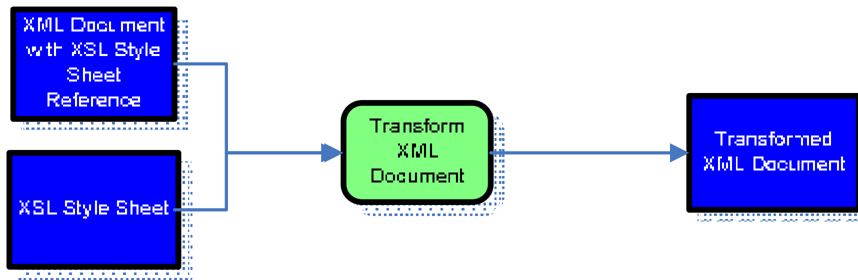


Figure 6.1.5-2: Transformation output as text.

001	<?xml version='1.0' ?>
002	<book>A DocBook Example</book>

Note that our resulting XML file is not yet a DocBook file. The schema reference is missing. We will have to add it manually and have to adjust our mapping to include <title> tags.

Figure 6.1.5-3: Activity Diagram - Transform XML Document



Input: XML Document with XSL stylesheet reference
Input: XSL stylesheet
Output: Transformed XML document

7 Process Comparison

In the table in section 7.1 the process steps for the four processes are shown in one comparison table. Process steps with the same functionality are entered at the same row.

7.1 Process Comparison Table

Generic Process Step	**XSLT** Process Step	**Dedicated Parser** Process Step	**Generic Parser** Process Step
1. Determine schema			
		1. Convert schema specification	1. Convert Schema Specification
		2. Develop ASF+SDF RelaxNGc schema validator	2. Develop ASF+SDF RelaxNGc schema validator
2.. Validate schema	1. Validate schema	3. Validate schema	3. Validate schema
		4. Develop ASF+SDF XML parser generator	4. Develop ASF+SDF XML Document validator
		5. Generate SDF XML parser	
3. Validate XML document		6. Validate XML document	6. Validate XML document
4, Deduce XML patterns			
	2. Develop XSL stylesheet		
4. Define transformation rules	3. Develop XSL template	7. Develop ASF+SDF XML transformer	6. Develop ASF+SDF XML transformer
6. Transform XML document	4 Transform XML document	8. Transform XML document	7. Transform XML document

7.2 Process Comparison

When we compare the processes for transforming XML documents with ASF+SDF with the process of transforming XML documents with commercial XSLT tool several differences comes to light.

- Transformers in ASF+SDF operate on syntax trees where as XSLT operates on a structured XML tree containing XML elements and attributes.
- XSLT uses XPATH for specifying positions within the XML tree. With ASF+SDF transformers using traversal functions we can only specify positions being the root node of the tree or the leaf nodes.
- Using traversal functions for substituting nodes in the tree limits us to substituting a node or sub tree of a type for a type that still allows for a valid specification. We can't just substitute any arbitrary type without first modifying the SDF specification.
- XSLT uses XSLT instructions and XSLT functions to allow for predefined functionality. Although ASF+SDF comes with specifications for Trees, Tables, Lists Booleans and Naturals, we suspect that this just isn't enough for quickly supporting transformations.
- The ASF+SDF Meta-Environment is an IDE used for development of formal languages and specifications. It uses term rewriting as its main instrument and is therefore rather slow. IDE's like *XML-Spy* or *Stylus Studio* which are dedicated to XML transformations using XML, XPATH and XSLT and XSL contain dedicated and smart XML editors, preview windows, XSLT processors in native code, mapping tools for mapping elements and attributes from one XML file to another, XML viewers and so on. In ASF+SDF a lot of tooling is needed before the transformation process has the same usability level as these dedicated tools.

The two ASF+SDF processes resemble each other. The first three process steps are identical. From process steps four and up the processes start to differ. Even though functions four and up sometimes have the same functionality, they differ on input an output of the process functions.

8 Implementation

8.1 Tooling

When implementing the three transformation processes we encountered no problems with the XSLT transformation process. The main reason for this is that all the tooling for this process already exists in the form of an IDE dedicated to XSLT transformations.

