

# Safe Template Processing of XML Documents

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der  
Technischen Universität Dresden  
Fakultät Informatik

eingereicht von

**Dipl.-Inform. Falk Hartmann**  
geboren am 2. April 1973 in Freital

Betreuender Hochschullehrer:  
Prof. Dr. rer. nat. habil. Uwe Aßmann, TU Dresden

Gutachter:  
Prof. Dr. rer. nat. habil. Uwe Aßmann, TU Dresden  
Prof. Dr. Welf Löwe, Linnaeus University

Tag der Verteidigung: 1. Juli 2011

Dresden, im September 2011



# Contents

<b>1. Preface</b>	<b>7</b>
1.1. Overview . . . . .	8
1.2. Problems . . . . .	9
1.3. Motivating Example . . . . .	10
1.4. Goals . . . . .	13
1.5. Contributions . . . . .	14
1.6. Related Work . . . . .	16
1.7. Typographic Conventions . . . . .	16
1.8. Outline . . . . .	17
<b>2. Introduction</b>	<b>19</b>
2.1. Definitions . . . . .	19
2.1.1. Templates and Related Terms . . . . .	20
2.1.2. Life Cycle Phases . . . . .	24
2.1.3. The Extensible Markup Language XML . . . . .	25
2.1.4. XML Schema Languages . . . . .	27
2.2. Applications . . . . .	30
2.2.1. Web Applications . . . . .	30
2.2.2. Code Generation . . . . .	30
2.3. Alternatives to Using Templates . . . . .	31
2.3.1. Transformations . . . . .	31
2.3.2. Aspect-Oriented Approaches . . . . .	32
2.3.3. Unparsers . . . . .	33
2.3.4. Comparison of Templates with Alternative Technologies . . . . .	34
2.4. Related Research Areas . . . . .	35
2.4.1. Macro Processing . . . . .	35
2.4.2. Templates as Programming Language Feature . . . . .	35
2.4.3. Invasive Software Composition . . . . .	36
2.4.4. Frame Processing . . . . .	37
2.5. Classification . . . . .	38
2.5.1. Target Language Awareness of Slot Markup . . . . .	39
2.5.2. Generality of the Slot Markup . . . . .	40
2.5.3. Entanglement Index . . . . .	40

## Contents

2.5.4.	Instantiation Data Access Strategy . . . . .	42
2.5.5.	Query Language . . . . .	43
2.5.6.	Instantiation Technique . . . . .	44
2.5.7.	Reuse in Templates . . . . .	44
2.5.8.	Further Features . . . . .	45
2.6.	Conclusion . . . . .	46
<b>3.</b>	<b>Safe Template Processing</b>	<b>49</b>
3.1.	Goals . . . . .	49
3.1.1.	Safe Authoring . . . . .	50
3.1.2.	Safe Instantiation . . . . .	50
3.1.3.	Separation of Concerns . . . . .	51
3.1.4.	Broad Applicability . . . . .	53
3.1.5.	Utilization of Existing Standards . . . . .	53
3.2.	Requirements . . . . .	54
3.2.1.	Preservation of Target Language Constraints . . . . .	54
3.2.2.	Coverage of Target Language . . . . .	55
3.2.3.	Computability . . . . .	55
3.2.4.	Expressiveness . . . . .	56
3.2.5.	Instantiation Data Type Safety . . . . .	56
3.2.6.	Independence of Query Language . . . . .	57
3.3.	Proposal of an Architecture fulfilling the Requirements . . . . .	57
3.4.	Conclusion . . . . .	60
<b>4.</b>	<b>Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language</b>	<b>61</b>
4.1.	General Design Decisions . . . . .	62
4.2.	Creation of Character Data . . . . .	65
4.2.1.	<code>xsl:text</code> . . . . .	66
4.2.2.	<code>xsl:attribute</code> . . . . .	67
4.2.3.	<code>xsl:include</code> . . . . .	69
4.3.	Conditional and Repeated Inclusion of Template Fragments . . . . .	71
4.3.1.	<code>xsl:if</code> . . . . .	71
4.3.2.	<code>xsl:for-each</code> . . . . .	73
4.4.	Reuse of Template Fragments . . . . .	76
4.4.1.	<code>xsl:macro</code> . . . . .	76
4.4.2.	<code>xsl:call-macro</code> . . . . .	77
4.5.	Advanced Features . . . . .	78
4.5.1.	Accessing multiple Instantiation Data Sources using Realms . . . . .	78
4.5.2.	Instantiation Pipelines using Bypassing . . . . .	80
4.6.	Definition of the Instantiation Semantics using XSL-T . . . . .	83
4.7.	Relation to Document Validation . . . . .	84
4.8.	Conclusion . . . . .	86
<b>5.</b>	<b>Safe Authoring of Templates</b>	<b>87</b>

5.1. Constraint Separation . . . . .	87
5.1.1. Introductory Example . . . . .	89
5.1.2. The Constraint XML Schema Language CXSD . . . . .	94
5.1.3. The Instantiation Data Constraint Language IDC . . . . .	98
5.1.4. Constraint Separation Process . . . . .	99
5.1.5. Proof of the Preservation of the Target Language Constraints . . . . .	103
5.1.5.1. Completeness of the Set of Required Attributes . . . . .	104
5.1.5.2. Compliance to the Content Model . . . . .	105
5.1.6. Visitor-based Implementation of the Constraint Separation . . . . .	107
5.1.7. Partial Templatization . . . . .	116
5.2. Template Validation . . . . .	117
5.3. Conclusion . . . . .	121
<b>6. Flexible, Efficient and Safe Template Instantiation</b>	<b>123</b>
6.1. Instantiation Data Evaluation . . . . .	123
6.1.1. Design of the PHP Interface . . . . .	124
6.1.2. The Identity PHP . . . . .	126
6.1.3. The XPath PHP . . . . .	127
6.1.4. The SPARQL PHP . . . . .	127
6.1.5. The System PHP . . . . .	128
6.2. Template Instantiation . . . . .	129
6.2.1. XML Access Technologies . . . . .	129
6.2.2. Operational Model of the XTL Engine . . . . .	130
6.2.3. Pipeline Implementation of the XTL Engine . . . . .	136
6.2.4. Memory and Runtime Complexity . . . . .	150
6.3. Instantiation Data Validation . . . . .	150
6.3.1. The IDC PHP . . . . .	151
6.3.2. Template Interface Generation . . . . .	152
6.3.2.1. Introductory Example . . . . .	153
6.3.2.2. An Algorithm for the Template Interface Generation . . . . .	155
6.3.2.3. Implementation using a PHP and an API-based Generator . . . . .	160
6.4. Conclusion . . . . .	163
<b>7. Validation</b>	<b>165</b>
7.1. Implementation of the Prototype . . . . .	165
7.1.1. The Constraint Separation Tool <code>xtlsc</code> . . . . .	167
7.1.2. The Template Validation Tool <code>cxsdvalidate</code> . . . . .	167
7.1.3. The Template Instantiation Tool <code>xtlinstantiate</code> . . . . .	168
7.1.4. The Template Interface Generation Tool <code>xtltc</code> . . . . .	169
7.2. Test Suites . . . . .	170
7.2.1. Constraint Separation Test Suite . . . . .	170
7.2.2. Template Validation Test Suite . . . . .	170
7.2.3. Template Instantiation Test Suite . . . . .	171
7.2.4. Template Interface Generation Test Suite . . . . .	172

## Contents

7.2.5. Round-trip Test Suite . . . . .	174
7.3. Applications of the Prototype . . . . .	175
7.3.1. SNOW: Use of XTL in a Staged Architecture . . . . .	175
7.3.2. EMODE: Use of XTL for Model-to-Text Transformations . . . . .	179
7.3.3. FeasiPLe: Use of XTL for Code Generation from Ontologies . . . . .	179
7.4. Proof of the Preservation of the Target Language Constraints . . . . .	180
7.5. Runtime and Memory Usage Measurements . . . . .	180
7.5.1. Runtime Measurement of Validation against a CXSD Schema . . . . .	181
7.5.2. Runtime Measurements of the Template Instantiation . . . . .	182
7.5.3. Memory Usage Measurements of the Template Instantiation . . . . .	184
7.6. Conclusion . . . . .	187
<b>8. Summary, Conclusion, and Outlook</b>	<b>189</b>
8.1. Summary . . . . .	189
8.2. Conclusion . . . . .	190
8.3. Suggested Improvements for XML Technologies . . . . .	191
8.4. Future Research Directions . . . . .	192
<b>A. Referenced XML Schemata and Instances</b>	<b>193</b>
A.1. XML Schema of XTL . . . . .	193
A.2. Purchase Order Schema . . . . .	201
A.3. Purchase Order Instance . . . . .	202
<b>B. Detailed Results of the Runtime and Memory Measurements</b>	<b>205</b>
<b>List of Acronyms</b>	<b>211</b>
<b>List of Figures</b>	<b>216</b>
<b>List of Listings</b>	<b>219</b>
<b>List of Tables</b>	<b>221</b>
<b>Bibliography</b>	<b>237</b>
<b>Index</b>	<b>239</b>

# 1

## Preface

But, how did the first template appear?

International Encyclopedia of Systems and Cybernetics, 1997 [69]

Almost two decades after the introduction of the World Wide Web by Tim Berners-Lee in 1989, the automatic generation of Web pages from dynamic data is still suffering the same problem as in the beginning: How can one be sure that the application produces valid HTML code? There have been several approaches to this problem, among them approaches that successfully solved the problem, thereby unfortunately violating other well-established design rules, like the Separation of Concerns principle. The consequences of this violation can hardly be managed in large applications developed in a cooperation of many participants assigned to multiple roles in the development process: therefore, the problem can still be considered unsolved in its generality.

The goal of this thesis has been to propose a solution that enables *Safe Template Processing*, i.e., a template technique that allows to be sure about the results a Web application produces. In addition, it was required that the solution complies to the mentioned design rules like the Separation of Concerns principle. The solution should furthermore be broadly applicable, i.e., it should not be restricted to the generation of Web pages. Finally, the approach should utilize existing standards and it should be a practical solution, i.e., acceptable to a non-academic user.

This thesis presents an approach that fulfills the described goals by extending a template-based mechanism (as it is well-known to Web engineers) with a validation technique that allows to give guarantees about the results the template is going to produce. Since these guarantees are given at the time the template is being authored, certain assumptions about the data that

## 1. Preface

is consumed within the template must be made. Additional techniques have been developed to check these assumptions.

This chapter starts with an overview of the research area in Section 1.1, including insights into its history. Afterwards, the problem addressed within this thesis is explained in more detail in Section 1.2. Section 1.3 illustrates the problem with a motivating example, whereas Section 1.4 outlines goals derived from the described problems. The contributions developed within this thesis are outlined in Section 1.5. Related research areas are introduced in Section 1.6. The chapter concludes with typographic conventions in Section 1.7 and the outline in Section 1.8.

### 1.1. Overview

It is widely accepted that Edsger W. Dijkstra introduced the term *Separation of Concerns* (SoC) in his groundbreaking article “On the role of scientific thought” [48]. It was pleading for a way of thinking about an aspect of a problem without considering other, related aspects. The idea has been adopted in the discipline of software engineering in various ways, e.g., as a maxim during system analysis or as a guideline for architectural design.

Parallel to Dijkstra, the SoC principle has been suggested by William W. Tunnicliffe as a principle to be used for applications in the publishing sector. Tunnicliffe held a presentation about the separation of information content of documents from their format at the Canadian Government Printing Office in September 1967 [76, Appendix A]. This presentation and the idea of SoC greatly influenced the development of the Standard Generalized Markup Language (SGML) and its successor, the Extensible Markup Language (XML).

There exist a number of techniques to separate aspects of a system, and with them, at least the same number of techniques to perform a necessary composition of the aspects. With respect to software, aspects typically separated from the core logic of a program include for example functionality for monitoring, transaction handling and security, but also constant data needed for program execution, if the data volume exceeds the amount comfortably manageable in the respective programming language.

A very popular technique to integrate data that has been divided due to the SoC principle is the template technology. From an unsophisticated point of view, a template is just an incomplete textual representation of data. Template engines have been used for a variety of purposes, especially in the areas of code generation and Web engineering.

There are three reasons for the widespread use of templates: (a) the learning effort for a particular template language is low, as templates closely resemble the syntax of the language that should be generated with them, (b) it is easy to keep arbitrary complex constant fragments of the document to be generated in the template itself, and (c) the possibility to adhere to the Separation of Concerns principle.

The amount of research conducted on template techniques does not reflect its importance and widespread use. Research on this area is nevertheless urgently needed, as template engines are very popular and, as it will be shown shortly, still suffer from the same problems as when they were introduced, especially with the lack of guarantees that can be given about the result of the instantiation of a template. The need for research is also indicated by the large number of existing template engines: at the time of writing, [96] alone listed 17 open-source, Java-



implemented engines. Some of them differ only in minor details and reflect the users' tendency to be picky about the particular syntax used, while others introduce novel concepts and designs.

Investigations on this area start with the question on the origins of the word *template* in the sense as sketched above. In general, the term seems to be in use since the early days of computer science, designating for example "...a plastic or stiff paper form that is placed over the function keys on a keyboard to identify their use..." [71]. Nowadays, the term is overloaded multiple times, primarily for the generic programming feature of C++ [174].

It seems to be impossible to track down who used the term *template* first to denote the technique described here, but the use of it was straightforward as the term had been in use in a very similar way in the area of electronic form processing, with forms also being a reincarnation of the SoC principle by drawing a border between the preprinted content of a form and the slots to be filled.

Most probably the first implementation of templates in the sense used here was the frame processing approach described by Paul Bassett in his landmark paper "Frame-Based Software Engineering" [17, also appeared in 46]. The term *frame* used by Bassett goes back to the conceptual frames introduced by Marvin Minsky [129]. It remains inscrutable who rebranded or reinvented Bassett's idea to the term *template* most widely used today.

It is possible to find even earlier related approaches. The  $\lambda$ -calculus introduced by Alonzo Church [35] can be seen as an early predecessor of template techniques, as in the term  $\lambda x.x+y$  the bound variable  $x$  is interpretable as representing data coming from a data source different from the source supplying the free variable  $y$ .

## 1.2. Problems

A well-known practical problem with the use of an arbitrary template technique is the possibility to violate the SoC principle. With respect to Web engineering, this corresponds to the architectural failure of not distinguishing clearly between view and controller (and sometimes even model) in an implementation of the Model-View-Controller (MVC) pattern [154]. The problem has been pointed out clearly in Terence Parr's remarkable paper "Enforcing strict model-view separation in template engines" [143].

A second problem can best be illustrated in the Java Server Pages (JSP) technology, a popular approach to building Web frontends in the Java technological space. It is shown in Section 2.1, that JSP documents are in fact templates. JSP users are often confronted with the difficulty of assuring the correctness of the results of the template instantiation. In the JSP world, one often ends up with a template that does not produce valid the Extensible Hypertext Markup Language (XHTML) output (at least not in all cases) or with a page that cannot be compiled by the JSP engine at all. Even with the tool support that became available in the last years, it is still easy to create such erroneous templates.

Three major categories of JSP documents producing invalid XHTML exist:

1. The page may not even produce valid XML, i.e., it may violate the requirements for well-formedness. (*violation of wellformedness*)

## 1. Preface

2. The page produces wellformed XML, but is failing to fulfill the structural constraints that are defined by the XHTML specification, irrespective of any data to be inserted into the template. (*violation of structural constraints*)
3. A template can produce invalid XHTML because of unsatisfied constraints on the data that should be inserted into it during the instantiation. (*violation of data constraints*)

Each of the three categories have different reasons and, more important, different techniques are applicable to deal with them. Alternative technologies like StringTemplate (ST) and XSL Transformations (XSL-T) address some of these issues, e.g., the first mentioned problem does not occur when using XSL-T.

Especially problems of the second and third category typically produce error messages that only state the invalidity of the document with respect to the expected target language. Unfortunately, the real cause of the problem, i.e., the information about the instantiation data value and its source, are missing in the error message. The loss of this information unnecessarily complicates tracing back the error.

Most of the existing alternative technologies are not easy to use. Template techniques are per se easy to use; however, this advantage is sometimes eliminated by rashly added or too many features. An example for such an impediment that severely violates the ease of use idea is the error handling exposed by JSP if an error from the third category mentioned above occurs: the data inserted during the execution of a JSP document is not checked by the typical JSP engine, resulting in XHTML errors that are shown in the users browser.

### 1.3. Motivating Example

Today's Web applications often make use of a MVC architecture that is similar to that shown in Figure 1.1: the model is represented by a database, the controller is implemented using some middleware like a servlet container and the view is shown to the user in form of XHTML pages in a browser.

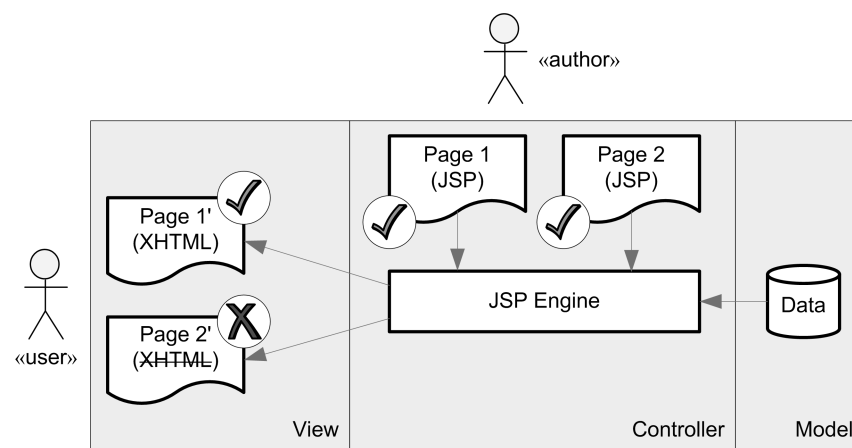


Figure 1.1.: A typical Web Application can produce both valid and invalid XHTML Documents

In the Java world, the servlet container typically used as middleware delivers the XHTML pages by using so-called *Java Server Pages* (JSP, [176]). The JSP documents are translated to XHTML in a multi-step process involving compiling them to Java classes and finally to Java class files. It is the execution of these class files that integrates the model data into the page finally emitted by the component that processes the JSP documents, the *JSP engine*.

As shown in Figure 1.1, there is a typical problem in the outlined scenario. The definition of the JSP language is quite imprecise, it is therefore not possible to check JSP documents in a way that guarantees that the instantiation of a JSP page yields a page conforming to the XHTML (or any other) standard. This enforces the development process that is shown in Figure 1.2.

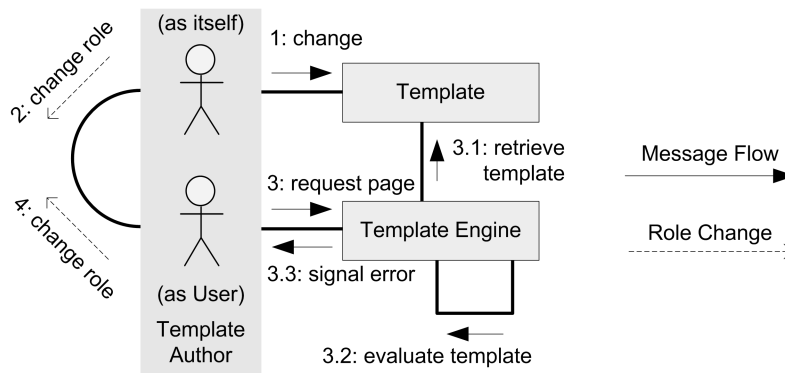


Figure 1.2.: The current Development Process for Templates

All of the three problems with template technologies introduced above, i.e., violation of wellformedness, violation of structural constraints as well as violation of data constraints, can be illustrated with this scenario.

An example for a JSP document causing a violation of wellformedness, i.e., not producing wellformed XML, is shown in Listing 1.1. The document yields a page containing a closing `</h1>` tag without a preceding opening `<h1>` tag. The problem is caused by the different conditions used for the inclusion of the opening and the closing tag. The main cause, however, is that JSP allows interweaving the XML syntax with its own special markup. Several approaches to the problem are possible, e.g., the use of model checkers [66] or control- and data-flow analysis. Interestingly, the problem may be completely solved by a *language design preserving syntax and semantics*. An example for a language which guarantees wellformedness is the XML Template Language (XTL) introduced in Chapter 4.

```

<html xmlns="http://www.w3.org/1999/xhtml">
<%!
    public boolean test1()
    {
        return false;
    }

    public boolean test2()
    {
  
```

## 1. Preface

```
        return true;
    }
%>
<head>
    <title>JSP not producing wellformed XHTML</title>
</head>
<body>
    <% if (test1()) { %><h1><% } %>
    Content
    <% if (test2()) { %></h1><% } %>
</body>
</html>
```

Listing 1.1: A JSP Document failing to produce wellformed XHTML Documents

An example for a violation of a structural constraint is shown in Listing 1.2. This example produces wellformed XML but fails to fulfill the requirements set by the XHTML specification. The `body` tag, which is required within the `html` tag, is included only conditionally, i.e., the document is obviously capable of producing documents not complying to the XHTML specification. The best solution for this type of problem is to disallow required elements to be subject to conditional inclusion in some way. This can be achieved by the newly developed *upfront verification* approach, which allows the verification of structural constraints during the authoring of a template. This approach is based on a technique called *separation of constraints*, which aims at deriving tests that can be applied to templates (like ‘body is not allowed to be subject to conditional inclusion’) and which is described in Section 5.1.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<%!
    public boolean test()
    {
        return false;
    }
%>
<head>
    <title>JSP producing invalid content (1)</title>
</head>
<% if (test()) { %>
<body>Content</body>
<% } %>
</html>
```

Listing 1.2: A JSP Document producing a Document that is not XHTML (1)

Finally, a violation of data constraints is illustrated in Listing 1.3. The problem is caused by character data not complying to a prescribed type. The listing produces an anchor (`<a>` tag) with a `name` attribute with a corresponding value ‘not an NMToken’. This attribute is restricted by the XHTML specification to be of the type `NMToken`, which is not allowed to contain spaces. Thus, the document produced by the JSP file is not valid XHTML. In general, this category of errors is in general impossible to be handled when the template is authored, as the value subject to the typing is instantiation data, which is *per definitionem* only known later at

the point in time when the template is instantiated. Nevertheless, two improvements over the current state of the art are possible.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<%!
    public String getName()
    {
        return "not an NMToken";
    }
%>
<head>
    <title>JSP producing invalid content (2)</title>
</head>
<body>
    <a name="<%= getName() %>" />
</body>
</html>
```

Listing 1.3: A JSP document producing a Document that is not XHTML (2)

First, the handling of errors can be substantially improved compared to what is currently usual in techniques like JSP by checking constraints imposed on the instantiation data within the template engine. This allows for generating error messages of much greater value. For instance, in the mentioned case, the error message could state that the instantiation data to be used as value for the name attribute of the a element does not comply to the expected type NMToken, and it could include the source of that instantiation data value. Current engines do not perform a check at all, which may lead to an error in the user's browser that can in the best case state that the value is not of the expected type NMToken. Unfortunately, this error would in most cases not be recorded in the server environment, which could increase the lifetime of that error considerably. An approach called *Instantiation Data Validation*, which improves the error handling, is described in Section 5.1 as well as in Section 6.3.

Second, if the template is fixed at instantiation time, i.e., can only be changed at build time of the system employing it, the type system of the language using the template engine may be used to assert the validity of the instantiation data. This approach called *Template Interface Generation* is described in Section 6.3.2. It relies on well-known XML binding approaches like Java Architecture for XML Binding (JAXB) and XML $\lambda$ , which have not been coupled with template techniques before.

## 1.4. Goals

The goal of this thesis is to develop a *safe template processing* approach that is easy to use and enables the user to be as safe as possible about the results produced by a particular template. More precisely, a list of five goals has been specified and is described in the following.

The first goal is to come up with a *safe authoring* approach. The approach should give a template author as much confidence about the result of the instantiation of a particular template as early as possible.

## 1. Preface

There are assumptions that must be made during the template authoring, as the validity of an instantiated template may depend on instantiation data which is by definition not available during authoring. Therefore, the second goal is to design a *safe instantiation* process that preserves the semantic information about the real cause of the invalidity of the instantiated template.

As already mentioned, template techniques are often used in order to separate concerns, but some of the existing approaches offer methods to overcome the separation in order to *seemingly* ease the use of the technique. For the approach to be developed, this is not acceptable. Therefore, the third goal is to maintain the *separation of concerns* in template processing, as this separation has proven to be a powerful concept to reduce the complexity of modern software architectures and to enable cooperation of different roles in the software development process.

The fourth goal is *broad applicability*. The approach should not be restricted to a particular application domain like Web engineering, but should also be usable for code generation. This implies that the approach should not be restricted to a particular XML dialect like XHTML.

Finally, the fifth goal is to maximize the *utilization of existing standards* for the approach. This has two major aspects: first, the techniques developed should either rely on existing standards or extend them, and second, an implementation of the core components should rely on available tools for these standards.

The first two goals address the problems introduced in Section 1.2, both goals together form the base for safe template processing. The last three goals further restrict the possible approaches in a way that enables an implementation and the practical use of the developed approach.

## 1.5. Contributions

First of all, an in-depth analysis of existing template approaches has been accomplished. This has led to a number of insights about template techniques that are not explicitly stated in today's literature. A new definition of the term template has been developed, which clearly separates templates from related approaches (transformation techniques, unparsers, aspect-oriented programming). A number of classification criteria has been found that have been used to evaluate existing approaches and to distinguish valuable features from questionable contributions.

In order to assert wellformedness of the result documents, a new universal, syntax- and semantics-preserving template language called XTL has been developed. XTL abandons the wide-spread use of a specific syntax for the markup of the template syntax and relies on the XML namespace [29] concept instead, thereby preserving the syntax and the semantics of the language it is used to instantiate. XTL is a universal language in the sense that it allows any XML dialect to be marked up as a template, it is not specific to a particular XML dialect like XHTML.

In addition to the guarantee on the wellformedness, the developed *upfront verification* approach allows guarantees about the compliance of a resulting document with a given XML Schema. This reduces the risk of undetected errors enormously and enables a new development process for templates, especially in the field of Web application development.

The *InstantiationDataValidation* has been developed to enable *safe instantiation* by simplifying the detection of the root causes for instantiation problems. Furthermore, even more guarantees about the instantiation of templates can be given if the template can only be changed at the build time of a system, i.e., if it can be considered part of the source code of the project using it. In this case, a derivative of the unparser approach called *Template Interface Generation* can be used to statically guarantee the correct types of the instantiation data.

The design criteria applied to XTL and the decisions made during its development are explained in detail. A semantics of XTL has been specified that avoids typical ambiguities which can be found in other template language specifications. This semantics also enables a formal proof of the correctness of the upfront verification approach. The ease of use of XTL is greatly improved by the fact that it does not prescribe a particular language used to describe the data to be inserted into a template, instead, any existing language like the XML Path Language (XPath) or the SPARQL Protocol and RDF Query Language (SPARQL) as well as simple names may be used.

No new approach should fall back behind the work of Terrence Parr [143] and the principles for SoC stated there. Among other contributions, Parr classifies the violations against SoC rules and introduces the *entanglement index* of a template technique as the number of rules violated by it. It turns out that improvements over Parr’s achievements are possible. In contradiction to Parr’s statement that each engine has at least an entanglement index of one, it is also possible to get an entanglement index of zero using a new technique called *partial templatization*.

A prototype for an XTL engine has been developed, supported by the EU project Services for Nomadic Workers (SNOW) and the BMBF project Enabling Model Transformation-Based Cost Efficient Adaptive Multimodal User Interfaces (EMODE). These projects have also been used to validate the practical usability of XTL. This validation clearly indicated that the reason for the ease of use of template languages is the fact that templates closely resemble the documents they are intended to be instantiated to. This resemblance, which also influenced the definition of the term template given in Section 2.1, is called the *prototypical nature* of templates. The extensive absence of a prototypical nature also hinders the wide-spread application of XSL-T even though the XSL-T specification [107] knows a prototypical mode called Simplified Stylesheet Module (SSM).

Due to the fact that the prototypical nature of template languages restricts their generative power (e.g., a purely prototypical XML template language is restricted to produce documents with a depth being a constant multiple of the depth of the template), the introduction of a macro feature became inevitable. The power regained by this feature is lowering the ease of use of the XTL but this is compensated by the shallow learning curve that results from the prototypical approach: users can start purely prototypical and start learning the macro feature when they need it, switching to a transformational style as it is used in XSL-T.

A document marked up using the XTL can also be used to validate other XML documents. In this case, XTL is interpreted as a schema language in the sense also used by XML Schema Definition (XSD) [59; 180; 26]. Again, the prototypical nature of XTL eases the creation of schema documents. The relationship between templates and schemas also shows a close connection between macros and schema types.

The major contributions of this thesis are the design of the universal, syntax- and semantics-preserving slot markup language XTL, the safe template processing technique enabled by the

## 1. Preface

upfront verification approach and the techniques for the safe instantiation, namely instantiation data validation and template interface generation, and the unification of document creation and validation that could be achieved using XTL.

### 1.6. Related Work

Even for the seemingly simple problem of constructing documents from multiple data sources, there is a variety of approaches. The most important and modern approaches will be described in short below.

Template engines are often used in Web engineering and for code generation. JSP [176] is the most-widely known representative in the Java technological space used in Web applications. One of the most advanced and scientifically backed-up template engines used for code generation as well as Web engineering is definitely ST [143].

An alternative approach is the use of a *transformation engine*. This approach is in wide use especially within the area of Web Content Management Systems (CMS). The difference to the template technique is rather subtle, but can be summarized best by describing the transformation approach as a more constructive approach: it composes the data sources by a specialized transformation, which has one of the data sources embedded, as opposed to the template approach, where the composition instructions can be considered as embedded into one data source. In the XML technological space, the most popular transformation language is XSL-T, which has been implemented using different languages.

The use of an *unparser* is a further option in use for the composition of data sources in Web applications. The technique is based on a specific compilation of one of the two data sources into the language that is also used for the composition program afterwards. As an example, if one wants to compose XML documents using Java, a number of tools exist that allow the creation of XML documents using a Java API. The SoC principle is somehow softened here, but could be restrengthened using standard design patterns [74]. Popular unparsers for XML files in Java include XMLBeans [8] and JAXB [105].

*Aspect-oriented Programming (AOP)* is a paradigm used to separate concerns within software engineering artifacts into a so-called core and one or more aspects. A component called aspect-weaver is used to compose the core and the aspects. These aspects can be seen as different data sources that need to be combined, which makes this approach related to template techniques, especially in the area of code generation. A wide-spread implementation of the Aspect-oriented Programming (AOP) approach is AspectJ.

### 1.7. Typographic Conventions

All code listings shown have been shortened to improve readability even at the expense of syntactic incorrectness. For example, in the XML listings, the prolog, the document type declaration as well as obvious namespace declarations have been omitted, even if this could cause some XML processing applications to emit error messages or warnings. The same is true for the package statement in Java listings.



In order to keep the text short and readable, fixed XML prefixes have been used throughout the document to refer to certain XML namespace. An overview of these prefixes is shown in Table 1.1.

Prefix	XML Dialect Namespace URI
<code>cxsd</code>	Constraint XSD CXSD (see Section 5.1.2) <a href="http://research.sap.com/cxsd/1.0">http://research.sap.com/cxsd/1.0</a>
<code>idc</code>	Instantiation Data Constraint Language IDC (see Section 5.1.3) <a href="http://research.sap.com/xtl/idc/1.0">http://research.sap.com/xtl/idc/1.0</a>
<code>s</code>	Namespace used for Result Splitting (see Section 6.2.3) <a href="http://research.sap.com/xtl/splitting">http://research.sap.com/xtl/splitting</a>
<code>xhtml</code> or none	XHTML™1.1 [4] <a href="http://www.w3.org/1999/xhtml">http://www.w3.org/1999/xhtml</a>
<code>xsd</code>	XML Schema [59; 180; 26] <a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>
<code>xsl</code>	XSL Transformations (XSL-T) [36] <a href="http://www.w3.org/1999/XSL/Transform">http://www.w3.org/1999/XSL/Transform</a>
<code>xtl</code>	XTL (see Chapter 4) <a href="http://research.sap.com/xtl/1.0">http://research.sap.com/xtl/1.0</a>

Table 1.1.: XML Namespaces and Prefixes

In the following, no assumptions are made about the encoding of XML documents, which can be declared by the author of the document. As the definition of strings depends on the set of characters available to express them, the symbol  $\mathbb{S}$  is used to denote the set of all strings that can be composed from the available characters, independently of the concrete encoding chosen.

A typewriter font has been used for in-line code snippets like `ContentHandler`, Uniform Resource Identifiers (URI) like `http://www.w3.org/2000/xmlns/` and file names like `XTL.xsd`.

In the index, the numbers of pages containing definitions are printed in **bold**.

## 1.8. Outline

Chapter 2 gives a definition of the term *template* followed by the discussion of the two main application areas of templates and alternative approaches. Classification criteria for template techniques are given. Finally, some ways to emulate complex features with simpler ones are shown.

## *1. Preface*

The safe template processing of XML documents is described in Chapter 3. After a motivating example, the goals of the approach are discussed and requirements for the solution are derived. A solution is proposed, and the building blocks of the solution are discussed in detail.

The design of the generic slot markup language XTL is discussed in detail in Chapter 4. Its instantiation semantics and indications for the correctness of the upfront verification approach are included in this chapter. An alternative use case for a generic slot markup language, namely the validation of documents, is discussed.

Chapter 5 shows the support which the proposed solution offers to a template author. It explains how the template technique is adapted to a particular usage scenario and how the upfront verification approach helps the template author detect mistakes in templates earlier. The chapter gives a proof of the correctness of the safe authoring approach. A further improvement, a technique named partial templatization is sketched.

Chapter 6 shows how templates are instantiated in an efficient, flexible and safe way. Here, efficiency refers to the reasonable consumption of memory and a fast execution. Flexibility refers to the implementation of design decisions that enable the wide-spread use of the proposed approach. Finally, safety means the realization of the adequate error handling introduced as a goal of the proposed approach. A further possible improvement, the Template Interface Generation, is introduced.

The work in this thesis has been validated as described in Chapter 7. This chapter discusses the implementation of the prototype, its application in the EU project SNOW and other projects, and shows the results of performance measurements.

In the final Chapter 8, conclusions are drawn and open research questions resp. directions are given.

# 2

## Introduction

Why is the customer just buying from you? And it is interesting that we had a few examples, for example, a company that is doing drilling machines, and if they sit back and ask themselves, "What does the customer really need? Does he need a drilling machine?" Then the answer is no, he needs holes. The company is now switching to sell holes, which is an entirely different business.

Henning Kagermann, 2006 [104]

This chapter aims at introducing the area of template techniques and at establishing the necessary vocabulary and formal foundations. Therefore, definitions of terms specific to this thesis are given in Section 2.1. Section 2.2 shows typical applications of template techniques. The Sections 2.3 and 2.4 introduce competing and related approaches. Finally, Section 2.5 sets up criteria for the classification of template techniques.

### 2.1. Definitions

Definitions of the term *template* and related terms are given in Section 2.1.1. Section 2.1.2 discusses the life cycles of both template techniques and templates. Section 2.1.3 introduces the formalization of XML documents used within this thesis. Finally, Section 2.1.4 gives a short introduction into the area of XML schema languages and defines a formalization of XML Schema, the most widespread XML schema language, which has been used thoroughly within this thesis.

## 2. Introduction

### 2.1.1. Templates and Related Terms

Finding a definition for the term *template* is a necessary precondition for the separation of template techniques from other code generation approaches. There are multiple ways to approach the term: from an etymological point of view as in [189], in a syntactic way as done in [143] or in a pragmatic sense as in [150] or [197].

In the following, a number of definitions will be given (partly newly expressed and partly taken from existing research) and discussed in order to find a definition that captures the notion of templates as concise as necessary within this thesis. Most importantly, the definition that is the result of the process is well-aligned with an intuitive point of view on templates and separates the template approach from related approaches that are not commonly considered as template techniques. First of all, the language that should be generated using a template technique is defined in Definition 2.1.

**Definition 2.1** ((Expected) Target Language). The *expected target language* or just *target language* is the language  $\mathcal{T}$  that is intended to be produced using a template technique.  $\square$

The Definitions 2.2 and 2.3, explaining the term *template* syntactically or pragmatically, are the starting point for the elaboration of concise definitions.

**Definition 2.2** (Template, *unsophisticated syntactic definition*). Every incomplete textual representation of data is a template.  $\square$

**Definition 2.3** (Template, *unsophisticated pragmatic definition*). A template is a means to compose concerns, i.e., a tool to reverse the separation of concerns.  $\square$

Both unsophisticated definitions cover a lot of approaches that would not be considered as template techniques after a more in-depth analysis. A good example is aspect-oriented programming (AOP, [63]), where a core program is woven together with advices from so-called aspects. For example, the unsophisticated Definition 2.2 would consider the core program as the template that is incomplete and the advice as the data to be filled into the core. However, the very nature of templates is their explicit incompleteness, i.e., the locations where the data is to be inserted in the templates are explicitly marked in the template itself. The following definition, which is an adapted version from [143], captures this aspect very well.

**Definition 2.4** (Template according to [143]). An unrestricted template,  $t^\circ$ , is an alternating list of output literals,  $t_i$ , and action expressions,  $e_i$ :

$$t_0 e_0 \dots t_i e_i t_{i+1} \dots t_n e_n$$

where any  $t_i$  may be the empty string and  $e_i$  is unrestricted computationally and syntactically. If there are no  $e_i$  in  $t^\circ$ , then  $t^\circ$  is just a single literal  $t_0$ .  $\square$

The last sentence in Parr's definition seems to try to emphasize another important property of templates: a document in the language that the template technique is supposed to produce is also considered a template. This aspect is called the *prototypical nature of templates*. Unfortunately, it is not really enforced by the definition, so documents using transformational approaches like an XSL-T stylesheet would fall under this definition. Parr explicitly states that XSL-T is not captured by his definition:

Interestingly, by this definition of template, XSL-T style sheets are not templates at all because style sheets specify a set of XSL-T tree transformations whose emer-

gent behavior is an XML or XHTML document. XSL-T style sheets are programs like servlets, albeit declarative in nature rather than imperative. [143]

This rationale is based on the semantics of XSL-T, but the definition relies on syntactic properties that are well satisfied by XSL-T. For this reason, the next definition explicitly expresses the prototypical nature of templates, which clearly rules out XSL-T by defining templates via the term *template language*. To accomplish this, the final Definition 2.5 roots the term template on the term *template language*, shifting the responsibility to capture the prototypical nature to the definition of the latter term.

**Definition 2.5 (Template, final).** A *template*  $t^\circ$  is a sentence from a template language  $\mathfrak{T}^\circ$ .  $\square$

Prerequisites for defining the term template language are furthermore the definitions of slots and slot markup languages.

**Definition 2.6 (Slot).** A *slot* is an area of variability in a document.  $\square$

It is important to note that the term document has been used in Definition 2.6 to avoid cyclic references between this definition and the definitions using it. This way of defining slots has the additional benefit of also capturing slots introduced in documents for other purposes, i.e., the definition also matches the incompleteness in a form intended to be filled out by humans. Based on Definition 2.6, it is possible to give the following definition for a slot markup language.

**Definition 2.7 (Slot Markup Language).** A *slot markup language*  $\mathfrak{S}$  is a non-empty set of features to denote slots within a document.  $\square$

This definition explicitly states that the markup of slots appears within the document itself, thereby excluding other methods for the designation of slots like pointcut languages used in AOP approaches [172]. This is also the motivation for the term *non-empty* in the definition: it prevents AOP approaches from being captured by the following definitions through just defining  $\mathfrak{S} = \emptyset$ . Elements from the slot markup language are also called *instructions* in the following.

The locations of slots and the locations of slot markup language sentences in a template may differ for several reasons. First, there may be imperative constructs in the slot markup language (like `for` loops) that mark their content as repeatable in an instantiated document (and thus define slots), but are not necessarily placed exactly at the location of a slot. Second, there may be slot markup in which the location of the slot markup is by design different from the location where the instantiation data should be inserted.

Given the definition of slot markup languages, the following more elaborated definition for the term template language can be given.

**Definition 2.8 (Template Language, elaborated).** A template language is the language produced from a target language and a slot markup language such that

1. each sentence from the target language is in the template language and
2. each sequence of literals from the target language interspersed with sentences from the slot markup language is in the template language.  $\square$

Unfortunately, the former definition captures only the syntactical aspect of the prototypical nature. In order to be concise, it is necessary to include the semantics of the prototypical nature: if a template that is actually a sentence from the target language is instantiated, it remains unchanged. In order to fix this problem of the definition, it is necessary to give a definition of the

## 2. Introduction

term *instantiation* as well. This definition also includes the definition of the term *instantiation data*.

**Definition 2.9** (Instantiation and Instantiation Data). *Instantiation* is the application of a function *instantiate* with  $\text{dom}(\text{instantiate}) = D \times \mathcal{T}^\circ$  that transforms a template  $t^\circ$  into  $\text{instantiate}(d, t^\circ)$  by replacing and thereby removing its slots. The data  $d \in D$  consumed during the slot replacement is called *instantiation data*.  $\square$

Obviously, the range of the function *instantiate* introduced above has remained unspecified. This range is the so-called *actual target language* defined below:

**Definition 2.10** (Actual Target Language). The *actual target language* is the range of the instantiation function *instantiate*, i.e.,  $\text{ran}(\text{instantiate})$ .  $\square$

The relation between actual and expected target language is discussed thoroughly in Chapter 3. Given the definitions of the expected target language and the instantiation function, the semantics of the prototypical nature of templates can be captured by the following formal definition of the term *template language*:

**Definition 2.11** (Template Language, final). Let  $T_{\mathcal{T}}$  be the set of terminal symbols from the target language  $\mathcal{T}$ , let  $\mathcal{S}$  be a non-empty slot markup language and let *instantiate* be an instantiation function. Then the template language  $\mathcal{T}^\circ$  is the smallest language constructed from the target language and the slot markup language  $\mathcal{S}$  such that the following conditions are satisfied:

$\forall t \in \mathcal{T} : t \in \mathcal{T}^\circ \wedge \text{instantiate}(d, t) = t$  for arbitrary instantiation data  $d$   
*(target language documents are templates and instantiate into itself)*

and

$\forall n \in \mathbb{N}, l_i \in T_{\mathcal{T}} \cup \{\epsilon\}, b_i \in \mathcal{S} \cup \{\epsilon\}, 0 \leq i \leq n : l_0 b_0 \dots l_i b_i l_{i+1} \dots b_n l_n \in \mathcal{T}^\circ$

*(templates are constructed from target language terminal symbols and slot markup language sentences)*  $\square$

This definition is still capturing a lot of approaches that include features that would not be considered *good* ideas (neither in an academic nor in a practical sense), however such approaches exist and are template techniques. In Figure 2.1, a comparison of the definitions in terms of captured popular approaches is shown.

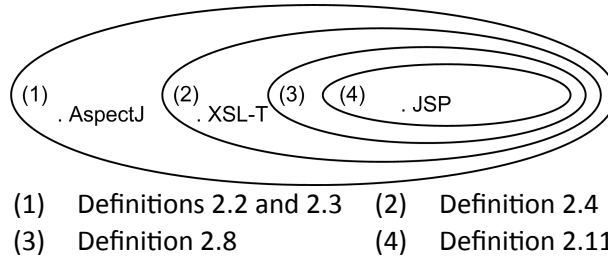


Figure 2.1.: Comparison of the Scopes of the Definitions of the Term *Template*

An important notion is also the term *template engine* as defined below:

**Definition 2.12** (Template Engine). The component<sup>1</sup> responsible for the instantiation of templates is called *template engine*.  $\square$

<sup>1</sup>The term *component* is used here and in the following in the sense defined in [51].

The relations between the terms just defined are illustrated in Figure 2.2. Basically, the transition from a document in the target language into a template is done by the introduction of elements from the slot markup language, whereas the instantiation transforms templates into documents in the actual target language.

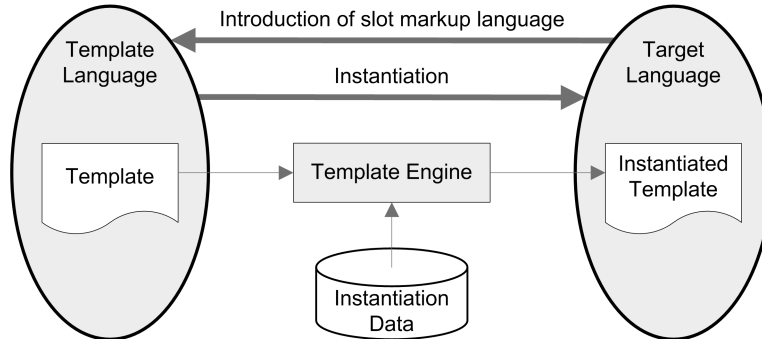


Figure 2.2.: Relations between Template and Target Language

Most template engines do not fully use the power enabled by the Definitions 2.11, they instead restrict the template language further, most notably by constraints on the nesting of slot markup language instructions.

A slot markup language typically comprises a second language that is used to refer to instantiation data, defined below. There are several ways to classify these so-called query languages—one classification is given in Section 2.5.5.

**Definition 2.13 (Query Language).** The part of the slot markup language used to identify or fetch instantiation data is called *query language*. □

```
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- ... -->
  <body>
    <xsl:for-each select="purchaseOrder/items/item">
      <p>
        <xsl:value-of select="@partNum" />
      </p>
    </xsl:for-each>
  </body>
</html>
```

Target language

Slot markup language

Query language

Listing 2.1: Origins of Fragments in a Template

As a template language is obviously a composition of the target language with the slot markup language, which itself incorporates instructions from the query language, it is possible to show the origin of fragments in a template as it is done in Listing 2.1. This listing shows a template

## 2. Introduction

producing Hypertext Markup Language (HTML) as the target language, using XSL-T as the slot markup language, which itself includes XPath as query language.

Please note that the listing shown is not an XSL-T stylesheet (as this would, by definition, not be a template), but rather an XSL-T *simplified stylesheet module* (SSM). These modules offer a prototypical use of the XSL-T language by allowing to embed a subset of XSL-T into target language documents.

### 2.1.2. Life Cycle Phases

To understand the following discussions and the overall structure of the thesis, both the life cycle of a template technique and the life cycle of a template as a document need to be defined and divided into phases. As the life cycle of a template is preceded by the life cycle of the underlying template technique, both life cycles are introduced together.

In Figure 2.3, the combined life cycles of a template technique and adhering templates is shown. For the scope of this thesis, there is no need to introduce end-of-life phases for template techniques or templates, so the life cycle ends with the use of the template technique or the validation of an instantiated template, respectively.

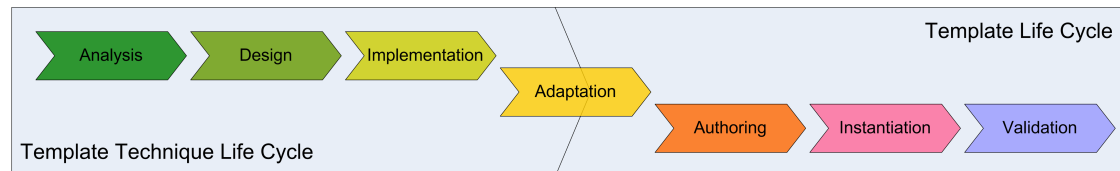


Figure 2.3.: Template Technique and Template Life Cycle

The first phase in the life cycle of a template technique is typically the *analysis phase* in which the goals and requirements induced by the scenario in which the technique should be used are captured. For the technique developed within this thesis, the goals are discussed in Section 3.1 and the requirements are introduced in Section 3.2.

After the analysis phase, the *design phase* typically proposes a solution fulfilling the goals and requirements found. This phase typically involves proposing features of the template engine that directly influence the design of the slot markup language. In this thesis, an architecture of a solution is sketched in Section 3.3, whereas the major sub-activity, the design of the slot markup language is described in detail in Chapter 4.

Obviously, the *implementation phase* consists of the actual development activities needed to create the software that implements the design created by the preceding phase. In this thesis, issues regarding the implementation phase have been split up and are contained in the Chapters 5 and 6.

For template techniques that are not fixed to support a single target language, an *adaptation phase* may be necessary which occurs between the life cycles of the technique and the templates. The proposed solution is an example for a technique that involves such an adaptation step—it is described in detail in Section 5.1.

The first phase in the life cycle of a template is the *authoring phase*, also referred to as *authoring time*. The person playing the role of the *template author* creates the template using some



tool, which may be a simple text editor or a sophisticated development environment with advanced features like syntax highlighting or text completion. The template validation that is part of the proposed solution and supports the author in creating correct templates is described in Section 5.2.

After the template has been authored, it is typically used within the *instantiation phase* to create target language documents. The solution elements addressing this phase are described in the Sections 6.1, 6.2 and 6.3.

Finally, some engines include a post-instantiation *validation phase* that checks whether the instantiated template conforms to the target language. The proposed solution guarantees the conformance of the instantiated template with the target language in other ways, so no part of this thesis is corresponding to this phase.

### 2.1.3. The Extensible Markup Language XML

XML is a general purpose markup language that evolved from SGML and has been published as a World Wide Web Consortium (W3C) recommendation 1998 [28]. The term *extensible* highlights the fact that XML allows the definition of arbitrary new languages, which are typically called *XML dialects*. A large number of XML dialects exist today, well-known XML dialects include, for example, XHTML [4], the Wireless Markup Language (WML) [90] and the Scalable Vector Graphics (SVG) language [61].

The mentioned specification [28] of XML defines only the concrete syntax of XML documents, whereas its abstract syntax, the so-called *XML information set*, is defined in [42]. A very simple XML document is shown in Listing 2.2.

```
<?xml version="1.0" encoding="UTF-8"?>
<address country="US">
  <name>Alice Smith</name>
  <street>123 Maple Street</street>
  <city>Mill Valley</city>
  <state>CA</state>
</address>
```

Listing 2.2: A simple XML file

In the following, a formalization for XML documents is given. Some low-level restrictions that are, for example, imposed by W3C specifications (e.g., the restriction that a namespace URI must not be the empty string in [29]) or by IETF specification (e.g., the syntactical structure of a URI defined in [23]) are omitted from the formalization for simplicity reasons. Thus, in the following, *NCNames* (see [29]) and URIs are modeled as strings.

Furthermore, the formalization does not consider the namespace prefixes, as they are only a syntactic simplification for the namespace binding of names. As a consequence, the term qualified name is used to denote what is called an expanded name in [29], i.e., as a tuple of a namespace URI and a local name.

**Definition 2.14.** Assume a set  $E$  of qualified names for elements, a set  $A$  of qualified names for attributes and a symbol  $\top$  indicating text. An XML document is defined to be  $D = (V, v_\bullet, \text{label}, \text{children}, \text{attr}, \text{value})$  where:

## 2. Introduction

- $V$  is a finite set of nodes.
- $v_\bullet$  is a distinguished node in  $V$  called the root node of  $D$ .
- $\text{label} : V \mapsto E \cup \top$  is a total function that maps each node to either the qualified name of an element or the symbol  $\top$ .
- $\text{children} : V \mapsto V^*$  is a total function that maps a node  $v \in V$  to a sequence of nodes  $v_0, \dots, v_{n(v)}$  such that
  1. no node occurs twice in the sequence:  
 $\forall 0 \leq i < j \leq n(v) : v_i \neq v_j,$
  2. every node besides  $v_\bullet$  has exactly one parent, whereas  $v_\bullet$  has none:  
 $\forall v \in V : \text{card}\{v' \mid \text{children}(v') = v_0, \dots, v, \dots, v_{n(v')}\} = [v = v_\bullet]$  and
  3. no cycles exist:  
 $\forall v \in V : v_p = \text{parent}(v) \Rightarrow v \notin \text{parent}^*(v_p)$   
 where  $\text{parent} : V - \{v_\bullet\} \mapsto V$  is the total function defined as follows  
 $\text{parent}(v) = v' \Leftrightarrow \text{children}(v') = v_0, \dots, v_i, v, v_j, \dots, v_{n(v')}.$
- $\text{attr} : V \times A \mapsto \text{String}$  is a partial function that is only defined for  $v \in V$  with  $\text{label}(v) \in E$ ,
- $\text{value} : V \mapsto \text{String}$  is a partial function that is only defined for  $v \in V$  with  $\text{label}(v) = \top$ .  $\square$

The helper function  $\text{hasAttr} : V \times A \mapsto \text{boolean}$  is true for a node  $v$  and a qualified name  $a$  of an attribute if and only if  $\text{attr}(v, a)$  is defined.

Given the XML document in Listing 2.2, Definition 2.14 yields a document  $D = (V, v_\bullet, \text{label}, \text{children}, \text{attr}, \text{value})$  with  $V = \{v_\bullet, v_1, v_2, v_3, v_4, v_{11}, v_{21}, v_{31}, v_{41}\}$  and the functions  $\text{label}$ ,  $\text{children}$ ,  $\text{attr}$  and  $\text{value}$  as shown in Figure 2.4.

$\text{label}(v_\bullet) = \text{"address"}$   
 $\text{label}(v_1) = \text{"name"}$   
 $\text{label}(v_2) = \text{"city"}$   
 $\text{label}(v_3) = \text{"street"}$   
 $\text{label}(v_4) = \text{"state"}$   
 $\text{label}(v_{11}) = \dots = \text{label}(v_{41}) = \top$   
 $\text{attr}(v_\bullet, \text{"country"}) = \text{"US"}$   
 $\text{value}(v_{11}) = \text{"Alice Smith"}$   
 $\text{value}(v_{21}) = \text{"123 Maple Street"}$   
 $\text{value}(v_{31}) = \text{"Mill Valley"}$   
 $\text{value}(v_{41}) = \text{"CA"}$

$\text{children}(v_\bullet) = v_1, v_2, v_3, v_4$   
 $\text{children}(v_1) = v_{11}$   
 $\text{children}(v_2) = v_{21}$   
 $\text{children}(v_3) = v_{31}$   
 $\text{children}(v_4) = v_{41}$

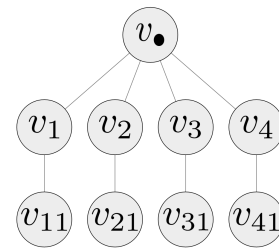


Figure 2.4.: Formalization of the XML document in Listing 2.2

The handling of whitespace (i.e., spaces, tabs and blank lines, see the definition of the non-terminal  $S$  in [28]) in XML documents is a rather complicated issue. In the following, the so-

called *ignorable whitespace* is completely omitted from the formal representation of XML instances. This greatly eases the readability of statements made against these formal representations. It should be clear that implementations of the techniques described using these formalizations must reconsider the applicability of this restriction, as such ignorable whitespace can be meaningful, e.g., if the indentation of the document must be considered.

#### 2.1.4. XML Schema Languages

As XML itself can be used to create markup languages called XML dialects, there is a need to capture vocabularies and rules of XML dialects. In the following, the term *schema* is used for any document which describes the permitted names for elements and attributes together with their permissible structure and values in an (in most cases, infinite) set of XML documents. Typically, a schema defines only the syntactic structure (and some static semantics) on top of the common XML grammar (as defined using EBNF in [28]).

Languages used to write schemas are called *meta languages*, or (more common in the XML community) *schema languages*. As the number of schema languages has grown to a considerable amount [98], the classification of schema languages is the subject of several publications [130; 116; 131].

One of the most important classification criteria is that between grammar- and pattern-based schema languages [116]. Grammar-based schema languages rely on tree grammars for the specification of the document structure, whereas pattern-based schema languages specify a number of patterns that are interpreted as properties that must be fulfilled by complying documents.

Grammar-based schema languages can be further divided by the formalism underlying the tree grammar. [131] classifies these formalisms to be either *regular*, *restrained-competition*, *single-type*, or *local* tree grammars. The expressive power of these languages is shown as a Venn diagram in Figure 2.5.

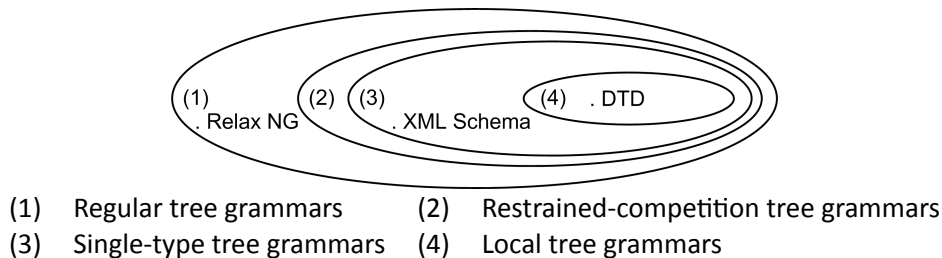


Figure 2.5.: Classification of Schema Languages [simplified, based on 131]

Typical schema languages are neither purely grammar- nor pattern-based. As an example, XML Schema [59; 180; 26], one of the most widely used schema languages defined by the W3C, is based on a single type grammar and adds pattern-based features for the specification of static semantics (so called *identity constraints*). Please note that in this thesis, the lower-case term *schema* refers to a schema in some, possibly unspecified, schema language, whereas the upper-case term *Schema* always refers to a schema specified using the XML Schema language.

## 2. Introduction

Other important schema languages are Document Type Definition (DTD), a schema language defined along with XML in [28], the Regular Language for XML Next Generation (RelaxNG), a regular tree grammar-based language [39] and Schematron, a pattern-based language [99].

A barely considered property of schema languages is their ability to express so-called attribute/element constraints [132]. An example for an attribute/element constraint is the statement „element *a* must carry an attribute named *b* or have a child element named *c*“. Constraints of this kind play a role in the definition of the template language XML. Unfortunately, attribute/element constraints can only be expressed by RelaxNG and Schematron, but not by XML Schema.

A number of formalizations of XML Schema can be found in today’s XML literature. The intent and completeness of these formalizations vary widely. The most general formalization of XML Schema as a so-called *XGrammar* can be found in [123]. The XGrammar concept is intended to formally capture most of the concepts in DTDs, XML Schema and RelaxNG. Other formalizations are restricted to a particular aspect of schema languages, like content models [168]. The Model Schema Language (MSL) is a formalization solely intended to specify the semantics of XML Schema [30; 31]. As there is no need to use a feature-complete specification in the following, a modification of the XGrammar concept is used.

In order to define the term schema, it is useful to give a formal definition of a simple type first. Simple types are types that can be used to validate character data, e.g., attribute values or text nodes [180]. The most prominent definition of a set of simple types is the library defined as part of XML Schema [26]. XML Schema also allows the definition of custom simple types. For the following, it is unnecessary to define how simple types are defined and how validation is performed. Therefore, Definition 2.15 is a very simplistic definition of simple types.

**Definition 2.15 (Simple Types).** A simple type  $\sigma$  is a possibly infinite subset of the set of strings, i.e.,  $\sigma \subseteq \mathbb{S}$ . A string  $s$  is said to be conforming to the simple type  $\sigma$ , if and only if  $s \in \sigma$ .  $\square$

In the following, a simple type will often be denoted by a qualified name  $q$ . Instead of writing  $s \in q$ , a predicate  $\text{Valid}(s, q)$  is used, which is true if and only if  $s \in q$ . Furthermore, *string* is used to denote the XML Schema’s string type `xsd:string` as defined by [26, Section 3.2.1]. Since `xsd:string` is syntactically unrestricted,  $\text{Valid}(s, \text{xsd:string})$  is true for all  $s \in \mathbb{S}$ .

A schema can be defined as in Definition 2.16. While this definition closely resembles the definition of an XGrammar [123], it differs from the latter in various ways. Most notable, the distinction between hedge and tree types has been removed. This is possible because in XML Schema, the appearance of hedge types—in XML Schema called model groups—is purely syntactic, because model groups are not allowed to be circular [180, Section 3.8.6]). Second, the distinction between element and attribute production rules has been omitted, because within the original definition of an XGrammar, the semantics of a derivation from the grammar was insufficiently specified. Finally, the terminal symbols have been divided into qualified names for attribute and elements.

It is necessary to point out that this definition of a schema is not exactly capturing the XML Schema language. On one hand, a schema, as defined in Definition 2.16, must fulfill further constraints in order to represent a valid XML Schema, e.g., its content models must comply to the Unique Particle Attribution (UPA) rule [180, Section 3.8.6]. The UPA rule demands the unambiguousness of the content models with respect to a parser without lookahead, making

XML Schema a restrained-competition tree grammar in the sense of [131]. On the other hand, some XML Schema documents can not be represented by a schema as defined below, e.g., if the XML Schema uses identity-constraint definitions [180, Section 3.11].

**Definition 2.16 (Schema).** A *schema* is the tuple  $S = (\Sigma, N, E, A, N_\bullet, R)$  where

- $\Sigma$  is a set of simple data types,
- $N$  is a set of non-terminal symbols,
- $E$  is a set of qualified names for elements,
- $A$  is a set of qualified names for attributes,
- $N_\bullet \subseteq N$  is the set of start symbols,
- $R$  is a set of production rules of the form  $X \rightarrow e(\text{ad}^*, \text{cm})$  where  $X \in N$ ,  $e \in E$ ,  $\text{ad}^* \subseteq \text{AD}(\Sigma, A)$  is a set of attribute declarations as defined in Definition 2.17 and  $\text{cm} \in \text{CM}(\Sigma, N, E)$  is a content model as defined in Definition 2.18.

The set of XML documents valid with respect to the schema  $S$  is denoted by  $\mathcal{L}(S)$ .  $\square$

Attribute declarations [180, Section 3.2] are defined in Definition 2.17. This definition focuses on the essentials of attribute declarations: a name, a type, whether the attribute is required or optional and whether it has been assigned a fixed value. All other aspects of attribute declarations have been omitted.

**Definition 2.17 (Set of attribute declarations).** The set of attribute declarations  $\text{AD}(\Sigma, A)$  is defined for the set of qualified names for attributes  $A$  and the set of simple data types  $\Sigma$  as the set of all tuples  $(a, \sigma, i, f)$  consisting of:

- $a \in A$  is a qualified name of an attribute,
- $\sigma \in \Sigma$  is a simple type to which the attribute value must comply,
- $i \in \{0, 1\}$  is the required cardinality of the attribute, and,
- $f \in \{\text{true}, \text{false}\}$  defines whether a fixed value is assigned to the attribute.  $\square$

A shorthand notation is defined, which allows to extract all attribute declarations that declare required attributes:  $\text{req}(A) = \{(a, \sigma, i, f) \in A \mid i = 1\}$ . Analogously, all attribute declarations that have been assigned a fixed value are denoted by  $\text{fixed}(A) = \{(a, \sigma, i, f) \in A \mid f\}$ .

Content models are defined in Definition 2.18. Again, this definition is not completely congruent with the XML Schema specification. On the one hand, it is missing the `xsd:all` model group [180, Section 3.7]. On the other hand, as this definition basically allows regular expressions as content models, it also allows content models that are not valid in XML Schema, but would be allowed in regular schema languages like RelaxNG. An example would be content models that define sequences of simple types mixed with elements, which is not specifiable in XML Schema. The definition still fulfills its goal, i.e., it allows XML Schema instances to be captured formally.

**Definition 2.18 (Set of content models).** The set of content models  $\text{CM}(\Sigma, N, E)$  over the set of simple types  $\Sigma$ , the set of non-terminal symbols  $N$  and the set of qualified names for elements  $E$  is defined recursively as follows:

- The empty sequence  $\epsilon$  is a content model:  $\epsilon \in \text{CM}$ .
- All simple types are content models:  $\forall \sigma \in \Sigma : \sigma \in \text{CM}$ .
- All non-terminal symbols are content models:  $\forall n \in N : n \in \text{CM}$ .
- All element names are content models:  $\forall e \in E : e \in \text{CM}$ .

## 2. Introduction

- For two content models  $cm_1 \in CM$  and  $cm_2 \in CM$ , the results of the following operations are also content models, i.e.,
  - $cm_1, cm_2 \in CM$ , meaning a sequence consisting of the two content models,
  - $cm_1 | cm_2 \in CM$ , meaning a choice between the two content models, and,
  - $cm_1\{i, j\} \in CM$ , meaning a repetition of the content model  $cm_1$ , where  $i \in \mathbb{N}$ ,  $j \in \mathbb{N}^+ \cup \{*\}$ , and  $j \neq * \Rightarrow i \leq j$ , with the special symbol  $*$  meaning unrestricted repetition.  $\square$

As shorthand notations,  $cm?$  is used to denote  $cm\{0, 1\}$  and  $cm^*$  to denote  $cm\{0, *\}$ .

## 2.2. Applications

Templates are ubiquitous. For reasons of simplicity, the two most prominent application classes are described in more detail below: the use within dynamic Web applications and for the generation of code. Other examples are easy to find on almost every desktop computer, e.g., the insertion of fields in OpenOffice documents [141]. Templates are also frequently used as a starting point for the creation of static Web pages, e.g., in tools like JAlbum [92].

### 2.2.1. Web Applications

Web applications are one of the major application areas where template techniques are used. With the advent of dynamic Web applications it has become more and more obvious that the separation of layout and content is vital for the maintenance of large and complex Web systems. This soon has led to the creation of languages providing features that allow the easy composition of documents for the Web (e.g., HTML or later XHTML pages). An early example for these languages is Perl [188], which facilitates the creation of Web pages using its variable interpolation feature.

The increasing number of Web sites containing user-generated content (paraphrased with the term *Web 2.0*) also creates new challenges for template techniques. One of the most important features today is the isolation of user-generated content from the surrounding page skeleton to satisfy minimum security requirements that are requested by users and operators of a Web site [101].

Template engines that can typically be found in today's Web applications include the Java-based engines JSP [176], Velocity [176], ST [173; 143], the PHP-based engine Smarty [165], the Python-based engine Document Template Markup Language (DTML) [114] and the Ruby-based engine Embedded Ruby (ERB) [50].

### 2.2.2. Code Generation

Code generation as a discipline started with the early work on compilers [79]. In the following, code generation should be understood as limited to produce textual representations of code. Template techniques have also been introduced into compiler construction—for an example see [171].

A recent event that fostered the generation of code was the advent of the Model Driven Architecture (MDA) [128], which led to an ever increasing number of models for all kinds of domains. A definition for the term model in the sense of the MDA can be found in [24]:

A model is a simplification of a system built with an intended goal in mind[...]. The model should be able to answer questions in place of the actual system.

A significant portion of these models is still used mainly for code generation [118]. MDA distinguishes between model-to-model transformations (M2M) and model-to-code transformations (M2C). The former typically transform higher level abstractions into each other whereas the latter typically transform from higher level abstractions into textual models, esp. into code. Obviously, the M2C transformations are a natural domain of template techniques.

M2M transformations are a typical domain of transformation languages (see Section 2.3.1), but in some cases, template techniques are used for M2M transformations. For example, the M2M transformation facility in the UML modeling tool *Enterprise Architect* is based on a template approach: templates are instantiated to create a textual specification of the target model which is afterwards parsed into some object representation of the model [166].

As already pointed out in [43], nearly all of the available MDA tools support some kind of template-based code generation. Template techniques that are in wide-spread use for the generation of source code are the Java-based engines Java Emitter Templates (JET) [148], XPAND [56] and ST [173; 143] as well as the Python-based Cheetah [158] engine.

## 2.3. Alternatives to Using Templates

There are several alternatives to using templates as a means for composing documents from multiple data sources that have been divided to achieve a separation of concerns. In the Sections 2.3.1, 2.3.2 and 2.3.3, the most widespread alternatives, i.e., transformation techniques, aspect-oriented approaches, and unparsers, are discussed.

### 2.3.1. Transformations

Transformation techniques are an established way to adapt data from one metamodel to another. This is supported by a domain-specific language used to specify a transformation program. Many languages that adhere to different programming paradigms have been used as a basis for transformation languages, e.g., SML [20], Ruby [68] and Prolog [80]. Transformation techniques are very popular in the Web engineering area, whereas examples for code generation using transformation technologies can hardly be found.

The most prominent transformation language used in the area of Web applications (especially in Web CMSs) is XSL-T. XSL-T is capable of transforming XML documents between XML dialects, but also into text and HTML documents. A lot of research has been performed on XSL-T. It turned out that XSL-T is in fact a functional programming language [135]. XSL-T has been criticized for its verbosity [68] and for its questionable usability [162]. With Xalan [10] and Saxon [106], two mature implementations are available.

## 2. Introduction

XSL-T has sometimes been perceived as a template technique. As has been argued in Section 2.1, this is not true, as XSL-T stylesheets are lacking the prototypical nature that is a vital property of template approaches. However, it is important to note that the XSL-T specification defines an alternative mode for the use of XSL-T in so-called SSMs, which are true templates with embedded XSL-T instructions.

The difference between the transformational and the template approach in the implementation of the view role in the MVC pattern is discussed in detail in [67], which introduces the approaches as patterns entitled *transform view* and *template view*, respectively.

An example for code generation using a transformation technique can be found in [25], where XSL-T has been used to generate Java from the XML Metadata Interchange (XMI) representation generated by a Meta-Object Facility (MOF)–based modeling tool.

### 2.3.2. Aspect-Oriented Approaches

The idea of aspect-orientation, which resulted in its own programming paradigm AOP, came up in the late 1990ies [108]. The primary idea of AOP is to separate the code into a *core* and several *aspects*. The core contains the main functional part of the program to be built, whereas the aspects implement several other functional or non-functional enhancements. Contrary to the transformation approaches above, aspect-oriented approaches are more often utilized for code generation than in Web engineering.

Technically, the aspects are implemented as *advices* and introduced into the core at sets called *pointcuts* of core code locations called *join points*. The introduction process is called *weaving*. There are several variations of this principle, that differ in the time the weaving is done, in how join points and pointcuts are selected, and so on. The notion of pointcuts allows the modularization of cross-cutting concerns, keeping together code fragments that are inherently related and would otherwise be tangled with the code at several locations.

Aspect-oriented code generation is very similar to template-based composition approaches. The core code can be seen as a template without explicit slots, and the advices can be seen as instantiation data. The major difference is that the linking between the two inputs is reversed: the advices are typically combined with the pointcuts, i.e., the instantiation data is bundled with the information where it should be inserted. This property of the core—not to contain explicit marks for the embedding of aspect code—is called *obliviousness*.

One could consider aspect-orientation as a template technique by aligning the linking directions found in aspect orientation (i.e., from aspect to core) and template approaches (i.e., from template to instantiation data), i.e., one could compare the aspect with the template (and the core with the instantiation data). This point of view violates Definition 2.11, as the core has the prototypical nature required from templates in the definition.

A problem with obliviousness is implementing repetition and conditional inclusion of parts of the core code. It is very easy to implement a loop with parametrized content in a template approach, whereas with aspect-orientation, this is typically not possible. Workarounds include the options to remove parts from the core code conditionally or to add to lists of grammatical elements.

An advantage of aspect-orientation over template techniques is the handling of crosscutting concerns. An advice can be applied implicitly against many code locations, e.g., using a regular



expression matching method names, whereas in template approaches, the embedding of the same instantiation data item at multiple places must be declared explicitly in the template.

Aspect-orientation has also been used within the area of Web engineering. An example use case is the localization of Web documents. Typically, such Web documents are prepared by native speakers and then translated into different languages by others. This translation does not change the structure of the document, it rather replaces content (i.e., text elements or attribute values in an XHTML document), therefore there is no need for the conditional or repeated inclusion of content. Using an aspect-oriented technique, the translations could be bundled with pointcuts declaring where to put them in the original documents to replace the content. This approach has the advantage that no templates must be created and maintained in order to translate the Web documents [47; 149].

Another example for the use of aspect-oriented approaches for the manipulation of XML documents are AspectXML [127] and the various approaches for XML update languages like XUpdate [115]. Both example languages use XPath to express pointcuts.

#### 2.3.3. Unparsers

Unparsers are another common option for code generation and Web applications. While a parser builds an object structure from a sentence of some grammar, an unparser reverses the process: it transforms some object structure into a sentence. The object structure can be a Concrete Syntax Tree (CST) or an Abstract Syntax Tree (AST). Other names for unparsers are serializers (e.g., used in Xerces [11]), API-based generators [169] or intra-level transformations [113, Figure 4.1c on page 71]. In the XML technological space, unparsers are typically called XML binding tools.

In the XML community, unparsers are also called XML binding frameworks, with JAXB [155; 105] and XMLBeans [8] being their most prominent representatives. XML binding frameworks typically work in two steps. First, there is a compilation step at build time which compiles an XML Schema into a corresponding object model in a target language, leveraging the type system of the target language as much as possible to reflect the constraints of the schema. Second, there are marshalling resp. unmarshalling steps at run time, which translate between XML documents and the created target language object model (or vice versa). Please note that marshalling corresponds to unparsing whereas unmarshalling corresponds to parsing. For more details and an overview and comparison of existing approaches see [111].

Comparable binding frameworks exist for other combination of languages, e.g., Jenerator for the generation of Java code from Java programs [187] and the RAP Widget Toolkit (RWT) for HTML generation from Java. The approach used by XMλ [125] is very similar, except for one difference: its type system has been designed to match that of XML.

The object model needed by an unparser to generate a document is often constructed using the visitor pattern [74] from another, existing object model. This is different from the template approach, where the template engines (e.g., XPAND [56]) often supply powerful mechanisms to visit such a model.

### 2.3.4. Comparison of Templates with Alternative Technologies

Figure 2.6 contains a comparison of four technologies that can be used to achieve a Separation of Concerns. One of the technologies, JSP, represents a template technique, the other ones, XSL-T, AspectJ, and JAXB, represent the alternative technologies introduced above, namely transformations, aspects-oriented approaches and unparsers, respectively. The four rightmost columns in the figure show (from left to right): the language of the first concern, the language in which the relation between the concerns is expressed, the language of the second concern and the language of the result, i.e., the language into which the concerns are combined. Obviously, the first and the second concern are interchangeable. Therefore, the language that is more closely related to the language of the result has been chosen as the first concern. The rectangles spanning the columns indicate how the concerns resp. the relating language are bundled together (typically in a file).

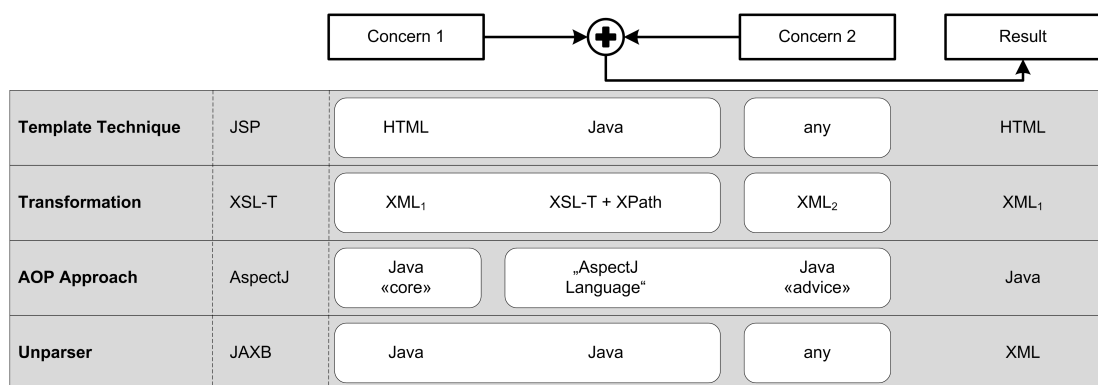


Figure 2.6.: Comparison of the Alternatives with Templates

JSP is typically used to create HTML from a JSP page, which is basically an HTML document with embedded Java instructions. As Java is a general-purpose language, the second concern can be any source of data. Taken together, the fact that the Query Language is bundled with the first concern and the fact that a standalone HTML document is a valid JSP file, make JSP a template technique.

XSL-T combines two XML languages, XML<sub>1</sub> and XML<sub>2</sub> into a resulting XML dialect. In most cases, this dialect is one of the input XML dialects, therefore named XML<sub>1</sub> w.l.o.g. here. However, in all cases in which XSL-T is used in its standard form (i.e., not as XSL-T SSM) and if the XML dialects are not XSL-T itself (an unusual, but not negligible case), XSL-T is not prototypical, i.e., no document from one of the XML dialects is itself a template.

In AspectJ, both concerns are documents or fragments in the Java language. The concern entitled *core* is typically more similar to the result than the concern named *advice*, as the latter concern is typically spread throughout the core concern (a phenomenon called *crosscutting*). AspectJ cannot be considered a template technique, as the concern that is typically bundled with the query language is not prototypical for the result.

Finally, in the unparser technique JAXB, the first concern is expressed in Java, which is basically a direct equivalent of an XML document to be augmented with the second concern using Java

instructions. The second concern can be any source of data (as it is the case with JSP, see above). The direct equivalent relationship between the first concern and the result is not a prototypical relationship, as the concern and the result are from different technological spaces.

## 2.4. Related Research Areas

In the following sections, research areas related to the notion of templates in the sense defined above, are discussed. Section 2.4.1 discusses the research in the field of macro processing. Section 2.4.2 discusses two languages in which the term template is used to denote a programming language feature. Section 2.4.3 introduces the Invasive Software Composition (ISC) approach, which is a compile-time composition technique. Finally, Section 2.4.4 discusses frame processing, a technique closely related to template techniques.

### 2.4.1. Macro Processing

The idea of macros was introduced by [124] in order to extend compiled languages with new language features. As such, and contrary to the approaches just introduced, macros are not primarily a technique for composing concerns, but are rather targeting reuse on a syntactical level.

An important improvement to the macro mechanism was its lifting to a level where syntactic constraints are preserved. An example for this lifting is the introduction of *hygienic macros* in LISP [110].

Today, the best-known example are the macros provided by C and its preprocessor. Unfortunately, C macros can easily break the syntactic structure of a program or interfere with the program in unexpected ways. The ideas from [110] were proposed to address the problems described in [40], but the development of C++ [174] is definitely the most important attempt to solve them. In order to make the use of the C++ preprocessor obsolete, templates (as a mean to express generic functions) and the `inline` and `const` specifiers (to replace constant definitions and the use of macros for repeated non-generic code fragments) have been added to the language. Unfortunately, modern C++ programs still leverage the preprocessor, sometimes by mixing its features with the features introduced to abolish it.

Macros are not a typical means for the composition of concerns, but they are part of a number of template techniques, as shown in Section 2.5.7.

### 2.4.2. Templates as Programming Language Feature

The aforementioned C++ templates are also a mean for separation: for example, they can be used to separate the behavior of a container data type from the actually contained type. Additionally, a C++ template can be considered to be of prototypical nature, as the result of the template instantiation basically yields the template with its formal type (as well as its non-type) arguments replaced with the actual ones, even if this instantiation result is normally not manifested by the C++ compilers.

A template mechanism is also the base of the modularization technique used in the BETA languages [112; 122]. Please note that it is strictly speaking not part of the BETA language, as the

## 2. Introduction

language used for modularization, the so-called *fragment language*, is independent from BETA. The fragment language is grammar-based, i.e., every correct sequence of terminal and nonterminal symbols from the grammar is called a *form* and represents a module. Non-terminals in the forms are called *slots*, as the non-terminals are openings where other forms may be inserted. An example for a form is shown in Listing 2.3.

```
Stack:
  (# Private: @<<SLOT private: descriptor>>;
    Push:
      (# e: @integer
        enter e
        do <<SLOT PushBody: descriptor>>
        #);
    Pop:
      (# e: @integer
        do <<SLOT PopBody: descriptor>>
        exit e
        #);
    New: (# do <<SLOT NewBody: descriptor>> #);
    isEmpty:
      (# Result: @boolean
        do <<SLOT isEmptyBody: descriptor>>
        exit Result
        #)
  #)
```

Listing 2.3: A BETA Form

If a form is associated with a name and a syntactic category (basically, the left hand side of the grammar rule from which the form has been derived), it is called a *fragment form*. *Fragment groups* bundle logically related fragment forms.

A fragment group  $F$  may specify a fragment group  $O$  as its *origin*. In this case, the slots of the fragment group  $O$  are substituted by the corresponding fragment forms in  $F$ . The origin fragment group  $O$  must have slots defined for all the fragment forms within the fragment group  $F$ .

This substitution process turns out to be extremely powerful: it subsumes the power available by C++ templates and exceeds it—for details, see [122].

Obviously, the substitution mechanism is a template instantiation, with the origin fragment group  $O$  being the template, whereas the fragment forms within  $F$  are the instantiation data. The term slot in BETA fits the Definition 2.6 of slots introduced above. A difference which should be noted, is the labeling of slots with syntactic categories, which enables (together with the syntactic category assigned to forms in fragment forms) a kind of safe authoring for BETA programs.

### 2.4.3. Invasive Software Composition

The ISC approach is a compile-time composition technique proposed in [15]. Its composition operators treat the components as so-called *grey-box* components, i.e., the components itself

can be modified (a property of *white-box* composition techniques), but this happens using well-defined interfaces (a property attributed to *black-box* composition techniques).

The ISC terminology has been formalized in [86], based on an extension of context-free grammars, called *context-free reuse-grammars*. Basically, the components of ISC are *fragments* of programs that correspond to sentential forms that can be derived from a non-terminal of a context-free grammar. Non-terminals in these sentential forms are considered as variation points.

Initially, ISC knows two composition operators called *bind* and *extend*, separating the set of variation points into *slots* and *hooks*. Slots can be bound, i.e., they can be replaced with single fragments using the *bind* operator. Hooks are extensible, i.e., fragments can be added to it repeatedly with the *extend* operator.

The main difference between ISC and template techniques is the handling of extensibility. Whereas ISC allows hooks to be bound multiple times, this is not possible in template techniques, which instead offer the possibility to iterate from within the template itself, thereby allowing a single slot to be replicated and bound to different values in the iterations. With a template approach, extensibility can be emulated using slots, if the template allows to reintroduce the slot markup when the slot is bound. Thus, instead of binding a slot denoted by some markup  $s$  directly to the value  $v$ , it is bound to the sequence  $vs$ , thereby reopening the slot for further values. Obviously, this emulation requires that there is a post-processing step in which all slots are removed, e.g., by binding them to empty values.

### 2.4.4. Frame Processing

Frame processing [17; 18; 46] can be considered as an early predecessor of template techniques. The technique has been named after its main source of inspiration, conceptual frames [129].

In [18], multiple views are given on frame processing. The most comprehensive one is to consider frames as software parts that should be assembled. Assembling frames is done by invoking frames from a root frame, the so-called *specification frame*. Invoked frames may in turn call further frames, making the assembly process a tree traversal over a set of frames.

During the assembly, some frames need to be adapted, i.e., modified or completed. To modify frames, two basic mechanisms are offered: the use of *variables* and the use of *named blocks*. Whereas variables do not have default values, named blocks have their content as default value. Variables can be set to arbitrary values by the invoking frame. Named blocks can be arbitrarily extended at the start or the end of the named blocks or replaced or deleted during assembly. Finally, frames can also include control statements to conditionally or repeatedly include parts of the frame, depending on the actual values of variables.

Obviously, frame processing closely corresponds to template processing as defined above. Frames correspond to templates. Variables correspond to slots. Named blocks are not typically found in template techniques. Control statements for conditional and repeated inclusion of fragments can be found in most template approaches. However, frame assembly is much more sophisticated than typical template instantiation, as it is possible to trigger the assembly of the frames recursively. This is a feature that is not typically found in template techniques. An exception is ST [173; 143], which could easily emulate the frame assembly process using its *template application* feature.

## 2. Introduction

Listing 2.4 shows the use of the XML-based Variant Configuration Language (XVCL), a frame processing language designed to foster reuse during the implementation of a software product line [93]. Further examples of frame processing technologies are given in [43].

Specification Frame
<pre>&lt;x-frame name="root"&gt;   public class Root   {     public static void main(String[] args)     {       &lt;set var="max" value="100"/&gt;       &lt;adapt x-frame="secondary-frame"&gt;         &lt;insert break="perform"&gt;           System.out.println("i =" + i);         &lt;/insert&gt;       &lt;/adapt&gt;     }   } &lt;/x-frame&gt;</pre>
Frame 'secondary-frame'
<pre>&lt;x-frame name="secondary-frame"&gt;   for (int i = 0; i &lt; &lt;value-of expr=""?@max?" /&gt;; i++)   {     &lt;break name="perform"/&gt;   } &lt;/x-frame&gt;</pre>
Result of the Frame Assembly Process
<pre>public class Root {   public static void main(String[] args)   {     for (int i = 0; i &lt; 100; i++)     {       System.out.println("i =" + i);     }   } }</pre>

Listing 2.4: Frame Processing Example with XVCL

## 2.5. Classification

In this section, a classification for template techniques is given. The classification focuses on template techniques and on defining orthogonal criteria. This distinguishes it from existing classifications like [43] and [169].

### 2.5.1. Target Language Awareness of Slot Markup

The most basic distinction to be made about slot markup is how its introduction influences the target language syntax and semantics. The target language syntax and semantics may be affected in several ways, which leads to the categories shown in Figure 2.7 and described in the following. Please note that the figure also includes *implicit* slot markup (the technique commonly used in AOP, see Section 2.3.2) for illustrative purposes, however, these are not marking up *slots* with respect to the Definition 2.6.

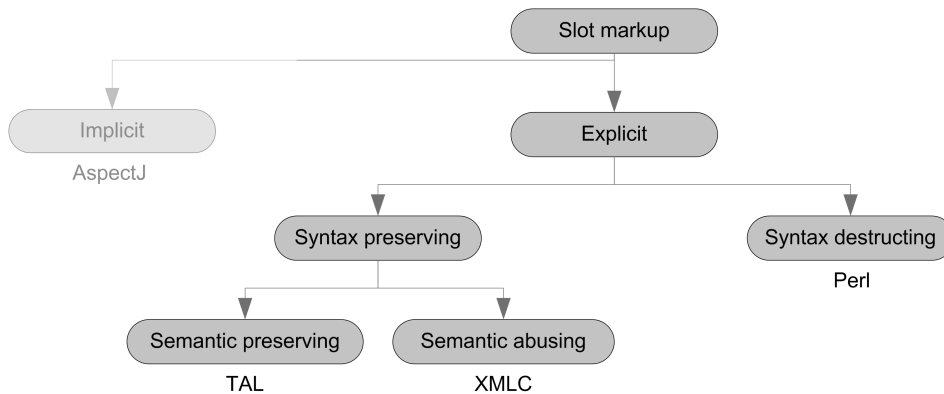


Figure 2.7.: Target Language Awareness of Slot Markup

Slot markup languages that do preserve the target language syntax are called *syntax preserving* slot markup languages, whereas slot markup languages changing the target language syntax are called *syntax destructing* languages.

Syntax preserving languages may be further classified into *semantics preserving* and *semantics abusing* languages, depending on whether semantic concepts from the target language are employed as they are intended to be or abused. The distinction between preservation and abuse is somehow fluent—an indication for abuse is that a language element that is not intended to carry any semantics is equipped with meaning by the slot markup language. A typical example for this is the abuse of comments for slot markup purposes.

Examples for syntax preserving slot markup languages can be found in approaches targeting markup languages like HTML or XML. The Template Attribute Language (TAL) [196] is using attributes from a distinct XML namespace, which is clearly a valid use of attributes and namespaces, i.e., TAL is *preserving* the semantics of the target language features.

On the other hand, XMLC [195] is using the `id` attribute and the `span` element of XHTML as slot markup, with the latter example clearly being an *abuse* of the target language semantics.

Languages that destruct the target language syntax typically rely on some kind of special separator (or pairs of separators) for slot markup. This goes back to the `$` symbol used in Unix shell scripts for variable interpolation and has been reused by Perl and many of its successors. A typical pair of separators that is used to bracket expressions from the query language are the strings `<%` and `%>`, which seemingly have been introduced by JSP but can also be found in a variety of other languages, e.g., in Tea [133] and Jxp [102].

## 2. Introduction

The classification of template techniques targeting XML documents may depend on the understanding of syntax. If syntax is understood as wellformedness, XSL-T SSM is clearly preserving it. On the other hand, if syntax is understood as compliance to an XML Schema, XSL-T SSM de-structs the target language syntax by embedding XSL-T elements at locations where they are not allowed to be placed.

### 2.5.2. Generality of the Slot Markup

Template approaches can also be classified by the relationship between the number of terminals in the target language and the number of terminals in the slot markup language.

Some approaches introduce a corresponding terminal in the slot markup language for each terminal in the target language [160; 156]. This approach makes the slot markup language *specific* to the target language.

On the other side, approaches exist that introduce only a small number of slot markup language terminals which allow to markup all kinds of slots possible within the target language. Moreover, the number of terminals introduced is independent from the target language. This typically makes this kind of approaches target language independent or *generic*. As an example, consider XSL-T SSM [107], which can be used to generate arbitrary documents using a fixed set of slot markup language instructions.

### 2.5.3. Entanglement Index

The *entanglement index* is a metric defined in [143] which classifies template approaches according to the number of violations against a set of rules guaranteeing a clean separation of concerns between the template engine and the application using it.

These so-called separation rules, which are formulated in [143] with respect to using templates as the view in the MVC pattern [154], are the following:

1. (*no modification*) The view can not modify the model neither by directly altering model data objects nor by invoking methods on the model that cause side effects.
2. (*no computations*) The view cannot perform computations upon dependent data values.
3. (*no comparisons*) The view cannot compare dependent data values.
4. (*no type assumptions*) The view cannot make data type assumptions.
5. (*no layout*) Data from the model must not contain display or layout information.

In order to ease the evaluation of these rules in scenarios other than Web applications, they have to be reformulated. The most specific rule (*no layout*) must be replaced by a rule capturing the actual intent of using a template engine:

1. (*no modification*) The template can not modify the instantiation data, neither directly nor by causing side effects.



2. (*no computations*) The template cannot perform computations upon dependent data values.
3. (*no comparisons*) The template cannot compare dependent data values.
4. (*no type assumptions*) The template cannot make data type assumptions.
5. (*separation of concerns*) Instantiation data must not contain information that should be separated from the application by the use of the template engine.

The first rule is easy to understand: the instantiation of a template should be free of side effects. That rule is reasonable as it is clear that moving application code into a template for whatever reasons is to be avoided. The rule does not prevent the instantiated template to communicate with the controller (as it is typically done using a Hypertext Transfer Protocol (HTTP) request within Web applications), it must only be followed during the instantiation of the template within the template engine.

The second, *no computations* rule can be subject to controversial discussions. On the one hand, it is obvious that certain computations (e.g., the computation of taxes on a basket in an e-commerce application) should not be subject of a template. On the other hand, some string computations like special character encoding may be performed within the template without violating the separation of concerns, especially since the encoding to be applied may depend on the context of the special characters, e.g., a string may need to be encoded differently as XML attribute value or as element content.

The *no comparison* rule is also subject to discussion. Obviously, for the comparison of a product price with a fixed value, the fixed value should not be part of the template. However, alternating the background colors of rows in a table requires a comparison like  $i \bmod 2 = 0$  and can well be done within a template. The ST engine solves this alternating colors task using a round-robin approach specifically designed to handle exactly this class of tasks. Comparisons can also be useful in code generation templates. For example, the use of separators is often associated with a comparison of a loop index with a fixed value (e.g.,  $i > 0$ ). Again, ST provides a solution for this problem using its `separator` feature [144].

The fourth rule implies that methods with arguments cannot be called, since type information is unavailable in templates. Furthermore, indexing of array elements using instantiation data is not possible, since this would require to assume that the data used as index is of a scalar type.

Finally, the fifth rule enforces separation of concerns between layout and content in a template application directly. The application logic should not be able to provide layout or display information. [143] states that there is no way to enforce this rule. However, for a particular target language, a distinction between parts of the language that are allowed to be generated from instantiation data and other parts may be possible. In this case, the partial templatization approach described in Section 5.1.7 can be used to enforce the *no layout* rule.

The IKAT engine from the Reasonable Server Faces (RSF) project [186] states about itself to have an entanglement index of *zero* [185]. As a reason for this, IKAT's inability to *permit computable control over output XML attributes* is given. This is based on the false assumption that attribute values can always be considered as layout. A counterexample for this is also given in Section 5.1.7.

### 2.5.4. Instantiation Data Access Strategy

There are basically two ways how the data needed to instantiate a template is passed to the template engine. The names for the strategies are taken from [143].

The first way is a *pull strategy*: the template engine calls the application using it to fetch data on demand. The second way is a *push strategy*: the applications passes all instantiation data to the template engine before the actual instantiation process is started. The differences between the approaches are summarized in the sequence diagrams in Figure 2.8. The definitions below are versions adapted from [143] by using the introduced terminology and removing clauses not substantially contributing to their clarity.