For the ASF+SDF processes however several tools have to be developed to support the process.

Table 8-1: Additional ASF+SDF tools needed for the Dedicated XML Parser process

#	Tool	Functionality
1	SDF RelaxNG Compact Syntax Parser	Parse RelaxNG Compact Syntax schema documents.
2	ASF+SDF Schema Validator	Validate RelaxNG Compact Syntax schema documents.
3	ASF+SDF Dedicated XML Parser Generator	Generate a dedicated XML parser given a RelaxNGc document as input.
4	ASF+SDF XML Transformer	Converts a XML document to another format using the SDF specification of the dedicated parser.

Table 8-2: Additional ASF+SDF tools needed for the Generic XML Parser process

#	Tool	Functionality
1	SDF RelaxNG Compact Syntax Parser	Parse RelaxNG Compact Syntax schema documents.
2	ASF+SDF Schema Validator	Validate RelaxNG Compact Syntax schema documents.
3	ASF+SDF XML Validator	Validate an XML syntax tree
4	ASF+SDF XML Transformer	Converts an XML document to another format using the SDF specification of the generic parser.

Except for the transformers all other tools have to be developed only once and can then be reused.

8.2 Implementation issues

With the actual implementation of the ASF+SDF processes we encountered problems at an early stage. The result is that only specific parts of the process could be considered for implementation. All tools rely heavily on a correct specification of RelaxNGc parser. The first priority was to implement this parser.

The actual work done on these implementations has been the following:

Conversion of the RelaxNGc EBNF specification to SDF
The RelaxNG EBNF specification is a lot smaller and more readable than the BNF specification. Therefore it was our first choice for conversion to SDF. Using a rather small test set in the early stages of development we later came to the conclusion that this informal non-normative specification could not be used for parsing large RelaxNG Compact Syntax specifications like DocBook. The EBNF specification is incomplete. It does not support comments, and contains ambiguities by default.

Validation of RelaxNGc schema with EBNF
The EBNF specification parsed several small RelaxNG schema code snippets. We used these snippets for developing parts of the validator. Due to the ambiguities and incompleteness of the specification we did not manage to deliver a completely working solution.

Parser generation
For the dedicated XML parser generator we started again with the EBNF specification. We developed several small parser generators that generate SDF production rules for parsing XML tag pairs and content. Due to the ambiguities and incompleteness of the specification we did not manage to deliver a working solution.

Extending the EBNF specification with elements from the BNF specification
In order to get our previous development attempts working we invested in extending the EBNF specification with elements from the BNF specification that were missing in the EBNF version. This resulted in more ambiguous constructs in the resulting specification and added to overall complexity. So this path was abandoned

Conversion of the RelaxNGc BNF specification to SDF
After our attempts with the EBNF specification we tried converting the BNF specification which is a much larger specification. Using larger specifications results in using more rewrite rules when implementing transformers. Conversion resulted in a parser containing ambiguities and parse errors of which most are now resolved. The parser was tested heavily with about eighty code snippets with different constructs from the RELAX NG Compact Syntax tutorial. It can now parse about 95% of the test set. One of the open issues is that the SDF implementation yields parse errors when parsing RelaxNGc documents containing comments and escape sequences. Another issue is that it displays ambiguities when parsing nested annotations. Other issues were resolved using

longest match and prefer annotations after inspecting ambiguous parse trees. It still fails on the DocBook RelaxNG Compact Syntax specification because it is heavily commented.

Testing the Generic XML Parser
The generic XML parser which will become part of the ASF+SDF library still contains some ambiguous constructs. We tested this parser with several XML document formats including DocBook to inspect the parse trees for developing transformers.

Implementing transformers
Several small transformers were implemented using rewrite rules and traversal functions for use with XML syntax trees.

9 Conclusions

In the previous chapters we have described four transformation processes and traced and compared them. We have discussed a generic transformation processes and two specific processes involving ASF+SDF and the Meta-Environment and the process using XSLT and an XSLT-engine.

Our final conclusion is that doing XML transformations with ASF+SDF from scratch is far more complex than using XSLT with an off the shelf XSLT processor. The complexity lies in the amount of time and effort that has to be spent developing tools like parsers, parser generators and tree transformers before the actual transformation can even take place. However, once we have those tools we should be able to reuse them and the processes should become less complicated.