**Definition 2.19** (Push Strategy, after [143]). A template uses the *push strategy* if all data used by the template is computed prior to template instantiation. □

**Definition 2.20** (Pull Strategy, after [143]). A template uses the *pull strategy* if any data used by the template is computed on demand by invoking application logic. □

The advantage of the push strategy is that it enforces the independence of the application logic from the order in which the instantiation data is accessed. The disadvantage is that some instantiation data items which may not be needed (e.g., because conditions prevent the data from being used) have to be calculated anyway. The pull strategy does not have this problem: it allows the lazy evaluation of instantiation data items.

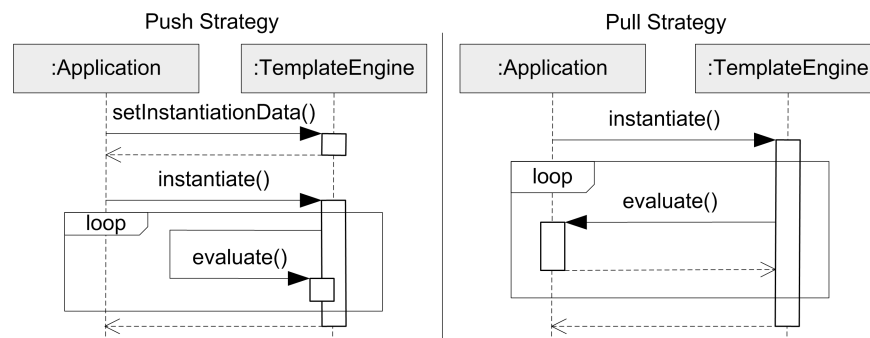


Figure 2.8.: Sequence Diagrams of Push resp. Pull Strategy

The classical example for the push strategy is ST, whereas the pull strategy is implemented in a variety of engines like JSP or Velocity. The push strategy can be emulated using the pull strategy—see Section 6.1.1.

A distinction between the strategies is sometimes impossible. An example are the SSMs defined in XSL-T. The data (i.e., the XML document addressed from the stylesheet via XPath expressions in `select` or other attributes) is typically *pushed* upfront into the XSL-T processor, but the evaluation of particular XPath expressions is performed on demand. Additionally, it is possible to access further data sources via the `document` function [36, Section 12.1], so the data can partly be *pulled* from the engine. That is, the data from the first source document is accessed using the push strategy (note that the evaluation within the template engine is not explicitly prohibited by Definition 2.19) whereas the documents retrieved via the `document` function are accessed using the pull strategy.

The discussion on the choice of strategy has not yet found a definitive answer. [143] argues that the pull strategy violates the *separation of concerns* design rule between application logic and template engine by allowing to build application logic that relies on a particular evaluation order of the instantiation data. While this argument is definitely true, it is questionable whether the benefits of disallowing this coupling outweighs the effort for the calculation of instantiation data that may not be needed during the instantiation. [117] emphasizes the problem that data calculated to be pushed into the engine may be unnecessary, thus wasting resources. Additionally, it is argued that functionalities that belong to the presentation layer shift into the application code. As an example, HTML/XML character escaping is mentioned, which is not correct, as the push approach itself does not prevent the engine to have features supporting character encoding, e.g., via its query language.

### 2.5.5. Query Language

Definition 2.13 introduces the notion of a query language that is used to refer to instantiation data from within the template. The query language used in a template can be classified in three ways, which are illustrated in Figure 2.9.

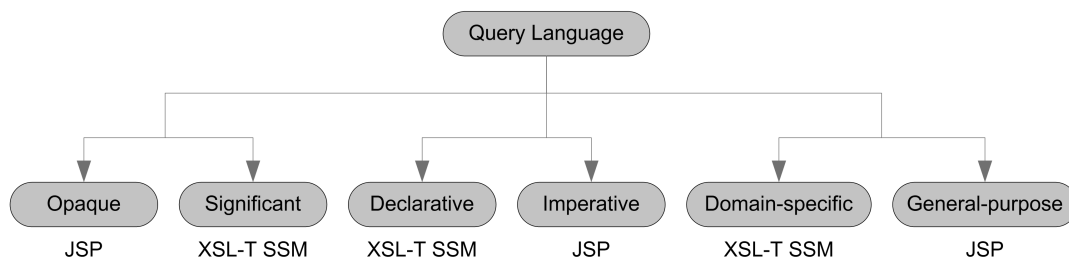


Figure 2.9.: Categories of Query Languages

First, the query language may be *opaque* or *significant* to the template engine. If the query language is opaque, the template engine either passes the query directly to the application incorporating itself or uses the query to search for instantiation data in a container passed by the application. In both cases, the template engine has no idea of the internal structure of the query. On the other hand, if the query language is significant, queries are executed by the template engine, i.e., the semantics of a query as well as its concrete syntax must be known to the engine.

Second, a query language may be *declarative* or *imperative*. In the first case, the instantiation data is described by the query, but the query does not define how to actually get the data. Differently, queries from imperative query languages define the exact way how to fetch the instantiation data.

Third, query languages may be *general-purpose* or *domain-specific* languages. In the first case, the query language is a general-purpose programming language on its own, whereas in the latter case, the query language is specific to a certain type of instantiation data source.

Two examples should clarify this classification: JSP is using Java as an imperative, general-purpose query language. Please note that Java is used in a way that makes it opaque to the JSP

## 2. Introduction

engine, as the engine itself is not interpreting the query language in any way. XSL-T SSM's are using XPath as a significant, declarative and domain-specific (i.e., XML instantiation data source specific) query language.

### 2.5.6. Instantiation Technique

Template techniques can also be classified by the way templates are instantiated. Typical instantiation techniques are compilers and interpreters.

Compilers transfer the template into a persistent intermediate form that is typically executable and emits the instantiated template during the execution. The best known example for this approach is JSP, which even generates multiple intermediate forms: first, a JSP document is typically translated into a Java source file, which is afterwards compiled into a class file directly executable by a Java Virtual Machine (JVM).

Interpreters instantiate the template directly, i.e., without translating it into a persistent intermediate form. An example for an interpreting template engine is ST.

The advantage of the compiling approach is an improvement of the instantiation speed, its disadvantage is the extra time needed for the compilation. In general, the decision for one of the two approaches depends both on the frequencies of template changes and template instantiation and their ratio.

### 2.5.7. Reuse in Templates

It is an important requirement that template fragments must be reused within a single template. Different approaches are in use to fulfill this requirement.

Many features supporting reuse within templates correspond to macros [124] in general purpose languages. These macro mechanisms can be further classified by their support for parameter passing and by their support for dynamic calls (i.e., the selection of the invoked macro depending on instantiation data).

Macro features can be classified by their parameter passing mechanisms. Some engines only support macros without parameters (like XTL, see Section 4.4), others allow passing arbitrary instantiation data and/or variable values (like XSL-T SSM), and some engines even allow template fragments to be passed into macros (like Tea [133]).

Another possible macro classification criterion is the selection mechanism of the macro to be called: the macro may be statically selected (like in Tea [133]), or it may be chosen depending on the context in which the template is applied (called *template polymorphism* in XPAND). XSL-T SSM even supports both types of selection with its static `xsl:call-template` and its dynamic `xsl:apply-templates` instructions.

ST [173; 143] offers an efficient object-oriented reuse technique called *group inheritance*. The motivation behind this technique is the use of ST as backend in the parser generator ANTLR[6; 142], where languages as similar as Java version 1.4 and version 5 should be generated without having to develop and maintain completely independent template sets. In ST, a template set is called a group. Groups can inherit templates from other groups. This way, it is possible to specify a common base group for both Java version and extract the differences between the languages into groups that inherit from the base group.

### 2.5.8. Further Features

Some template approaches offer unique features, which should be mentioned shortly.

Jostraca [157] offers capabilities to search and replace text within the whole template during the instantiation process. This is clearly not a template-typical feature, but merely an addition of a common text processing features.

ST [143; 144] offers a feature called *group interfaces* which allows the specification of parameters a set of templates must have. Together with ST's feature of *group inheritance*, this mechanism enables an object-oriented reuse technique in templates. Please note that the term *interface* here relates to the contract between the template and another template that uses it, which is different from how the term is used in Section 6.3.2, where it refers to the contract between the template and its instantiation data.

XPAND knows a special syntax to prevent newlines from being taken over from the template into the instantiated template. For example, for the use of this feature consider Listing 2.5, which shows an excerpt from an XPAND template and from the Java code produced by it. Please note the difference between the two lines creating the private methods: in the second one, the XPAND expression is closed with “-»”, meaning that the following whitespace should be omitted in the output.

	XPAND Template	
<pre>public class Test {     «LET 'doA()' AS method»         private void «method»         {         }     «ENDLET»     «LET 'doB()' AS method»         private void «method-»         {         }     «ENDLET» }</pre>		
	Instantiation Result	
<pre>public class Test {     private void doA()     {     }      private void doB() {     } }</pre>		

Listing 2.5: Suppression of Newlines in XPAND

## 2. Introduction

Repleo [14] is a template engine that also provides syntax-safe template instantiation. In contrast to the approach introduced in this thesis, it proposes a restricted slot markup language (called *template meta language* in this context) that destructs the syntax of the target language (called *object language*). Repleo also introduces an adaptation phase, which combines the input grammars in a common template grammar. Repleo validates the instantiation data only during the instantiation time, it does not offer a technique equivalent to the Template Interface Generation introduced in Section 6.3.2. Repleo uses an XPath-like query language.

An approach to generating safe template languages is also proposed in [85]. The approach is very similar to the approach proposed in this thesis, but uses a syntax-destructing slot markup language. This is motivated by the fact that the approach in [85] is not restricting the target language. The query language proposed in [85] is the Object Constraint Language (OCL).

## 2.6. Conclusion

This chapter defined the terms that are used throughout this thesis. The concise definition of the template term captures the intuitive meaning of this term in the context of Web applications and code generation very well, which distinguishes the definition from existing ones like [144], which makes the definition a contribution in its own right. An introduction to the typical applications of template techniques has been given. The alternatives to using a template technique have been described, both with their advantages and disadvantages. The related research areas have been introduced. Finally, classification criteria have been given, which allow to describe the properties of existing and new template techniques concisely. The classification goes beyond existing classifications like [169], as it defines orthogonal properties and exhaustively covers the area of template techniques.

Cib

## *2. Introduction*



# 3

## Safe Template Processing

An diesen einfachen Beispielen wird jene Eigenschaft von Web-Templates deutlich, die zugleich ihre *pragmatische Stärke* und *formale Schwäche* ist: *Web-Templates können einfach hingeschrieben werden*, eine formale Validierung ist nur auf der Ebene der fertig gestellten Web-Dokumente sinnvoll möglich.

Karsten Wendland, 2006 [189]

In this chapter, an approach for a development technique named *safe template processing* is shown. Section 3.1 defines goals for an approach that enables safe template processing. From these goals, requirements are derived in Section 3.2. Based on these requirements, an architecture is proposed in Section 3.3.

### 3.1. Goals

The motivating example shown in Section 1.3 can be used to define a number of goals which a design for a template technique should address. Some of the goals may contradict what is typically expected from template approaches, others are suggested by common sense and are, nevertheless, not respected by all existing techniques.

As it has already been mentioned in Section 2.3, several other approaches exist that address some or all issues in the scenario above. Each particular goal is discussed with respect to existing technologies.

### 3. Safe Template Processing

#### 3.1.1. Safe Authoring

Unfortunately, the relation between the template and the target language shown in Figure 2.2 does not reflect today's reality. Instead, the instantiation of a template *may* lead to a document in the target language, but this is not guaranteed in every case. This leads to the typical *trial and error* process shown in Figure 1.2, as it is executed by Web developers and designers regularly: A template is changed, and afterwards the result of the instantiation process is checked.

As the experience with techniques like JSP and XSL-T shows, this process is error-prone for several reasons: the executing person may consider a change small enough not to be worth checking and, more important, the change may not be covered by the instantiation, e.g., because the changed part is not instantiated at all due to the used instantiation data.

The goal that can be derived from this problem is called *safe authoring*, reflecting that the fulfillment of this goal gives an author the highest safety possible that a created template will actually instantiate into the target language. This safety is nevertheless restricted, as the instantiation data has substantial influence on the instantiated template, i.e., every guarantee given to the author is given under the presumption that the instantiation data fulfills certain properties (which will be explained in Section 3.2.5).

The term safe authoring is informally defined as follows: A template approach enables *safe authoring* if it gives (under the presumption of certain instantiation data properties) an author a clear indication whether a particular template will instantiate into the target language or not.

The most popular example of an approach not targeting the goal of safe authoring is JSP. By its typical mixture of XHTML as target language, Java as query language and several notations to distinguish between the languages, there is a high risk of creating templates not instantiating into the target language.

On the other hand, XML binding technologies like JAXB fulfill the safe authoring goal very well, because they employ the type system of some programming language to guarantee instantiation results, making it impossible for an author to create documents that fail to instantiate into the target language.

#### 3.1.2. Safe Instantiation

The instantiation of a safely authored template may fail because the instantiation data used does not fulfill the assumptions that have been made during the template authoring. These instantiation failures must be communicated as error messages. The asynchronism of template authoring and template instantiation complicates relating the omitted error messages to the cause of the error.

Furthermore, the person encountering the error (e.g., the user of a Web application) is most probably different from the person that caused the error (e.g., the application developer), which makes understandable error messages even more valuable.

The informal definition of safe instantiation is as follows: A safe instantiation checks the instantiation data and emits error messages that

1. clearly describe the problems that occurred,
2. show their root causes and

3. allow determining the person that is responsible to fix them.

Existing approaches differ widely in their error handling. JSP seems to be the worst approach in this respect: depending on whether the error in the template leads to a compilation error during template instantiation or just to a malformed XHTML document, different errors may occur. For examples of error messages caused by invalid JSP pages, see Figures 3.1(a), 3.1(b) and 3.1(c). For an unexperienced user, it is hard to decide which of the error message is due to an invalid change of the template and which one is due to incorrect instantiation data.

```
org.apache.jasper.JasperException: Unable to compile class for JSP

An error occurred at line: 6 in the jsp file: /randomdivbyzero.jsp
Generated servlet error:
C:\Java\jakarta-tomcat-5.0.28\work\Catalina\localhost\jsp-examples\org\apache\jsp\randomdivbyzero_jsp.java:18:
    return Math.((float)(2*Math.random()));
                   ^
1 error

org.apache.jasper.compiler.DefaultErrorHandler.javacError(DefaultErrorHandler.java:84)
org.apache.jasper.compiler.ErrorDispatcher.javacError(ErrorDispatcher.java:332)
org.apache.jasper.compiler.Compiler.generateClass(Compiler.java:412)
org.apache.jasper.compiler.Compiler.compile(Compiler.java:472)
org.apache.jasper.compiler.Compiler.compile(Compiler.java:451)
org.apache.jasper.compiler.Compiler.compile(Compiler.java:439)
org.apache.jasper.JspCompilationContext.compile(JspCompilationContext.java:511)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:295)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:292)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:236)
javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

(a) Compilation problem

```
org.apache.jasper.JasperException: / by zero
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:372)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:292)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:236)
javax.servlet.http.HttpServlet.service(HttpServlet.java:802)

root cause
java.lang.ArithmeticException: / by zero
org.apache.jsp.randomdivbyzero_jsp._jspService(randomdivbyzero_jsp.java:64)
org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:94)
javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:324)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:292)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:236)
javax.servlet.http.HttpServlet.service(HttpServlet.java:802)
```

(b) Runtime exception

**XML Parsing Error: mismatched tag. Expected: </html>.**  
**Location:** http://localhost:4040/jsp-examples/randomdivbyzero.jsp  
**Line Number 6, Column 3:**

```
</invalid>
--^
```

(c) Parse problem

Figure 3.1.: Error Messages caused by JSP Pages

### 3.1.3. Separation of Concerns

As mentioned above, templates are frequently used to achieve a separation of concerns. The separation typically occurs between development artifacts, responsibilities (or roles), or life cycle phases. The actual concerns to be separated depend on the application area in which the

### 3. Safe Template Processing

approach should be used. An overview of actual concerns in the two most important usage scenarios for template approaches is shown in Figure 3.2.

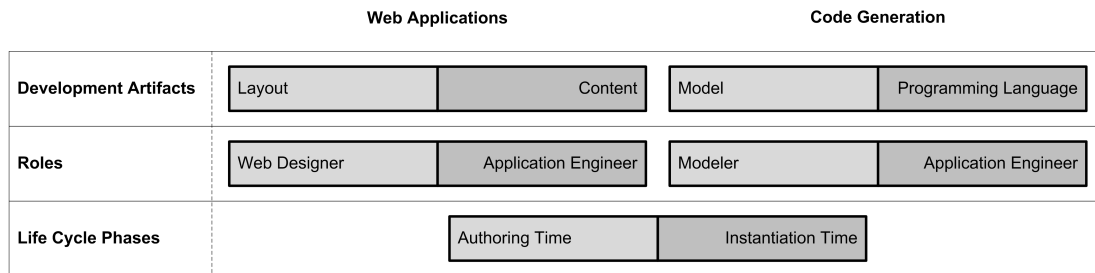


Figure 3.2.: Separation of Concerns in Different Scenarios

Part of this goal is not only to provide means for the separation of concerns: it is equally important to ensure that the separation is not circumvented by users of a template technique. The importance of the enforcement of separation of concerns has been described in [143].

The following informal definition considers both parts of this goal: A template approach fulfills the goal of *Separation of Concerns* if it

1. *enables* the separation of concerns, i.e., allows the distributed, asynchronous and simultaneous work on the concerns and
2. *enforces* the separation of concerns, i.e., restricts the consequences of changing a concern for related concerns as far as possible.

Often, the separated concerns are particular documents or other storage units, but the concerns to be separated may also be rather abstract views of stakeholders on a single result of the development process. E.g., if the template technique is used in a Web application, the concerns to be separated are typically layout and content, a separation that has been recognized as being essential in the publishing sector as early as 1967 by Tunnicliffe [76] and that also holds in the field of Web engineering. The separation between these concerns can lead to separated storage units, but the separation can as well take place within a single storage unit.

An analogous separation is desirable if the template engine is used for M2C transformations in a generative scenario: programming language specifics should be separated from programming language independent information stored in a model (described in Section 2.2.2).

The separations between concerns described above correspond to the separation of responsibilities of stakeholders (or roles). In the Web application scenarios, the layout concern is typically the responsibility of a Web designer, whereas the creation of content is typically the responsibility of an editor (e.g., in a Web CMS scenario) or a software engineer (e.g., acting as the developer of the model and the controller in a Web application).

For the code generation scenario, the responsibility for the creation of the artifacts described above may be distributed between a programming language specialist (for the programming language specifics) and a model developer (responsible for the model as such).

In both scenarios, the authoring of a template and its instantiation typically occur asynchronously, i.e., the life cycle phases not necessarily overlap each other. It is not atypical that a

template still gets instantiated when the author of the template is no longer available to maintain it.

Approaches like XML binding technologies (e.g., JAXB) completely fail with regard to this goal, both in enabling separation as well as in enforcing it. For example, if an unparser-related library like RWT is used to build a Web user interface, the Web designer and the application developer role are unified.

On the other hand, JSP enables the separation of concerns, but fails to enforce it (which is indicated by its high entanglement index of 5), as the embedded access to Java allows the template author to accomplish arbitrary tasks, including the tasks that belong to the model or the controller in an MVC-based application.

#### 3.1.4. Broad Applicability

The architecture should be applicable in a wide range of applications, from Web CMSs to UML tools. Therefore, assumptions about particular uses of the architecture and its implementations have to be avoided. On the other hand, this design goal had to be restricted in order to create a prototype implementing the approach, i.e., the set of target languages addressable has been limited to XML dialects.

The definition of broad applicability is therefore as follows: A template approach satisfies the goal of *Broad Applicability* if it is usable in different application scenarios and capable of generating various target languages.

Velocity is an example for a broadly applicable template technique, as it has been widely used for Web applications [77] as well as for code generation [175]. The languages that have been generated using Velocity include various XML dialects like XHTML, Java, C++ as well as plain text (e.g., for the generation of emails).

JSP is restricted in its use by its reliance on Web application servers. There have been experiments to separately use JSP, but the coupling to Web application servers has proven to be strong, which complicates the stand-alone use of JSP. Besides this restriction, JSP has been used to generate several web-typical languages like XHTML and WML.

#### 3.1.5. Utilization of Existing Standards

During the design and implementation of the Safe Template Processing approach, the question of whether a (*de facto*) standard or component should be reused often arises. Typically, the consequences are as follows: if the standard is not reused, a completely new way of template processing could be introduced (along with the necessary standards, tools and processes), which leaves more design options at the cost of reducing the chances of the new approach to become widely accepted. Alternatively, adapting the existing standards or components reduces the degrees of freedom for the design, whereas the chances for establishing the approach are much higher. The alternatives can be considered as revolutionary or evolutionary trials to establish a new template processing approach.

In the following, for a design question that can only be decided in the described ways, the latter alternative, i.e., the evolutionary improvement, is taken. Or, informally defined: A tem-

### 3. Safe Template Processing

plate technique fulfills the *Utilization of Existing Standards* goal if it minimizes the changes to the standards, tools and processes left to users adopting the technique.

XML binding technologies like RWT can be seen as a way to a revolutionary change to the Web engineering process. This can also serve as an explanation, why these approaches failed to prevail widely. Advanced JSP editors as found in modern Web development IDEs are a typical example for the evolutionary approach, as they do not try to change the process of JSP authoring.

## 3.2. Requirements

From the goals introduced in Section 3.1, a number of requirements can be deduced, which are described below. These requirements are to be fulfilled by the developed approach. The requirements address different goals—a summary about these dependencies is shown in Figure 3.3<sup>1</sup>. The dependencies for each requirement are discussed in detail in the corresponding section. If possible, examples that fulfill the requirement or fail to fulfill it are given.

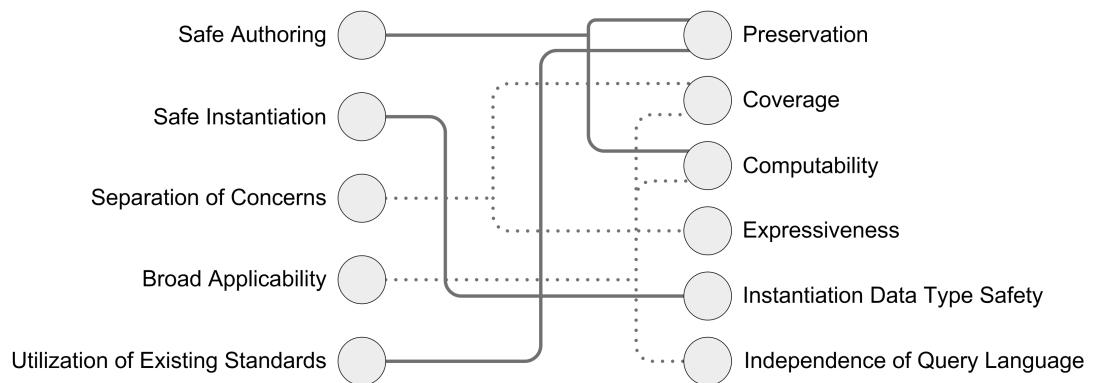


Figure 3.3.: Relations between Goals and Requirements

#### 3.2.1. Preservation of Target Language Constraints

In order to guarantee that the instantiated template complies to the target language, all constraints that are inherent to the target language (i.e., which form the schema of the target language) must also be valid within the template language. This does not mean that the constraints can be mapped one-to-one from the target into the template language. Instead, every constraint from the target language will lead to an equivalent, maybe more complicated, constraint in the template language.

Formally, this requirement can be defined as follows:

**Definition 3.1** (Preservation of target language constraints). A template technique preserves the target language constraints, if for each template  $t^\circ$  the instantiation instantiate with the

<sup>1</sup>The different styles for the lines in the Figures 3.3, 3.6 and 7.1 have been chosen to improve the perceivability of these figures, they are *not* semantically important.

instantiation data  $d \in D$  yields a document from the target language:  $\forall t^\circ \in \mathcal{T}^\circ : \forall d \in D : \text{instantiate}(d, t^\circ) \in \mathcal{T}$ .  $\square$

Obviously, this *preservation* requirement addresses the safe authoring goal. There are several ways to fulfill this requirement, which differ in their reuse level of existing standards. Therefore the utilization of existing standards is also related to this requirement.

XSL-T SSM can be seen as an example for a template language that is completely ignoring target language constraints within the templates: it is possible to generate any XML dialect from an XSL-T SSM. If a specific dialect defined by some XML Schema is constructed by a stylesheet, none of the constraints from this XML Schema are checked within the stylesheet. This makes XSL-T SSM both a powerful and an error-prone template technique.

### 3.2.2. Coverage of Target Language

A template engine must be able to produce *all documents* of the target language. Definitions 2.5 and 2.11 already state that the target language is covered by the templates as the set of templates is a subset of the template language, i.e.,  $\mathcal{T}^\circ \subset \mathcal{T}$ .

The fragments contained in the instantiated template originate, however, both from the template as well as from the instantiation data. Thus, the requirement must be fulfilled independently of which fragments of the instantiated template originate from the template.

The distribution of fragments between the template and the instantiation data is itself restricted by the separation of concerns goal and has not been formalized. The following is therefore only a semi-formal definition of the *coverage* requirement, as it relies on the unspecified notion of a set of valid instantiation data  $D_t$  that reflects which parts of the target language document  $t$  could originate from the instantiation data:

**Definition 3.2** (Coverage of target language). A template technique covers a target language  $\mathcal{T}$  if for each document  $t$  from the target language  $\mathcal{T}$  and for any instantiation data  $d$  from the set of valid instantiation data  $D_t$ , there exists a template  $t^\circ$  that instantiates to  $t$ :  $\forall t \in \mathcal{T} : \forall d \in D_t : \exists t^\circ \in \mathcal{T}^\circ : \text{instantiate}(d, t^\circ) = t$   $\square$

The *coverage* requirement clearly addresses the goal of broad applicability, as a template engine that is not capable of creating the complete target language is only useful in very special cases. Furthermore, the requirement is influenced by the separation of concerns goal, because this goal determines the distribution of target language fragments between the template and the instantiation data.

### 3.2.3. Computability

As the requirement *preservation* in Section 3.2.1 indicates, constraints imposed by the target language have to be transformed to be applicable to validate documents with respect to the template language. The user of the template engine should not be burdened with the process of adapting a template technique to a particular target language.

Thus, the template language syntax must be automatically computable from the target language syntax. As a side effect, this requires the target language syntax to be available in a machine-readable form (like an XSD document or some other grammar description).

### 3. Safe Template Processing

The *computability* requirement therefore addresses the broad applicability goal, as it enables the use of the template technique for generating documents from arbitrary languages. Furthermore, it also contributes to the fulfillment of the safe authoring goal.

#### 3.2.4. Expressiveness

In order to be actually usable, a template language and hence a slot markup language must offer a well-balanced amount of expressiveness. The language must support control statements, especially for the *conditional* and *repeated* inclusion of template parts.

The absence of these control features typically leads to a violation of the separation of concerns goal. Without the control features, an author basically needs to separate conditional and repeated parts into (sub-)templates that are instantiated conditionally or repeatedly into fragments that are, in turn, used as instantiation data for the (master-)template. The effort of creating and maintaining these (sub-)templates as separate resources then leads to the embedding of template parts into the code using the template engine.

There is a risk of exaggerating the expressiveness of the slot markup language. This extra amount of power available to the template developer typically leads to application code being embedded in a template, a situation which is hard to detect and even harder to eliminate.

With respect to the area of Web applications with their typical division between application and presentation layer (resp. controller and view in the MVC pattern), the consequences of both insufficient and exaggerated expressiveness are shown in Figure 3.4.

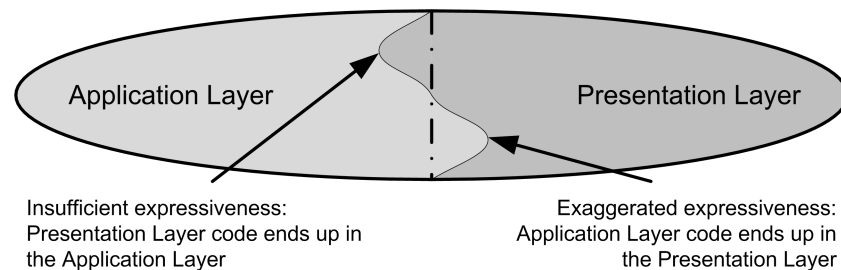


Figure 3.4.: Consequences of Insufficient or Exaggerated Expressiveness

The *expressiveness* requirement addresses the separation of concerns goal, as both insufficient and exaggerated control lead to violations of the goal.

Almost all existing template techniques support the conditional and the repeated inclusion of template fragments.

#### 3.2.5. Instantiation Data Type Safety

Incorrect treatment of unexpected instantiation data items is a major source for problems during the instantiation of templates. Unfortunately, the instantiation data is, by definition, not available during the authoring time of a template. In order to detect problems with the instantiation data, it is therefore necessary to specify a contract between a template and the used instantiation data.



### 3.3. Proposal of an Architecture fulfilling the Requirements

Such a contract consists of constraints asserting properties of the instantiation data, especially concerning the type of the data. The *instantiation data type safety* requirement enforces that the type of the instantiation data items must be checked by a template technique.

This requirement addresses the *safe instantiation* goal: the required contract between the template and the instantiation data can be used to clearly communicate problems with the instantiation data to users of the technique.

#### 3.2.6. Independence of Query Language

To be usable independently of a specific source of instantiation data, a template technique should be designed to be capable of dealing with *any* query language.

This is especially important since different types of instantiation data may have completely different access mechanisms. For example, the query language for accessing an XML document as data source can be XPath, while a template that should directly access a relational database would use the Structured Query Language (SQL) for the same purpose.

It is important to note that this requirement can only be fulfilled to a certain degree. Allowing a query language to alter the state of the data used to fill the template seriously injures the separation of concerns goal. This issue was discussed in detail in [143].

The *independence of query language* requirement addresses the broad applicability goal.

Existing approaches differ in their independence of the query language. Some approaches use a general programming language as query language, i.e., these approaches are itself strictly bound to a particular query language, which, however, allows employing arbitrary query languages using its own language means. For example, a JSP engine can use XPath [9] within Java to access XML documents using XPath.

Other approaches have a fixed query language that is capable of operating on different meta-models. An example for this approach is XPAND [56]. In these approaches, the query language itself delivers the flexibility of accessing multiple instantiation data sources.

### 3.3. Proposal of an Architecture fulfilling the Requirements

Based on the outlined requirements, we propose an architecture that enables safe template processing [82]. The architecture consists of six solution elements, which are addressing the various requirements described in Section 3.2. In the following, the meaning of each element of the architecture is explained in detail. As the architecture itself is independent of a particular target language, the descriptions can be applied to any implementation of the architecture.

The solution elements can be assigned to the life cycle phases introduced in Section 2.1.2. Information is passed between the particular solution elements from elements in earlier life cycle phases to latter ones. Figure 3.5 shows the solution elements, their assignment to life cycle phases and the flow of information between the elements. The relations between the solution elements and the requirements are shown in Figure 3.6.

The *Slot Markup Language Design* process creates the basis for the template technique: the slot markup language itself. Both concrete syntax and semantics must be designed carefully to allow other solution elements to rely on it. This process must deliver a grammar for the

### 3. Safe Template Processing

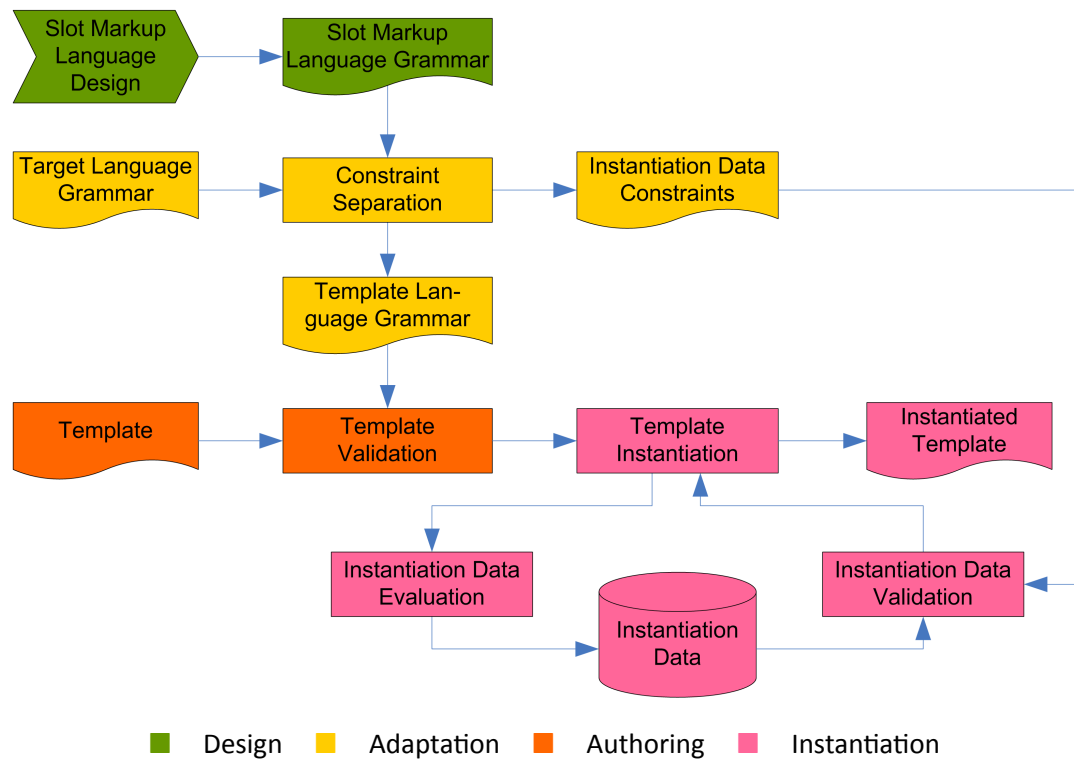


Figure 3.5.: The Proposed Architecture

language elements in the slot markup language in a machine-readable form. The design of the slot markup language can be considered a part of the design phase of a template technique.

The requirements Preservation, Coverage, Expressiveness and Independence of Query Language had substantial influence on the Slot Markup Language Design. It therefore contributes to the goals of Safe Authoring, Separation of Concerns, Broad Applicability, and Utilization of Existing Standards.

If the target languages to be produced by the template technique are restricted to be XML dialects, the machine-readable form of the slot markup language grammar would preferably be an XML Schema. Chapter 4 describes the design of a slot markup language targeting arbitrary XML dialects in detail.

Next, we propose a *Constraint Separation* component, which adapts the template technique to a particular target language by combining the grammars of the slot markup language and the target language and transforming them into the grammar of the template language and a set of instantiation data constraints. This component is part of the adaptation phase of a template technique.

The design of the Constraint Separation largely depends on the Preservation requirement and addresses the goals of Safe Authoring and Utilization of Existing Standards. The component is described in more detail in Section 5.1.

### 3.3. Proposal of an Architecture fulfilling the Requirements

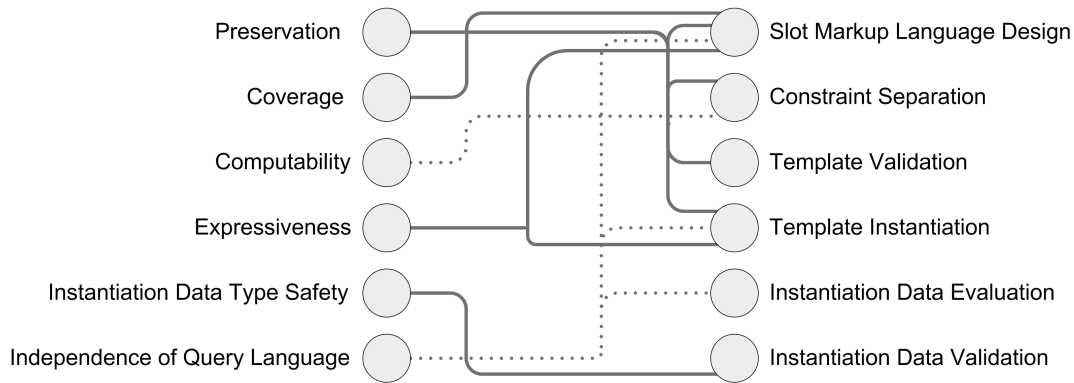


Figure 3.6.: Relations between Requirements and Solution Elements

The *Template Validation* component applies the template language grammar produced by the Constraint Separation process to check the validity of the templates created by an author. It belongs to the authoring phase. A successful validation asserts the author that the template will instantiate into the target language as long as the instantiation data complies to the instantiation data constraints emitted by the Constraint Separation.

The component performing the Template Validation is influenced by the requirement Preservation, i.e., it addresses the goals of Safe Authoring and Utilization of Existing Standards. A detailed description of the component can be found in Section 5.2.

A validated template can be used by the *Template Instantiation* process to produce a document from the target language within the instantiation phase. For the instantiation, instantiation data is needed, which is delivered by the Instantiation Data Evaluation process.

The template engine, the component performing the Template Instantiation process, is determined by the requirements of Expressiveness, Independence of Query Language and Preservation. It therefore addresses the goals of Separation of Concerns and Broad Applicability. A detailed description of an efficient template engine design is given in Section 6.2.

As already mentioned, the instantiation data consumed by the Template Instantiation process is delivered by the *Instantiation Data Evaluation* process, which is fetching the data from some instantiation data source.

The Independence of Query Language requirement is the main determinant for the component implementing this evaluation process. The component therefore contributes to the Broad Applicability goal. A design for this component is introduced in Section 6.1.

The Instantiation Data Evaluation only delivers instantiation data from a data source, but is not capable of asserting its properties; instead, the *Instantiation Data Validation* process is responsible for these assertions.

Obviously, the component implementing the Instantiation Data Validation process depends on the Instantiation Data Type Safety requirement and addresses the Safe Instantiation goal. The component is described in detail in Section 6.3. An alternative approach for addressing the same requirements and goals is the Template Interface Generation approach described in Section 6.3.2.

### 3.4. Conclusion

This chapter analyzed the problems introduced in Section 1.3 in order to define goals for the approach to be developed by this thesis. The goals have been used to set up a number of requirements. Based on the requirements, an architecture has been proposed that is (for the moment, presumably) fulfilling the requirements and therefore helps reaching the goals. The relations between the goals, the requirements and the solution's elements have been discussed in detail, which is important for understanding tradeoffs made during the design and the implementation of the approach.

The following chapters are structured as follows. Chapter 4 discusses the design of a slot markup language. Chapter 5 describes the solution elements of the architecture that are assigned to the adaptation phase or the authoring phase. Finally, Chapter 6 discusses the solution elements assigned to the instantiation phase. These relations are illustrated in Figure 3.7. Please note that starting with Chapter 4, the thesis deals with XML target languages, thereby restricting the general discussions and proposals made so far to the XML technological space.

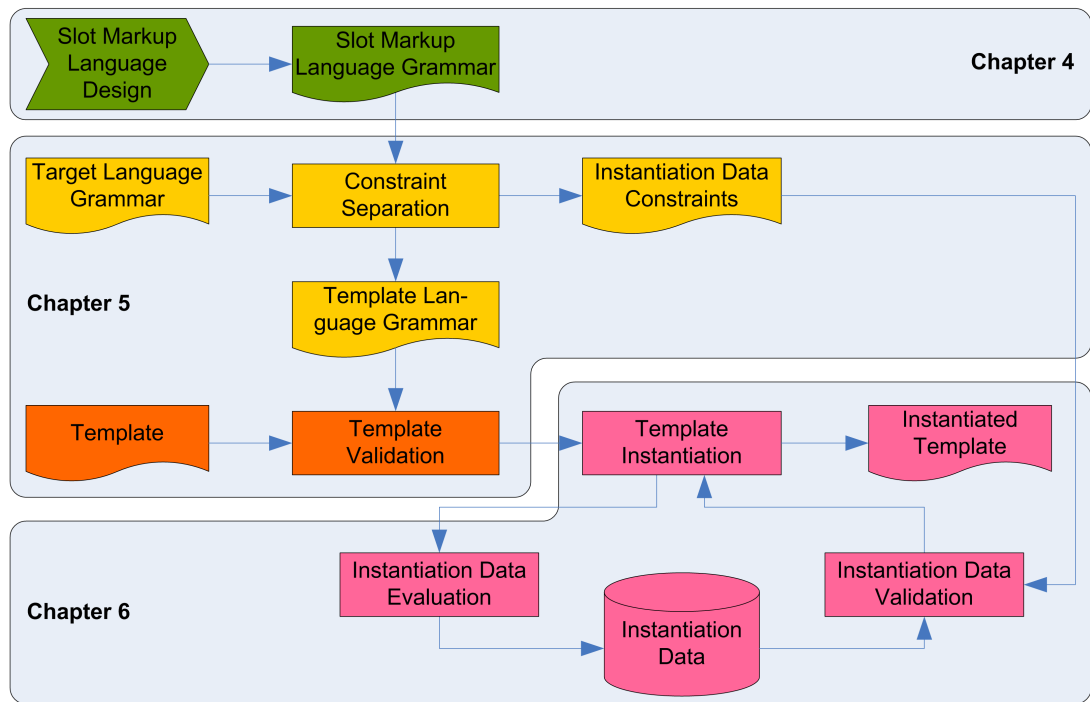


Figure 3.7.: Relations between the Solution Elements and the Following Chapters

# 4

## Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

Ein Loch ist da, wo etwas nicht ist.

Kurt Tucholsky, 1931 [184]

One of the most important steps in the implementation of the approach proposed in Chapter 3 is the design of a slot markup language. The Separation of Concerns goal proposed in Section 3.1.3 requires the slot markup language to enable the user to incorporate the template engine without having to violate the intended separation of concerns. Thus, the design of this language determines whether the architecture is acceptable for a given purpose. Furthermore, the design of the slot markup language influences the solution elements of the approach as described in Section 3.3. The template engine must be implemented depending on the features of the slot markup language. In addition to this, the expressive power of the grammar needed to describe the template language also depends, besides on the target language, on the design of the slot markup language. An example for such a language, named XML Template Language (XTL) is shown in detail in this chapter.

Section 4.1 explains the decisions made during the design of the XTL. The following Section 4.2 introduces the language features that allow for creating XML document character data, whereas Section 4.3 shows features for the conditional or repeated inclusion of template fragments. Section 4.4 introduces macros, which enable reuse within XTL templates. Two special XTL features,

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

namely realms and bypassing, are explained in Section 4.5. Furthermore, Section 4.6 outlines how the semantics of XTL can be described translationally in relation to XSL-T. Finally, Section 4.7 discusses the application of XTL to a different domain, namely as a schema language for the validation of XML documents.

### 4.1. General Design Decisions

As mentioned in Section 1.3, most template techniques available today are not guaranteeing the wellformedness of instantiated templates. In order to fulfill the goal of safe authoring, the wellformedness of the instantiation result must be guaranteed. There are several ways to achieve this, which can be divided into syntax-preserving and syntax-destructing approaches, as introduced in Section 2.5.1.

Syntax-destructing approaches have the disadvantage of being hard to implement (as most of the tools have to be reimplemented to reflect the template language syntax) but often offer a template language with slightly improved readability. Syntax-preserving approaches have the advantage of enabling the reuse of existing tools (i.e., incidentally also addressing the Utilization of Existing Standards goal) for the syntax of the target language. The readability of these template languages can often be improved by lightweight changes like adjustments of the syntax highlighting for the slot markup parts of the template language.

It has been decided that XTL should be a syntax-preserving slot markup language. The language feature that should be used by XTL is the XML namespace `http://research.sap.com/xtl/1.0` particularly reserved for the XTL. Thus, XTL templates are wellformed XML documents that contain slots designated by nodes that belong to the XTL namespace. In addition to asserting the wellformedness of the instantiation results, using an XML namespace and refraining from non-XML slot markup syntax enables the use of standard XML schema languages (like XML Schema [59; 180; 26]) to describe the template language grammar in the first place.

Next, the expressiveness of XTL had to be decided. Therefore, it was necessary to define which features should be supported by the language. Basically, these features fall into one of three categories: features supporting the creation of XML nodes, features allowing to control the instantiation and features for the reuse of template fragments. The design of the particular features is described in Section 4.2, 4.3 and 4.4, respectively.

Several additions to the XTL language have been considered. Basically, they can be grouped into two categories: elements which may be added without injuring the safe authoring approach and elements that will seriously harm this approach.

#### Syntax

The normative definition of XTL is the XML Schema document `XTL.xsd`. As the XML Schema syntax itself is very verbose, the syntax of the elements is explained textually instead of by showing fragments of the schema. The complete schema can be found in Appendix A.1.

## Semantics

For the core language elements of XML, a denotational semantics for the instantiation is given below. This semantics is given as a Haskell [182] program. It operates on a simplified XML model that uses the type shown in Figure 4.1 to represent XML documents. This data type closely resembles the XML data model introduced in Section 2.1.3.

```
data Node =
  Text String |
  Comment String |
  Element QName (Map QName String) [Node]
```

Listing 4.1: Representation of XML documents in the Instantiation Semantics

Please note that the `QName` is representing a triple consisting of three strings: a prefix, a local name and a namespace URI, i.e., it extends a qualified name in the sense of [29] with the capability of keeping the prefix. An example for a `QName` would be the triple `("xt1", "text", "http://research.sap.com/xt1/1.0")`.

Furthermore, the denotational semantics accesses instantiation data using a clearly defined interface named `IDS`. The interface is defined using a Haskell class and is shown in Listing 4.2. It basically consists of five functions which are explained at the parts of the semantics where they are used.