ASF+SDF lacks mechanisms other than selecting part of a term to start transformation from a pre selected node. In XSLT selection of a starting node can be done using XPATH.

The system for rewriting between XSLT and ASF+SDF processes differs a lot. In the XSLT process XSL and XSLT functions are used on a structured XML tree, in ASF+SDF term rewriting and traversal mechanisms are used on a syntax tree. Before we can use these rewriting and traversal mechanisms thorough understanding of the SDF specification is needed.

For developing ASF+SDF tools a working SDF specification is a must. All tools are derived form such a specification. ASF+SDF developers need skills for converting meta-syntaxes like BNF or EBNF to SDF and for disambiguating them.

10 Topics for further study

The topics listed below address several issues encountered during implementation but weren't solved. These issues deserve investigating in more detail and could be used as research topics for further study.

10.1 Improvements for the SDF2 SGLR parser

Extending SDF with Unicode support
A one-to-one conversion of BNF or EBNF to SDF can be a problem using the current SDF SGLR parser when these specifications contain UTF16 or Unicode characters because the parser can only cope with specifications that use the Latin-1 character set. Implementing these types of specifications involves manual alteration of the specification thereby omitting all UTF16 or Unicode lexical or context-free syntax constructs. When language specifications contain Unicode or UTF16 characters it is almost impossible to create a specification compliant parser since it will only parse a subset of the possible input characters. To cope with such situations a solution would be to extend the SDF SGLR with multi character set support thus allowing SDF parsers to parse e.g. Unicode or UTF16 characters.

10.2 Improvements for ASF+SDF

10.2.1 Extending ASF+SDF with an include mechanism

Many languages support mechanisms for allowing modularity often in the form of an include mechanism. Parsing documents written in those languages can result in incomplete parse trees when a vital part of the document is situated in another file. ASF+SDF has not got support for substituting a reference to an include file with its contents. The only way to parse dependent include files in the current situation is to do manual substitution of the include statement with its contents and save the resulting file for use as input for the parser. Another commonly used practice is to use external preprocessing tools.

A solution to this problem may be to extend ASF+SDF with auxiliary functions to replace a term matching a predefined pattern with the contents of a file using a filename that is extracted form the term during parsing.

10.2.2 Extending ASF+SDF with URI support

Many specifications have references to external data sources on the internet. The current version of ASF+SDF lacks http support for gathering such documents. Documents have to be downloaded manually from the internet before they can be passed to the parser.

Extending ASF+SDF with auxiliary functions for getting and putting files over http using Universal Resource Identifiers (URIs) and substituting them for the appropriate term might reduce the number of downloads and substitutions on has to execute beforehand.

10.3 Improvements for the ASF+SDF Meta-Environment

10.3.1 Displaying Partial Parse Trees

When an SDF specification can be parsed without errors the resulting parse tree can be displayed and subsequently inspected by zooming in and out on the tree. When the SDF specification contains errors one would expect that a partial parse tree would be available for viewing and inspecting. The partial parse tree would consist of the nodes that had been added to the tree up to the point the error occurred during parsing. Displaying a partial parse tree might give some useful leads as were to look for errors in the SDF specification. It has come to our attention that this problem was recently solved an will be included in one of the next releases of the Meta-Environment.

10.4 Generating ASF+SDF XML parse tree transformer

In our process description we've had to implement the transformer on XML documents manually. The transformer contains the transformation rules in ASF+SDF. Using tree traversal functions or rewrite rules the document can be transformed. These rules work on the SDF syntax tree of an XML document. Supposing we only want to transform XML to XML then we could use the more human readable XSLT syntax for defining our transformations and generate and ASF+SDF transformer program form these transformation rules. We would only have to implement this transformation program once.

11 Bibliography

[META05] M.G.J. van den Brand and P. Klint, ASF+SDF Meta-Environment User Manual. January 2005.

[TRAV04] M.G.J. van den Brand, P. Klint and J.J. Vinju. Term rewriting with Traversal Functions. ACM transactions on Computational Logic. 2004.