```
type IDS a = (a -> String -> String, a -> String -> [a], a -> String
  -> Bool, a -> String -> [Node], a)

evalText :: IDS a -> (a -> String -> String)
evalText (text, _, _, _, _) = text

evalForEach :: IDS a -> (a -> String -> [a])
evalForEach (_, forEach, _, _, _) = forEach

evalIf :: IDS a -> (a -> String -> Bool)
evalIf (_, _, if_, _, _) = if_

evalInclude :: IDS a -> (a -> String -> [Node])
evalInclude (_, _, _, include, _) = include

root :: IDS a -> a
root (_, _, _, _, root) = root
```

Listing 4.2: Definition of the `IDS` class

The denotational semantics starts with the function `instantiateDocument` shown in Listing 4.3. This function takes an Instantiation Data Source (`IDS`) and a representation of an XML document (serving as template) and yields the instantiated template. This function triggers the

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

instantiation by calling the function `instantiateNodes` on the children of the root node (which can, by definition, never be an element defined by XTL).

The function `instantiateNodes` has a quite complicated signature: it takes three arguments and delivers a 3-tuple as result. The first parameter is a map of macros, which is explained in more detail in the Sections 4.4.1 and 4.4.2. The second parameter is the IDS needed to evaluate instantiation data. The third parameter is the list of nodes to be instantiated. The 3-tuple returned by the function contains a (possibly modified) map of macros, a mapping from expanded names to strings representing a set of attributes and a list of nodes created during the instantiation. The function takes the first element from the list of nodes to be instantiated and processes it by calling the `instantiateNode` function. Afterwards, it calls itself on the remainder of the list. The return value is created from the result of both calls by combining the returned attributes and concatenating the returned children.

```
type AttrMap = Map QName String
type MacroMap = Map String [Node]

instantiateDocument :: IDS p -> Node -> Node
instantiateDocument ids (Element qn attributes children) =
  let
    (_, attributes1, children1) = instantiateNodes ids (root
      ids) empty children
  in
    Element qn (union (transformNamespaceAttributes attributes)
      attributes1) children1

instantiateNodes :: IDS p -> p -> MacroMap -> [Node] -> (MacroMap,
  AttrMap, [Node])
instantiateNodes ids context macros (node:further) =
  let
    (macros1, attributes1, children1) = instantiateNode ids
      context macros node
    (macros2, attributes2, children2) = instantiateNodes ids
      context macros1 further
  in
    (macros2, attributes1 'union' attributes2, children1 ++
      children2)
instantiateNodes ids context macros [] = (macros, empty, [])

instantiateNode :: IDS p -> p -> MacroMap -> Node -> (MacroMap,
  AttrMap, [Node])
```

Listing 4.3: Preamble of the Denotational Instantiation Semantics

The `instantiateNode` function has a signature similar to that of `instantiateNodes`, but takes only a single node as its third parameter. The implementation of that function is given below in the Sections 4.2.1 to 4.5.2. The treatment of text, comment nodes and element nodes is shown in Listing 4.4. Since there is a special treatment of elements assigned to the bypassing



namespace explained in Section 4.5.2, a boolean guard is used to restrict the element processing by these default rules to elements not assigned to the bypassing namespace. Elements from the XTL namespace are treated by special rules, which are shown below, but are to be found *before* the default rules in the complete semantics.

```

instantiateNode ids context macros (Text text) = (macros, empty,
  [Text text])
instantiateNode ids context macros comment@(Comment _) = (macros,
  empty, [comment])
instantiateNode ids context macros (Element qn@(QN prefix
  namespaceURI localName) attributes children) | not (isBypassURI
  namespaceURI) =
  let
    (macros1, attributes1, children1) = instantiateNodes ids
      context macros children
  in
    (macros1, empty, [Element qn (union attributes attributes1)
      children1])

```

Listing 4.4: Semantics for Text, Comment and Element Nodes

### Examples

Since XTL has been designed to fulfill the requirement of independence of the query language, the query language is arbitrary. In the examples below, XPath is used as the query language. As the instantiation data, the *purchase order example* document `po.xml` (from [59], see also Section A.3) is used. This means that the values of the `select` attributes below must be read as XPath expressions targeting `po.xml`.

## 4.2. Creation of Character Data

For each XML node type (like element, comment, attribute etc.), there could have been a corresponding XTL language feature allowing to dynamically create the node from the instantiation data instead of statically including it in the template. Fully supporting this 1:1-relationship would violate the separation of concerns goal, as it would allow the arbitrary creation of element nodes (identified via their names) from instantiation data.

As element nodes are not character data in the sense of [28, Section 2.4], but rather markup, they should never be subject to dynamic creation. The same is true for attribute names. On the other hand, text nodes and attribute values are character data in an XML document, and their creation from instantiation data must therefore be supported by the XTL. Therefore, XTL supports the dynamic creation of text as described in Section 4.2.1 and the dynamic creation of attribute values as described in Section 4.2.2, but does not offer a feature to dynamically create elements. An exception to this latter statement is the dynamic inclusion of XML fragments as described in Section 4.2.3, where the drawbacks of this feature are explained as well.

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

The dynamic creation of comments and processing instructions would be possible, but has not been included in the current version of XTL.

##### 4.2.1. `xtl:text`

There are basically four design options for an instruction intended to create a text node. First, an element of the target language can be used for that purpose—the instantiation data would then be denoted via an attribute of that element (e.g., XMLC uses HTML’s `span` tag with its `id` attribute for that purpose). Second, an attribute at the parent element could theoretically be used, but this attribute has to denote several things: where to insert the text node, if multiple children exist, and where to get the instantiation data from. Third, a comment could be used to denote the position of the text node to be created—the comment content could then be used to denote the instantiation data. Finally, the slot markup language could contain an instruction solely designed for the purpose of creating text nodes.

The reuse of a target language element is not possible in a *generic* slot markup language, as the target language is arbitrarily exchangeable by definition. Furthermore, the use of an attribute at the parent element would severely harm the understandability of the XML template and in addition to that, the position at which the text node is to be inserted must be updated together with the content of the element, which is a potential source of errors. The third option—abusing comments for the creation of text nodes—contradicts the requirement of preservation, as comments are not enforceable using XML schema languages.

Therefore, XTL follows the straightforward approach and contains an instruction `xtl:text` that is replaced with instantiation data during the instantiation.

##### Syntax

The `xtl:text` element supports two attributes: one for the description of the instantiation data to be used to replace the element and one for the support of realms, which is described in detail in Section 4.5.1.

The `select` attribute contains a string of the query language which is passed to the instantiation data evaluator. The string is evaluated within a certain context: if the `xtl:text` is not contained in any `xtl:for-each` instruction, the context is the entirety of the instantiation data. For the treatment of `xtl:text` within `xtl:for-each`, refer to the description of `xtl:for-each` in Section 4.3.2.

##### Semantics

In Listing 4.5, the function `evaluateText` is used to determine the instantiation data item to be used for replacing the `xtl:text` instruction. In order to prevent `xtl:text` to be used to create markup, the instantiation data item evaluated must be processed in the way described in [28, Section 2.4], i.e., all ampersand characters `&` and the left angle bracket `<` must be replaced by the strings `&amp;` and `&lt;`, respectively, by all valid XTL engine implementations. In the denotational semantics, this escaping process is performed by the call to the function `escapeText`.

The `evaluateText` method has to return a string value. The conversion of non-string values returned by the evaluation into a string value is up to the concrete implementation of this function. For XPath, a natural choice would be to follow XSL-T in its use of the XPath function `string` to convert the query result into a boolean value [38, Section 4.2].

```
instantiateNode ids context macros (Element (QN _ "text"
  "http://research.sap.com/xtl/1.0") attributes _) =
  let
    selectExpr = attributes ! (QN "" "select" "")
  in
    (macros, empty, [Text (escapeText (evalText ids context
      selectExpr))])
```

Listing 4.5: Semantics of `xtl:text`

### Example

Listing 4.6 shows how `xtl:text` could be used to create a text node—in this case as a subnode to the `name` element literally contained in the template.

	Template		
	<pre>&lt;?xml version="1.0"?&gt; &lt;sample xmlns:xtl="http://research.sap.com/xtl/1.0"&gt;   &lt;name&gt;&lt;xtl:text select="/purchaseOrder/shipTo/name"/&gt;&lt;/name&gt; &lt;/sample&gt;</pre>		
	Instantiation Result		
	<pre>&lt;?xml version="1.0" encoding="UTF-8"?&gt; &lt;sample&gt;   &lt;name&gt;Alice Smith&lt;/name&gt; &lt;/sample&gt;</pre>		

Listing 4.6: Example Use of `xtl:text`

#### 4.2.2. `xtl:attribute`

Creating attributes is substantially more complicated than creating text. Basically, two options arise. First, the slot may be marked up using the attribute value itself (e.g., via a special syntax like in `href="$url"`). This has the advantage of being easy to read, but the problem that the special syntax must be encoded by the author if it is used without being meant as slot markup. The second option is to use an XML element to dynamically create the attribute from instantiation data. This refrains the user from encoding any special markup: if the attribute exists at the element, it has to be taken into the instantiated template *as is*. The decision to make XTL use the second option via an `xtl:attribute` element as defined here supports the goal of Safe Authoring, as the author is freed from dealing with encoding special markup.

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

The Separation of Concerns goal is not affected by this decision, as `xtl:attribute` only allows the dynamic creation of the attribute value, but not of the attribute name.

The construction of attributes is supported by the XTL using the element `xtl:attribute`, which is adding an attribute to its parent element. The attribute has a fixed name (i.e., the name is not taken from the instantiation data) and a value taken from the instantiation data. In Listing 4.8, `xtl:attribute` is used to create a date attribute at the order element.

##### Syntax

The `xtl:attribute` element supports four attributes. As with `xtl:text`, one of the attributes is for the support of realms, which is described in detail in Section 4.5.1.

The name attribute defines the name of the attribute to be created by `xtl:attribute`. Its value must be a QName, which allows for inserting qualified and unqualified attributes. As already mentioned above, the value is static, i.e., it is not possible to create an attribute with a name taken from the instantiation data.

As `xtl:text`, `xtl:attribute` supports the `select` attribute. Its value is used to get the instantiation data to be used as the attribute value. Again, this string is evaluated within a certain context: if the `xtl:attribute` is not contained in any `xtl:for-each` instruction, the context is the entirety of the instantiation data. For the treatment of `xtl:attribute` within `xtl:for-each`, refer to the description of `xtl:for-each` in Section 4.3.2.

Finally, a mode attribute could be used to define the precedence of an attribute created by `xtl:attribute` compared to a literally specified attribute of the same QName. By default, the literally contained attribute would be overwritten. Using this attribute, it is possible, for example, to append the dynamically created value to the literal one. As this functionality is possibly harming the requirement of preservation, its use is only allowed if the attribute to be created is defined to be of the XML Schema type `String`.

##### Semantics

In Listing 4.7, the function `evaluateText` is *reused* to determine the instantiation data item to be used as the value for the attribute to be created. In order to prevent `xtl:attribute` to create multiple attributes (an attack typically used against Web applications known as markup injection), the evaluated instantiation data must be processed in the way described in [28, Section 2.4]. This is similar to the processing in `xtl:text`, but in addition to the replacements made there, single quotes `'` and double quotes `"` must also be replaced by the strings `&apos;` and `&quot;`, respectively. This escaping is performed by the call to the function `escapeAttributeValue` in Listing 4.7.

The processing of the mode attribute is not included in the denotational syntax for readability reasons.

```
instantiateNode ids context macros (Element (QN prefix "attribute"
  "http://research.sap.com/xtl/1.0") attributes _) =
  let
    name = attributes ! (QN "" "name" "")
    selectExpr = attributes ! (QN "" "select" "")
```

```

in
  (macros, singleton (mkQName name) (escapeAttrValue (evalText
    ids context selectExpr)), [])
where
  mkQName :: String -> QName
  mkQName s = case elemIndex ':' s of
    Nothing -> QN "" s ""
    Just idx -> QN "" (drop (idx+1) s) (take (idx-1) s)

```

Listing 4.7: Semantics of `xtl:attribute`**Example**

In Listing 4.8, the `xtl:attribute` element is used to create a data attribute at the element containing the `xtl:attribute`, namely the `order` element literally contained in the template.

Template
<pre> &lt;?xml version="1.0"?&gt; &lt;sample xmlns:xtl="http://research.sap.com/xtl/1.0"&gt;   &lt;order&gt;     &lt;xtl:attribute name="date"       select="/purchaseOrder/@orderDate" /&gt;   &lt;/order&gt; &lt;/sample&gt; </pre>
Instantiation Result
<pre> &lt;?xml version="1.0" encoding="UTF-8"?&gt; &lt;sample&gt;   &lt;order date="1999-10-20"/&gt; &lt;/sample&gt; </pre>

Listing 4.8: Example Use of `xtl:attribute`**4.2.3. `xtl:include`**

XTL offers an `xtl:include` element that can be used to dynamically include complete XML fragments (consisting of multiple XML nodes) into the instantiation result.

Strictly speaking, `xtl:include` is also a partial violation of the independence of query language requirement, as it can not be asserted that every query language is capable of delivering an XML fragment that could be inserted by the template engine. Despite of this, the `xtl:include` statement has been added for two reasons: fragment inclusion is a very powerful language feature and query languages not capable of delivering XML fragments could be adapted to create XML from query results.

Even more questionable is the fact that `xtl:include` can be used to generate markup and character data. This may violate the requirement of separation of concerns. Therefore, the

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

`xtl:include` element is not included in the subset of XML supported by the safe authoring approach (for details, see Section 5.1).

##### Syntax

The `xtl:include` element supports the same two attributes as `xtl:text`: a `select` attribute for the description of the instantiation data to be used to replace the element and one for the support of realms, which is described in detail in Section 4.5.1.

##### Semantics

In Listing 4.9, the function `evaluateInclude` is reused to determine the instantiation data item to be used as the value for the attribute to be created. This function has to return nodes of an XML document, therefore there is no need for escaping special XML characters: they must already have been replaced in the instantiation data.

```
instantiateNode ids context macros (Element (QN _ "include"
  "http://research.sap.com/xtl/1.0") attributes _) =
  let
    selectExpr = attributes ! (QN "" "select" "")
  in
    (macros, empty, evalInclude ids context selectExpr)
```

Listing 4.9: Semantics of `xtl:include`

##### Example

In Listing 4.10, the `xtl:include` element is used to include all name elements together with the contained text node from `po.xml` into the instantiated template. Please note that this example shows that character data *as well as* markup are created using this instruction.

Template
<pre>&lt;?xml version="1.0" encoding="UTF-8"?&gt; &lt;sample xmlns:xtl="http://research.sap.com/xtl/1.0"&gt;   &lt;xtl:include select="//name"/&gt; &lt;/sample&gt;</pre>
Instantiation Result
<pre>&lt;?xml version="1.0" encoding="UTF-8"?&gt; &lt;sample&gt;   &lt;name&gt;Alice Smith&lt;/name&gt;   &lt;name&gt;Robert Smith&lt;/name&gt; &lt;/sample&gt;</pre>

Listing 4.10: Example Use of `xtl:include`

### 4.3. Conditional and Repeated Inclusion of Template Fragments

Following the argumentation in Section 3.2.4, there is a need to support the conditional and repeated inclusion of template fragments in order to fulfill the expressiveness requirement. Obviously, both features have their counterparts in general purpose programming languages like Java.

#### 4.3.1. `xsl:if`

Typical conditional statements include simple if statements (with or without else branches), expanded if statements (with multiple conditions like `if...elseif...`) and switch statements. If one abstracts from the concrete syntax, these statements can be classified by how many of their branches can be selected. For example, a standard `if...then...else...` statement chooses exactly one of the two branches, whereas the `switch` statement in Java may select none, one or more branches (mostly depending on the use of `break` statements within the branches).

The independence of query language and the preservation requirement prohibit the introduction of a statement that allows the selection of multiple branches into XTL. The first requirement prevents deciding how many branches can be selected. The second requirement would be hard to fulfill in the presence of such a statement as all combinations of branches must be checked for their validity within the target language.

Therefore, XTL can only support conditional inclusion statements that select *at most one* of its branches. This only requires that the content of each branch is valid within the target language.

From the syntactical point of view, it must be decided whether the conditional statement should be implemented as an attribute (like in TAL) or as an element (comparable to the `if` statement in XSL-T). The second approach is more comfortable, but also harder to implement.

In fact, the current XTL version only supports a very simple `xsl:if` statement that only allows one branch to be included or not.

#### Syntax

The `xsl:if` element supports the two attributes also known from `xsl:text`: one for the description of the instantiation data to be used to replace the element and one for the support of realms, which is described in detail in Section 4.5.1.

The `select` attribute contains a string from the query language that is passed to the instantiation data evaluator. The string is evaluated within a certain context: if the `xsl:if` is not contained in any `xsl:for-each` instruction, the context is the entirety of the instantiation data. For the treatment of `xsl:if` within `xsl:for-each`, refer to the description of `xsl:for-each` in Section 4.3.2.

As opposed to the XTL elements described above, the `xsl:if` element is not declared to be empty, but rather allows a sequence of arbitrary elements as its content. These elements are the content that is conditionally included in the instantiated template, depending on the result of the evaluation of the `select` attribute. The children of `xsl:if` are evaluated during the

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

instantiation, i.e., other XTL statements can be included. This also includes the use of `xtl:attribute`, which allows the conditional creation of attributes.

##### Semantics

In Listing 4.11, the function `evaluateIf` is used to determine the instantiation data item needed to decide whether the children of the `xtl:if` element are processed and inserted into the instantiated template or not.

Since this decision has two alternatives, the `evaluateIf` method has to return a boolean value. It is up to the concrete implementation of the `evaluateIf` function whether true or false has to be returned if the query string does not evaluate into a boolean value. For XPath, a natural choice would be to follow XSL-T in its use of the XPath function `boolean` to convert the query result into a boolean [38, Section 4.3].

If the evaluation of the instantiation data item into a boolean value yields true, the content of the `xtl:if` element is processed by the `instantiateNodes` function and the result of this processing becomes the result of processing the `xtl:if` element. If the evaluation yields false, an empty attribute map as well as an empty child list is returned.

```
instantiateNode ids context macros (Element (QN _ "if"
      "http://research.sap.com/xtl/1.0") attributes children) =
  let
    selectExpr = attributes ! (QN "" "select" "")
  in
    if (evalIf ids context selectExpr)
      then
        instantiateNodes ids context macros children
      else
        (macros, empty, [])
```

Listing 4.11: Semantics of `xtl:if`

##### Example

An example for `xtl:if` is shown in Listing 4.12, which also shows that an else branch can be simulated with the most query languages.

---

Template

---

```
<?xml version="1.0" encoding="UTF-8"?>
<sample xmlns:xtl="http://research.sap.com/xtl/1.0">
  <xtl:if select="count(//items/item)=2">
    <fulfilled id="1"/>
  </xtl:if>
  <xtl:if select="not(count(//items/item)=2)">
    <fulfilled id="2"/>
  </xtl:if>
</sample>
```



## Instantiation Result

```
<?xml version="1.0" encoding="UTF-8"?>
<sample>
  <fulfilled id="1"/>
</sample>
```

Listing 4.12: Example Use of `xtl:if`**4.3.2. `xtl:for-each`**

In typical programming languages, especially in languages following an imperative or object-oriented paradigm, a variety of statements for the repeated execution of fragments can be found. These statements can be classified into being controlled by conditions, by a counter or by a collection.

Condition-controlled statements are typically classified by the time at which the condition is evaluated: at the start of the statement (in Java represented by a mere `while` statement) or at its end (in Java represented by the `do...while` statement). The statements may also support an option for an early exit, which allows the repetition to be immediately exited.

The condition-controlled statements are typically used to make repetitions depend on evaluation results obtained inside the statement. As there is no possibility to perform calculations in XTL itself, such a statement does hardly make sense.

Count-controlled statements can be considered a special form of collection-controlled statements, if collections of a given size (corresponding to *count*) can be constructed.

Syntactically, the situation is similar to `xtl:if`: the statement for repeated inclusion can be implemented as an attribute (like in TAL) or as an element.

For that reasons, XTL only supports one statement for the repeated inclusion of template fragments, namely `xtl:for-each`. Currently, there is no statement for an early exit. Thus, XTL is quite similar to XSL-T in its support for repetition.

**Syntax**

The `xtl:for-each` element supports four attributes: one for the description of the instantiation data item to be used as the collection for controlling the repetition, two for the specification of ordering the collection before using it and one for the support of realms, which is described in detail in Section 4.5.1.

The `select` attribute contains a string from the query language that is passed to the instantiation data evaluator. The string is evaluated within a certain context: if the `xtl:for-each` is not contained in any `xtl:for-each` instruction, the context is the entirety of the instantiation data. For the treatment of `xtl:for-each` within `xtl:for-each`, refer to the semantics of `xtl:for-each` described below.

The `order-by` attribute also contains a string from the query language that is intended to be evaluated by the instantiation data evaluator. The result is used to sort the elements of the collection obtained from the `select` attribute. The `order` attribute determines whether the elements should be sorted in ascending or descending order.

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

Differently from the XTL elements described above, the `xtl:for-each` element is not declared to be empty, but rather allows a sequence of arbitrary elements as its content. For obvious reasons, the use of `xtl:attribute` as a (direct) child is prohibited, nevertheless, `xtl:attribute` elements may be (indirect) descendants of `xtl:for-each` elements.

The children of `xtl:for-each` are the content that is to be repeated in the instantiated template, depending on the result of the evaluation of the `select` attribute.

##### Semantics

In Listing 4.13, the function `evaluateForEach` is used to determine an instantiation data item collection. If the result returned by evaluating the query string taken from the `select` attribute is not a collection, it is the responsibility of the concrete implementation of `evaluateForEach` to convert it into a list (which may be empty).

The collection is interpreted as a list of contexts, as they are passed as an argument to the `evaluateText`, `evaluateInclude`, `evaluateIf` and `evaluateForEach` functions. Thus, `xtl:for-each` is the only XTL element that is capable of changing the context which determines the root for the evaluation of `select` attributes.

Optionally, it may be necessary to sort the collection returned by `evaluateForEach`. This happens if an `order-by` attribute has been specified. The value of this attribute is evaluated by calling `evaluateText` for each of the elements in the collection as context and sorting the collection corresponding to the returned values.

The content of the `xtl:for-each` element is evaluated once for each element from the collection by calling `instantiateNodes` and passing the current element from the collection as the context for the instantiation.

The mechanism of establishing a new context within `xtl:for-each` is basically similar to the notion of the context item in XSL-T. As a consequence, relative XPath expressions are similarly evaluated in XTL and XSL-T.

Since the syntax disallows the use of `xtl:attribute` as a child of `xtl:for-each`, the `instantiateNode` method returns always an empty attribute map for `xtl:for-each`.

It is also important to note that the use of the `order-by` attribute can seriously slow down the instantiation of a template, as the whole collection must be evaluated before sorting can take place. If no `order-by` attribute is specified, the evaluation of the collection can instead take place lazily.

```
instantiateNode ids context macros (Element (QN _ "for-each"
  "http://research.sap.com/xtl/1.0") attributes children) =
  let
    selectExpr = attributes ! (QN "" "select" "")
    contexts = evalForEach ids context selectExpr
    orderedContexts = orderContexts ids attributes contexts
    result = map (\currentContext -> instantiateNodes ids
      currentContext macros children) orderedContexts
    allChildren = map (\(macros, attributes, children) ->
      children) result
  in
```

### 4.3. Conditional and Repeated Inclusion of Template Fragments

```
(macros, empty, concat allChildren)
where
  orderContexts :: IDS p -> AttrMap -> [p] -> [p]
  orderContexts ids attributes contexts =
    if (QN "" "order-by" "") 'member' attributes
    then
      let
        orderBy = attributes ! (QN "" "order-by" "")
        order = findWithDefault "ascending" (QN ""
          "order" "") attributes
        ascOrdering c1 c2 = compare (evalText ids c1
          orderBy) (evalText ids c2 orderBy)
        ordering = (if order == "ascending" then id else
          flip) ascOrdering
      in
        sortBy ordering contexts
    else
      contexts
```

Listing 4.13: Semantics of `xtl:for-each`

#### Example

In Listing 4.14, `xtl:for-each` is used to create a number of empty `item` elements which have an attribute named `price` which has the value of the `USPrice` element corresponding to the item.

---

#### Template

---

```
<?xml version="1.0" encoding="UTF-8"?>
<sample xmlns:xtl="http://research.sap.com/xtl/1.0">
  <xtl:for-each select="//items/item">
    <item>
      <xtl:attribute name="price" select="USPrice/text()"/>
    </item>
  </xtl:for-each>
</sample>
```

---

#### Instantiation Result

---

```
<?xml version="1.0" encoding="UTF-8"?>
<sample>
  <item price="148.95"/>
  <item price="39.98"/>
</sample>
```

Listing 4.14: Example Use of `xtl:for-each`

## 4.4. Reuse of Template Fragments

An important addition is the introduction of a macro mechanism, as this allows to use a *transformational* style within the template and, as a consequence, abolishes the limitation of pure prototypical templates, in which the depth of the instantiated template is a linear function of the depth of the template.

### 4.4.1. `xtl:macro`

Many of the slot markup languages support to reuse fragments of the template or target language. As explained in Section 2.5.7, there is a variety of design options for macro mechanisms.

XTL only supports the most basic notion of macros. Neither parameter passing nor other advanced techniques are supported. Macros are defined using `xtl:macro`.

#### Syntax

`xtl:macro` supports only one attribute, the name attribute declaring the name of the macro. This value must be a string. The content of the `xtl:macro` element is assigned to the name literally. The use of `xtl:macro` is restricted: `xtl:macro` can only be used as direct child of a template's root element and no element nodes besides the root element and other `xtl:macro` (as well as `xtl:init`) elements are allowed to precede it.

#### Semantics

Listing 4.15 shows that `instantiateNode` evaluates the `xtl:macro` element by just returning a modified map of macros, in which the value of the name attribute is associated with the list of children of the `xtl:macro` element. No elements or attributes are generated by the instantiation of `xtl:macro`.

```
instantiateNode ids context macros (Element (QN _ "macro"
  "http://research.sap.com/xtl/1.0") attributes children) =
  let
    name = attributes ! (QN "" "name" "")
  in
    (insert name children macros, empty, [])
```

Listing 4.15: Semantics of `xtl:macro`

#### Example

As the definition of macros does not change the instantiated template, the use of `xtl:macro` is shown below in Listing 4.17 together with the use of `xtl:call-macro` to invoke the defined macro.

### 4.4.2. xtl:call-macro

In order to invoke a macro defined with `xtl:macro`, XTL offers the `xtl:call-macro` element.

#### Syntax

`xtl:call-macro` supports only one attribute, the `name` attribute declaring the name of the macro. This value must be a string. No children are allowed in `xtl:macro`. The use of `xtl:call-macro` is unrestricted.

#### Semantics

Listing 4.16 shows that the instantiation of `xtl:call-macro` instantiates the children of the `xtl:macro` with the same value of the `name` attribute at the location of the `xtl:call-macro` element. Please note that this instantiation may occur in a different context (i.e., within an `xtl:for-each` element) than the one that was active at the `xtl:macro`.

```
instantiateNode ids context macros (Element (QN _ "call-macro"
  "http://research.sap.com/xtl/1.0") attributes _) =
  let
    name = attributes ! (QN "" "name" "")
    nodes = macros ! name
  in
    instantiateNodes ids context macros nodes
```

Listing 4.16: Semantics of `xtl:call-macro`

#### Example

In Listing 4.17, `xtl:macro` is used to define a macro with the name `simple`. Afterwards, `xtl:call-macro` is used to invoke the defined macro. The listing demonstrates that the children of `xtl:macro` are instantiated at the location where the macro is actually invoked using `xtl:call-macro`, as the attribute created using `xtl:attribute` occurs at the parent element of `xtl:call-macro`.

---

#### Template

---

```
<?xml version="1.0"?>
<sample xmlns:xtl="http://research.sap.com/xtl/1.0">
  <xtl:macro name="simple">
    <xtl:attribute
      name="date"
      select="/purchaseOrder/@orderDate"/>
  <date>
    <xtl:text select="/purchaseOrder/@orderDate"/>
  </date>
```

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

```
</xtl:macro>
<order>
  <xtl:call-macro name="simple"/>
</order>
</sample>
```

##### Instantiation Result

```
<?xml version="1.0" encoding="UTF-8"?>
<sample>
  <order date="1999-10-20">
    <date>1999-10-20</date>
  </order>
</sample>
```

Listing 4.17: Example Use of `xtl:macro` and `xtl:call-macro`

### 4.5. Advanced Features

During the practical use of XTL in the projects SNOW and EMODE, a lot of features have been added. Some of the features turned out to be very valuable, while others have been removed due to unexpected problems or have been replaced by more powerful ones.

The most important advanced features are XTL's capability to handle multiple instantiation data sources using *realms* described in Section 4.5.1 and the support for instantiation pipelines using *bypassing* described in Section 4.5.2.

#### 4.5.1. Accessing multiple Instantiation Data Sources using Realms

It has turned out to be very beneficial to be able to access multiple instantiation data sources from within one template. XTL supports any number of instantiation data sources. For example, it is possible to access several XML documents using XPath in one template. Furthermore, each source could be accessed using a different query language, allowing to access an XML document using XPath and an ontology using SPARQL. A combination of a data source with an instantiation data evaluator capable of evaluating the queries from the associated query language is called a realm.

##### Syntax

XTL supports realms by two syntactical means: a `realm` attribute and an element named `xtl:init`.

The `realm` attribute can be used with all XTL elements that support the `select` attribute. The value of the `realm` attribute is interpreted by an implementation of an XTL template engine in order to know which instantiation data evaluator is capable of interpreting the value of the `select` attribute.

The `xtl:init` instruction can be used to initialize a realm, more exactly, its assigned instantiation data evaluator. `xtl:init` can only be used as a direct child of the template's root element and no elements except `xtl:macro` or `xtl:init` may precede it.

### Semantics

The handling of realms has not been made part of the denotational semantics of XTL in order to keep the semantics short and easy to understand. Furthermore, adding realms would not add much value to the semantics, as it would only influence the way an IDS is chosen to call its functions like `evaluateText` etc.

The use of multiple realms leads to multiple contexts. Each of the realms has its own context, i.e., the evaluation of an `xtl:attribute` element with a `realm` attribute with a value of `a` uses as its context *either* the context set by the innermost `xtl:for-each` with a `realm` attribute with the value `a` *or* the instantiation source in its entirety, if there is no suitable `xtl:for-each`.

An implementation should introduce the notion of a default realm, which is used when a template contains XTL elements with `select` attributes, but without explicit `realm` attributes.

The children of the `xtl:init` instruction are passed to the instantiation data evaluator responsible for the realm denoted by the value of the `realm` attribute of `xtl:init`. An XTL template engine implementation must not interpret this content in any way.

### Example

Listing 4.18 illustrates the use of two realms within a single template. The first realm is named `po` and refers to the `po.xml` file known from the previous examples. The second realm is named `id` and is assigned to an instantiation data evaluator named `identity` that returns the `select` attribute's value in its `evaluateText` function and a collection of length  $n$  from its `evaluateForEach` function, if the corresponding `select` attribute has a value of  $n$ .

It should be noted that the inner `xtl:for-each` instruction obviously does not change the context that is used by the `xtl:text` instruction with the `realm` attribute of the value `po`.

---

#### Template

---

```
<?xml version="1.0"?>
<sample xmlns:xtl="http://research.sap.com/xtl/1.0">
  <xtl:for-each select="//items/item" realm="po">
    <xtl:for-each select="2" realm="id">
      <item>
        <xtl:text select="productName" realm="po"/>
      </item>
      <item>
        <xtl:text select="productName" realm="id"/>
      </item>
    </xtl:for-each>
  </xtl:for-each>
</sample>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<sample>
  <item>Lawnmower</item>
  <item>productName</item>
  <item>Lawnmower</item>
  <item>productName</item>
  <item>Baby Monitor</item>
  <item>productName</item>
  <item>Baby Monitor</item>
  <item>productName</item>
</sample>
```

Listing 4.18: Example Use of Realms

#### 4.5.2. Instantiation Pipelines using Bypassing

Applications performing complex XML transformations are often arranging multiple XML transformers (like XSL-T processors) in transformation *pipelines*, an architectural pattern also known as *staged architecture* [16]. There are two ways to arrange transformers into such a pipeline, which are shown in Figure 4.1.

The first type, a *horizontal* pipeline, transforms a document  $XML_A$  into a second document  $XML_B$  and afterwards into a third document  $XML_C$ . Each of the three documents can comply to a different XML dialect, as long as none of the documents represents an XSL-T stylesheet. The second type, the *vertical* pipeline, differs exactly in the XML dialect produced by the transformation of the first document  $XML_A$ : the result of this first transformation is itself a stylesheet that is then used to transform  $XML_B$  into  $XML_C$ .

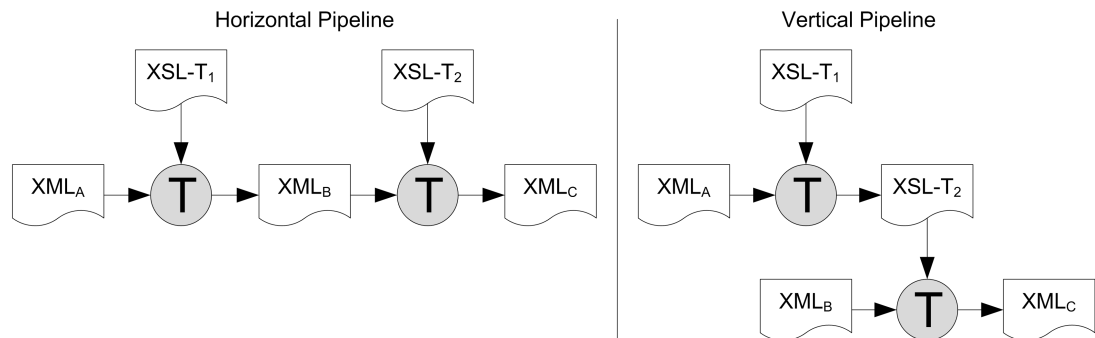


Figure 4.1.: Types of XML Transformation Pipelines

The first type corresponds to the pipelines typically used in Cocoon [7] to implement complex XML transformations, whereas the second type corresponds to the ideas proposed in [65] and [64] to implement XSL-T language extensions transparently. A vertical XSL-T pipeline can also be used to partially define the semantics of XTL. This is demonstrated in Section 4.6.



Bypassing is a feature that helps writing templates that are intended to be instantiated using a multi-stage (vertical) pipeline of XTL instantiation engines.

### Syntax

Syntactically, XTL defines a special namespace URI that is parameterizable with the number of instantiations that should be passed until the element associated with the namespace is actually processed. This URI has the form `http://research.sap.com/xtl/1.0/bypass/n` where *n* is the number of XTL template engines that should be passed before the element from this namespace should actually be processed. The number *n* is called generation number. If *n* is omitted, a default of 1 is assumed.

### Semantics

Listing 4.19 shows the denotational semantics of the bypassing feature. This is an extension to the default processing of elements shown in Listing 4.4: the difference is that the function is guarded by the expression `isBypassURI namespaceURI`. This guard asserts that this rule is only applied if the element is from a namespace complying to the namespace URI shown above.

If the element is from a bypassing namespace, it is copied into the instantiated template with a namespace with a generation number decreased by one. If the generation number reaches 0, the element is assigned to the standard XTL namespace. The attributes of the element are directly transferred to the instantiated document, whereas the content of the element is instantiated and the result is added as child to the element in the instantiation result.

```

instantiateNode ids context macros (Element (QN prefix namespaceURI
  localName) attributes children) | isBypassURI namespaceURI =
  let
    (macros1, attributes1, children1) = instantiateNodes ids
      context macros children
    newNamespaceURI = transformBypassURI namespaceURI
  in
    (macros1, empty, [Element (QN prefix newNamespaceURI
      localName) (union attributes attributes1) children1])

transformBypassURI :: String -> String
transformBypassURI uri =
  if uri == "http://research.sap.com/xtl/1.0/bypass/" ||
    uri == "http://research.sap.com/xtl/1.0/bypass/1"
  then "http://research.sap.com/xtl/1.0/"
  else case matchRegex (mkRegex
    ("http://research.sap.com/xtl/1.0/bypass/([0-9]+)")) uri of
    Nothing ->
      uri
    Just nodes ->
      "http://research.sap.com/xtl/1.0/bypass/" ++ show (read
        (nodes !! 0) - 1)

```

```
transformNamespaceAttributes :: AttrMap -> AttrMap
transformNamespaceAttributes =
  mapWithKey (\key -> \value ->
    case key of
      QN _ _ "http://www.w3.org/2000/xmlns/" ->
        transformBypassURI value
      _ -> value)

isBypassURI :: String -> Bool
isBypassURI = isPrefixOf "http://research.sap.com/xtl/1.0/bypass/"
```

Listing 4.19: Bypassing Semantics

### Example

Listing 4.20 shows the use of bypassing. In the first instantiation step, some element names are collected from the instantiation data document. Furthermore, these names are used to dynamically construct `select` attributes which are evaluated in the second instantiation, where the number of elements with that particular name in the instantiation data document is counted.

The example shows that XTL elements within XTL elements marked for bypassing are evaluated, thereby allowing the dynamic construction of queries. This feature also works over different query languages and greatly enhances the expressive power of the templates. However, care should be taken as it is easy to construct unreadable templates this way. Section 7.3.1 demonstrates a use case where bypassing is valuable.

#### Template

```
<?xml version="1.0" encoding="UTF-8"?>
<sample
  xmlns:xtl="http://research.sap.com/xtl/1.0"
  xmlns:xtl-bp="http://research.sap.com/xtl/1.0/bypass/1">
  <xtl:for-each select="//*[starts-with(local-name(), 'item')]">
    <count>
      <xtl:attribute name="name" select="local-name(.)"/>
      <xtl-bp:attribute name="count">
        <xtl:attribute name="select"
          select="concat('count('//',local-name(.),'')'"/>
        </xtl-bp:attribute>
      </count>
    </xtl:for-each>
  </sample>
```

#### Instantiation Result after First Instantiation

```
<?xml version="1.0" encoding="UTF-8"?>
<sample xmlns:xtl="http://research.sap.com/xtl/1.0">
  <count name="items">
    <xtl:attribute name="count" select="count('//items)" />
  </count>
</sample>
```

```

</count>
<count name="item">
  <xsl:attribute name="count" select="count(//item)" />
</count>
<count name="item">
  <xsl:attribute name="count" select="count(//item)" />
</count>
</sample>

```

---

Instantiation Result after Second Instantiation

---

```

<?xml version="1.0" encoding="UTF-8"?>
<sample xmlns:xsl="http://research.sap.com/xsl/1.0">
  <count count="1" name="items"/>
  <count count="2" name="item"/>
  <count count="2" name="item"/>
</sample>

```

Listing 4.20: Bypassing Example

## 4.6. Definition of the Instantiation Semantics using XSL-T

The instantiation semantics of XTL can also be defined using XSL-T. Because of the fact that XSL-T is limited to using XPath as its query language, this translational definition of XTL's semantics is restricted to the particular query language XPath.

The easiest way of defining a translational semantics would be to implement a single XSL-T stylesheet that takes an XTL template and an additional XML document as instantiation data source and outputs the instantiated template. Unfortunately, this is not possible as XSL-T (as it is currently defined in [107]) is not capable of dynamically evaluating XPath expressions embedded in its *source* documents. It is possible to circumvent this restriction in two ways. First, an XSL-T extension function (like `saxon:evaluate()` implemented in [106]). Second, a vertical two-stage transformation process could be used to avoid the necessity of dynamic XPath evaluation.

Such a two-stage transformation process is shown in Figure 4.2. The left hand side shows a template engine that transforms an XTL template using an XML instantiation data source  $XML_A$  into an instantiated template  $XML_B$ . The right side of the figure shows the implementation of the same process using an XSL-T processor: first, the XSL-T processor compiles the XTL template into an XSL-T stylesheet, which can afterwards be used to transform  $XML_A$  into the instantiated template  $XML_B$ .

The stylesheet  $XSL-T_1$  represents the generic translation process between XTL and XSL-T: it writes the XPath expressions contained in the XTL template as values of `select` attributes into the stylesheet  $XSL-T_2$ . As these XPath expressions are now no longer part of the source document, but rather of the stylesheet, they can be evaluated by a standard XSL-T processor.

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

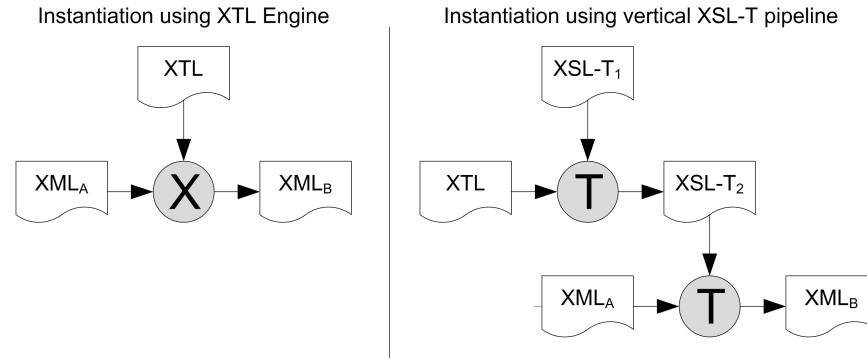


Figure 4.2.: Using a Vertical XSL-T Pipeline to Emulate the XTL Engine

### 4.7. Relation to Document Validation

A generic XML slot markup language like XTL can also be used to check the validity of XML documents, i.e., as a schema language [81]. The prototypical nature of templates in general and of XTL templates in particular makes this schema language easy to learn and use. Figure 4.3 illustrates the differences between template instantiation and validation against a schema.

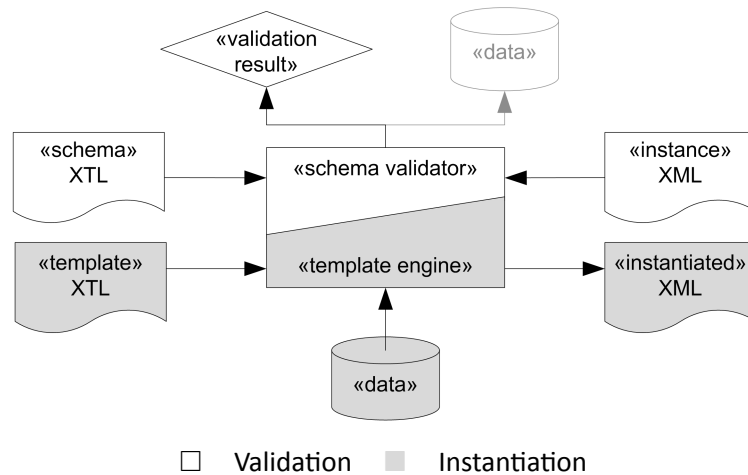


Figure 4.3.: Schema Validation and Template Instantiation

Instantiation transforms an XTL document and an instantiation data source into an instantiated template, whereas validation takes an XTL document and an XML document and answers the question: *Could this XML document be produced by the XTL document?* In some specific cases, even the reconstruction of the instantiation data may be possible.

The validation semantics of XTL can be given denotationally [81]. Another option is to give a translational semantics by transforming XTL to RelaxNG. This transformation can easily be implemented using XSL-T. XTL as a schema language allows to define the languages that can be defined by regular tree languages (see [81]). It is even possible to express certain attribute/ele-

ment interdependencies in XTL, a feature missing in many other schema languages (see Section 2.1.4).

As Figure 4.3 suggests, in most cases the result of the validation process will only yield a yes/no answer. This leads to the question whether the query language terms embedded in the `select` attributes in the XTL document used as schema are significant or opaque (see Section 2.5.5) to the validator. Whereas [81] treats the `select` attributes as opaque (which makes them meaningless, as there is no instantiation data source supplied by the application using the validator), they can be used for other purposes. First, the `select` attributes could be used to establish a link to simple types, allowing attribute values or text element content to be restricted. Second, the `select` attributes could also be used to establish a mechanism to validate static semantics within the instance. For example, a simple identifier could be used that is bound when it is first referenced in a `select` attribute. In subsequent references, the current corresponding value is checked for equality against the bound value.

Giving the `select` attributes a semantics may also lead to the opportunity to partially reconstruct the instantiation data source, as it has been suggested in Figure 4.3. An important condition that must be fulfilled by the query language is reversibility of the queries: an XPath expression like `//author/@name` is clearly not reversible, as it is unclear how many nodes have been consumed by the `//` operator before the `author` element has been found. For an example of a reversible XPath subset, see Section 6.3.2.

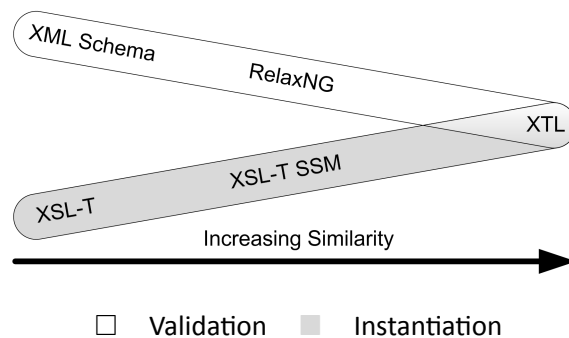


Figure 4.4.: Similarity between Schema/Template and Instance

Comparing XTL to other schema languages like RelaxNG and XML Schema indicates a continuum of XML Schema languages in terms of similarity between instance and schema. Similarity between the instance and a schema/template is mainly violated in two ways: first, by the reification of elements or attributes using a metaelement and second, by the introduction of macros. The similarity between a RelaxNG document and an instance is greater than that between an XML Schema document and a corresponding instance, because of XML Schema's strict distinction between element declarations and type definitions. Furthermore, the similarity between an XTL document and a corresponding instance is greater than that between a RelaxNG document and an instance, as RelaxNG enforces the reification of all elements and attribute names using `rng:element` and `rng:attribute`, whereas XTL allows to literally include elements

#### 4. Design of a Universal, Syntax- and Semantics-Preserving Slot Markup Language

and attribute names. The similarity relation between the mentioned schema languages is illustrated in Figure 4.4.

This figure also illustrates the similar relationship between transformation languages and prototypical template languages. Again, the similarity between an XSL-T SSM and an instance produced by it is greater than that between an XSL-T stylesheet and its corresponding result, because XSL-T SSM does not enforce a new top level document structure like XSL-T does.

According to the relationships described, it should also be clear that XTL's `xtl:macro` and `xtl:call-macro` instructions correspond to RelaxNG's `rng:define` and `rng:ref` instructions. Even further, these instructions also correspond to the definition of types in XML Schema. When considering instantiation, XTL's macros correspond to the *template rules* in an XSL-T stylesheet.

### 4.8. Conclusion

This chapter dealt with the design of XTL, a broadly applicable, syntax- and semantics-preserving slot markup language. Starting with general design decisions, the various features of XTL have been introduced by defining their syntax and semantics and by giving examples for their use. The semantics has been defined denotationally. As Haskell has been used to express this denotational semantics, a first implementation of XTL is possible just based on this chapter. The semantics has also been given by translating XTL into XSL-T, which is only possible in a two-stage process because of technical limitations of XSL-T. Finally, the relation of slot markup languages to document validation has been discussed.

The precise definition of the semantics is a contribution of this thesis, as other template techniques typically do not define the semantics formally. The denotational semantics has also been used to check the validity of a later Java implementation of the XTL instantiation process (see Section 6.2 and 7.2).

# 5

## Safe Authoring of Templates

Better safe than sorry.

*(English proverb)*

This chapter explains the processes Constraint Separation and Template Validation from Figure 3.5, which are the processes that support the safe authoring of templates. The Constraint Separation process, which adapts the template engine to a particular target language is shown in detail in Section 5.1, where it is also described formally as a transformation based on the XML Schema formalization introduced in Section 2.1.4. Section 5.2 introduces the Template Validation process, which checks the validity of a template with respect to the target language to which the template engine has been adapted.

In the following, the target language is assumed to be defined by an XML Schema. XML Schema is widely used for the definition of XML dialects and is well-supported by a number of tools, including validators and editors with support for both the creation of XML Schema documents and documents complying to a certain schema. Therefore, this decision directly addresses the Utilization of Existing Standards goal.

### 5.1. Constraint Separation

The Constraint Separation component is responsible for converting the grammar of the target language into grammars that can be used to validate templates, as well as into constraints on the instantiation data. The inferred grammar is used by the template validator to perform the authoring time validation of the templates, whereas the instantiation data constraints are used

## 5. Safe Authoring of Templates

to check the instantiation data in the instantiation data validator. The grammar transformer is therefore separating the authoring time from the instantiation time constraints.

The separation process is designed such that the conclusion illustrated in Figure 5.1 can be drawn: *if a template conforms to the template language and the instantiation data conforms to the instantiation data constraints (both emitted by the Constraint Separation), then the instantiated template conforms to the target language grammar (which has been used as input for the Constraint Separation) process.* The process is amazingly simple—for a discussion of its correctness see Sections 5.1.5 and 7.2.5.

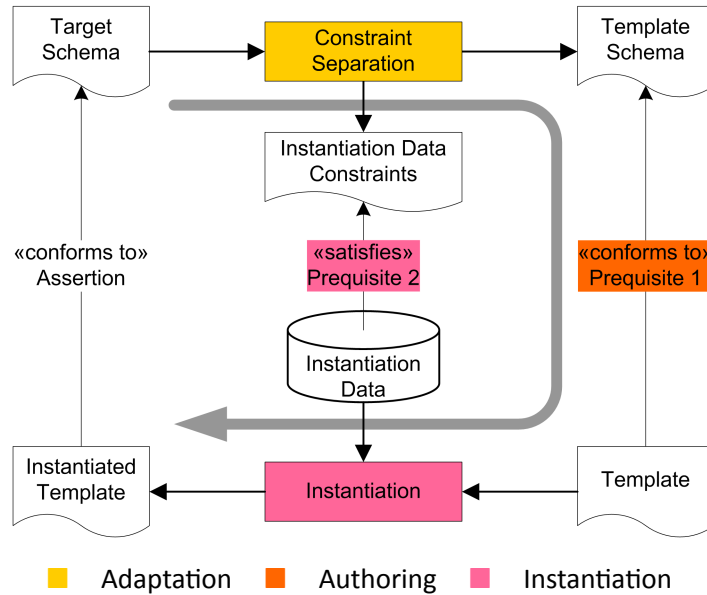


Figure 5.1.: Conclusion Enabled by the Constraint Separation Process

The Constraint Separation process described here relies on the separability of parts of a document which could be created dynamically and parts of the documents which are always part of the template. It is assumed that for all target languages to be created, markup is always part of the template, whereas character data can be part of the template or subject to dynamic creation.

For declarative text markup languages like XHTML, the assumption stated above is reasonable. The template author—in this scenario a Web designer—is responsible for describing the layout and the structure of the document, which is described in XHTML by markup and character data. The content of this document is typically delivered by the application that is using the template to render its output. Therefore, it must also be possible to create character data dynamically.

It is important to note that structural differences in documents of the target language can still be expressed by using XTL. However, the stated assumption prevents the proposed template approach to be used in scenarios, in which the elements of the markup itself are dynamic. An example for such scenarios are applications that must be capable of producing arbitrary XML



languages, which are not known before instantiation time. These applications therefore remain the domain of transformation techniques like XSL-T.

### 5.1.1. Introductory Example

In the following, the separation of constraints is shown in an example. The XML Schema used for the example is the purchase order schema `po.xsd` (see Listing A.3, [59]). An instance document for this schema is shown in Listing 5.1. In the listing, four parts of the instance document are shown that should be changed by the Constraint Separation process in order to allow them to be dynamically set or to be influenced by the instantiation data. The four cases are discussed in the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder
  orderDate="1999-10-20"> ①
  <shipTo country="US">
    <name>Alice Smith</name> ②
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment> ③
  <items> ④
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

Listing 5.1: A Purchase Order with Potentially Dynamic Parts Highlighted

## 5. Safe Authoring of Templates

First, the Constraint Separation process must ensure that the value of the attribute `orderData` ① in Listing 5.1 could be created dynamically using an `xtl:attribute` instruction.

The same must be allowed by the Constraint Separation for the content of the `name` ② element—the XTL instruction that could be used here is `xtl:text`.

Furthermore, an element declared to be optional, like `comment` ③, should be replacable by an `xtl:if` instruction containing the same element: in this example, a `comment` element. Optionality is declared by the underlying XML Schema, in this case `po.xsd`, by setting the `minOccurs` and `maxOccurs` attributes to 0 and 1, respectively.

Finally, repeatable elements, like the `item` elements within the `items` ④ elements, should be producable by an `xtl:for-each` instruction with appropriate content. In this example, appropriate means conforming to the rules for the `item` element. An element is considered repeatable when the underlying XML Schema sets `maxOccurs` to a value greater than 1.

The Constraint Separation process should produce a template language grammar that allows the documents in both Listing 5.1 and Listing 5.2 as instances.

What are the modifications the Constraint Separation process needs to execute to transform the target language grammar into the correspondings template language grammar? In the following, it is just considered how an XML Schema may look like, if it allows both documents in the Listings 5.1 and 5.2 as instances.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder xmlns:xtl="http://research.sap.com/xtl/1.0">
  <xtl:attribute name="orderDate" select="date"/>
  <shipTo country="US">
    <name><xtl:text select="shipTo/name"/></name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <xtl:if select="length(comment) > 0">
    <comment>
      <xtl:text select="comment"/>
    </comment>
  </xtl:if>
  <items>
    <xtl:for-each select="items/item">
      <item>
        <xtl:attribute name="partNum" select="partNum"/>
        <productName>Lawnmower</productName>
        <quantity>1</quantity>
      </item>
    </xtl:for-each>
  </items>
</purchaseOrder>
```

```

        <USPrice>148.95</USPrice>
        <comment>Confirm this is electric</comment>
    </item>
</xtl:for-each>
</items>
</purchaseOrder>

```

Listing 5.2: A Purchase Order XTL Template

Enabling the use of `xtl:attribute` in order to create attribute values is rather easy. The attribute `orderDate` is defined in the complex type `PurchaseOrderType`, which is shown in Listing 5.3.

```

<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

```

Listing 5.3: The PurchaseOrderType from po.xsd

Fortunately, the content model of the `PurchaseOrderType` is a sequence, so it is easy to allow the use of `xtl:attribute` at its beginning. The resulting modified `PurchaseOrderType` is shown in Listing 5.4. There is no use attribute at the definition of the `orderDate` attribute, which makes it an optional attribute. If the attribute `orderDate` would have been defined as required using `use='required'`, the Constraint Separation process would have to change this to `use='optional'` in order to allow the attribute to be omitted when an `xtl:attribute` instruction is present to create it.

```

<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element ref="xtl:attribute" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element ref="shipTo"/>
    <xsd:element ref="billTo"/>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element ref="items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

```

Listing 5.4: The Modified PurchaseOrderType, Allowing the Use of `xtl:attribute`

## 5. Safe Authoring of Templates

The modified type allows the use of `xtl:attribute` as shown in Listing 5.2. However, the modified type introduces a lot of inaccuracies. First, there is no relation between the attributes created using `xtl:attribute` instructions and the set of attributes permitted on `xtl:attribute`'s parent element. Second, there is no guarantee that an `orderDate` attribute created using `xtl:attribute` has a value that is, as required by the original attribute definition, a valid value of the type `xsd:date`. Whereas the first problem can be addressed in the template language grammar (i.e., it can be validated during the authoring phase), the latter problem can only be addressed during the instantiation phase, as the data used to create the attribute is only available at this point.

The text node in Listing 5.1 that has been replaced by an `xtl:text` in Listing 5.2 is defined in the type `USAddress`, which is shown in Listing 5.5.

```
<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
    fixed="US"/>
</xsd:complexType>
```

Listing 5.5: The `USAddress` Type from `po.xsd`

In order to allow the replacement of the text node by an `xtl:text` instruction, a complex type with mixed content could be used. An example for such a type definition is shown in Listing 5.6.

```
<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="name">
      <xsd:complexType mixed="true">
        <xsd:sequence>
          <xsd:element ref="xtl:text" minOccurs="0" maxOccurs="1">
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
```

Listing 5.6: The Modified `USAddress` Type, Allowing the Use of `xtl:text`

The relaxed `USAddress` type allows the introduction of `xtl:text` as shown in Listing 5.2, but introduces similar problems like the introduction of `xtl:attribute` above. First of all, the mixed content model does not exclusively allow the use of either literal text or the `xtl:text` instruction, but rather allows a mixture of both. Second, there is again no guarantee on the validity of the created text with respect to the originally defined type.

Allowing the use of `xtl:if` and `xtl:for-each` to surround optional or repeatable elements requires introducing a choice between the original element and the respective XTL instruction. For example, to allow the use of `xtl:if` to surround the optional comment element (see Listings 5.1 and 5.2, ③), a type definition like the one shown in Figure 5.7 can be used.

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress" />
    <xsd:element name="billTo" type="USAddress" />
    <xsd:choice minOccurs="0">
      <xsd:sequence>
        <xsd:element minOccurs="1" ref="comment" />
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element ref="xtl:if" minOccurs="1"
          maxOccurs="1" />
      </xsd:sequence>
    </xsd:choice>
    <xsd:element name="items" type="Items" />
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date" />
</xsd:complexType>
```

Listing 5.7: The Modified `PurchaseOrderType`, Allowing the Use of `xtl:if`

Again, the introduction of the XTL instruction has opened up a number of new problems. First of all, the syntactic definition of `xtl:if` defines its content model based on an `xsd:any` wildcard. Therefore, the modified type does not guarantee that the content of `xtl:if` is actually a `comment` element. For the introduction of `xtl:for-each`, there is no guarantee that the number of evaluations of the `xtl:for-each` instruction is between the values of `minOccurs` and `maxOccurs` of the original type. Finally, if a sequence of multiple optional or repeatable elements is enabled for the use of `xtl:if` and `xtl:for-each`, violations of the UPA may occur.

The problems opened up by the simple relaxation of the target language grammar just demonstrated are addressed with the means introduced in the sections below. Section 5.1.2 introduces the Constraint XML Schema Definition Language (CXSD), an extension of XML Schema, which extends XML Schema in its expressive power in a way needed by the Constraint Separation process. Furthermore, Section 5.1.3 introduces the Instantiation Data Constraint language (IDC), a very simple XML dialect that is used by the Constraint Separation process to express constraints determined on the instantiation data. Based on these prerequisites, the Constraint Separation

## 5. Safe Authoring of Templates

process is introduced in Section 5.1.4. Section 5.1.5 demonstrates that the proposed process preserves that constraints defined by the target language. Finally, Section 5.1.6 describes the implementation of the Constraint Separation and Section 5.1.7 introduces an extension of the Constraint Separation process called Partial Templatization.

### 5.1.2. The Constraint XML Schema Language CXSD

As shown in the example above, validating a template with respect to its instantiation results requires a powerful schema language. XML Schema itself is not powerful enough to express the constraints that must hold in order to assert that the instantiated template conforms to the target language XML Schema. In other words, XML Schema is *not closed under composition* with the XML Schema of XTL.

There are three reasons for this: first, the composed schema must be able to express complex constraints between attributes and elements like the one above: *either* some element has an attribute named `attr` or it has a child element named `xtl:attribute` and a name attribute with the value `attr`. XML Schema is not capable of expressing such complex constraints. RelaxNG allows to express choices between attributes and child elements, but is not able to express the further condition on the `xtl:attribute` element. Schematron [99] would be capable of expressing the constraint above.

Second, the composed schema must also be able to express alternatives for the element content. One of such constraints occurs when the constraint separation process enables the use of `xtl:text` in an element that is defined to be of a simple type in the target language (see Section 5.1.4). The constraint basically states that an element's content is *either* complying to some simple type *or* is an `xtl:text` element. Such alternatives are also not expressible using XML Schema (even not using mixed content elements). RelaxNG is capable of expressing such alternatives, whereas Schematron cannot express this, as it has no features to check text nodes against simple types.

Finally, the introduction of `xtl:if` and `xtl:for-each` statements leads to UPA violations. Therefore, it is necessary to relax the UPA constraint. This makes the resulting schema harder to evaluate, but validation is still possible—as is shown in Section 5.2—because the unique particle attribution is still intact, but its application is delayed until the evaluation of further constraints.

In order to fulfill the goal of Utilization of Existing Standards, it has been decided to add a constraint language to XML Schema. The basic idea is to use OCL [136] constraints embedded as annotations in an XML Schema in order to strengthen its expressiveness. The resulting schema language is called CXSD. Technically, the embedded constraints are always invariants in the sense defined by the specification.

There are multiple reasons why the use of OCL is beneficial. First, there exist powerful implementations like the Dresden OCL Toolkit [183] and the implementation of the Model Development Tools (MDT) [53] subproject of Eclipse. Second, the OCL language allows the easy adaptation of meta-models, which allows arbitrary capabilities to be built in to the language, e.g., the capability to check simple types.

Even further, CXSD can improve the currently unsatisfying transformation of the Unified Modeling Language (UML) models into XML Schema [34; 21]. If the UML model is enhanced using

OCL constraints, which is an important technique to build concise models that could be used in MDA processes, the constraints are typically ignored when transforming into XML Schema. With CXSD, the constraints could simply be transformed into corresponding constraints in the schema, thereby greatly enhancing the conformity of the transformation result with the UML model.

By design, CXSD schema is unable to relax the constraints imposed on a complying document by its underlying XML Schema, i.e., each of the OCL constraints is restricting the number of complying instances (or leaves it unchanged). More specifically, if a document is not complying to the underlying XML Schema, it will also not comply to the CXSD document, while the reverse is not true. To distinguish between the language accepted by a CXSD schema from that of the underlying XML Schema, the first schema is denoted by  $S^+$  and its language therefore by  $\mathcal{L}(S^+)$ , whereas the latter is designated by  $S$  with its accepted language being  $\mathcal{L}(S)$ . With these notations, the fact that CXSD only restricts the constraints contained in its underlying schema can be formally notated as  $\mathcal{L}(S_{\mathcal{T}^+}^+) \setminus \mathcal{L}(S_{\mathcal{T}^+}) = \emptyset$ .

To understand the meaning of the OCL constraints which are introduced by the constraint separation process, it is necessary to consider the underlying metamodel. An UML class diagram of the metamodel is shown in Figure 5.2.

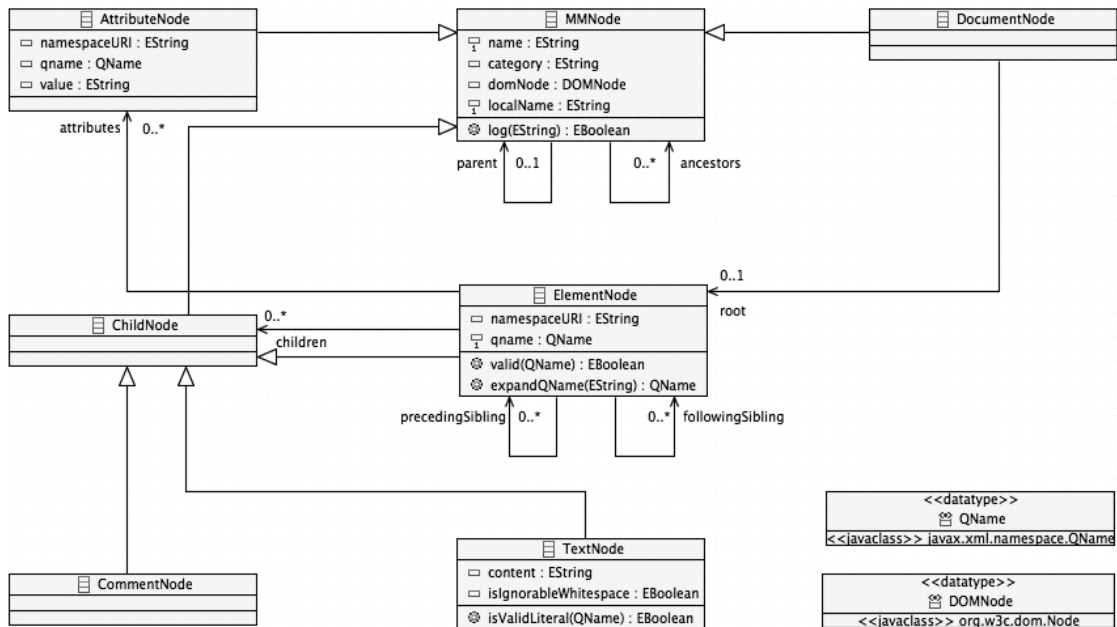


Figure 5.2.: Meta-model for the CXSD constraints

The metamodel is mostly self-explaining, as it closely resembles the Document Object Model (DOM). The node class from DOM is represented by the MMNode class. In additions to DOM's capabilities, additional relations corresponding to XPath axes have been added: for example the capability to access the ancestors of each node that has been added to the MMNode. Furthermore, a method `isValidLiteral` has been added to the TextNode that allows to check whether the text of the node is compliant to a simple type denoted by a passed QName.

## 5. Safe Authoring of Templates

As already has been stated above, all constraints in the CXSD are invariants. This makes it unnecessary to attach the keyword `inv` to the constraints, especially since the naming of constraints is a part of CXSD (as already described). Furthermore, the OCL specification [136] defines that invariants are relative to a contextual type. In the standard use of OCL as an extension to a UML model, this context is given using the `context` keyword along with a textual specification of the context type. In CXSD, the context is inferred from the position of the constraint within the XML Schema. CXSD allows constraints to be attached to attributes and elements as well as to types. In the latter case, the constraint must hold for all elements complying to that type. Additionally, constraints can be added to restrictions (which fits the fact that constraints are restrictive by nature), but not to extensions (as no syntax for the revocation of constraints has been defined to be part of CXSD).

The embedding of the OCL constraints into XML Schema is based on XML Schema's `appinfo` feature [180, Section 3.13]. Typically, the embedding is done in a `CDATA` section to avoid having to escape characters like `<`, which are quite important in OCL. The CXSD language allows the assignment of a simple name and a message to each constraint. The simple name is intended to be used to denote the constraint in error messages. The message should give more details and is proposed to be shown as a detailed error description when the constraint fails.

A complete `element` declaration with an embedded OCL constraint is shown in Listing 5.8. It shows the declaration of a `test` element. The element declaration contains an `xsd:annotation` element, which in turn contains an `xsd:appinfo` element having its source attribute set to `http://research.sap.com/cxsd/1.0`. This element contains the root element of the CXSD invariant, `cxsd:inv`. This element has two subelements, `cxsd:ocl`, containing the OCL representation of the constraint as a `CDATA` section, and `cxsd:message`, a human-readable message that is reported if the constraint fails during the CXSD validation process. The evaluation of the constraint takes place whenever a node in an XML document is successfully validated against the element declaration into which it has been embedded. Then, the context of the OCL evaluation is this node.

```
<xsd:element name="test">
  <xsd:annotation>
    <xsd:appinfo source="http://research.sap.com/cxsd/1.0">
      <cxsd:inv name="restricted-attribute-count">
        <cxsd:ocl>
          <![CDATA[
            self.attributes->size() <= 2
          ]]>
        </cxsd:ocl>
        <cxsd:message>
          At maximum, 2 attributes can be present.
        </cxsd:message>
      </cxsd:inv>
    </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence />
    <xsd:attribute name="a" type="xsd:string" />
  </xsd:complexType>
</xsd:element>
```



```

        <xsd:attribute name="b" type="xsd:string" />
        <xsd:attribute name="c" type="xsd:string" />
    </xsd:complexType>
</xsd:element>

```

Listing 5.8: A complete CXSD Element Declaration with an Embedded OCL Constraint

The constraint restrains the number of attributes allowed at the `test` element to two out of three attributes `a`, `b` and `c` that are allowed by the schema itself. Therefore, the XML snippet `<test a='1' b='2' />` would be valid, whereas `<test a='1' b='2' c='3' />` would not be valid, because the OCL constraint evaluates to `false`.

In the following, two additional examples show how the CXSD can be leveraged to express constraints that would not be expressible using XML Schema alone. First, an example demonstrates how CXSD can be used to express an exclusive-or between attributes, which is part of the XML Schema specification. Afterwards, an exclusive-or relation between an element and an attribute is shown, which is part of the XSL-T specification. Both examples show the expressive power and value of CXSD.

### Refining the XML Schema Self Description `XMLSchema.xsd`

XML Schema does not allow to express exclusive-or relations between attributes. An example for such an exclusive-or relation can be found in the XML Schema specification itself: at an element declaration, *either* a default or a fixed attribute can be present [180, Section 3.3.3]. Note that this lack of expressiveness makes it impossible to define an `XMLSchema.xsd` completely equivalent to the specification, as not all constraints given by the XML Schema specification can be expressed in it. While this is inexpressible in XML Schema, it can well be expressed using CXSD, as is shown in Listing 5.9.

```

let
    fixedPresent:Boolean =
        self.attributes->select(name='fixed')->size() > 0,
    defaultPresent:Boolean =
        self.attributes->select(name='default')->size() > 0
in
    not(fixedPresent and defaultPresent)

```

Listing 5.9: Expressing a Constraint from the XML Schema Specification with CXSD

It is easy to see that the constraint first checks whether the `fixed` and `default` attributes are present by selecting them from the `attributes` axis defined in the meta-model and stores the result in the Boolean variables `fixedPresent` and `defaultPresent`. Afterwards, the constraint states that `not(fixedPresent and defaultPresent)` must be true, which prevents both attributes from being present at the same element declaration.

Another example that can be formulated in CXSD is the complex relation between the attributes `name`, `ref` and the text content of an `xsd:element` element stated in [180, Section

## 5. Safe Authoring of Templates

3.3.3] (which basically states that a non-global element has either a `name` or a `ref` attribute, and is empty besides the `xsd:annotation` element in the latter case).

Even the most complex restriction dictated by the XML Schema specification, the UPA, could be expressed using CXSD. This could be achieved by applying the concept of Brzowski derivatives [33] to XML Schema processing. This suggestion stems from [168], and the concept of *initial determinism* introduced there could easily be implemented in OCL (as the calculation mostly includes set operations, which are well-supported in OCL), allowing the monitoring of the UPA constraint in CXSD.

### Implementing Constraints of the XSL-T 2.0 Specification

XSL-T 2.0 (as well as its predecessor version) defines a lot of elements in which the existence of mixed content implies the absence of the `select` attribute. An example for this is the `xsl:with-param` element which can be used to pass parameters to a template, for example when calling a named template with `xsl:call-template`. The value of the parameter is either retrieved by evaluating the `select` attribute (if present) or by evaluating the so-called *sequence constructor* parented by the `xsl:with-param` element [107, Section 10.1.1]. Other elements, for which a similar constraint is contained in the specification, include `xsl:attribute`, `xsl:comment`, etc.

A constraint enforcing this syntax is shown in Listing 5.10. The constraint basically counts the number of text or element nodes within the `xsl:with-param` element and determines whether it carries a `select` attribute, and checks that the existence of text or element nodes implies the absence of `select`.

```
let
  selectPresent:Boolean =
    self.attributes->select(name='select')->size() > 0,
  childrenPresent:Boolean = self.children->select(
    oclIsTypeOf(ElementNode) or oclIsTypeOf(TextNode)
  )->size() > 0
in
  childrenPresent implies not(selectPresent)
```

Listing 5.10: Expressing a Constraint from the XSL-T 2.0 Specification with CXSD

### 5.1.3. The Instantiation Data Constraint Language IDC

In the previous section, the validity of the instantiation data has been assumed. In order to satisfy this assumption, a second language called IDC has been developed. This simple language allows to specify properties of the instantiation data which could later be validated in the instantiation data validator (see Section 6.3) or asserted using the alternative approach of Template Interface Generation (see Section 6.3.2).

As opposed to the CXSD, the IDC language has been specifically designed to fit exactly the needs of the constraint separation process. It is therefore a very domain specific language,

which is not intended to be reused in other scenarios. This also explains the limited capabilities of the IDC.

Similar to the CXSD, IDC statements are embedded into an XML Schema using its `appinfo` element [180, Section 3.13]. An IDC statement is always embedded into the reference to an element from the `XTL.xsd`. A particular IDC statement starts with a single `constraints` element that contains a sequence of `constraint` elements. Each of these constraints is itself empty, but has the following attributes:

1. The `type` attribute gives the qualified name of a (simple or complex) type to which the instantiation data item must comply. If a multiplicity (see below) is specified, each of the instantiation data items must comply to that type.
2. The `min` and `max` attributes are used to specify the lower and the upper limit of the multiplicity of this instantiation data item. The attributes default to a value of 1, which means that exactly one single instantiation is expected. The types of the attributes `min` and `max` are the same as the types of the XML Schema attributes `minOccurs` and `maxOccurs`.
3. The `for-name` attribute allows to specify an additional qualified name. This attribute is only valid if the IDC statement is embedded into an `xtl:attribute` reference. It restricts the particular constraint to be valid only for the creation of the attribute with the specified name. This way, multiple IDC constraints can be formulated for a single position at which `xtl:attribute` can be used to create several, differently typed, attributes.

Figure 5.11 shows an example of an IDC fragment. It shows an IDC statement that is embedded into a reference to `xtl:attribute`. It contains exactly one constraint that restricts the instantiation data value to be used for the creation of the `order` (value of `for-name`) attribute to be of type `date` (value of `type`).

```
<xsd:element ref="xtl:attribute" minOccurs="0" maxOccurs="unbounded">
  <xsd:annotation>
    <xsd:appinfo source="http://research.sap.com/xtl/idc/1.0">
      <idc:constraints>
        <idc:constraint type="xsd:date"
          for-name="orderDate"/>
      </idc:constraints>
    </xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

Listing 5.11: An Instantiation Data Constraint in an XML Schema fragment

#### 5.1.4. Constraint Separation Process

In order to understand the construction of the template language schema, it is necessary to consider the relation between the target language schema and the template language schema (viewed both as CXSD document and as its underlying XML Schema), which is shown in Figure 5.3. The figure assumes that the instantiation data is valid with respect to the instantiation

## 5. Safe Authoring of Templates

data constraints. Please note that the figure already reflects that the target language schema is exactly defined by the CXSD document, i.e.,  $\mathcal{T}^\circ = \mathcal{L}(S_{\mathcal{T}^\circ}^+)$ . There are four cases:

- (a) This case is reflecting the fact that templates are prototypical by definition (see Definitions 2.5 and 2.11): each target language document must also be part of the template language, i.e.,  $\forall t \in \mathcal{T} : t \in \mathcal{T}^\circ$ , or, in terms of languages,  $\mathcal{T} \subseteq \mathcal{T}^\circ$ .
- (b) All documents which are complying to the template language must instantiate into documents from the target language. This is reflecting the requirement of preservation, meaning that all constraints from the target language schema have been successfully transferred into the template language schema (considered as an CXSD document). Formally  $\forall t^\circ \in \mathcal{T}^\circ : \text{instantiate}(d, t^\circ) \in \mathcal{T}$  for valid instantiation data  $d$ .
- (c) Documents that only comply to the template language schema considered as an XML Schema but do not comply to it considered as CXSD document do not instantiate into the target language. As these are documents that may be constructed using an editor that only supports XML Schema but not CXSD, the number of documents falling into this case should be minimized. This helps creating valid templates with standard tools, a goal called approximation. Formally,  $\mathcal{L}(S_{\mathcal{T}^\circ}) \setminus \mathcal{L}(S_{\mathcal{T}^\circ}^+)$  should be minimized.
- (d) Documents that do not even comply to the template language schema considered as XML Schema do not instantiate into the target language:  $\forall t^\circ \notin \mathcal{T}^\circ : \text{instantiate}(d, t^\circ) \notin \mathcal{T}$  independently of the instantiation data  $d$ .

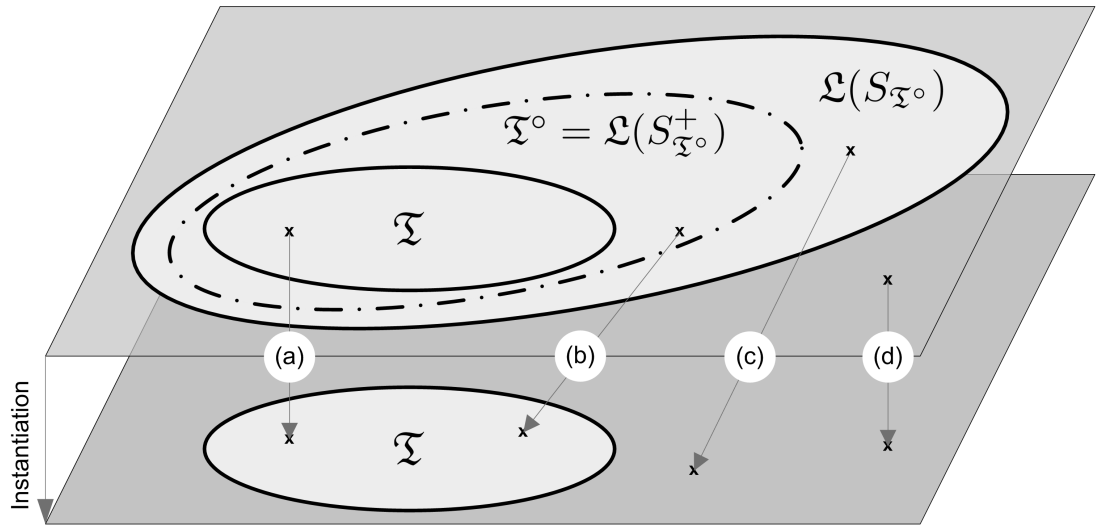


Figure 5.3.: Set Relations between Template and Target Language

To restrict the complexity of the Constraint Separation step while still being able to present a working solution, it is necessary to restrict both the XML Schema features used in the definition of the target language as well as the complexity of the XTL itself.

The four most important instructions of XTL are considered by the following description of the constraint separation process: `xtl:attribute`, `xtl:text`, `xtl:if` and `xtl:for-each`. Support for `xtl:include` is not considered, but `xtl:include` could easily be enabled at all places in which `xsd:any` wildcards are used in the XML Schema. Additionally, `xtl:macro` and `xtl:call-macro` are not supported by the constraint separation process described here.

On the other side, the set of XML Schema features used to describe the target language had to be restricted as well. Most notably, `xsd:all` and mixed content models and substitution groups are not considered. Identity-constraint definitions are also not considered, as their treatment would require advanced XPath rewriting techniques.

To describe the constraint separation process, an XML Schema instance as defined by Definition 2.16 is assumed as a target language grammar. The template language grammar is no longer an XML Schema document, as it contains OCL constraints (expressing *authoring* constraints) as well as IDC constructs (expressing *instantiation data* constraints). Definition 5.1 below contains extension points at which both types of constraints can be stored. The constraints themselves will not be formalized beyond what has been said in Section 5.1.2 and 5.1.3, as this is not necessary to document the basic idea of the Constraint Separation process.

**Definition 5.1** (Set of extended content models). The set of extended content models  $ECM(\Sigma, N, E)$  over the set of simple types  $\Sigma$ , the set of non-terminal symbols  $N$  and the set of qualified names for elements  $E$  is defined recursively as follows:

- The empty sequence  $\epsilon$  is an extended content model:  $\epsilon \in ECM$ .
- All simple types are extended content models:  $\forall \sigma \in \Sigma : \sigma \in ECM$ .
- All non-terminal symbols are extended content models:  $\forall n \in N : n \in ECM$ .
- All element names are extended content models:  $\forall e \in E : e \in ECM$ .
- All element names equipped with an authoring constraint  $c_a$  and an instantiation data constraint  $c_i$  are extended content models:  $\forall e \in E : e\langle c_a, c_i \rangle \in ECM$ .
- For two extended content models  $cm_1 \in ECM$  and  $cm_2 \in ECM$ , the results of the following operations are also extended content models, i.e.,
  - $cm_1, cm_2 \in ECM$ , meaning a sequence consisting of the two extended content models,
  - $cm_1 \mid cm_2 \in ECM$ , meaning a choice between the two extended content models, and,
  - $cm_1\{i, j\} \in ECM$ , meaning a repetition of the extended content model  $cm_1$ , where  $i \in \mathbb{N}$ ,  $j \in \mathbb{N}^+ \cup \{*\}$ , and  $j \neq * \Rightarrow i \leq j$ , with the special symbol  $*$  meaning unrestricted repetition.  $\square$

A CXSD schema with embedded IDC constructs is an XML Schema as defined in Definition 2.16, but with the notion of the content model as defined in Definition 2.18, replaced by that of an extended content model as defined in Definition 5.1.

The Constraint Separation process transforms the target language grammar  $T = (\Sigma, N, E, A, N_\bullet, R)$  into the template language grammar  $T^\circ = (\Sigma, N, E^\circ, A, N_\bullet, R^\circ)$ , where

- $E^\circ = E \cup \{\text{xtl:attribute}, \text{xtl:text}, \text{xtl:if}, \text{xtl:for-each}\}$ , i.e., the set of elements from the target language grammar extended by the elements `xtl:attribute`, `xtl:text`, `xtl:if` and `xtl:for-each`,

## 5. Safe Authoring of Templates

- $R^\circ = \bigcup_{r \in R} \{X \rightarrow cs(e, ad^*, cm) \mid r = X \rightarrow e(ad^*, cm)\}$ , with the constraint separation function  $cs$  as described below.

The constraint separation function  $cs(e, ad^*, cm)$  is defined as follows:  $cs(e, ad^*, cm) = e(ad^{*\circ}, cm^\circ)$  where

- $ad_{fixed}^* = \text{fixed}(ad^*)$  is the set of attribute declarations that assign a fixed value to the declared attribute,
- $ad_{required}^* = \text{req}(ad^*) \setminus ad_{fixed}^*$  is the set of attribute declarations that declare required attributes without an assigned fixed value,
- $ad_{relaxed}^* = \bigcup_{ad \in ad^*} \{(a, \sigma, 0, f) \mid ad = (a, \sigma, i, f)\}$  is the a set of attribute declarations that corresponds to  $ad_{required}^*$ , but has all its attribute declarations' required cardinalities relaxed to optional (or 0 according to Definition 2.17),
- $ad_{other}^* = ad^* \setminus ad_{required}^* \setminus ad_{fixed}^*$  is the set of attribute declarations that are neither required nor have a fixed value assigned,
- $ad^{*\circ} = ad_{fixed}^* \cup ad_{relaxed}^* \cup ad_{other}^*$  is the new set of attribute declarations after the constraint separations, which is the union of the sets of the fixed, the other and the relaxed attribute declaration sets,
- $cm^\circ = \begin{cases} cs'(cm) & \text{if } n = 0, \\ \text{xtl:attribute}\{m, n\} \langle c_a^{attr}(ad_{relaxed}^*), c_i^{attr} \rangle, cs'(cm) & \text{otherwise} \end{cases}$   
where  $m = |ad_{required}^*|$  and  $n = m + |ad_{other}^*|$ .

The helper function  $cs'(cm)$  is in turn recursively defined over the structure of content models (according to Definition 2.18, as its arguments are content models from the target language grammar) as follows:

- $cs'(\epsilon) = \epsilon$ ,
- $cs'(\sigma) = (\mathbb{S} \mid \text{xtl:text}) \langle c_a^{text}(\sigma), c_i^{text} \rangle$
- $cs'(N) = N$ ,
- $cs'(e) = e$ ,
- $cs'(cm_1, cm_2) = cs'(cm_1), cs'(cm_2)$ ,
- $cs'(cm_1 \mid cm_2) = cs'(cm_1) \mid cs'(cm_2)$ ,
- $cs'(cm\{i, j\}) = \begin{cases} cs''(e\{i, j\}) & \text{if } cm = e \text{ where } e \in E, \\ cs'(cm)\{i, j\} & \text{otherwise} \end{cases}$

It should be noted that the relaxation of the simple type  $\sigma$  to the general string type  $\mathbb{S}$  in the definition of  $cs'(\sigma)$  arises from the fact that for elements with mixed content, no simple type restricting the eventual text content can be specified in XML Schema [180]. The restriction has therefore been moved into the corresponding authoring constraint  $c_a^{text}(\sigma)$ .

The function  $cs'$  introduced above basically performs an identity transformation on the content model, except for the case in which a reference to an element is found within a content model. In this case, the processing is actually done by the helper function  $cs''$ , which is defined as follows:

$$cs''(e\{i, j\}) = \begin{cases} \text{xtl:if} \langle c_a^{if}(e), c_i^{if}() \rangle \mid e\{0, 1\} & \text{if } i = 0 \text{ and } j = 1, \\ e\{1, 1\} & \text{if } i = j = 1, \\ \text{xtl:for-each} \langle c_a^{for-each}(e), c_i^{for-each}(i, j) \rangle \mid e\{i, j\} & \text{otherwise} \end{cases}$$

The construction of the referenced authoring and instantiation data constraints follows below. It is important to note that the authoring constraints are given as expressions over a formal representation (see Section 2.1.3) of the XML instance to be validated, whereas the instantiation data constraints are given as sets of tuples  $(a, \sigma, i, j)$ , where  $a$  is an attribute name,  $\sigma$  is a simple type and  $i$  and  $j$  are minimum and maximum cardinalities for the instantiation data. The free variable  $v$  in the authoring constraints refers to the node against which the constraint is validated.

$$\begin{aligned}
c_a^{attr}(ad_{relaxed}^*) &= \forall (a, \sigma, i, f) \in ad_{relaxed}^* : c_1(a, v) \Leftrightarrow c_2(a, v), \text{ where} \\
c_1(a, v) &= \neg \text{hasAttr}(v, a) \\
c_2(a, v) &= \exists v' \in \text{children}(v) : \text{label}(v) = \text{xsl:attribute} \wedge \text{attr}(v', \text{name}) = a \\
c_i^{attr}(ad_{relaxed}^*) &= \{(a, \sigma, -, -) \mid (a, \sigma, i, f) \in ad_{relaxed}^*\} \\
c_a^{text}(\sigma) &= \begin{cases} \text{Valid}(\sigma, \epsilon) & \text{if } \text{children}(v) = \epsilon, \\ \text{Valid}(\sigma, \text{value}(v')) & \text{if } \text{children}(v) = v' \wedge \text{label}(v') = \top, \\ \text{true} & \text{if } \text{children}(v) = v' \wedge \text{label}(v') = \text{xsl:text}, \\ \text{false} & \text{otherwise} \end{cases} \\
c_i^{text}(\sigma) &= \{(-, \sigma, -, -)\} \\
c_a^{for-each}(e) &= \text{children}(v) = v' \wedge \text{label}(v') = e \\
c_i^{for-each}(i, j) &= \{(-, -, i, j)\} \\
c_a^{if}(e) &= \text{children}(v) = v' \wedge \text{label}(v') = e \\
c_i^{if}() &= \emptyset
\end{aligned}$$

The `xsl:text` authoring constraint  $c_a^{text}$  is validating the content of the `xsl:text` element against the simple type  $\sigma$  expected at the location in the document at which `xsl:text` has been allowed by the Constraint Separation process, whereas the `xsl:for-each` authoring constraint  $c_a^{for-each}$  is only checking for the name of the contained element. The reason for this behavior is that the `xsl:text` element is declared to allow mixed content, in which case the simple type to which the content should comply could not be specified. On the other hand, the `xsl:for-each` declaration within the `XSL.xsd` schema enforces a *strict* processing (see [180, Section 3.10.1]) of the elements complying to the wildcard within the `xsl:for-each` content model. The strict processing allows the `xsl:for-each` authoring constraint to check only for the name of the element within it. The same argumentation also holds for the `xsl:if` authoring constraint  $c_a^{if}(e)$ .

### 5.1.5. Proof of the Preservation of the Target Language Constraints

The following argumentation shows that instantiating a template which has been validated as suggested by the mechanisms in Chapter 5 yields a document from the target language. This

## 5. Safe Authoring of Templates

argumentation, together with the roundtrip test case described in Section 7.2 shows the fulfillment of the Safe Authoring goal.

For the argumentation, the following assumptions are made:

- The target language should be denoted  $\mathfrak{T}$  and is described by the schema  $T = (\Sigma, N, E, A, N_\bullet, R)$ .
- The template language is denoted by  $\mathfrak{T}^\circ$  and described by the CXSD schema  $T^\circ = (\Sigma, N, E^\circ, A, N_\bullet, R^\circ)$ . This expanded schema is derived from  $T$  by applying the process described in Section 5.1.4.
- A template  $t^\circ$  is an XML document  $(V^\circ, v_\bullet^\circ, \text{label}^\circ, \text{children}^\circ, \text{attr}^\circ, \text{value}^\circ)$  with  $t^\circ \in \mathfrak{T}^\circ$  that belongs to the template language, i.e., it belongs to the target language's schema  $t^\circ \in \mathfrak{L}(T^\circ)$ .
- The instantiation data is denoted by  $d$  and is assumed to satisfy the instantiation data constraints  $I$ .

Given these assumptions, it can be shown that the instantiated template belongs to the target language, i.e.,  $\text{instantiate}(d, t^\circ) \in \mathfrak{T}$ . This fact can be shown by giving two proofs for sub statements: First, it must be shown that each node in the instantiated template has at least the attributes it is required to have and that the attribute values are of the correct type. Second, it must be shown that each node satisfies its content model, i.e., that its children have the exact type (simple or complex). Proving both statements is equivalent to proving the main statement, as they define *local validity* against the defined XML Schema subset in the sense of [180, Section 2.1].

In the following, the instantiated template is named  $t = \text{instantiate}(d, t^\circ)$  and is also a well-formed XML document, i.e.,  $t = (V, v_\bullet, \text{label}, \text{children}, \text{attr}, \text{value})$ . Furthermore, the one-to-one relationship between rules in the target and the template language grammar established by the Constraint Separation process in Section 5.1.4, is formalized by a function  $\text{cr}$  (“corresponding rule”) that maps a rule  $r \in R$  from the target language grammar to a rule  $r^\circ \in R^\circ$  of the template language grammar, i.e.,  $\text{cr}(X \rightarrow e(ad^*, cm)) = X \rightarrow \text{cs}(e, ad^*, cm)$ .

### 5.1.5.1. Completeness of the Set of Required Attributes

The following is valid for each  $v \in V$ . Assume that  $v$  has been instantiated from the node  $v^\circ$ , which has been produced by a rule  $r^\circ = X \rightarrow e(ad^{*\circ}, cm^\circ) \in R^\circ$ . Because of the one-to-one relationship between rules in the template and the target language grammar maintained by the constraint separation process, the node  $v$  must comply to the rule  $r \in R$  with  $r^\circ = \text{cr}(r)$ .

Then it must be shown that  $\forall(a, \sigma, i, f) \in \text{req}(ad^*) : \text{hasAttr}(v, a)$ . Depending on whether the required attribute is or is not part of the template  $t^\circ$ , the following two cases must be considered:

- If the attribute is contained literally in the template, i.e, if  $\text{hasAttr}(v^\circ, a)$ , the attribute will also be part of the instantiated template  $t$ , i.e.,  $\text{hasAttr}(v, a)$ , as the instantiation never removes attributes (see Chapter 4). The attribute will always be contained literally in the template if it is required and has an assigned fixed value, since this attribute is also required by the template language grammar.



- If the attribute is missing in the template, i.e., if  $\neg \text{hasAttr}(v^\circ, a)$ , it is necessary to further consider the rule  $r^\circ$ . If the rule  $r$ , from which  $r^\circ$  is originating, had attribute declarations for required attributes, the rule  $r^\circ$  will have an extended content model on the right side, which adds the constraint  $c_1(a, v^\circ) \Leftrightarrow c_2(a, v^\circ)$  (see Section 5.1.4) for each attribute required by the target language grammar (but without a fixed value) to the right hand side of  $r^\circ$ . As  $c_1(a, v^\circ) = \neg \text{hasAttr}(v^\circ, a)$  is true since it follows from  $\neg \text{hasAttr}(v^\circ, a)$  and the fact, that the template instantiation is not removing attributes from the template,  $c_2$  must also be *true* to satisfy  $c_1(a, v^\circ) \Leftrightarrow c_2(a, v^\circ)$ , therefore  $c_2(a, v^\circ) = \exists v' \in \text{children}(v^\circ) : \text{label}(v') = \text{x tl:attribute} \wedge \text{attr}(v', \text{name}) = a$  must evaluate to *true*. This means that  $v^\circ$  is parent to an `x tl:attribute` instruction which has a `name` attribute with the value  $a$ . From the existence of this `x tl:attribute` child element  $\text{hasAttr}(v, a)$  can be inferred using the semantics of `x tl:attribute` in Listing 4.7.

### 5.1.5.2. Compliance to the Content Model

Compliance of the nodes in the instantiated template to their proposed content model as defined by  $T$  can be shown by induction over the nodes in  $t$ . The induction starts with nodes which have simple content (i.e., the leaves of the tree formed by the XML document  $t$ ). The induction step shows that a node which is containing only nodes that fulfill their proposed content models also fits its own content model.

**Induction start** For each rule  $r \in R$  producing simple content, i.e., for each rule  $r = X \rightarrow e(\text{ad}^*, \sigma)$ , there is a corresponding rule  $r^\circ = \text{cr}(r) \in R^\circ$ . In the following it is shown that if a node  $v^\circ$  in the template is produced by the rule  $r^\circ$ , the node  $v$  instantiated from  $v^\circ$  will be valid with respect to the rule  $r$  in terms of its content, i.e., the node  $v$  will have content which complies to  $\sigma$ .