[SYNT06] P. Klint. Syntax Analysis. 13 December 2006.

[COM01] M.G.J. van den Brand, A. van Deursen, J.Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, Scheerder, J.J. Vinju, E. Visser, and J. Visser, The ASF+SDF Meta-Environment: a Component Based Language Development Environment. Proceedings of Compiler Construction, 2001.

[EXT07] Paul Klint, Jurgen Vinju. The extension points of the Meta-Environment. January 2007.

[FILT94] Paul Klint, Eelco Visser. Using Filters for the Disambiguation of Context-free Grammars.

[ATRM00] M.G.J. van den Brand, H.A. de Jong, P. Klint, P.A. Olivier. Efficient Annotated Terms. SEN-R0003 February 28, 2000.

[DSTK] J.J.Vinju. SDF Disambiguation Medkit for Programming Languages.

[ARCH06] Paul Klint, Jurgen Vinju. The Architecture of The Meta-Environment. December 13, 2006.

[LANG88] Thomas A. Sudkamp, Languages and Machines, An Introduction to the Theory of Computer Science. p46-53, p237-247. Addison Wesley 15768. 1988.

[SDF00] Joost Visser, Jeroen Scheerder. A Quick Introduction to SDF. April 25, 2000

[SDFXX] Mark van den Brand, Paul Klint, Jurgen Vinju. The Syntax Definition Formalism SDF.

[SGLR97] Eelco Visser. Scannerless Generalized LR-Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam. 1997.

[EPC06] Jan Mendling, Wil van der Aalst. Towards EPC Semantics based on State and Context. 2006.

[XTRM03] M. Bravenboer. Connecting XML Processing and Term Rewriting with Tree Grammars. Utrecht University. November 2003

[TERM06] Paul Klint, Term Rewriting. 13 December 2006.

[CGEN03] Jonne van Wijngaarden. Code Generation from a Domain Specific Language, Designing and Implementing Complex Program Transformations. July 8, 2003

[RNC02] James Clark. RELAX NG Compact Syntax. Committee Specification 21. OASIS. November 2002.

[RNG01] James Clark, Murata Makoto. RELAX NG Specification. Committee Specification. OASIS. 3 December 2001.

[XMLN99] Tim Bray, Dave Hollander, and Andrew Layman. Namespaces in XML. W3C (World Wide Web Consortium). 1999.

[XMLN06] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin. Namespaces in XML 1.0 (Second Edition). W3C Recommendation. 16 August 2006.

[XM0S04] David C. Fallside, Priscilla Walmsley. XML Schema Part 0: Primer Second Edition, W3C Recommendation 28 October 2004

[XM1S04] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn. XML Schema Part 1: Structures Second Edition, W3C Recommendation 28 October 2004

[XMSD01] Paul V. Biron, Ashok Malhotra. W3C XML Schema Datatypes. XML Schema Part 2: Datatypes. W3C (World Wide Web Consortium). 2001.

[XMSD04] Paul V. Biron, Kaiser Permanente, Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. W3C Recommendation. 28 October 2004

[XML00] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, Eve Maler. Extensible Markup Language (XML) 1.0 Second Edition. W3C (World Wide Web Consortium). 2000.

[XML06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation. 16 August 2006.

[XMCS05] Mary Holstege, Asir S. Vedamuthu. XML Schema: Component Designators, W3C Working Draft 29 March 2005.

[XSLT99] James Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation. 16 November 1999.

[XSLT07] Michael Kay. XSL Transformations (XSLT) Version 2.0. W3C Recommendation 23 January 2007

[DCBK99] Norman Walsh and Leonard Muellner. DocBook, The definitive guide. 1999.

[URI98] T. Berners-Lee, R. Fielding, L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. IETF (Internet Engineering Task Force). 1998.

[XPAT99] James Clark, Steve DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation 16 November 1999.

[XPAT07] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, Jérôme Siméon. XML Path Language (XPath) 2.0. W3C Recommendation 23 January 2007.

[XSL06] Anders Berglund. Extensible Stylesheet Language (XSL) Version 1.1. W3C Recommendation 05 December 2006.