As  $r^\circ$  is created from  $r$  using the Constraint Separation process described in Section 5.1.4, it has the form  $X \rightarrow e(\text{ad}^*, (\mathbb{S}|\text{x tl:text})(c_a^{\text{text}}(\sigma), c_i^{\text{text}}))$ . As the template  $t^\circ$  is valid with respect to  $T^\circ$  by assumption, the constraint  $c_a^{\text{text}}(\sigma)$  evaluates to *true* for  $v^\circ$ .

It must be shown that *either*  $\text{children}(v) = \epsilon$  with  $\text{Valid}(\sigma, \epsilon)$  *or*  $\text{children}(v) = v_0$  with  $\text{label}(v_0) = \top$  and  $\text{Valid}(\sigma, \text{value}(v_0))$ . There are three cases depending on the number and type of the children of  $v_0$ :

- If the node  $v^\circ$  has no children, i.e., if  $\text{children}^\circ(v^\circ) = \epsilon$ , then the node  $v$  in the instantiated template will also have no children (due to the semantics of XML), i.e.,  $\text{children}(v) = \epsilon$ . The empty node  $v$  is valid with respect to  $r$ , as  $c_a^{\text{text}}(\sigma)$  is true in this case exactly if  $\text{Valid}(\sigma, \epsilon)$ , which has been reasoned above.
- If the node  $v^\circ$  has exactly one child, which is a text node, i.e., if  $\text{children}(v^\circ) = v_0^\circ$  and  $\text{label}(v_0^\circ) = \top$ , the value of this node,  $\text{value}(v_0^\circ)$ , is literally transferred into the instantiated template, i.e.,  $\text{children}(v) = v_0$ ,  $\text{label}(v_0) = \top$  and  $\text{value}(v_0) = \text{value}(v_0^\circ)$ . From the fact that the authoring constraint  $\text{Valid}(\sigma, \text{value}(v_0^\circ))$  is *true* by assumption, it follows that  $\text{Valid}(\sigma, \text{value}(v_0))$  holds, too. The last expression means that  $v$  is valid with respect to the rule  $r$ .

## 5. Safe Authoring of Templates

- If the node  $v^\circ$  has exactly one child, i.e.,  $\text{children}(v^\circ) = v_0^\circ$ , and if this child node is an `xtl:text` element node, i.e.,  $\text{label}(v_0^\circ) = \text{xtl:text}$ , the instantiated node  $v$  will have one child node  $v_0$  with  $\text{children}(v) = v_0$ , which is a text node, i.e.,  $\text{label}(v_0) = \top$  which has a value  $\text{value}(v_0) = d$  taken from the instantiation data. The instantiation data constraint  $c_i^{\text{text}} = (-, \sigma, -, -)$  asserts that  $d \in \sigma$  and therefore, that  $v$  complies to the rule  $r$ .

There are no other cases in which  $c_a^{\text{text}}$  evaluates to *true*, therefore, there are no other sequences of children  $v^\circ$  can have without violating the assumed authoring constraint  $c_a^{\text{text}}$ .

**Induction step** For each rule  $r \in R$  producing non-simple content, i.e., for each rule  $r = X \rightarrow e(\text{ad}^*, \text{cm})$  with  $\text{cm}$  being one of  $\epsilon, X \in N, e' \in E, (\text{cm}_1, \text{cm}_2), (\text{cm}_1 | \text{cm}_2), \text{cm}_1\{i, j\}$  with  $\text{cm}_1, \text{cm}_2 \in \text{CM}(\Sigma, N, E)$ , there is a corresponding rule  $r^\circ = \text{cr}(r) \in R^\circ$ . It must be shown that the node  $v$  instantiated from  $v^\circ$  is valid with respect to the rule  $r$ , if  $v^\circ$  is valid with respect to  $r^\circ$ .

Depending on the concrete value of  $\text{cm}$ , the following cases must be considered:

- If  $\text{cm} = \epsilon$ , the transformed rule  $r^\circ$  will also be of the form  $X \rightarrow e(\text{ad}^*, \epsilon)$ . Thus, the node  $v^\circ$  will have an empty child sequence  $\text{children}(v^\circ) = \epsilon$ , which is instantiated into an empty child sequence on the instantiated node  $v$ , i.e.,  $\text{children}(v) = \epsilon$ . Obviously, this sequence satisfies the content model defined by  $r$ .
- If  $\text{cm}$  is a non-terminal  $X \in N$ , the rules  $r$  and  $r^\circ$  are again identical. Since the instantiation of an arbitrary non-XTL node in a template yields the node itself, with the children in the instantiation being the instantiated children from the template, validity is not affected in this case. Therefore, if  $v^\circ$  complies to rule  $r^\circ$ ,  $v$  complies to  $r$  as well.
- If  $\text{cm}$  is an element name  $e' \in E$ , the transformed rule  $r^\circ$  will be of the form  $X \rightarrow e(\text{ad}^*, e')$ , i.e., the child sequence of node  $v^\circ$  will be  $\text{children}(v^\circ) = v_0^\circ$  with  $\text{label}(v_0^\circ) = e'$ . The instantiation of this node will lead to node  $v$  with the child sequence  $\text{children}(v) = v_0$  with an equally named child:  $\text{label}(v_0) = e'$ . Therefore,  $v$  complies to  $r$ .
- If  $\text{cm}$  is the concatenation  $\text{cm}_1, \text{cm}_2$  of two content models  $\text{cm}_1$  and  $\text{cm}_2$ , the corresponding transformed rule will be of the form  $X \rightarrow e(\text{ad}^*, (\text{cm}'_1, \text{cm}'_2))$ . If the node  $v^\circ$  has the child sequence  $\text{children}(v^\circ) = v_{10}^\circ v_{11}^\circ \dots v_{1n}^\circ v_{20}^\circ v_{21}^\circ \dots v_{2m}^\circ$ , where the nodes  $v_{i0}^\circ v_{i1}^\circ \dots v_{in}^\circ$  are valid with respect to the content model  $\text{cm}'_i$  for  $i \in \{1, 2\}$ , the instantiated node  $v$  will have the child sequence  $\text{children}(v) = v_{10} v_{11} \dots v_{1n} v_{20} v_{21} \dots v_{2m}$ , which will be valid with respect to rule  $r$  if and only if the child nodes  $v_{i0} v_{i1} \dots v_{in}$  are valid with respect to the content model  $\text{cm}_i$  for  $i \in \{1, 2\}$ . In other words, the recursive instantiation process does not change the validity of sequences.
- For content models  $\text{cm}$  which are the alternative  $\text{cm}_1 | \text{cm}_2$  of two content models  $\text{cm}_1$  and  $\text{cm}_2$ , the argumentation for sequences above holds analogously.
- If  $\text{cm}$  is a content model with cardinalities,  $\text{cm}_1\{i, j\}$  for some content model  $\text{cm}_1$ , the argumentation further depends on the content model  $\text{cm}_1$ . If this is not just an element  $e$ , the argumentation for sequences and alternatives above can be applied. If  $\text{cm}_1$  is just

an element  $e$ , several cases depending on the minimum and maximum cardinality must be considered:

- If  $i = j = 1$ , then this case degenerates to the case where  $cm$  is an element name  $e'$  considered above already.
- If  $i = 0$  and  $j = 1$ , the content model  $cm$  will be of the form  $\text{x t l : i f } \langle c_a^{if}(e), c_i^{if}() \rangle | e\{0, 1\}$ . The child sequence  $\text{children}(v^\circ)$  can take three forms:
  - For an empty child sequence, i.e.,  $\text{children}(v^\circ) = \epsilon$ , the instantiated node  $v$  will also have an empty child sequence, i.e.,  $\text{children}(v) = \epsilon$ . This node  $v$  is valid with respect to the rule  $r$ , as the minimum cardinality for the element  $e$  was 0.
  - If the child sequence contains only one node labeled  $e$ , i.e.,  $\text{children}(v^\circ) = v_0^\circ$  with  $\text{label}(v_0^\circ) = e$ , the instantiation will give a node  $v$  with the child sequence  $\text{children}(v) = v_0$  and  $\text{label}(v_0) = e$ , which is again valid with respect to the rule  $r$ , as the maximum cardinality for the element  $e$  is 1.
  - If the child sequence consists of a single node with the label  $\text{x t l : i f}$ , i.e.,  $\text{children}(v^\circ) = v_0^\circ$  with  $\text{label}(v_0^\circ) = \text{x t l : i f}$ , the authoring constraint  $c_a^{if}(e)$  defines the child sequence of  $v_0^\circ$  to be  $\text{children}(v_0^\circ) = v_1^\circ$  with  $\text{label}(v_1^\circ) = e$ . After instantiation, this yields a node  $v$  with either an empty child sequence (if the instantiation data  $d$  used to instantiate the  $\text{x t l : i f}$  instruction evaluated to false) or a single-element child sequence  $\text{children}(v) = v_0$  with  $\text{label}(v_0) = e$  (if  $d$  evaluated to *true*). In both cases, the node  $v$  is valid with respect to the rule  $r$ , as it has been shown in the two cases above.
- In any other case,  $cm$  will be of the form  $\text{x t l : f o r - e a c h } \langle c_a^{for-each}(e), c_i^{for-each}(i, j) \rangle | e\{i, j\}$  and the child sequence  $\text{children}(v^\circ)$  can take two forms:
  - If the child sequence is of the form  $\text{children}(v^\circ) = v_0^\circ v_1^\circ \dots v_n^\circ$  with  $i \leq n \leq j$  and  $\forall 0 \leq k \leq n : \text{label}(v_k^\circ) = e$ , the instantiated node  $v$  has an analogous child sequence  $\text{children}(v) = v_0 v_1 \dots v_n$  with  $\forall 0 \leq k \leq n : \text{label}(v_k) = e$ , which is also a valid sequence with respect to the rule  $r$ .
  - If the child sequence consists of a single node labeled  $\text{x t l : f o r - e a c h}$ , i.e.,  $\text{children}(v^\circ) = v_0^\circ$  with  $\text{label}(v_0^\circ) = \text{x t l : f o r - e a c h}$ , the authoring constraint  $c_a^{for-each}(e)$  restricts the child sequence of  $v_0^\circ$  to be  $\text{children}(v_0^\circ) = v_1^\circ$  with  $\text{label}(v_1^\circ) = e$ . The instantiation of this gives  $v$  a child sequence  $\text{children}(v) = v_0 v_1 \dots v_n$  with  $\forall 0 \leq k \leq n : \text{label}(v_k) = e$ . The instantiation data constraints asserts that  $i \leq n \leq j$ . Taken together, this asserts that the node  $v$  complies to the rule  $r$ .

### 5.1.6. Visitor-based Implementation of the Constraint Separation

An implementation of the Constraint Separation process is much harder than it may look after considering the description in Section 5.1.4. There are two reasons for this: first, the implementation has to deal with the “syntactic sugar” that is removed from the description in the

## 5. Safe Authoring of Templates

previous section (actually, it is hidden in the transformation from an XML Schema to an XGrammar). Second, an implementation has to choose between a number of libraries available for XML Schema, each with their own peculiarities and advantages.

There is a number of libraries for the manipulation of XML Schemas, most notably, the schema manipulation library built into XMLBeans [8] and the analogous library built into Xerces [11]. Furthermore, it is possible to compile the XML Schema metamodel, i.e., `XMLSchema.xsd`, with JAXB, leading to another possibility for the treatment of XML Schema as an object model. The most important difference is the level of abstraction of the library, e.g., the JAXB-generated library represents the concrete schema syntax, whereas the other libraries represent a more abstracted view on the XML Schema.

While it would be very helpful to work on the more abstract syntax level, it is necessary to manipulate the XML Schemas on the concrete syntax level in order to introduce the CXSD and IDC constraints. Unfortunately, the libraries providing the abstract view on the syntax encapsulate the concrete syntax via their Application Programming Interface (API). Therefore, the Constraint Separation component has been implemented using a JAXB-generated XML Schema object model.

As it can be seen in Section 5.1.4, the Constraint Separation is a process which can easily be separated in multiple steps, i.e., for enabling the use of the particular XTL instructions. The Constraint Separation component is therefore implemented as a sequence of steps, where each step operates the XML Schema object model produced by its predecessor. Figure 5.4 shows the processing steps that implement the complete Constraint Separation process.

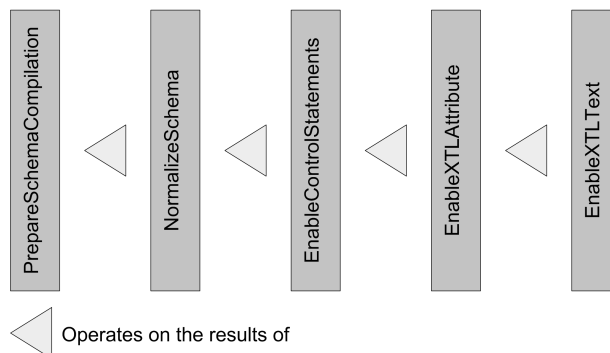


Figure 5.4.: The Constraint Separation Processing Steps

The particular steps are implemented using a slightly modified visitor design pattern [74]. The necessary modifications and their rationals are described in the following.

The first modification is caused by the fact that the XML Schema object model has been generated using JAXB. Therefore, the effort to add `accept` methods to all the classes in the object model was too high. Instead, an `org.lixlil.xtl.compiler.schema.Schema-Acceptor` has been implemented that contains the `accept` methods of all visitable elements.

The `org.lixlil.xtl.compiler.schema.SchemaVisitorBase` contains implementations of `visit` methods for all visitable elements. These methods are intended to be

overwritten by subclasses (i.e., Constraint Separation process steps). Furthermore, the `visit` methods are equipped with two parameters: the first is the visited object, the second its parent object (in the XML document sense). This allows for easy access of the parent object, which would otherwise not be possible because JAXB is not providing access to it by default. Furthermore, it allows concrete implementations to visit elements depending on the context, for example, to visit only `xsd:sequence` elements embedded into `xsd:complexType`, but not into `xsd:restriction` elements.

Finally, the subclasses of `org.linuxltx.compiler.schema.SchemaVisitorBase` can configure the traversal order. In the top-down configuration, the `visit` methods are called before the subelements' `accept` methods are invoked; the bottom-up configuration calls the `visit` methods after the `accept` methods of the subelements. The configuration of the `SchemaVisitorBase` is passed to the `SchemaAcceptor` when the traversal starts.

Besides operating on the same XML Schema, the steps share access to an implementation of the interface `org.linuxltx.compiler.schema.ConstraintSeparationContext` shown in Listing 5.12.

```
public interface ConstraintSeparationContext
{
    /* Lookup of types in the schema */
    public ComplexType getComplexType(String name);

    public SimpleType getSimpleType(String name);

    /* Creation of names. */
    public QName createTNSQName(String localPart);

    public String createTypeName(String suggestion);

    /* Information about namespaces. */
    public String getOriginalTargetNamespace();

    public String getTargetNamespace();

    public boolean isFromOriginalTargetNamespace(QName qname);

    /* Information about related schemata. */
    public String getCXSDSchemaLocation();

    public String getIDCSchemaLocation();

    public String getXTLSchemaLocation();

    /* Accessing the ConstraintFactory. */
    public ConstraintFactory getConstraintFactory();
}
```

Listing 5.12: The `ConstraintSeparationContext` Interface

## 5. Safe Authoring of Templates

The operations in this interface serve one of five purposes. There are methods for the lookup of types in the XML Schema currently processed, for the creation of names, for retrieving information about namespaces, for the management of the location of related schemata and for the retrieval of the `ConstraintFactory` (see below). The particular methods are as follows:

- The `getComplexType` method retrieves the object representing the complex type with the passed name or returns `null` if no such complex type exists.
- The `getSimpleType` method performs the same function as `getComplexType`, but for simple types.
- The `createTNSQName` method creates a `QName` from the passed local part and the target namespace of the XML Schema currently processed.
- The `createTypeName` method creates a name (more exactly, its local part), such that it is unique within the XML Schema currently processed. The method guarantees that there is no complex type or simple type with the same name within the schema (please note that complex and simple types share a common symbol space [180, Section 2.5]). The passed suggestion is first tried as the name, if it already exists, the method repeatedly tries to create a unique name by concatenating an increasing number (starting with 0).
- The `getOriginalTargetNamespace` method returns the target namespace of the XML Schema currently processed as it was originally set.
- The `getTargetNamespace` method returns the target namespace as it should be after processing. In order to prevent confusions, the original target namespace is prefixed with `xtl:` to get the namespace that the XML Schema should have after the process of Constraint Separation, if a target namespace has been defined by the schema. If no namespace has been defined, this method returns `null`.
- The `isFromOriginalTargetNamespace` method can be used to check whether a particular `QName` is defined in the target namespace originally specified by the currently processed XML Schema.
- The `getCXSDSchemaLocation`, `getIDCSchemaLocation` and `getXTLSchemaLocation` methods return the location of the CXSD, IDC or XTL schema, resp., if such locations have been externally configured (e.g., via the command line).
- The `getConstraintFactory` method returns an implementation of the `ConstraintFactory` interface described below.

The interface `org.linuxltx.xtl.compiler.schema.ConstraintFactory` referred to above is shown in Listing 5.13. Using this interface, the Constraint Separation steps can create both authoring and instantiation time constraints (or CXSD and IDC constraints, respectively).

```
public interface ConstraintFactory
{
    /* Authoring Time Constraint Construction */
    public Inv getControlStatementsAuthConstraint(QName
        elementQName);

    public Inv getExpandedSimpleTypeAuthConstraint(QName typeQName);

    public Inv getRequiredAttributesAuthConstraint(Set<QName>
        attributeNames);
}
```

```

/* Instantiation Time Constraint Construction */
public Constraints getAttributesInstConstraint(Map<QName, QName>
    attributesToTypes);

public Constraints getExpandedSimpleTypeInstConstraint(QName
    typeQName);
}

```

Listing 5.13: The ConstraintFactory Interface

The ConstraintFactory provides the following methods:

- The `getControlStatementsAuthConstraint` method returns an authoring time constraint checking that the content of the context node is an element with the passed name `elementQName`.
- The `getExpandedSimpleTypeAuthConstraint` method returns an authoring time constraint that checks that the content of the context node is either an `xtl:text` instruction or a text node with a value complying to the simple type named `typeQName`.
- The `getRequiredAttributesAuthConstraint` method returns an authoring time constraint which makes sure that for each of the attribute names passed (as a `Set`), either an attribute exists at the context node or an `xtl:attribute` instruction exists, which creates the attribute (i.e., has a name attribute set to that particular attribute name).
- The `getAttributesInstConstraint` method returns an instantiation time constraint which checks for each key/value pair in the passed `Map`, whether the attribute named equal to the key is of the type indicated by the value.
- The `getExpandedSimpleTypeInstConstraint` method returns an instantiation time constraint which checks whether the data used for the instantiation of the `xtl:text` instruction is valid with respect to the type passed by its name.

Different technologies have been used to implement the creation of authoring and instantiation time constraints. The authoring time or CXSD constraints are generated from XTL templates, whereas the instantiation time or IDC constraints are constructed programmatically using an object model generated from the IDC XML Schema using JAXB.

### PrepareSchemaCompilation

The first process step in the Constraint Separation process is implemented in the class `org.linuxltxl.compiler.schema.steps.PrepareSchemaCompilation`. This step has two responsibilities: adding import statements for related schemas and changing the target namespace of the schema.

As the Constraint Separation process adds elements from the CXSD and IDC namespaces and adds references to elements from the XTL namespace, this step adds `xsd:import` instructions for these three namespaces. The namespace attribute is automatically set, but the `location` attribute is only set if a location (absolute or relative) has been externally configured (e.g., via the command line). An example for the addition of these `xsd:import` statements is shown in Listing 5.14.

## 5. Safe Authoring of Templates

```
<xsd:import namespace="http://research.sap.com/cxsd/1.0"
            schemaLocation="../../../schemas/CXSD.xsd"/>
<xsd:import namespace="http://research.sap.com/xtl/idc/1.0"
            schemaLocation="../../../schemas/IDC.xsd"/>
<xsd:import namespace="http://research.sap.com/xtl/1.0"
            schemaLocation="../../../schemas/XTL.xsd"/>
```

Listing 5.14: Added `xsd:import` Statements

The `PrepareSchemaCompilation` step is also responsible for changing the target namespace of the processed schema. If the schema to be processed has a target namespace (called *original target namespace* in the following), the namespace URI is appended to the prefix `xtl:`, in order to get the namespace URI for the schema after the Constraint Separation process (called just *target namespace* in the following). As this namespace URI is typically also assigned to another namespace prefix (typically `tns`, an abbreviation for *target namespace*), these other namespace prefixes are also assigned to the target namespace. Furthermore, all qualified names within this schema that point to the original target namespace must be rewritten to point to the target namespace. If no original target namespace has been set, the schema will also lack a target namespace after processing, which makes the processing of prefixes and qualified names obsolete.

### NormalizeSchema

In order to allow the following steps to work under all circumstances, additional changes to the schema being processed are necessary. The `org.lixlix.xtl.compiler.schema.steps.NormalizeSchema` step is responsible for making sure that all simple types which need to be referenced from CXSD constraints are identifiable by name (and not anonymous types) and for making sure that all elements which need to be able to act as content for an `xtl:if` or `xtl:for-each` instruction are declared at the top-level.

The first responsibility mentioned is to create referencable top-level simple type declarations from anonymous type declarations. This is necessary to allow the authoring time constraint embedded during the enablement of `xtl:text` instructions to refer to the simple type (see below). An example for an extracted simple type is shown in Listing 5.15.

```
<xsd:simpleType name="simpleTypeOfQuantity">
  <xsd:restriction base="xsd:positiveInteger">
    <xsd:maxExclusive value="100"/>
  </xsd:restriction>
</xsd:simpleType>
```

Listing 5.15: Top-level Declaration of a Previously Anonymous Simple Type

The second responsibility is to create top-level elements from all `xsd:element` elements which will be enabled for the use within `xtl:if` or `xtl:for-each` statements later on. This is necessary since the `xtl:if` and `xtl:for-each` definitions within the XTL XML Schema



contain a wildcard (`xsd:any`) with its `processContents` attribute set to `strict`, which requires a top-level declaration of the element to fulfill the wildcard [180, Section 3.10.1].

This process of promoting local elements to top-level elements can cause name clashes, as the element is moved from the namespace opened by a complex type or element declaration into the single global namespace for elements within the schemas' target namespace. The alternative to this promotion of local element declarations is to relax the wildcard within the XTL schema (i.e., to set its `processContents` attribute to `lax` or `skip`), and to check the local validity of the content with regards to its element name and its complex type from within a CXSD constraint. This variant has not been implemented, as the standard XML parsers with validation do not expose a simple interface to check local validity, which would make the implementation of the necessary extension of the CXSD metamodel much harder.

### EnableControlStatements

The processing step `org.lixi.xtl.compiler.schema.steps.EnableControlStatements` has the responsibility to enable the use of `xtl:if` statements for all conditional elements and to enable the use of `xtl:for-each` statements for all repeatable elements. Conditional elements are defined as elements with a cardinality of `0..1`, repeatable elements as having a maximum cardinality greater than 1.

The whole schema is traversed for elements fulfilling the conditions for conditional or repeatable elements. Each occurrence of such an element is replaced by a choice between the element and a reference to the `xtl:if` or `xtl:for-each` element from the XTL schema respectively. The `xtl:if` and `xtl:for-each` references are further restricted by inserting a CXSD constraint defining that the content of the XTL instruction must have a particular qualified name. This constraint is constructed using the `getControlStatementsAuthConstraint` method from the `ConstraintFactory`. An example showing a processed optional element is shown in Listing 5.16.

```
<xsd:choice>
  <xsd:element minOccurs="0" ref="comment"/>
  <xsd:element ref="xtl:if">
    <xsd:annotation>
      <xsd:appinfo source="http://research.sap.com/cxsd/1.0">
        <cxsd:inv name="control-statement-constraint">
          <cxsd:ocl><![CDATA[
            let
              elementChildren:Sequence(ElementNode) = self.children->
                select(oclIsTypeOf(ElementNode))->collect(oclAsType(
                  ElementNode))
            in
              elementChildren->size() = 1 and
              elementChildren->at(0).qname = self->expandQName('
                comment')
          ]]></cxsd:ocl>
        <cxsd:message>
      </cxsd:message>
    </xsd:annotation>
  </xsd:element>
</xsd:choice>
```

## 5. Safe Authoring of Templates

```
        </cxsd:inv>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
</xsd:choice>
```

Listing 5.16: Choice between comment and `xtl:if`

### EnableXTLAttribute

The next processing step, `org.lixi.xtl.compiler.schema.steps.EnableXTLAttribute`, is responsible for enabling the use of the `xtl:attribute` instruction. To achieve this, four modifications are made to all complex types in the schema.

First, an element reference to `xtl:attribute` is added to the content model of the complex type. If the complex type is already defined to be a sequence, the reference is just inserted at its beginning. If the complex type is defined to be a choice, this choice is wrapped by a sequence that contains the reference to `xtl:for-each` followed by the choice. Unfortunately, the insertion of `xtl:attribute` is not possible if the complex type inherits from another complex type, as the insertion would only be possible at the end of the content model, which is not acceptable. Therefore, inherited types are excluded from this processing (see Chapter 8 for the suggestion to change XML Schema with respect to this).

Second, the usage attribute of all attribute declarations for required attributes is relaxed to optional, which allows the template author to omit required attributes (and create them via `xtl:attribute` instead). In the `po.xsd` example, the `partNum` attribute is relaxed to become an optional attribute.

Third, an IDC constraint is added, containing the expected types for all attributes. In the `po.xsd` example, the instantiation data used to create the `partNum` attribute is bound to be of type `SKU`. The result of these first three processing steps is shown in Listing 5.17.

```
<xsd:element name="item">
  <xsd:complexType>
    <xsd:annotation>
      <xsd:appinfo source="http://research.sap.com/cxsd/1.0">
        <!-- See Listing 5.18 -->
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" minOccurs="0" ref="
        xtl:attribute">
      <xsd:annotation>
        <xsd:appinfo source="http://research.sap.com/xtl/idc/1.0">
          <idc:constraints>
            <idc:constraint for-name="partNum" type="SKU"/>
          </idc:constraints>
        </xsd:appinfo>
      </xsd:annotation>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

```

</xsd:element>
<!-- ... -->
<xsd:attribute name="partNum" type="SKU" use="optional"/>
</xsd:complexType>
</xsd:element>

```

Listing 5.17: Enabled `xtl:attribute` with IDC Constraints

Finally, the processing step adds an additional CXSD constraint which ensures that the required attributes are either directly specified or created via an appropriate `xtl:attribute` instruction. In the `po.xsd` example, it is ensured that `partNum` is either directly specified or created via an `xtl:attribute` instruction (which therefore needs to have a `name` attribute with a value of `partNum`). The OCL part of the CXSD constraint is shown in Listing 5.18.

```

let
  xtlAttributeChildren:Sequence(ElementNode) = self.children->select(oclIsTypeOf(
    ElementNode))->collect(oclAsType(ElementNode))->select(localName='attribute'
    and namespaceURI='http://research.sap.com/xtl/1.0')
in
  OrderedSet{
    self.expandQName('partNum')
  }->forAll( attributeQName |
    let
      xtlAttributeChild:ElementNode = xtlAttributeChildren->any(attributes->select(
        name='name' and self.expandQName(value)=attributeQName)->size() > 0),
      attributePresent:Boolean = self.attributes->select(qname=attributeQName)->size()
        > 0
    in
      not(attributePresent) implies not(xtlAttributeChild->isEmpty())

```

Listing 5.18: A CXSD Constraint for Required Attributes

### EnableXTLText

The last processing step, `org.lixi.xtl.compiler.schema.steps.EnableXTLText`, allows the use of `xtl:text` to create the text content of elements with simple content. To this end, the processing step processes all top-level elements referencing a non-anonymous simple type (both preconditions are asserted by the `NormalizeSchema` step). The reference to the simple type is replaced with a reference to a complex type with mixed content. This newly created type allows the use of `xtl:text` within. An IDC constraint is added which restricts the instantiation data used to replace the `xtl:text` instruction during the Template Instantiation to the simple type originally referenced by the element. In the `po.xsd` example, the type `xtlTextOrDecimal` is created to replace elements with simple content complying to the `xsd:decimal` type. The result of this processing is shown in Listing 5.19.

```

<!-- ... -->
<xsd:element name="zip" type="xtlTextOrDecimal"/>
<!-- ... -->
<xsd:complexType mixed="true" name="xtlTextOrDecimal">
  <xsd:annotation>

```

## 5. Safe Authoring of Templates

```
<xsd:appinfo source="http://research.sap.com/cxsd/1.0">
  <!-- See Listing 5.20 -->
</xsd:appinfo>
</xsd:annotation>
<xsd:sequence>
  <xsd:element maxOccurs="1" minOccurs="0" ref="xtl:text">
    <xsd:annotation>
      <xsd:appinfo source="http://research.sap.com/xtl/idc/1.0">
        <idc:constraints>
          <idc:constraint type="xsd:decimal"/>
        </idc:constraints>
      </xsd:appinfo>
    </xsd:annotation>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>
```

Listing 5.19: Enabled `xtl:text` for the Creation of the Content of the `zip` Element

Furthermore, the processing steps add an CXSD constraint that restricts the use of text content within the newly created type to be compliant to the simple type originally referenced by the element. In the `po.xsd` example, the constraint asserts that either an `xtl:text` element or a text node complying to the type `xsd:decimal` is present. The OCL part of the CXSD constraint is shown in Listing 5.20.

```
let
  expectedQName:QName = self.expandQName('{http://www.w3.org/2001/XMLSchema}decimal'),
  textChildren:Sequence(TextNode) =
    self.children->select(oclIsTypeOf(TextNode))->collect(oclAsType(TextNode)),
  xtlTextChildren:Sequence(ElementNode) =
    self.children->select(oclIsTypeOf(ElementNode))->collect(oclAsType(ElementNode))
->select(localName='text' and namespaceURI='http://research.sap.com/xtl/1.0')
in
  if (xtlTextChildren->size() = 0)
  then
    textChildren->size() = 1 and
    textChildren->at(1).isValidLiteral(expectedQName)
  else
    xtlTextChildren->size() = 1 and textChildren->size() =
    textChildren->select(isIgnorableWhitespace)->size()
  endif
```

Listing 5.20: A CXSD constraint for Simple Content

### 5.1.7. Partial Templatzation

In [143], it has been stated that every template engine has at least an entanglement index of 1, as it is impossible for the template engine to decide whether a value which is instantiated into a template plays a role in the content or layout of the instantiated template.

As a side effect of the safe authoring approach described above, a technique that enables an entanglement index of 0 becomes possible. In the XHTML 1.0 document shown in Listing 5.21,

two attribute values are highlighted. First, the value of the `style` attribute, which obviously contains style information. Second, the value of the `alt` attribute is highlighted, which contains content, as it gives an alternative representation for the `img` element it is assigned to.

For every well designed XML language like XHTML, there must be a way to decide whether a particular piece of information (like a text or an attribute value) is content or layout. In general, the use of a template engine implies that separation of concerns is an issue, so there must be a set of rules to decide whether an information is allowed to be filled from instantiation data or whether it must be part of the template.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Sample XHTML Document</title>
  </head>
  <body>
    <p style="text-indent:1em;"> ①
      Some content.
       ②
    </p>
  </body>
</html>
```

Listing 5.21: A Simple XHTML 1.0 File

The constraint separation process described above can be adjusted to allow the dynamic creation of only certain character data items. In contrast to the standard constraint separation process, partial templatzation allows the use of the XTL instructions only partially. This allows the refinement of the contract between the template author and the application using the template technique: the `style` ① attribute would not be allowed to be created dynamically from the instantiation data by the Constraint Separation process (as it is *layout* in the sense of Figure 3.2), whereas the `alt` ② attribute would be allowed (as it is *content* in the sense of the aforementioned figure).

Technically, partial templatzation could be best configured using a language that allows to select the XML Schema parts (attribute and element declarations, type definitions etc.) which should be subject to the constraint separation process. A subset of the Path Language for XML Schema (SPath) [126] would be a good choice for such a language.

## 5.2. Template Validation

The *Template Validation* process is the responsibility of the template validator component. The template validator allows the change of the template development process to the one shown in Figure 5.5. The new process is more straight-forward than the current process (cf. Figure 1.2): the template author does not need to change its role to that of the user of the template technology like in the traditional process. Instead, he gets direct feedback about the template. Furthermore, the validation of the template does no longer depend on a particular set of instantiation data, since the validation result makes a general statement about the template of the form: *if* the instantiation data is as specified by the instantiation data constraints emitted by the Con-

## 5. Safe Authoring of Templates

straint Separation process, *then* the instantiated template will comply to the target language grammar. Thus, the Template Validation process contributes to the Safe Authoring goal stated in Section 3.1.1.

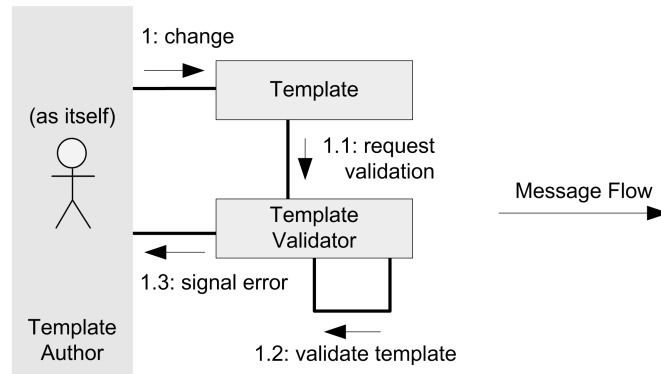


Figure 5.5.: The Proposed Development Process for Templates

The design of the template validator has two responsibilities. First, it has to check the validity of XML documents against the CXSD schemas as emitted by the Constraint Separation process. Furthermore, the template validator component is also responsible for establishing a link between the XTL instructions in the template and the instantiation data constraints in the schema.

The Constraint Separation process defined the extension of XML Schema with OCL constraints (see 5.1.2) and relaxed the UPA rule in order to enhance the expressive power of XML Schema. The template validator must therefore validate XML Schema without the restrictions introduced by the UPA and evaluate the embedded OCL constraints. After an analysis of existing XML parsers capable of validating the XML input against a given XML Schema, it became obvious that it is easier to reimplement a validating XML parser than to extend one of the existing implementations like Xerces. This is mostly due to the fact that the existing XML parsers are not designed to be extensible, which makes extending them a time-consuming and tedious task.

As mentioned above, it is also the responsibility of the template validator to establish a link between the elements of the XTL template and the instantiation data constraints. This link is necessary to allow the Instantiation Data Validation component to evaluate the instantiation data constraints. This link is comparable to the link between an XML document and its XML Schema established by the Post-Schema-Validation Infoset (PSVI). The difference is that the link between the template and its CXSD schema is more specific. It is more specific, as it links only XTL instructions to the type that the instantiation data must comply to, whereas PSVI is directly linking every element to the corresponding type in the XML schema.

In order to enable the Instantiation Data Validation component to facilitate the established link, it has to be persisted. There are basically two options for this. First, it is possible to persist the link in an extra document that for example maps XPath expressions pointing into the template to the expected type of the instantiation data. Second, it is possible to transfer the link data within the XTL template itself. As the link is only starting from XTL instructions, the second alternative has been chosen. To enable this, the affected XTL instruction has been extended to

allow a `type` attribute that can denote the required instantiation data type via its QName. This approach is similar to the approach proposed to persist the PSVI in [167].

The following additional attributes are defined to carry the instantiation data constraint link: for the `xtl:text` and `xtl:attribute` instructions, the `type` attribute is defined, which allows a QName as value which denotes the type that the value used to create the text or the attribute value must comply to. There is no need to define the `type` attribute for `xtl:if` and `xtl:for-each`, since the type to be delivered by the Instantiation Data Evaluator is defined implicitly (see Section 4.1). For the `xtl:for-each` instruction, the attributes `min` and `max` are defined, which are of the same type as `minOccurs` and `maxOccurs` defined in [180].

An XTL template augmented with instantiation data constraints is shown in the upper part of Listing 5.22. Obviously, the `xtl:for-each` instruction can be executed an arbitrary number of times ①. The `partNum` attribute must be created with a value complying to the `SKU` type ②, whereas the text content of the `productName` element must comply to the `xsd:string` type ③. As an example for a persistent PSVI, a fragment of the `po.xml` document is shown in the lower part of Listing 5.22. Interestingly, the questionable use<sup>1</sup> of an attribute like `psvi:atttypes` ④ is not necessary in the proposed approach, since the expected type of a particular attribute is attached to its `xtl:attribute` instruction rather than to the element that the attribute is going to be assigned to during the Template Instantiation process.

The actual Template Validation process is implemented by creating a DOM of the instance to be validated first. This DOM is used for two purposes: for the creation of a Streaming API for XML (StAX) event stream that is used for the standard XML Schema validation and to build a representation of the document as an instance of the meta model introduced in Section 5.1.2 (see Figure 5.2, especially). As the UPA can be violated by a CXSD document, the validation component is implemented using a backtracking algorithm. The backtracking algorithm is in turn supported by a component implementing the `org.lixlix.xtl.cxsd.TransactionalReader` interface that allows the validator to read events from the StAX stream transactionally. The `TransactionalReader` interface could easily be implemented by subclassing the component that implements the `org.lixlix.xtl.engine.impl.ReadWindow` interface described in detail in Section 6.2.2.

The mere XML Schema validation process of an XML document has been tested against a subset of the XML Schema Test Suite [181]. This subset excludes the tests in which XML Schema documents include features that are not supported by the Constraint Separation process. The Template Validation component complies to 97% of the remaining subset of approx. 24000 documents. The documents in which the Template Validation component fails are mostly documents with literals that are checked against particular simple types, especially with literals in Japanese or Chinese encodings. In these tests, the validation results returned by the XML Schema Test Suite and by the Xerces simple type validation facility disagree, which in turn causes the Template Validation component to disagree with the result proposed by the XML Schema Test Suite.

---

<sup>1</sup>Sperberg-McQueen [167]: “It should be noted that I am fully aware that this solution is ugly. Long meditation on this problem has convinced me that every available solution to this problem is ugly: attributes were designed to have atomic or simple list values, not to have attributes of their own, and I no longer expect to find a pretty way to go against the grain of XML here.”

```

<items>
  <xtl:for-each select="items" min="0" max="unbounded"> ①
    <item>
      <xtl:attribute name="partNum" select="id" type="SKU"/> ②
      <productName>
        <xtl:text select="productName" type="xsd:string"/> ③
      </productName>
      <!-- ... -->
    </item>
  </xtl:for-each>
</items>

```

```

<shipTo
  country="US"
  psvi:type="po:USAddress"
  psvi:atttypes="country xsd:NMTOKEN" ④
>
  <name psvi:type="xsd:string">Alice Smith</name>
  <street psvi:type="xsd:string">123 Maple Street</street>
  <city psvi:type="xsd:string">Mill Valley</city>
  <state psvi:type="xsd:string">CA</state>
  <zip psvi:type="xsd:decimal">90952</zip>
</shipTo>

```

Listing 5.22: Linked Instantiation Data Constraints Compared with Embedded PSVI

In addition to the mere XML Schema validation process, the DOM instance is also used to evaluate the OCL expressions embedded into the CXSD document. As the foundation for the evaluation of the OCL expressions, the OCL library from the Eclipse MDT project has been used. This OCL library operates on a model compliant to the metamodel shown in Figure 5.2. The model is basically an Eclipse Modeling Framework (EMF)-based view on the DOM instance. The overall validation process has been validated as described in Section 7.2. The extra effort needed to evaluate the OCL constraints has been measured, the results of this measurement are presented in Section 7.5.1.

The validation algorithm is rather straightforward. A virtual `xsd:choice` between all root level element declarations in the XML Schema is created, which is used to start the validation process. The template validator reads from the StAX stream and tries to validate these events against the alternatives within the choice. As an alternative might turn out to be invalid, a transaction is started on the StAX event stream using the `TransactionReader` component. The transaction will be committed if an alternative proves to be matching or it will be rolled back otherwise. If an OCL constraint is attached to an element in the XML Schema, the constraint will be evaluated with its context set to the element currently validated. If the constraint evaluates to false, the validation process will fail at this point, again causing a rollback of the innermost transaction. The overall validation process returns true if there is an assignment of element



and types in the XML Schema against the content of the document where the OCL constraints within the CXSD schema evaluate to true at all nodes in the document where the enclosing XML Schema instruction is used to validate the instance.

In addition to the boolean result of the validation, during a successful validation, the IDC constraints embedded in the CXSD schema are transferred into the validated XML instance. As the IDC constraints may be reassigned when a transaction is rolled back, the validation process writes the augmented XML events into an implementation of the interface `org.lxlxl.xtl.cxsd.TransactionalWriter`, which defers the further processing of the events until all transactions are committed.

### 5.3. Conclusion

This chapter introduced the components of the proposed architecture involved during the authoring time of a template, namely the Constraint Separation and the Template Validation component.

The Constraint Separation process, responsible for separating authoring time from instantiation time constraints, has been described precisely, which involved the definition of two supporting languages, CXSD and IDC. The CXSD extends XML Schema in two ways: by allowing embedded OCL constraints to more precisely specify properties to be fulfilled by instances that should comply to the described language. The IDC is a helper language used to transfer instantiation data constraints to the instantiation phase of the proposed process. A slight modification of this process called Partial Templatization has been introduced, which allows to reach an entanglement index (in the sense of [144]) of 0.

The Template Validation component is responsible for validating the authoring time constraints, i.e., for validating a template against a CXSD document. Technical reasons and the relaxation of the UPA rule by the CXSD language complicated the rather elementary implementation of this component.

## *5. Safe Authoring of Templates*

# 6

## Flexible, Efficient and Safe Template Instantiation

*Hofstadter's Law:* It always takes longer than you expect, even when you take into account Hofstadter's Law.

Douglas Hofstadter, 1979 [87]

This chapter discusses the processes of Instantiation Data Evaluation, Template Instantiation and Instantiation Data Validation from Figure 3.5. All these processes correspond to the *template instantiation* process in the same sense as used in existing approaches. For each process, a different main issue can be defined: flexibility is most important for the Instantiation Data Evaluation, efficiency is the key issue in the Template Instantiation process, and safety is guaranteed by the Instantiation Data Validation process. As in Chapter 5, the target language is required to be an XML dialect defined by an XML Schema.

### 6.1. Instantiation Data Evaluation

The evaluation of the instantiation data is performed by the *instantiation data evaluator* component. The architecture requires this component to decouple the template engine from a particular query language. In an implementation of this architecture, the instantiation data evaluator can be realized as a plugin of the template engine, allowing a single implementation to cooperate with data sources as distinct as XML documents, relational databases or UML models. The Java implementation of the XTL Engine calls the plugin a Placeholder Plugin (PHP).

## 6. Flexible, Efficient and Safe Template Instantiation

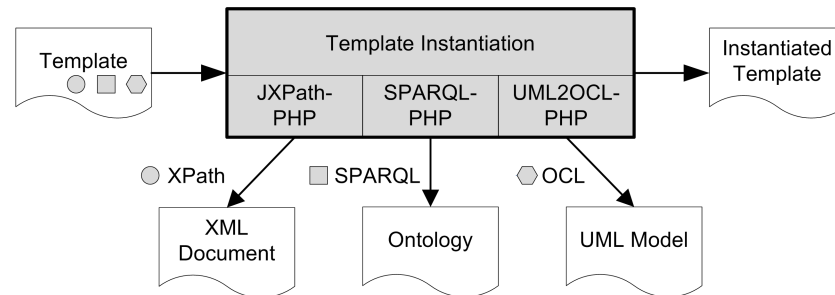


Figure 6.1.: Accessing Multiple Instantiation Data Sources Using Multiple PHPs

### 6.1.1. Design of the PHP Interface

Using a plugin approach for the adaptation of the template engine to a particular query language directly addresses the Independence of Query Language requirement. Besides this, the approach turned out to have a number of additional advantages.

First of all, the XTL's realm feature can be used together with multiple plugins to access data from multiple instantiation data sources at once. There is no need to use the same query language for the instantiation data sources, instead, the query language can be chosen arbitrarily for each source. For example, an XML document, an ontology and a UML model can be accessed from within one template using the query languages XPath, SPARQL and OCL, resp., as illustrated in Figure 6.1. It is also possible to access multiple XML documents as instantiation data sources represented by realms—an approach which provides a more uniform access to multiple XML documents than the approach provided by XSL-T with its `document` function [107, Section 16.1].

Furthermore, a plugin can also be used to build an intermediate view on the instantiation data source. An example is the SPARQL plugin which is supplied as part of the XTL Engine. This plugin allows to query ontologies. As an option, the plugin is capable of building the transitive closure on its underlying ontology. This equips SPARQL with the capability to execute transitive queries, which is originally not part of the language. In general, plugins can be used for the creation of all kinds of intermediate, transient models.

A plugin can also be used to change the instantiation data access strategy (see Section 2.5.4). Due to the design of the XTL, the XTL Engine implements a pull strategy: every time an expression from the query language is found as the value of a `select` attribute, it is executed immediately. An application could calculate all the instantiation data before actually invoking the XTL Engine, bundle the data into a custom PHP and pass it to the template engine. The engine would still pull the data from the PHP, but effectively, the engine with the PHP now follows the push strategy. This is basically an application of the Move Copy of Data pattern as described in [178] (but used at design level, which is different from its description, where it is applied to the architectural level). Such an inversion of the data access strategy can also be achieved using the Template Interface Generation approach described in Section 6.3.2.

A PHP is defined by the interface `PlaceholderPlugin` shown in Figure 6.1. It mirrors the functionality of the `IDS` class from the denotational semantics of the XTL with two minor differences.

First, the IDS has been defined using Haskell's type classes (see Listing 4.2), whereas the `PlaceHolderPlugin` uses Java's generic types feature. For this reason, the interface has a type parameter named `Type`. An implementation of the interface would use an appropriate context type as the actual value for this parameter. As an example, a `JXPath` implementation would use a class that represents the concept of the current node from XSL-T.

Furthermore, the `PlaceHolderPlugin` interface distinguishes between the evaluation of the `select` attribute at an `xtl:attribute` and an `xtl:text` element, whereas the IDS knows only one of them. This is caused by the fact that the `xtl:attribute` and `xtl:text` instructions are each represented by their own Java classes, which do not share a common interface from which the value of the `select` attribute could be fetched.

```
public interface PlaceHolderPlugin<Type>
{
    public String evaluateAttribute(
        XTLLAttribute xtlAttribute, Type argument);

    public Iterator<Type> evaluateForEach(
        XTLLForEachStart xtlForEach, Type argument);

    public boolean evaluateIf(
        XTLLIfStart xtlIf, Type argument);

    public List<XMLEvent> evaluateInclude(
        XTLLInclude xtlInclude, Type argument);

    public String evaluateText(
        XTLLText xtlText, Type argument);

    public void init(List<XMLEvent> event);

    public void onEndDocument();
}
```

Listing 6.1: The `PlaceHolderPlugin` Interface

Please note that the `evaluate` operations all share analogous arguments. The first argument corresponds to the `XTL` instruction for which they are responsible. The hierarchy of the classes for these arguments is explained in Section 6.2.3 (see esp. Figure 6.6). The second argument is the current value of the innermost enclosing `xtl:for-each` instruction, which accesses the same realm. If no such plugin exists, a `null` value will be passed to the PHP. The second argument is of the generic type `Type`, as the actual value representing an iteration in an `xtl:for-each` loop depends on the query language and the PHP implementation.

The operations of the `org.lixlix.xtl.php.PlaceHolderPlugin` are described below:

- The `evaluateAttribute` operation has to be implemented by the PHP to support the evaluation of `xtl:attribute` instructions. The corresponding function within the IDS

## 6. Flexible, Efficient and Safe Template Instantiation

is the `evalText` function. An implementation of a PHP should not include the name of the attribute into the calculation of its value in any way.

- The `evaluateForEach` operation has to be implemented by the PHP to support the evaluation of `xtl:for-each` instructions. In order to allow nested `xtl:for-each` instructions, this operation also gets the current value of the innermost enclosing `xtl:for-each` instruction as described above. The implementation of this operation must return an `Iterator` over values of the generic type `Type`, as a value from this operator will be passed to other `evaluate` methods as second argument (see above). The corresponding function within the IDS is the `evalForEach` function.
- The `evaluateIf` operation has to be implemented by the PHP to support the evaluation of `xtl:if` instructions. The corresponding function within the IDS is the `evalIf` function.
- The `evaluateInclude` operation has to be implemented by the PHP to support the evaluation of `xtl:include` instructions. The corresponding function within the IDS is the `evalInclude` function. The implementation of this operation must return a `java.util.List` of `javax.xml.stream.XMLEvents`. For the rationale of this return type see Section 6.2.1.
- The `evaluateText` operation has to be implemented by the PHP to support the evaluation of `xtl:text` instructions. The corresponding function within the IDS is the `evalText` function.
- The `init` operation has to be implemented by the PHP to support the evaluation of `xtl:init` instructions. There is no corresponding function in the IDS, as the denotational semantics for XTL does not include realms. The only argument of this operation is a `java.util.List` of `javax.xml.stream.XMLEvents`. For the rationale of this type see again Section 6.2.1.
- The `onEndDocument` operation has to be implemented by the PHP to get notified when the XTL template to be instantiated has been completely processed. This can be used by the PHP to perform cleanup operations or to conclude diagnostical information.

To make implementing a new PHP as easy as possible, an abstract base implementation of the `PlaceholderPlugin` interface is provided via the class `org.linux.xtl.php.impl.PlaceholderPluginImpl`, which mainly provides default implementations for most of the methods and aggregates the processing of `xtl:attribute` and `xtl:text` methods via a common `evaluateSelect` method.

### 6.1.2. The Identity PHP

The Identity PHP serves multiple purposes. It can be used for testing, to explain the concept of a PHP, and it can be used for syntactical reasons, e.g., to create an `xtl:for-each` instruction which repeats its content exactly  $n$  times for an arbitrary but fixed  $n$ .

The Identity PHP got its name from its easy implementation of the `evaluateText` and `evaluateAttribute` operations described above: it just returns the value of the `select` attribute of the `xtl:text` or `xtl:attribute` instruction evaluated. The `evaluateIf` operation is implemented in a similarly easy way: if the `select` attribute has a (text) value of `true`, it returns the (boolean) value `true`, and `false` otherwise. The `evaluateForEach`

operation parses its `select` attribute as an integer and returns an iterator containing the integers from 1 to the parsed value or an empty iterator if no value could be parsed or the parsed value was 0.

From the implementation of the `evaluateForEach` operation, which is returning an iterator over integers, it is clear that the type parameter `Type` has the actual value `java.lang.Integer` for the Identity PHP.

### 6.1.3. The XPath PHP

The XPath PHP allows the XTL to use XPath expressions in order to access the instantiation data source. As XPath [9] is used for the implementation of the PHP, the instantiation data source does not necessarily have to be an XML document, as XPath allows the evaluation of XPath expressions against any object model using a well-defined mapping between the XPath syntax and Java object properties and associations.

The XPath PHP makes XTL comparable to XSL-T and therefore enables the definition of the instantiation semantics in Section 4.6 and the time comparison with JSP and XSL-T in Section 7.5. Furthermore, as XPath is well supported by Haskell (more specifically, by the *Haskell XML Toolbox* [163]), it also enables the direct comparison of the Java implementation of the Template Instantiation component with the denotational semantics given in Chapter 4.

The `evaluateAttribute` and `evaluateText` operations work exactly as they would do in XSL-T. The notion of the context item [107, Section 5.4.3.1] is reused, the context item corresponds to the notion of a *context* introduced in Section 4.3. The `evaluateForEach` operation establishes a new context. The `evaluateIf` method follows the suggestion from Section 4.3 and uses the XPath function `boolean` to determine the boolean value from the result of the evaluation of the `select` attribute. The XPath PHP also implements the `evaluateInclude` method—the result is similar to the result returned by `xsl:copy-of` [107, Section 11.9.2], when applied to the current context.

The type parameter named `Type` of the `PlaceholderPlugin` interface is assigned the actual value `org.apache.commons.xpath.Pointer` by the XPath PHP. This `Pointer` class is an implementation of the context item from XPath in XPath, which perfectly matches the use of it as the context in the PHP implementation.

The XPath PHP implementation is an alternative implementation for using XPath expressions using the XPath implementation of Xalan. The XPath PHP is only usable to evaluate XPath expression on XML documents represented as DOM. Therefore, the type parameter `Type` has the actual value `org.w3c.dom.Node`.

### 6.1.4. The SPARQL PHP

The SPARQL PHP enables the use of SPARQL expressions to fetch instantiation data from within XTL templates. The motivation for this plugin came from the Feature-getriebene, aspektorientierte und modellgetriebene Produktlinienentwicklung (FeasiPLe) project (see Section 7.3). The handling of XML namespaces within SPARQL also motivated the introduction of the `xtl:init` instruction.

## 6. Flexible, Efficient and Safe Template Instantiation

In contrast to the XPath plugin, the SPARQL PHP can not process `xtl:attribute`, `xtl:text` or `xtl:if` instructions which are not enclosed in an `xtl:for-each` instruction. This is a design decision caused by the fact that SPARQL queries return a relation, whereas XPath queries return a node. Therefore, there has to be a distinction between the building of a relation (selection) and the access to particular values (projection). This distinction is mapped to the PHP operations by using the `xtl:for-each` instruction in order to retrieve the relation by selection, and by letting `xtl:attribute`, `xtl:text` and `xtl:if` access particular attributes from the result. The retrieval of the relation is performed by the `SELECT` statement of SPARQL, whereas the access to a particular attribute is performed using a subset of the SPARQL syntax, namely its query variable syntax [151, Section 4.1.3]. As the existence of a relation is a prerequisite for the projection, an `xtl:for-each` is required around all `xtl:attribute`, `xtl:text` and `xtl:if` instructions.

As SPARQL itself has a restricted expressive power, the SPARQL PHP supports the execution of an inference process on the ontology before it is queried. The inference process is parametrized by a set of rules that can be specified in any rule language supported by the underlying ontology processing framework. The framework being used to implement the SPARQL PHP is Jena [100]. The class `com.hp.hpl.jena.query.QuerySolution` is the actual value of the `Type` type parameter of the `PlaceholderPlugin` interface.

### 6.1.5. The System PHP

The System PHP is a PHP implementation that supports special functions that are sometimes useful but are not supported by some query languages. This PHP supports only `xtl:attribute`, `xtl:text` and `xtl:if` instructions—it cannot be used as the realm for `xtl:for-each` and `xtl:include` instructions. Its query language syntax is based on XPath, but as no context node exists, only a fixed set of predefined functions can be used. The functions are as follows:

- The `build-number` function can be used to retrieve the build number of the XTL Engine currently in use.
- The `file-exists` function can be used to check whether there exists a file with the passed path.
- The `last` function can be used to check whether the innermost enclosing `xtl:for-each` instruction is currently processed for the last time.
- The `position` function can be used to retrieve the current index of the control variable value in the sequence of control variable values for the innermost enclosing `xtl:for-each` instruction.
- The `version` function can be used to return the version of the XTL Engine currently in use.

The System PHP is a subclass of the XPath PHP, but as it does not support `xtl:for-each`, the actual value of the `Type` parameter of the `PlaceholderPlugin` interface does not matter.



## 6.2. Template Instantiation

The template engine is the component performing the Template Instantiation process. It is therefore influenced by the required expressiveness offered by the slot markup language. Furthermore, the template engine invokes the Instantiation Data Evaluation and Instantiation Data Validation processes in order to retrieve and verify the instantiation data.

### 6.2.1. XML Access Technologies

The technology used to read the templates and to emit the instantiation result has the greatest influence on the efficiency of an XML template engine implementation. This is especially true if the template engine interprets the templates rather than compiling them (see Section 2.5.6).

Typically, XML access technologies are classified in event-driven approaches and object-oriented XML representations. *Event-driven approaches* are typically efficient (both with respect to memory and time) and provide a sequential access to XML documents. The most prominent example for an event-driven technology is the Simple API for XML (SAX). SAX has been originally designed for read-only access. Other event-driven approaches like StAX have been designed to also support the creation of XML documents.

*Object-oriented approaches* provide a complete, random-access view on an XML document. The abstraction level of this view depends on the actual technology. The most prominent example for an object-oriented access technology is DOM, which provides a view which closely resembles the XML structures as defined in [28]. XML binding tools like JAXB and XMLBeans (see also Section 2.3.3) can also be considered object-oriented access technologies with a higher level of abstraction. Object-oriented approaches are typically designed to support both reading and writing XML documents. The higher level of abstraction provided by these technologies typically slows down reading, whereas the provided random-access view typically causes significantly higher memory consumption.

Typically, a template engine produces arbitrary large documents from relatively small ones. The necessity to create arbitrary large documents leads to the decision for an event-driven approach. This decision was eased by the fact that the constructed document is not changed after its creation, i.e., there is no need for random access to the document under construction.

For reasons of simplicity, it has been decided to use the same XML access technology for both reading the template and creating the output document. The first evaluation of SAX for reading XML documents exposed one of its major weaknesses. SAX dictates the user to structure its algorithm along its `ContentHandler` interface, which obscures the algorithm. Furthermore, if look-ahead is necessary to complete the processing of a particular XML structure, the interface enforces the implementation of additional data structures which have to be examined in all of its methods. This problem has also been described in [152].

In contrast to the statements in [152], a solution for this problem exists with the StAX technology since 2003 [94; 145]. StAX is an event-driven XML access technology which is designed to read and write XML documents and which incorporates ideas from the object-oriented approaches by providing an object model for its events. StAX does not prescribe a fixed structure for the XML processing algorithm. While SAX pushes events into the processing logic, a StAX implementation is pulled by the processing logic when it is ready to continue. For this reason,

## 6. Flexible, Efficient and Safe Template Instantiation

StAX belongs to the family of so-called pull parsers (in contrast to the push parsing approach in SAX). The difference between both types of parsers is shown in Figure 6.2.

The choice of StAX allowed the uniform treatment of reading and writing XML documents and fulfilled the requirement of being able to implement lookahead efficiently.

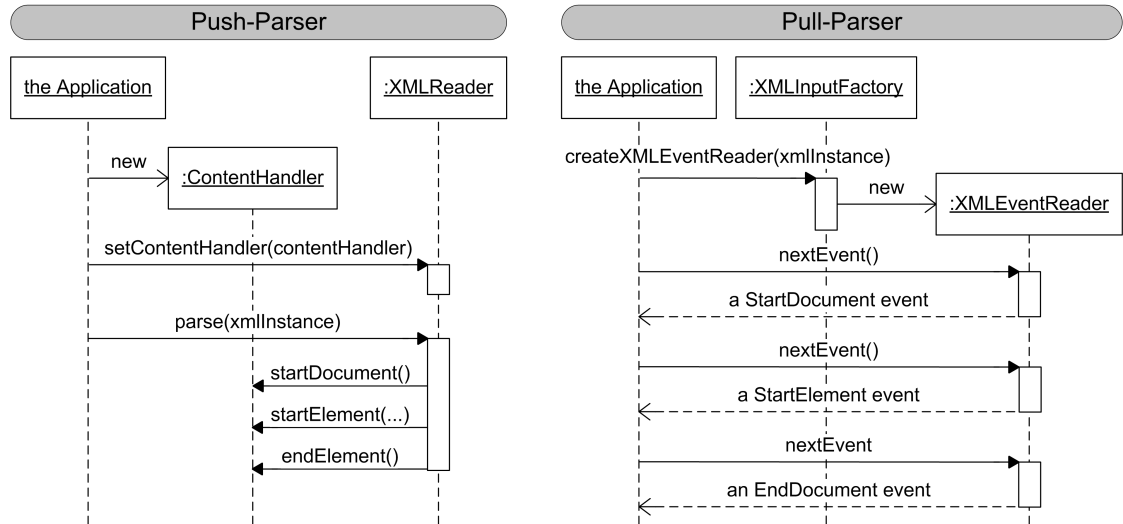


Figure 6.2.: Push- and Pull-Parser

### 6.2.2. Operational Model of the XTL Engine

The denotational semantics of the XTL given in Chapter 4 has been valuable in the implementation of the engine, as a working prototype could easily be compiled. Unfortunately, implementing a Java-based variant of a template engine for XTL from its denotational semantics is non-trivial. It turns out that a Java implementation is more complicated than one would expect from the semantics. Therefore, an operational model has been developed that forms the basis for a maintainable and efficient implementation of the Template Instantiation process. The XTL instructions can easily be implemented on the operations supplied by the operational model.

The abstract machine used by the operational model of the XTL Engine is shown in Figure 6.3. The template engine basically reads from the input XTL document (the template) and writes to the output document (the instantiated template). The component responsible for the evaluation of the instantiation data is called PHP here, for more details on the underlying plugin mechanism refer to Section 6.1.1. To fulfill its task, the engine sets up four helper components:

- a *read window*, which is easing access to the template input stream,
- a *loop stack*, which keeps track of the nesting of loop statements (i.e., `xtl:for-each` instructions) during the instantiation process,
- a *map of macros*, which is used to store macro definitions made by `xtl:macro` statements, and finally
- a number of *PHPs*, maintained in a map indexed by their realm names.

The operational model will be shown using excerpts from the Java implementation of the XTL Engine. First, the four helper components are discussed. Afterwards, the implementations of the particular XTL instructions using the four helper components of the XTL Engine will be described.

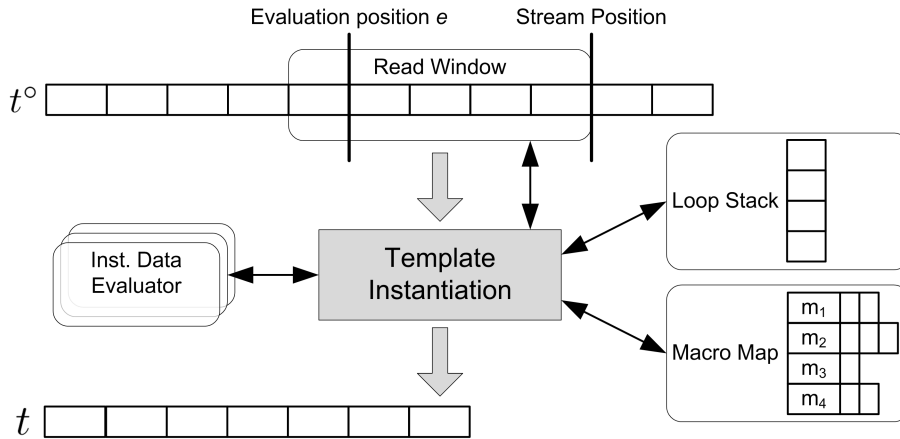


Figure 6.3.: XTL Engine with Input and Output Streams

### Read Window

The read window allows the template engine to look at a certain fragment of the template instead of a particular element. Parts of the template which are located before the content of the read window (in document order) are no longer accessible by the template engine, whereas parts that are behind the content of the read window are not yet read. Thus, the end of the read window marks the current read position within the template input stream. The evaluation position is the location of the template that the engine is currently evaluating and that must necessarily be within the read window.

The read window is, for example, of importance during the processing of `xtl:for-each` statements. As the content of `xtl:for-each` statements can potentially be instantiated several times, it must be kept inside the read window until the `xtl:for-each` has been fully evaluated.

```
public interface ReadWindow
{
    /* Marker management. */
    public void markReadPosition(Object marker);

    public void returnTo(Object marker);

    public void removeMarker(Object marker);

    /* Read and skip. */
}
```

## 6. Flexible, Efficient and Safe Template Instantiation

```
public List<XMLEvent> readUntilBeforeAndSkipOnce()  
    throws XMLStreamException;  
  
public void skipUntilAfter() throws XMLStreamException;  
  
/* Content modification. */  
public void replaceLastRead(List<XMLEvent> events);  
}
```

Listing 6.2: The ReadWindow Interface

The operations shown in Listing 6.2 basically fall in three categories: operations for the management of markers within the template input stream, operations for reading from or skipping in the stream and operations that modify the stream by replacing or inserting `XMLEvents`. A visualization of all operations supplied by the `ReadWindow` implementation is shown in Figure 6.4.

- The `markReadPosition(marker)` operation marks the current position within the read window with some marker object `marker`. The read window ensures by itself that a return to the marked position is always possible, i.e., the start of the read window will not be moved behind the first marked position (in document order).
- The `returnTo(marker)` operation changes the read position within the read window to the read position which has been stored by a preceding call to `markReadPosition` with the same argument `marker`.
- The `removeMarker(marker)` operation removes the marked position associated with the marker object `marker`. If the marked position to be removed is the first marked position in document order, the read window will shrink itself by moving its beginning to the next marked position.
- The `skipUntilAfter()` operation moves the read position within the read window to after the element closing the last read start element. The end of the read window will be extended by the read window itself if the end element is not part of the current read window content.
- The `readUntilBeforeAndSkipOnce()` operation reads from the current read position to the position before the element closing the last read start element and afterwards skips the closing element. The end of the read window will be extended by the read window itself if the end element is not part of the current read window content. The part of the read window content read will be returned.
- The `replaceLastRead(events)` operation removes the last element read from the read window and replaces it with the passed XML events `events`. Furthermore, the read position is set back to the beginning of the events that has been used to replace the last read event.

The read window implementation is designed to keep the content of the read window as small as possible, while still allowing all described operations to be performed at any time. This is described by two contracts. First, the read window will be empty if neither a marker has been set nor an insert operation has filled the read window with some content. Second, all marked positions will always be part of the read window. While the first contract minimizes the memory

used by the read window, the second one guarantees that elements within the read window, which could be required at some time in the future, are still available.

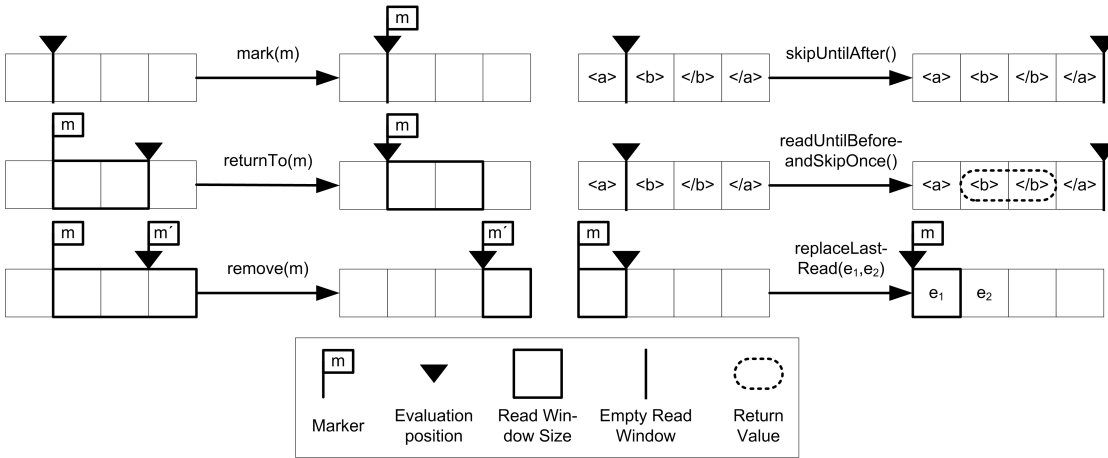


Figure 6.4.: Examples of Read Window Operations' Execution

### Loop Stack

In order to keep track of nested loop statements, the engine maintains a stack of them. Currently, as the XTL language design only supports one type of loop statement, the loop stack is used to keep track of `xtl:for-each` statements only. The topmost stack frame always corresponds to the innermost active `xtl:for-each` statement. In this stack, the execution information for each `xtl:for-each` statement is saved, i.e., the following information is stored per `xtl:for-each`:

- an iterator of objects, which is used to retrieve all values that the control variable of the `xtl:for-each` statement takes during the execution of the `xtl:for-each` loop,
- an object representing the value of the control variable of the surrounding `xtl:for-each` (or null, if there is no such `xtl:for-each` statement),
- the current position of the control variable value within the iterator of all values,
- the Instantiation Data Evaluator which has been used to retrieve the iterator for the control variable values, and finally
- the name of the realm which has been assigned to the `xtl:for-each` statement.

Listing 6.3 shows the operations supplied by the loop stack.

```
public interface LoopStack
{
    /* Loop lifecycle management. */
    public <T> void openLoop(Iterator<T> iterator,
        PlaceholderPlugin<T> php, String realm);

    public void reenterLoop();
}
```

## 6. Flexible, Efficient and Safe Template Instantiation

```
public void closeLoop();

/* Information about the innermost loop. */
public Iterator<?> getActiveIterator();

public int getPosition();

/* Information about an arbitrary loop. */
public Object getControlVariableValue(String realm);

public PlaceholderPlugin<?> getPlaceholderPlugin(String realm);
}
```

Listing 6.3: The LoopStack Interface

As indicated in the listing, three categories of operations exist: first, operations for managing the loop lifecycle, second, operations for giving access to information about the innermost loop and, finally, operations which return information about an arbitrary loop on the stack. The particular operations are described below.

- The `openLoop(iterator, php, realm)` operation adds a new stack frame with the sequence of upcoming control variable values `iterator`, the PHP `php` and the realm name `realm`. The first value of the control variable is taken from `iterator` immediately—thus, the passed iterator must not be empty. The position of the current value of the control variable is initialized to 0.
- The `closeLoop()` operation removes the topmost stack frame from the loop stack.
- The `reenterLoop()` operation updates the topmost stack frame by retrieving a new value for the control variable from the sequence of control variable values passed to `openLoop`. As a side effect, the stored position of the control variable within the iterator of possible control variable values is also increased.
- The `getActiveIterator()` operation returns the iterator which is contained in the topmost entry of the `xtl:for-each` stack. This so-called *active* iterator is the one which should be used to retrieve the next element when the next `xtl:for-each` end element is encountered.
- The `getPosition()` operation returns the position of the current control variable's value within the sequence of control variable values for the innermost loop. Obviously, this always corresponds to the number of completed evaluations of the loop's content.
- The `getControlVariableValue(realm)` operation returns the value of the control variable in the innermost `xtl:for-each` statement which has been assigned to the realm `realm` during a call to the `openLoop` operation. This is needed to supply evaluations of context-dependent XTL statements like `xtl:text` with the current context value as described in Section 4.2.1.
- The `getPlaceholderPlugin(realm)` operation returns the PHP of the topmost stackframe which has been assigned to the realm `realm` during the `openLoop` call. This operation is only used internally for the implementation of the System PHP (see Section 6.1.5).

### Map of Macros

The use of macros in XTL templates involves two instructions: `xtl:macro` and `xtl:call-macro`. While the first instruction assigns a sequence of XTL instructions to a macro name, the latter calls the macro by executing all XTL instructions which have been assigned a particular name by a preceding `xtl:macro` instruction.

```
public interface MacroMap
{
    public List<XMLEvent> get(String name);

    public void put(String name, List<XMLEvent> events);
}
```

Listing 6.4: The MacroMap Interface

In order to implement this transfer of XTL instructions with a name from the `xtl:macro` instructions to corresponding `xtl:call-macro` instructions, the map of macros as shown in Listing 6.4 is used. It supports the following operations:

- The `put(name, events)` operation assigns the name `name` to the sequence of events `events`.
- The `get(name)` operation retrieves the sequence of events which has been assigned for the name `name`.

### Map of PHPs

The engine uses a number of instantiation data evaluators to evaluate `select` attributes. For details of these instantiation data evaluators, see Section 6.1. The selection of an appropriate instantiation data evaluator may depend on `realm` attributes—for details see Section 4.5.1.

```
public interface PlaceholderPluginMap extends
    Iterable<PlaceholderPlugin<?>>
{
    public PlaceholderPlugin<?> get(String realm);
}
```

Listing 6.5: The PlaceholderPluginMap Interface

The `PlaceholderPluginMap` interface shown in Listing 6.5 contains only a single operation. This `get(realm)` operation returns the PHP instance responsible for the realm with the name `realm`.

The operations exposed by the PHPs itself closely correspond to the `IDS` interface which has been defined as part of the denotational semantics of XTL (see Listing 4.2). For a detailed discussion of the PHP operations, refer to Section 6.1.1.

### 6.2.3. Pipeline Implementation of the XTL Engine

The XTL Engine itself is implemented as a pipeline composed of separate processing steps, which will be discussed in the following. The chaining of the particular steps is shown in Figure 6.5. The steps interact by reading or writing `XTLEvents` for its predecessor or to its successor, respectively. The pipeline is driven by the `XMLPipelineDriver`, which merely reads events from its predecessor and writes these events directly to its successor. The actual instantiation is performed by the `XTLProcessingReader`. The other components prepare the event stream (like the `XTLEventReader`), perform optional features of the XTL processing (like the `BypassProcessingReader` or the `IndentingXMLEventWriter`) or solve technical issues (like the `ReassigningAttributesWriter`).

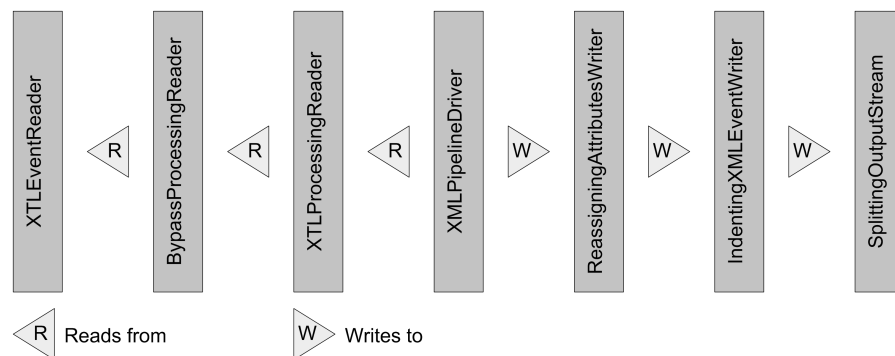


Figure 6.5.: The XTL Engine's Processing Pipeline

A great simplification in the implementation of the XTL Engine has been achieved by introducing `XMLEvent` subclasses for the particular XTL instructions. The object model formed by these subclasses is shown in Figure 6.6. There are three categories of subclasses: representations of start elements of XTL instructions (like `XTLForEachStart`, left column in Figure 6.6), representations of end elements of XTL instructions (like `XTLForEachEnd`, right column) and representations of empty XTL instructions (like `XTLAttribute`, center column). All classes inherit from a common base class `XTLEventImpl`, which implements the `javax.xml.stream.event.XMLEvent` interface and can therefore be handled by `XMLEventReader` and `XMLEventWriter` implementations. This common base class implements basic methods enforced by the `XMLEvent` interface and declares abstract methods like the `process` method which must be implemented by subclasses (see below for details).

For the non-empty XTL instructions, there are additional base classes in the hierarchy inheriting from `XTLEventImpl`: `XTLStartElementImpl` and `XTLEndElementImpl` for start and end elements, respectively. These base classes implement the methods enforced by the `StartElement` and `EndElement` interfaces from the package `javax.xml.stream.events`.

The leaf classes basically implement the `process` method, which is used in the main part in the XTL processing pipeline, the `XTLProcessingReader`. Via this method, the XTL Engine is easily extendable with new XTL instructions: the actual interpretation of the instruction is kept



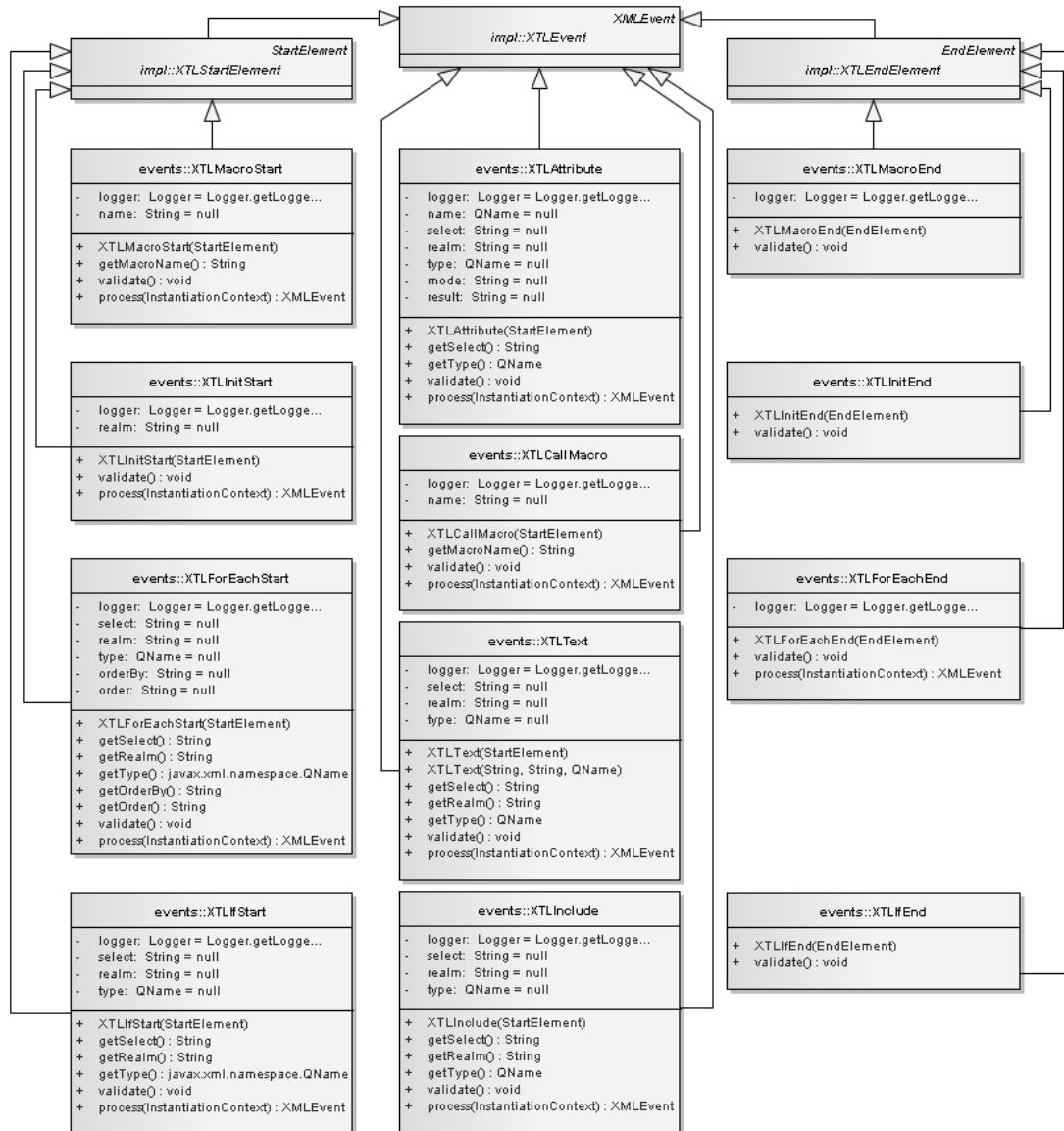


Figure 6.6.: The XMLEvent Hierarchy

## 6. Flexible, Efficient and Safe Template Instantiation

together with its representation in the object model, which makes it unnecessary to change the `XMLProcessingReader` class when new functionalities are added.

It can easily be seen from the operational model and from the description of the particular processing steps below, that the implementation of the XTL Engine, while being possible in a well-structured and easily maintainable design, is much more complex than the implementation of the denotational semantics as a Haskell program. Interestingly, it seems that reimplementing Haskell-based XML processing applications in Java always leads to programs which are larger and very differently structured. This insight also seems to be true for other reimplementations, e.g., for the reimplementation of the RelaxNG validation given by [37]:

The Relax NG derivative algorithm is implemented in a few hundred elegant declarative functional lines of Haskell, and also in tens of thousands of lines and hundreds of classes of highly abstract complex Java code. [89, spelling corrected]

### XTLEventReader

The first step in the instantiation process is the replacement of all StAX XML events that correspond to XTL elements with specific XTL events. The implementation is really simple. During the construction of the `XTLEventReader`, all classes registered as representations of XTL events are retrieved. This is achieved via the service provider mechanism [177, Section “Service Provider”], i.e., by inspecting all files named `/META-INF/services/org.li4x.xtl.engine.XTLEvent` in the classpath. All of these classes are inspected for their constructors via reflection. If a class provides a constructor with a single `javax.xml.stream.events.StartElement` argument, it is considered a representation of an XTL start element. If the class provides a constructor with a single `javax.xml.stream.events.EndElement`, it is considered a representation of an end element, respectively. Classes providing none of these constructors are invalid implementations and are excluded from further processing.

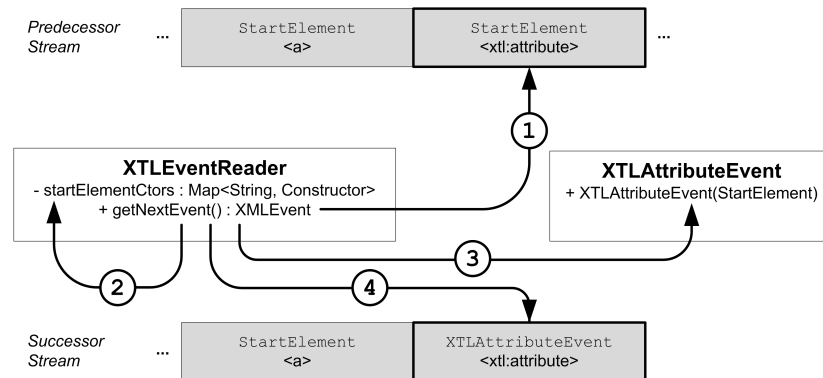


Figure 6.7.: Activities during a Call to `XTLEventReader.getNextEvent`

Each class must denote a local element name by using the Java annotation `org.li4x.xtl.engine.XTLEventDescription`. If no such annotation is attached to the class, the class is considered an illegal implementation and is excluded from further processing. If an

annotation is attached, its `localName` attribute is used to store the constructor in either a start element or an end element constructor map.

During the template instantiation process, the `getNextElement` method prescribed by the `javax.xml.stream.XMLEventReader` interface is invoked. An example of such an invocation is shown in Figure 6.7. The implementation of `getNextElement` first fetches an element from the `XMLEventReader`s predecessor in the processing chain ①. If the element is not from the XTL XML namespace, it is simply returned to the caller. If the element has the XTL namespace URI, its local name is used to look up a constructor from either the start element or the end element constructor map, depending on whether the element is a start or an end element ②. This constructor is then called with the element from the predecessor ③. If the construction succeeds, its result is returned to the caller, i.e., the successor in the processing chain ④. If construction fails, a warning message is emitted, stating that an unprocessable element from the XTL namespace has been encountered and the predecessor is asked for a new element, which is then processed in the described way. This process continues until an element can be returned or the predecessor fails to deliver further elements. In that case an `javax.xml.stream.XMLStreamException` is thrown.

### BypassProcessingReader

The next step in the processing chain is the `BypassProcessingReader`, which is responsible for implementing the bypassing feature as described in Section 4.5.2. An example of the operations executed by the `BypassProcessingReader` is shown in Figure 6.8.

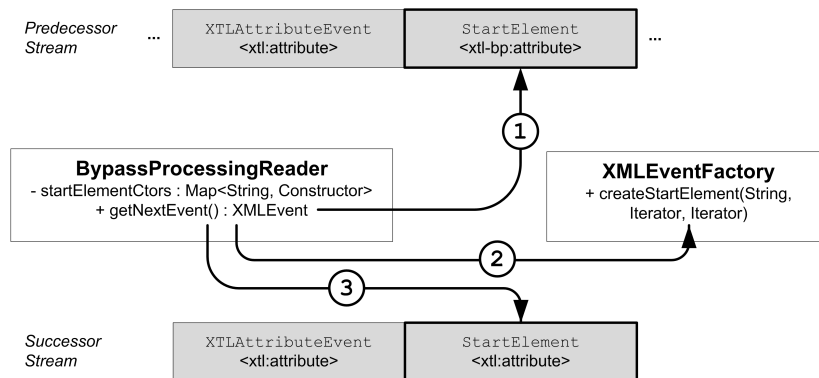


Figure 6.8.: Activities during a Call to `BypassProcessingReader.getNextEvent`

When the `getNextEvent` method of the `BypassProcessingReader` is called, it first fetches an element from its predecessor in the processing chain ①. If the element is not from one of the bypassing namespaces, i.e., if its namespace URI is not `http://research.sap.com/xtl/1.0/bypass/n`, the element is simply returned to the caller. Please note that this is also true for elements from the XTL namespace itself. If the element is from one of the bypassing namespaces, its generation number is decreased. Then, a new start element is created using the `javax.xml.stream.XMLEventFactory` ②. This element is either from the XTL namespace, if the decreased generation number is 0, or from the bypassing namespace

## 6. Flexible, Efficient and Safe Template Instantiation

with the decreased generation number. It is important to note that elements read from the bypassing namespace with the generation number 1 are converted to mere start elements, not to specific XTL events. This prevents the elements from this namespace from being processed in the following processing chain, and this is also the reason why the `XTLEventReader` precedes the `BypassProcessingReader` in the processing chain. Finally, the newly created element is returned to the caller ③.

### XTLProcessingReader

The `XTLProcessingReader` is the core of the XTL engine. This is the part of the XML processing pipeline in which the XTL instructions are actually executed. To grant the components performing the actual processing of XTL events access to the operations of the abstract machine described in Section 6.2.2, the `XTLProcessingReader` implements the `InstantiationContext` interface shown in Listing 6.6.

```
public interface InstantiationContext
{
    public LoopStack getContextStack();

    public MacroMap getMacroMap();

    public PlaceholderPluginMap getPlaceholderPluginMap();

    public ReadWindow getReadWindow();
}
```

Listing 6.6: The `InstantiationContext` Interface

An example of the operation of the `XTLProcessingReader` is shown in Figure 6.9.

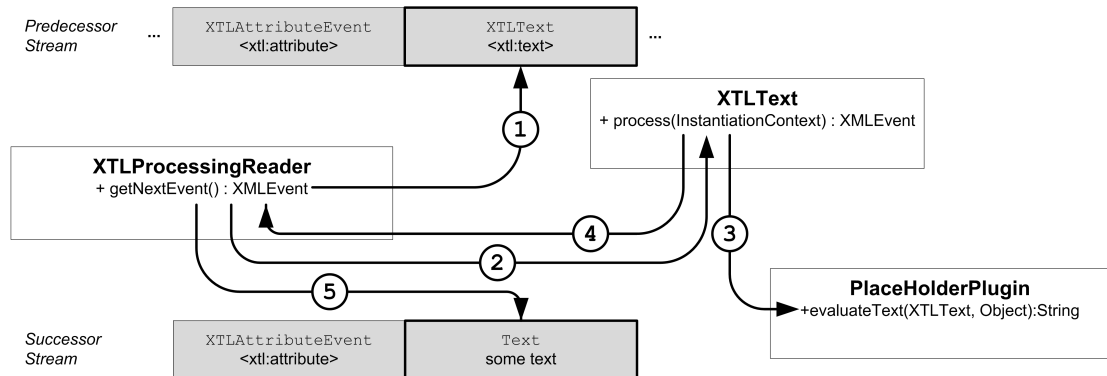


Figure 6.9.: Activities during a Call to `XTLProcessingReader.getNextEvent`

The `XTLProcessingReader`'s `getNextEvent` method operates in a loop. First, it fetches an `XMLEvent` from its predecessor ①. If the fetched event is an instance of `java.xml.`

`stream.events.Comment`, the loop is reentered, i.e., XML comments from the template are omitted from the output. If the fetched event is an XML instruction, its `process(InstantiationContext):XMLEvent` method is called ②. As the `InstantiationContext` parameter, the `XTLProcessingReader` passes itself.

The `process` method of the `XTLText` class invoked by the `XTLProcessingReader`, which is shown in Listing 6.7, uses the passed `InstantiationContext` interface to retrieve the `PlaceholderPlugin` responsible for evaluating the expression in the `select` attribute of the `xtl:text` instruction ③. Afterwards, this PHP's `evaluateText` method is called to actually evaluate the `select` expression ④. The string value returned by the PHP is used to construct an `javax.xml.stream.events.Text` object, which is returned to the `XTLProcessingReader`'s `getNextEvent` method, which returns it to its invoker ⑤.

If the `process` method of the `XTLEvent` would have returned `null`, the `getNextEvent`'s loop would have been reentered.

```
public XMLEvent process(InstantiationContext context) throws
    XMLStreamException
{
    logger.debug("Processing XTLText.");

    PlaceholderPlugin php =
        context.getPlaceholderPluginMap().get(realm);
    String result = php.evaluateText(this,
        context.getContextStack().getControlVariableValue(realm));
    return eventFactory.createCharacters(result);
}
```

Listing 6.7: The `process` Method in `XTLText`

The processing of `xtl:attribute` instructions is very similar to the processing just explained, i.e., the `process` method of `XTLAttribute` is almost identical to the one in `XTLText`.

The implementation of the `process` method in `XTLIfStart` is shown in Listing 6.8. The processing starts similarly to the processing of an `xtl:text` instruction, but the `evaluateIf` method from the PHP returns a boolean value rather than a string. If this boolean is `false`, the processing of the preceding stream is skipped until after the `xtl:if` end element that corresponds to the current start element using the method `skipUntilAfter` from the `ReadWindow` class. The `process` method returns `null` to signal the `XTLProcessingReader` that it has not produced anything that should be part of the instantiated template. The `XTLIfEnd` class itself has an empty `process` method, as no action has to be executed when a closing `xtl:if` element is encountered.

```
public XMLEvent process(InstantiationContext context) throws
    XMLStreamException
{
    logger.debug("Processing XTLIf select='"+select+"'");
```

## 6. Flexible, Efficient and Safe Template Instantiation

```
PlaceholderPlugin php =
    context.getPlaceholderPluginMap().get(realm);
boolean conditionFullfilled = php.evaluateIf(this,
    context.getContextStack().getControlVariableValue(realm));
if (!conditionFullfilled)
{
    // Fast forward.
    context.getReadWindow().skipUntilAfter();
}

return null;
}
```

Listing 6.8: The process Method in XTLLifStart

The implementation of the process method of the `XTLForEachStart` and `XTLForEachEnd` classes are shown in the Figures 6.9 and 6.10, respectively.

An `xtl:for-each` start element is processed as follows. First, a PHP is retrieved using the `realm` attribute value. Then, the method `evaluateForEach` is called to get an iterator over all control values for which the `xtl:for-each` instruction content should be instantiated. If the iterator is empty, the `xtl:for-each` instruction is skipped completely using the `skipUntilAfter` method from the `ReadWindow` class. Otherwise, the current read position is marked with the iterator in the read window using the `markReadPosition` method, and an entry on the `ContextStack` is made. This entry contains the iterator itself, its first element as current value of the control variable, 0 as current position within the `xtl:for-each`, the PHP used to evaluate the `xtl:for-each` and the value of the `realm` attribute. The process method returns null in both cases, as `xtl:for-each` itself does not directly contribute to the instantiated template.

```
public XMLEvent process(InstantiationContext context) throws
XMLStreamException
{
    logger.debug("Processing XTLForEachStart
        select='"+select+"'");

    PlaceholderPlugin php =
        context.getPlaceholderPluginMap().get(realm);
    Iterator<?> it = php.evaluateForEach(this,
        context.getContextStack().getControlVariableValue(realm));
    if (it.hasNext())
    {
        // Mark the current read position.
        context.getReadWindow().markReadPosition(it);

        // Register the execution of a loop on the context stack.
        context.getContextStack().openLoop(it, php, realm);
    }
}
```

```

else
{
    // Fast forward.
    context.getReadWindow().skipUntilAfter();
}

return null;
}

```

Listing 6.9: The process Method in XTLEachStart

If an `xtl:for-each` end element is encountered, its process method first retrieves the iterator of the innermost `xtl:for-each` instruction from the `ContextStack` using its `getActiveIterator` method. The iterator's `hasNext` method is then called to determine whether the `xtl:for-each` instruction's content should be executed once more. If `hasNext` returns false, the topmost entry on the `ContextStack` and the position in the `ReadWindow` marked with the iterator are removed. Otherwise, if `hasNext` returns true, the next value for the control variable is retrieved and the topmost entry on the `ContextStack` is updated. Finally, the read position in the `ReadWindow` is reset to the position marked with the iterator, i.e., to the position directly after the corresponding `xtl:for-each` start element. In either case, the process method returns null.

```

public XMLEvent process(InstantiationContext context) throws
XMLStreamException
{
    logger.debug("Processing XTLEachEnd.");

    // Get the context stack.
    LoopStack contextStack = context.getContextStack();

    // Get the active iterator.
    Iterator<?> it = contextStack.getActiveIterator();

    // Do we have a new context object?
    if (it.hasNext())
    {
        // Reenter the loop.
        contextStack.reenterLoop();

        // Jump to the event following the for-each start.
        context.getReadWindow().returnTo(it);
    }
    else
    {
        // Remove the context.
        contextStack.closeLoop();

        // Allow the read window to compact itself.
    }
}

```

## 6. Flexible, Efficient and Safe Template Instantiation

```
        context.getReadWindow().removeMarker(it);
    }

    return null;
}
```

Listing 6.10: The process Method in `XTLForEachEnd`

The macro handling is also very easy to implement: the processing of the `xtl:macro` start element is shown in Listing 6.11, the process method of the `xtl:macro` end element is empty and the processing of `xtl:call-macro` is shown in Listing 6.12.

The process method of `XTLMacroStart` uses the `readUntilBeforeAndSkipOnce` method to get its content and stores it under the name given by the value of its name attribute in the macro map. Nothing is returned from the process method, since the `xtl:macro` instruction does not directly contribute to the instantiated template.

```
public XMLEvent process(InstantiationContext context) throws
    XMLStreamException
{
    logger.debug("Processing XTLMacroStart name='"+name+"'");

    // Get content of the macro until the closing element.
    List<XMLEvent> content =
        context.getReadWindow().readUntilBeforeAndSkipOnce();

    // Store this macro.
    context.getMacroMap().put(name, content);

    return null;
}
```

Listing 6.11: The process Method in `XTLMacroStart`

When an `xtl:call-macro` instruction is encountered, the macro map is used to retrieve the events stored by an `xtl:macro` instruction with a name attribute of the same value. This content is then used to replace `xtl:call-macro` instructions using the `replace-LastRead()` method of the read window. Nothing is returned from the process method, so the `XTLProcessingReader` is going to fetch the next element from the read window, which will be the first element from the currently inserted macro definition.

```
public XMLEvent process(InstantiationContext context) throws
    XMLStreamException
{
    logger.debug("Processing XTLCallMacro name='"+name+"'");

    // Get the content of the macro definition.
    List<XMLEvent> events = context.getMacroMap().get(name);
}
```



```

        // Replace xtl:call-macro with the macro content.
        context.getReadWindow().replaceLastRead(events);

        return null;
    }

```

Listing 6.12: The process Method in XTLCallMacro

The processing of the `xtl:include` instruction is shown in Listing 6.13. The process method determines the events to be included using the `evaluateInclude` method of the PHP and uses the retrieved events to replace the last read event in the read window. Afterwards, the method returns null to force the `XTLProcessingReader` to fetch the first event from the read window, which will now be the first element evaluated by the PHP.

```

public XMLEvent process(InstantiationContext context) throws
    XMLStreamException
{
    logger.debug("Processing XTLInclude.");

    PlaceholderPlugin php =
        context.getPlaceholderPluginMap().get(realm);

    // Get events to be included.
    List<XMLEvent> events = php.evaluateInclude(this,
        context.getContextStack().getControlVariableValue(realm));

    // Insert ourself with the new events.
    context.getReadWindow().replaceLastRead(events);

    return null;
}

```

Listing 6.13: The process Method in XTLInclude

The processing of `xtl:init` is implemented as shown in Listing 6.14. The process method fetches the content of the `xtl:init` instruction using `readUntilBeforeAndSkipOnce` from the read window and passes the retrieved events to the PHP denoted by its `realm` attribute. As `xtl:init` does not contribute to the instantiated template, the process method returns null.

```

public XMLEvent process(InstantiationContext context) throws
    XMLStreamException
{
    logger.debug("Processing XTLInitStart.");

    PlaceholderPlugin php =
        context.getPlaceholderPluginMap().get(realm);
}

```

```
List<XMLEvent> content =  
    context.getReadWindow().readUntilBeforeAndSkipOnce();  
  
    php.init(content);  
  
    return null;  
}
```

Listing 6.14: The process Method in XTLInit

### **XMLPipelineDriver**

The `XMLPipelineDriver` is a very simple component responsible for driving the XML processing pipeline by reading `XMLEvents` from its predecessor and writing them to its successor. This mechanism is implemented in its `execute` method. As soon as no more events can be read from its predecessor, the `execute` method returns.

### **ReassigningAttributesWriter**

The `ReassigningAttributesWriter` solves a purely technical problem: some StAX implementations cannot handle standalone `java.xml.stream.events.Attribute` instances, i.e., instances which are directly embedded into the StAX event stream. These implementations expect all `Attribute` instances to be assigned to `javax.xml.stream.events.StartElement` instances. Since the processing of `xml:attribute` instructions creates such standalone instances, the `ReassigningAttributesWriter` has been introduced to keep the implementation of the XTL engine's core components as clean and clear as possible.

The `ReassigningAttributesWriter` performs the assignment of standalone `Attribute` instances to their preceding start element. To achieve this, the writing of the last `StartElement` is deferred until a new `StartElement` or an `EndElement` is encountered. During deferral, all `Attribute` instances in the stream are removed from the stream and assigned to the deferred start element.

### **IndentingXMLEventWriter**

In order to produce a human readable XML output, the `IndentingXMLEventWriter` component can be inserted into the processing pipeline. Indentation of XML can be implemented in many ways. In order to keep the memory consumption low, a streaming indentation algorithm has been implemented. The indentation process has been subdivided into two components, the actual `IndentingXMLEventWriter` and an inner class called `PostProcessor`. Therefore, the indentation step in the processing pipeline shown in Figure 6.5 is shown in more detail in Figure 6.10. The `IndentingXMLEventWriter` itself is responsible for keeping track of the element nesting in the `XMLEvent` stream, whereas the `PostProcessor` is responsible for performing the actual indentation (i.e., the indentation on the syntactical level).

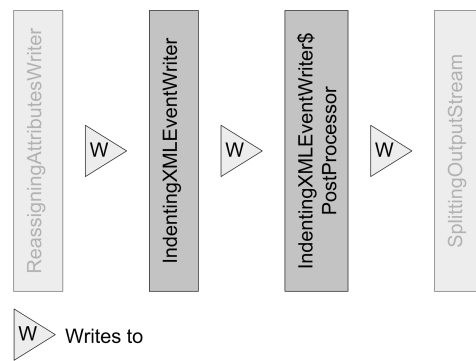


Figure 6.10.: Indentation Parts of the XTL Processing Pipeline

The `PostProcessor` provides the following operations to the `IndentingXMLEventWriter`:

- `increaseIndent` increases the indentation for all following elements by one tabulator.
- `decreaseIndent` decreases the indentation for all following elements by one tabulator.
- `indent` inserts the number of tabulators currently set as a `Text` event into the target `XMLEvent` stream.
- `assertNewLine` ensures that the target event stream is on a newline. If this is not the case, the `PostProcessor` inserts a newline.
- `add` adds a particular `XMLEvent` to the event stream.

The `IndentingXMLEventWriter` obeys the state chart shown in Figure 6.11. The following actions are performed on the transitions between the states:

- ① The only valid event in the initial state is the `StartDocument` event, which causes a transition into the corresponding state. The event itself is added to the `PostProcessor`.
- ② `Text` events occurring in the `START_DOCUMENT` state are ignored.
- ③ A `StartElement` event makes sure that the target stream is on a newline, adds the event to the `PostProcessor` and increases the indent level. Furthermore, a transition to the `START_ELEMENT` state is performed.
- ④ If `Text` events occur in the `START_ELEMENT` state, their content is aggregated in a text buffer.
- ⑤ If another `StartElement` event occurs, the content of the text buffer is flushed (see below). Afterwards, `indent` is called on the `PostProcessor`, the `StartElement` event is added to the `PostProcessor` and the indentation is increased.

## 6. Flexible, Efficient and Safe Template Instantiation

- ⑥ If an `EndElement` event occurs, the content of the text buffer is flushed, the indentation is decreased, `indent` is called and the event is added to the `PostProcessor`. A transition to the `END_ELEMENT` state is performed.
- ⑦ If a `StartElement` occurs in the `END_ELEMENT` state, the text buffer content is flushed, `indent` is called, the event is added to the `PostProcessor` and the indentation is increased. A transition to the `START_ELEMENT` state is performed.
- ⑧ If `Text` events occur in the `END_ELEMENT` state, their content is aggregated in a text buffer.
- ⑨ If another `EndElement` event occurs, the content of the text buffer is flushed. The indentation is decreased, `indent` is called and the event is added to the `PostProcessor`.
- ⑩ If an `EndDocument` event occurs, the event is added to the `PostProcessor`. The final state of the start model has been reached.

All transitions not shown in Figure 6.11 are considered illegal and cause an error message. This also includes events which will never occur in the `XMLEvent` stream originating from the `XTLEventProcessingReader`, like `Comment` or `ProcessingInstruction` events.

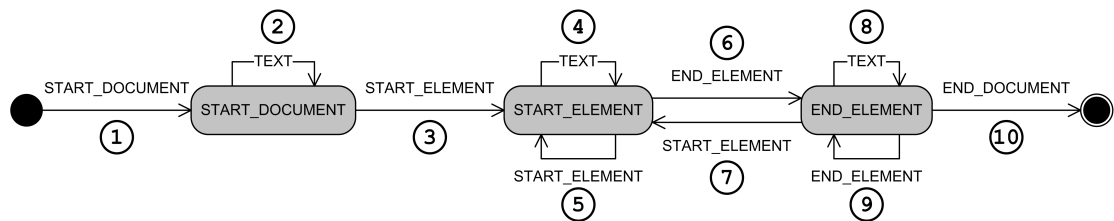


Figure 6.11.: State Chart of the `IndentingXMLEventWriter`

The treatment of the content of the text buffer during flushing to the successor `XMLEventWriter`, which occurs in the transitions ⑤, ⑥, ⑦ and ⑨, depends on the text which has been collected in the text buffer. If the text to be flushed consists of a single line and is not longer than a configurable size, it is emitted as is. This results in an XML snippet like the following:

```
<a>text</a>
```

If the text is longer or contains newlines, it is tokenized and its parts are emitted with indentation, resulting in XML like the following:

```
<a>
  text
  more text
</a>
```

### SplittingOutputStream

A special feature of the XTL Engine is its capability to split the instantiation result into multiple result documents. This feature roughly corresponds to the `result-document` instruction of XSL-T 2.0. Syntactically, this feature is based on an XML Schema which defines elements that allow bundling multiple XML documents into one. An example document is shown in Listing 6.15.

```
<?xml version="1.0" encoding="UTF-8"?>
<s:files xmlns:s="http://research.sap.com/xtl/splitting">
  <s:file name="simple1_1.xml">
    <a>This is simple1_1.xml!</a>
  </s:file>
  <s:file name="simple1_2.xml" encoding="iso-8859-1">
    <a>This is simple1_2.xml!</a>
  </s:file>
  <s:file name="simple1_3.xml" encoding="utf-8">
    <a>This is simple1_3.xml!</a>
  </s:file>
  <s:file name="simple1/simple1_4.xml">
    <a>This is simple1/simple1_4.xml!</a>
  </s:file>
</s:files>
```

Listing 6.15: A Template Instantiation Result before Splitting

The root element of a template instantiation result which should be split into multiple files must be `files` from the splitting namespace (here prefixed with `s`, the namespace URI is `http://research.sap.com/xtl/splitting`). Within this root element, multiple `s:file` elements are allowed. Each `s:file` element must have a `name` attribute that declares the file name into which all content parented by this `s:file` element should be written. The `s:file` element can also carry an `encoding` attribute which sets the encoding of this particular file to be generated.

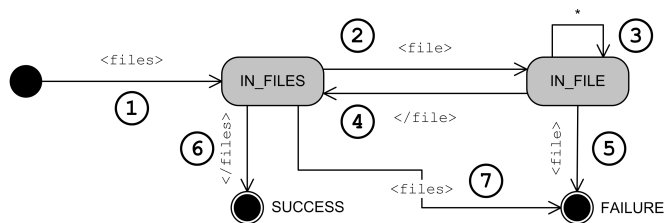


Figure 6.12.: State Chart of the SplittingOutputStream

The implementation of the `SplittingOutputStream` is simple and follows the state chart shown in Figure 6.12. From the initial state, the only valid transition is to the `IN_FILES` state via an `s:files` element ①. From the `IN_FILES` state, an `s:file` element causes the creation of a file and a transition into the `IN_FILE` state ②, in which basically all occurring elements are written to the file created ③. The `IN_FILE` is only left when a closing `s:file` or `s:files` tag is encountered. The first case causes a transition back to the `IN_FILES` state ④,

## 6. Flexible, Efficient and Safe Template Instantiation

whereas the latter case is obviously an error, which causes the `SplittingOutputStream` to enter a final `ERROR` state ⑤. From the `IN_FILES` state, an occurring closing `s:files` tag causes the transition to the final `SUCCESS` state ⑥, whereas a closing `s:file` is an error and leads to the final `ERROR` state ⑦.

### 6.2.4. Memory and Runtime Complexity

From the operational view on the XTL Engine, the limits for the memory and time consumption shown below can be derived. The underlined parts of the equations are to be interpreted as XPath expressions.

The memory consumption limit of the XTL Engine is determined by the size of the read window. As explained above, the read window must be capable of holding the content of the largest `xtl:for-each` statement of a template, i.e., the memory is limited by the maximum number of nodes contained in an `xtl:for-each` statement in the template  $t^\circ$ .

$$memory(t^\circ) = O \left( \max_{x \in t^\circ // \underline{xtl:for-each}} |x // \underline{node()}| \right)$$

A limit for the time consumption can only be given by abstracting from the time needed for the evaluation of the expressions from the query language. Under this restriction, the time needed for the instantiation of the template is linear in the size of the instantiated template. Obviously, the size of the instantiated template can not be estimated from the template, as this size depends on the result of the evaluation of the instantiation data.

$$time(t^\circ) = O \left( |instantiate(t^\circ) // \underline{node()}| \right)$$

Measurements of the implementation show the correctness of these estimations, for details see Section 7.5.

### 6.3. Instantiation Data Validation

The *instantiation data validation* process is responsible for the validation of the instantiation data. The instantiation data validator component verifies the data supplied by the instantiation data evaluator against the constraints determined during the Constraint Separation process. Non-compliance of the instantiation data has to be reported by this component.

As Figure 3.5 shows, the instantiation data validator gets activated during the instantiation phase. This means that no corrective actions can be taken anymore if a problem is detected with respect to the instantiation data. The component is beneficial nonetheless, because it is able to deliver the exact reason why the instantiated template will not comply to the target language. Furthermore, the error is detected within the application which incorporates the template engine, and not, as for example in a classical Web application, in some user's browser (see Figure 1.1). Thus, validating the instantiation data constraints contributes to the safe instantiation goal.

### 6.3.1. The IDC PHP

The process of validating the particular instantiation data constraints is simple. The constraints are simply taken from the template into which they have been augmented by the Template Validation process as shown in Section 5.2. Afterwards, the values returned by the instantiation data validation process are validated against these constraints. This process is implemented by the IDC PHP, which is a PHP that wraps another PHP and validates its return values as shown in Figure 6.13 (see Section 6.2.3 for the process without involvement of the IDC PHP). The IDC PHP is a decorator [73]. If multiple PHPs are in use during XTL instantiation (see Section 4.5.1 for details), each PHP is wrapped with its own IDC PHP.

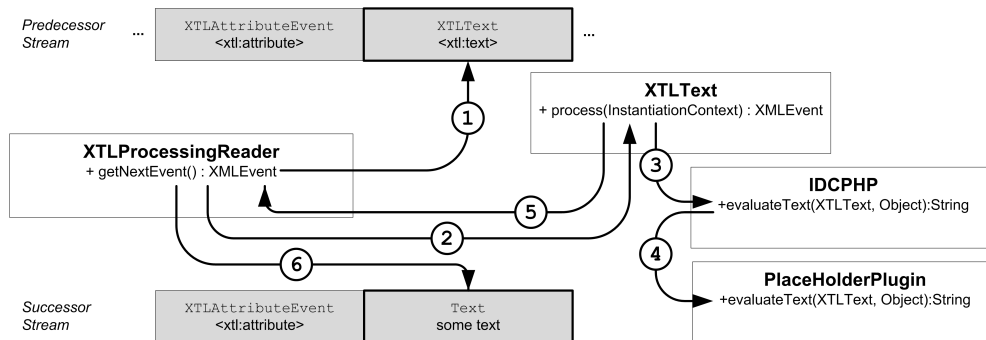


Figure 6.13.: XTL Instantiation with enabled Instantiation Data Validation

When the `XtlProcessingReader` encounters an `xtl:text` instruction ①, it calls the `process` method of the `XtlText` event implementation ②. This implementation now no longer retrieves the instantiation data directly from the responsible PHP (as in Figure 6.9), but rather calls the `evaluateText` method of the IDC PHP to evaluate it ③. The IDC PHP in turn calls the `evaluateText` method of the responsible PHP ④ to actually get the instantiation data value, which it validates using the `type` attribute augmented to the `xtl:text` instruction. If the instantiation data is valid with respect to the instantiation data constraint, the processing method returns ⑤ and the `XtlProcessingReader` creates a text element ⑥. If the instantiation data is not valid, the instantiation fails, thereby producing an error message telling which instantiation data constraint has been validated.

The actual processing of the value returned by the wrapped PHP depends on the affected XTL instruction:

1. If an `xtl:text` or `xtl:attribute` instruction is processed, the instantiation data is validated against the XML Schema simple type denoted by the `type` attribute of the XTL instruction. This validation is performed using the `validate` method from the corresponding `org.apache.xerces.impl.dv.XSSimpleType` instance, which is retrieved based on the `type` attribute.
2. If an `xtl:if` instruction is processed, no validation is performed as the PHP interface already restricts the return value to be of boolean type (see Listing 6.1), which makes it impossible to retrieve an invalid instantiation data value here.

## 6. Flexible, Efficient and Safe Template Instantiation

3. If an `xtl:for-each` instruction is processed, the wrapped PHP returns an iterator over elements of some type determined by a generic parameter of the PHP interface. It must be checked that this iterator returns a number of elements between the values of the `min` and the `max` attribute of the `xtl:for-each` instruction (the latter value may be unbounded, which makes the interval of allowed values for the number left-bound). As the retrieval of elements from the iterator does not take place in the PHP itself, but rather in the `process` methods of the `XTLForEachStart` and `XTLForEachEnd` event classes, the iterator is wrapped in a `org.linux.xml.util.SizeCheckingIterator`. This decorator class just delegates calls to its `next` and `hasNext` methods to the decorated iterator, and counts the number of elements already retrieved and adds an additional check that is executed when the `hasNext` method of the decorated iterator returns `false` for the first time. The check ensures that the number of elements retrieved from the decorated iterator is within the interval specified by the `min` and `max` attributes of the `xtl:for-each` instruction.

### 6.3.2. Template Interface Generation

There is an alternative approach to ensure that the instantiation data satisfies its constraints. Assuming that the frequency of template modifications is low (or that the modifications are of a special type, see below), an interface to the template that asserts the instantiation data constraints can be generated. This interface ensures the instantiation data constraints by mapping them to the type system of the language that is using the template engine.

This technique is called *Template Interface Generation* and slightly changes the template technique as proposed in Figure 3.5. The changed architecture is shown in Figure 6.14. The difference to the previously proposed architecture is the extension of the adaptation phase: the adaptation of the template technique now also includes an adaptation to the authored template. After the template has been authored and validated, it is compiled in the Template Interface Generation step.

This step yields a template interface that fulfills the functions of both instantiation data evaluation *and* instantiation data validation. For that reason, the template interface is connected with the Template Instantiation bidirectionally. Strictly speaking, the generated template interface appears in two life cycle phases. It is generated in the adaptation phase and used in the instantiation phase, which is indicated in Figure 6.14 by the bicolour box used for it. Furthermore, it is important to note that the aforementioned adaptation phase is different from the one introduced in Section 2.1.2, since it is adapting the template engine to a particular template rather than to a particular target language.

Template Interface Generation combines the best features of both XML binding tools and template techniques. XML binding tools guarantee that a generated document complies to a given schema by translating the constraints contained in the schema to the type system of the programming language using the XML binding tool (see Section 2.3.3 and [155]). This principle has also been called *intra-level transformation* between technological spaces [113]. The disadvantage of XML binding tools is that the entire document must be created in the host language. On the other hand, template techniques offer an easy way to generate a document only partially using a programming language: the remainder of the target document is literally contained in



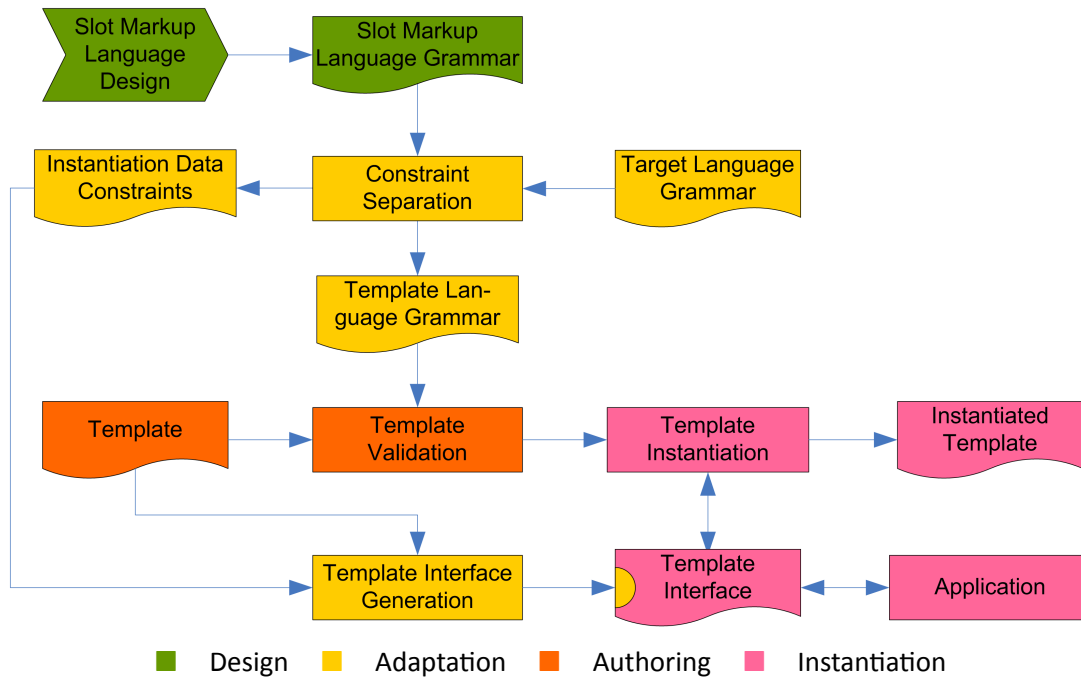


Figure 6.14.: Architecture with Template Interface Generation

the template. Template Interface Generation allows to generate target language documents partially using a programming language, partially from a template using a generated API which ensures the validity of the instantiation data.

A difference between the originally proposed architecture in Figure 3.5 and the architecture in Figure 6.14 is the missing instantiation data source component in the latter architecture. It has been replaced by the application employing the template engine. The application is connected bidirectionally to the template interface and the directions correspond to the push and the pull strategy introduced in Section 2.5.4.

If the pull strategy is used, Template Interface Generation just generates an interface which must be implemented by the application that is using the template. In this case, the instantiation data is queried from the application when it is needed. On the other hand, if the push strategy is used, the Template Interface Generation process will generate a data model which is instantiated afterwards and populated by the application, and that is passed to the template engine upon invocation. Obviously, the data model corresponds to the Move Copy of Data pattern also mentioned in Section 6.1.1.

### 6.3.2.1. Introductory Example

As an introductory example, consider the XTL template shown in Listing 6.16. The template is obviously intended to generate an XHTML document. The template is augmented with instantiation data constraints, e.g., with `type` attributes at the `xtl:text` instructions. The Template

## 6. Flexible, Efficient and Safe Template Instantiation

Interface Generation process interprets the `select` attributes in this listing as XPath expressions and tries to build an object model which can be accessed using these `select` attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<html
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xtl="http://research.sap.com/xtl/1.0"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
>
  <body>
    <ul>
      <xtl:for-each select="books" min="1" max="unbounded">
        <xtl:for-each select="authors" min="1" max="unbounded">
          <xtl:text select="name" type="xsd:string"/>
        </xtl:for-each>
        <xtl:text select="name" type="xsd:string"/>
        (publ. <xtl:text select="publicationDate" type="xsd:date"/>)
        <xtl:if select="instock">Buy</xtl:if>
      </xtl:for-each>
    </ul>
  </body>
</html>
```

Listing 6.16: Example Template for Template Interface Generation

As the root for the XPath expressions within the documents, the root class of the object model to be generated will be used. As this usage is implicit (as opposed to being declared using some XPath expression), a name for the root class can not be deduced from the XTL template. As a convention, the Template Interface Generation process names the root class `ObjectModelRoot`.

Since the outermost `xtl:for-each` instruction has a `select` attribute with a value of `books`, the `ObjectModelRoot` class must have a property named `books`. As the multiplicity of the `xtl:for-each` instruction is restricted by `min` and `max` attributes deduced during the Template Validation process, the `books` property must have a multiplicity of  $1 \dots n$ . There is no `type` attribute at the `xtl:for-each` instruction, since this instruction merely changes the context of the PHP during the Template Instantiation. The Template Interface Generation process therefore generates a new class `Book` for this property—the name is put into singular in order to make the object model more human readable.

The `Book` class is used as the context for the `xtl:text` instructions with the `select` attribute values `name` and `publicationDate`. Therefore, the `Book` class has properties with these names and the types `String` and `Date`, respectively. The Java type is deduced from the XML Schema type in the same way JAXB maps XML Schema types to Java types (see below for the details). Additionally, there is an `xtl:if` instruction with a `select` attribute with the value `instock`, which is turned into a boolean property of the same name at the `Book` class. Finally, the outermost `xtl:for-each` statement contains a further `xtl:for-each` statement carrying a `select` attribute with the value `authors`. Analogously to what has been

described above, this statement adds an `authors` property to the `Books` class typed by the newly introduced `Author` class.

The `xtl:for-each` statement referring to the `authors` property of the `Book` class contains only a single `xtl:text` instruction, which causes the Template Interface Generation process to add a `name` property to the `Author` class. This attribute has the type `String`. The overall object model which can be deduced from the template is shown in Figure 6.15.

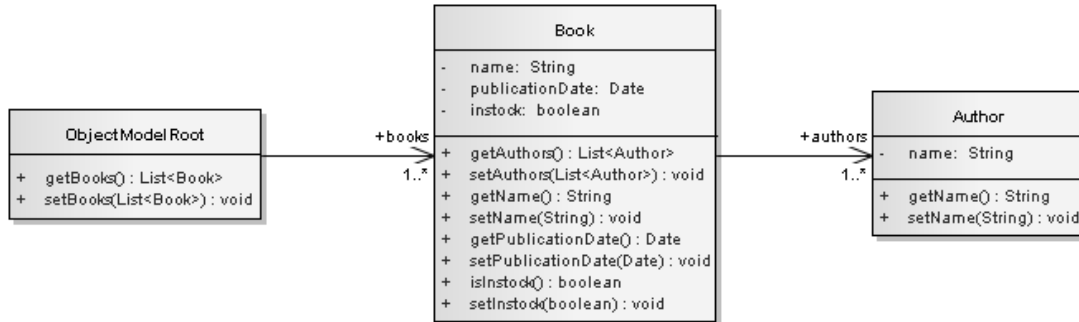


Figure 6.15.: The Object Model Deduced from the Template in Listing 6.16

It is important to note that the Instantiation Data Type Safety can not only be guaranteed by the Java types of the properties within the generated object model. The problem is that the XML Schema type model is much more fine-grained than the Java type model, e.g., there is no exact equivalent of the `xsd:nonNegativeInteger` type. The solution is to select the Java type which has the smallest value range including all the values from the XML Schema type, and to check the value for its correctness with respect to the XML Schema type either when the value changes (i.e., within the `set` method) or when the object model is passed into the Template Instantiation process.

### 6.3.2.2. An Algorithm for the Template Interface Generation

It is important to note that the Template Interface Generation process restricts the syntax of XPath expressions used within the template. For example, the XPath axis `descendant` can not be allowed in the XPath expressions in a template for which an interface should be generated, since this axis allows an XPath to evaluate to a node arbitrarily deep within the context node. There is no equivalent to such an arbitrary descent within an object model.

The syntax accepted by the Template Interface Generation is shown in Figure 6.16. The figure contains a subset of the syntactical productions in the XPath specification [38]. The first column in the figure contains the number of the production rule within the specification. On the right hand side of the rules, terminal and non-terminal symbols have been greyed out if they are not allowed within the XPath expressions. If a non-terminal has been greyed out, its corresponding production rule has been omitted. The rule has also been omitted if its left-hand side is not from the XPath specification—this is the case for the `QName` non-terminal symbol, which is actually defined in [29].

## 6. Flexible, Efficient and Safe Template Instantiation

[1]	LocationPath	::=	RelativeLocationPath   AbsoluteLocationPath
[2]	AbsoluteLocationPath	::=	'/' RelativeLocationPath?   AbbrAbsoluteLocationPath
[3]	RelativeLocationPath	::=	Step   RelativeLocationPath '/' Step   AbbrRelativeLocationPath
[4]	Step	::=	AxisSpecifier NodeTest Predicate*   AbbrStep
[5]	AxisSpecifier	::=	AxisName '::'   AbbrAxisSpecifier
[6]	AxisName	::=	'ancestor'   'ancestor-or-self'   'attribute'   'child'   'descendant'   'descendant-or-self'   'following'   'following-sibling'   'namespace'   'parent'   'preceding'   'preceding-sibling'   'self'
[7]	NodeTest	::=	NameTest   NodeType '(' ')'   'processing-instruction' '(' Literal ')'
[12]	AbbrStep	::=	'.'   '..'
[13]	AbbrAxisSpecifier	::=	'@'?
[37]	NameTest	::=	'*'   NCName ':' '*'   QName

Figure 6.16.: The XPath Syntax Accepted by the Template Interface Generation Process

There are several reasons why a part of the right-hand side of a production rule has been greyed out. First of all, all axes that allow an arbitrary depth selection within an XML document have been omitted (*ancestor*, *ancestor-or-self*, *descendant*, *descendant-or-self*, *preceding*, *following*). The *namespace* axis has been disallowed as its use does not make any sense when accessing an object model. Furthermore, the use of predicates to further classify the selected nodes has been disallowed in order to simplify the actual implementation of the Template Interface Generation algorithm, although it would be possible to include predicates in the allowed XPath subset. The abbreviated location paths have been disallowed since they are referring to the disallowed *descendant-or-self* axis. Finally, node tests have been restricted to name tests, since this is the only node test applicable to an object model.

The Template Interface Generation also slightly changes the semantics typically assigned to these expressions. Typically, a conversion from the node selected by an XPath expression depending on the context of the expression is applied. An `xsl:if` instruction will convert a `null` value returned by an XPath expression into the boolean value `false`, and an arbitrary non-null value into the boolean value `true`. Adapting the Template Interface Generation process to this behaviour would mean that no type information could be deduced from `xsl:if` statements at all, which would complicate the type deduction during the generation process.

In order to understand how an XML Schema type is mapped to a Java type, it is necessary to look at the type mapping defined in [105]. The mapping used during the Template Interface Generation process is analogous, with the exception that the `xsd:IDREF` type is also mapped to the Java type `java.lang.String`, i.e., the `IDREF` value is represented textually within

the document model (as it would within the XML document representing the document model), and not by resolving the referenced object (which would require the type to be mapped to the `java.lang.Object` type).

The Template Interface Generation algorithm basically tries to construct an object model from the set of XPath expressions contained in an XTL template. As an intermediate step, an in-memory model similar to a UML model is constructed (which is later used to generate the actual template interface code using an M2C transformation). The intermediate model is a tree of instances of a data structure called *property descriptor*. A property descriptor has the following properties:

- A *parent*, which may also be absent.
- A list of *children*. Each of the child property descriptors of property descriptor *A* has *A* as its parent.
- A *name*, which is a String.
- A *type*, which is a QName. The type may also be absent.
- A *minimum* and *maximum* cardinality. The latter one may also take a value of  $-1$  to denote unrestricted cardinality.

The construction of the property descriptor tree works as follows. A *current* and a *root property descriptor* are maintained. Initially, they are both initialized with a single property descriptor that forms the root of the property descriptor tree (and, thus, is the only property descriptor without parent).

If an XPath expression complying to the syntax shown in Figure 6.16 is given, a property descriptor is retrieved using the current and root property descriptor as described below. In order to compute the result, a result property descriptor is introduced, which may be reassigned during the process.

1. The XPath expression is interpreted using rule [1] from Figure 6.16. If it is an absolute location, the root property descriptor is used to initialize the result property descriptor, otherwise, the current property descriptor is used as initialization value.
2. For each abbreviated step (AbbrStep) within RelativeLocationPath (see rules [3], [4], [12]), the result property descriptor is either unchanged (for the abbreviated step `.`) or the parent of the current property descriptor is assigned as its new value (for the abbreviated step `..`). It is an error if this parent property result descriptor is absent.
3. For a non-abbreviated step, i.e., a combination of an AxisName or an AbbrAxisSpecifier with a QName (see rules [4]...[7], [37]), the following happens, depending on the type of the axis:
  - a) If the axis is the `attribute` axis (which may also be denoted using the AbbrAxisSpecifier `@`), the QName is used to search for an equally named property descriptor within the children of the current result property descriptor. If such a property descriptor exists, it becomes the new result property descriptor. It is an error if the property descriptor found has an absent type. If no such property descriptor exists, a new one is created, a parent-child relationship between it and the current property descriptor is created and the newly created property descriptor becomes the new result property descriptor.

## 6. Flexible, Efficient and Safe Template Instantiation

- b) If the axis is the `child` axis, the calculation of the result property descriptor happens exactly as with an `attribute` axis, with the only difference that it is an error if an existing property descriptor has a non-null type.
- c) If the axis is `self`, `following-sibling` or `preceding-sibling`, the result property descriptor is unchanged. However, it is an error if the current result property descriptor has a name distinct from the QName in the NameTest.
- d) If the axis is `parent`, the result property descriptor is assigned to the parent of the current property descriptor. It is an error if the result property descriptor becomes null during this operation or if the new property descriptor has a name distinct from the QName in the NameTest.

The XTL document is then processed in its document order. Non-XTL parts of the template are ignored. If an XTL instruction is encountered, it is interpreted depending on its type as described below:

1. If an `xtl:for-each` instruction is encountered, a property descriptor is retrieved using the XPath expression from the XTL instruction with the algorithm described above. It is an error if the retrieved property descriptor has a non-null type. If the property descriptor has just been created by the retrieval algorithm described above, the last NameTest is assigned to its name attribute, and the `type`, `min` and `max` attributes are transferred to its respective properties. Otherwise, the following operations are performed:
  - a) The minimum cardinality of the property descriptor is set to be the maximum of its previous value and the `min` attribute value of the `xtl:for-each` instruction.
  - b) The maximum cardinality of the property descriptor is set to be the minimum of its previous value and the `max` attribute value of the `xtl:for-each` instruction.
  - c) The previous value of the type attribute of the property descriptor and the type denoted by the `type` attribute of the `xtl:for-each` instruction are compared with regard to their lexical spaces (in the sense of [26]). It is an error if these lexical spaces are incomparable (i.e., when denoting the spaces with  $A$  and  $B$ ,  $A \setminus B \neq \emptyset \wedge B \setminus A \neq \emptyset$  is true). If the lexical spaces are comparable, the type with the smaller lexical space is stored as the type value of the property descriptor.Finally, the newly retrieved property descriptor is set to be the new current property descriptor. It is an error if the minimum cardinality of this property descriptor is greater than its maximum cardinality (this is the case if the template declares two cardinality intervals  $a \dots b$  and  $c \dots d$  with either  $b < c$  or  $d < a$ , i.e., two non-overlapping intervals).
2. If an `xtl:if` instruction is encountered, a property descriptor is retrieved using the XPath expression from the XTL instruction with the algorithm described above. If the property descriptor has been newly created, it is assigned the type `xsd:boolean`. If an existing property descriptor has been retrieved, it is an error if the retrieved property descriptor has a type different from `xsd:boolean`.

3. If either an `xtl:attribute` or an `xtl:text` instruction is encountered, a property descriptor is retrieved using the XPath expression from the XTL instruction with the algorithm described above. If the property descriptor has just been created by the retrieval algorithm, its name attribute is set to the latest NameTest in the XPath expression of the `select` attribute and its type attribute is set to the value of the `type` attribute of the XTL instruction. If the property descriptor is not newly created, it is an error if the common super class of the Java mappings of both types is not equal to one of the mapped classes (as with the `xtl:for-each` instruction, see above). The most specific type in the sense of *derivation by restriction* [180] is stored as the type value of the property descriptor.

If the algorithm described above is applied to the template in Listing 6.16, the property descriptor tree shown in Figure 6.17 is created.

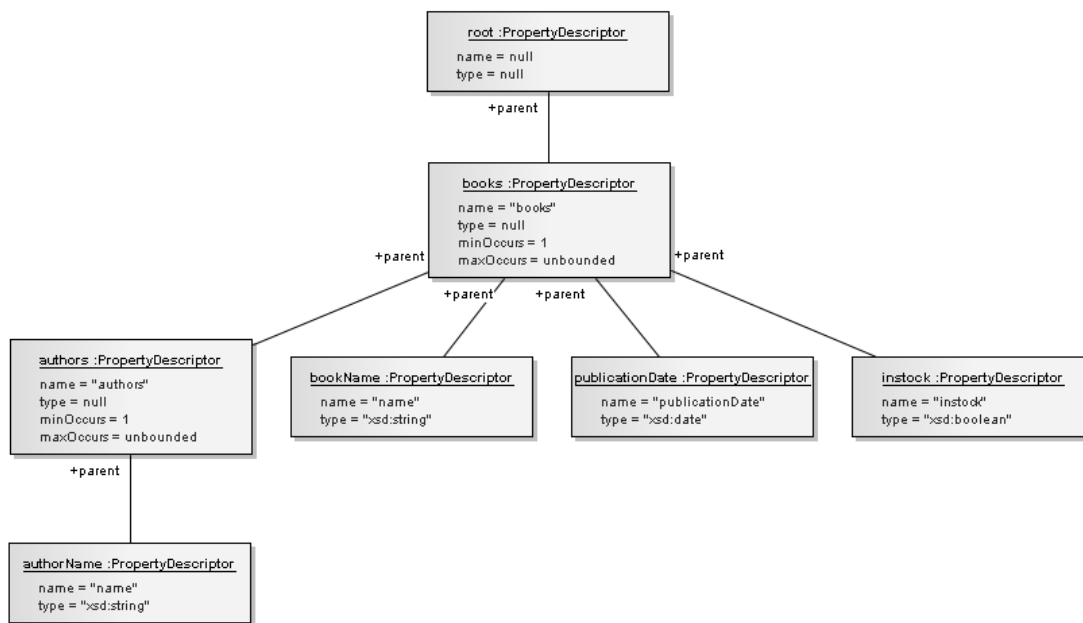


Figure 6.17.: The Tree of Property Descriptors Built from the Template Shown in Listing 6.16

This property descriptor tree can easily be transformed into a Java object model using the following mapping:

1. For property descriptors with an absent type, a class is created with the name determined by the name value of the property descriptor. If there is no such name, i.e., if the property descriptor is the root property descriptor, the class is named `ObjectModelRoot`. The name (as described in Section 6.3.2.1) is chosen as a convention, since the template itself contains no information about the naming of this root class.
2. For each of the children of a property descriptor, a property at the generated class is added. If the child property descriptor has a multiplicity, a collection class is used to

## 6. Flexible, Efficient and Safe Template Instantiation

type the property; otherwise, the value of the type attribute of the property descriptor is mapped to a Java type which is then used to type the property.

This simple mapping schema transforms the property descriptor tree in Figure 6.17 into the Java object model shown in Figure 6.15. Obviously, the type safety given by the Java object model does not completely ensure that the instantiation data constraints are fulfilled. There are two reasons for this. First, there are no exact counterparts for some of the XML Schema types. For example, there is no Java type with exactly the value range provided by `xsd:nonNegativeInteger`. Second, the cardinalities of the collections determined from the template are not checked by the Java collection types. In order to close these gaps and to enforce the instantiation data constraints, an additional `validate` method is generated for all classes within the object model. This `validate` method ensures the correct types and cardinalities of all property values within the object itself and subsequently calls the `validate` method on all children within the object model. The `validate` method must be called by the template engine before starting with the actual instantiation process.

### 6.3.2.3. Implementation using a PHP and an API-based Generator

The implementation of the Template Interface Generation process is straightforward and consists of two steps: the analysis step, which constructs the property descriptor tree, and the generation step, in which the property descriptor tree is transformed into a Java object model.

The analysis step has been implemented using a PHP named `org.lixlil.xtl.compiler.template.AnalyzerPHP`. This approach makes it easy to iterate over the template and to react to the embedded XTL instructions. The property descriptor implementation is the class `org.lixlil.xtl.compiler.template.PropertyDescriptor`. This is also the actual value for the `Type` type parameter of the `PlaceholderPlugin` interface. The context mechanism of the PHP is used to keep track of the current property descriptor (in the sense described above), whereas the root property descriptor is kept in a private field `root` in the PHP.

The algorithm for retrieving a property descriptor from an XPath expression is implemented in the method `retrievePropertyDescriptor` as shown in Listing 6.17. The method first determines whether to use the root or the current property descriptor as the base for the retrieval of the new property descriptor. Afterwards, the XPath expression is parsed and the method iterates over its steps. It is checked that the steps do not contain predicates, since the use of predicates has been excluded from the accepted XPath syntax (see Figure 6.16), if the check fails, an exception is thrown. Depending on the axis referenced in the step, a new property descriptor is retrieved. In case a forbidden axis or a forbidden node test is encountered, the method also throws an exception. If all steps are processed, the resulting property descriptor is returned.

```
private PropertyDescriptor retrievePropertyDescriptor(PropertyDescriptor current,
    String xpathExpression)
    throws XPathExpressionException
{
    PropertyDescriptor result;
    LocationPath locationPath = parse(xpathExpression);
```



```

result = locationPath.isAbsolute() || current == null ? root : current;
for (Step step : locationPath.getSteps())
{
    assertTrue(step.getPredicates().length==0, "Predicated are not allowed.");

    switch (step.getAxis())
    {
        case Compiler.AXIS_ANCESTOR:
        case Compiler.AXIS_ANCESTOR_OR_SELF:
        case Compiler.AXIS_DESCENDANT:
        case Compiler.AXIS_DESCENDANT_OR_SELF:
        case Compiler.AXIS_FOLLOWING:
        case Compiler.AXIS_NAMESPACE:
        case Compiler.AXIS_PRECEDING:
            String axisName = Step.axisToString(step.getAxis());
            fail("Use of axis '"+axisName+"' not allowed.");
            break;
        case Compiler.AXIS_ATTRIBUTE:
        case Compiler.AXIS_CHILD:
            assertTrue(step.getNodeTest() instanceof NodeNameTest,
                "Only NameTests are allowed as NodeTests.");
            NodeNameTest nodeNameTest = (NodeNameTest)step.getNodeTest();
            assertTrue(nodeNameTest.getNodeName().getPrefix() == null,
                "Prefixes are forbidden.");
            assertFalse(nodeNameTest.isWildcard(),
                "The '*' NameTest is not an allowed.");
            String name = nodeNameTest.getNodeName().getName();
            PropertyDescriptor newResult = result.getChild(name);
            if (newResult == null)
            {
                newResult = new PropertyDescriptor(name, result);
            }
            result = newResult;
            break;
        case Compiler.AXIS_PARENT:
            assertFalse(result.isRoot(),
                "Use of parent axis at root node is forbidden.");
            result = result.getParent();
            break;
        case Compiler.AXIS_SELF:
        case Compiler.AXIS_FOLLOWING_SIBLING:
        case Compiler.AXIS_PRECEDING_SIBLING:
            // Do nothing.
            break;
    }
}

return result;
}

```

Listing 6.17: The retrievePropertyDescriptor Method in the AnalyzerPHP

The `retrievePropertyDescriptor` method is called by the implementations of the various `evaluate` methods of the PHP. As an example, the implementation of the `evaluateForEach` method is shown in Listing 6.18. The method uses the `retrievePropertyDescriptor` method to get a property descriptor and afterwards transfers the cardinality and type information to the property descriptor (which is responsible for monitoring the restrictions put on the cardinality and the type). The method returns an iterator containing only the

## 6. Flexible, Efficient and Safe Template Instantiation

retrieved property descriptor, thereby causing the elements contained within the `xtl:for-each` instruction to be evaluated once with this property descriptor as context argument.

```
public Iterator<PropertyDescriptor> evaluateForEach(XTLForEachStart xtlForEach,
    PropertyDescriptor context)
{
    try
    {
        String select = xtlForEach.getSelect();

        logger.debug("evaluateForEach(...) called with select='"+select+"'");

        PropertyDescriptor propertyDescriptor =
            retrievePropertyDescriptor(context, select);

        propertyDescriptor.setType(UNKNOWN);
        propertyDescriptor.setMinOccurs(xtlForEach.getMin());
        propertyDescriptor.setMaxOccurs(xtlForEach.getMax());

        return Collections.singletonList(propertyDescriptor).iterator();
    }
    catch (XPathExpressionException xpe)
    {
        log(xpe);
        return Collections.<PropertyDescriptor>emptyList().iterator();
    }
}
```

Listing 6.18: The `evaluateForEach` Method in the `AnalyzerPHP`

After the `AnalyzerPHP` has completed processing the XTL template, its root property descriptor is taken and the tree of property descriptors is transformed into a Java object model starting from the root property descriptor. The Java code itself is constructed using an API-based Java generator [97]. The transformation process also generates the `validate` method described above. An example for such a method, in which a cardinality of 2...4 is assumed to be allowed for the authors property, is shown in Listing 6.19. In addition to the classes from the object model, a PHP named `ObjectModelRootPHP` is generated, which serves as an adapter between the generated object model and the Template Instantiation component. This PHP can be created with an instance of the `ObjectModelRoot` class as argument. The PHP calls the `validate` method on the passed instance immediately, causing the complete validation of the passed object model.

```
public void validate() {
    if (authors == null) {
        throw new IllegalStateException("Missing required
            attribute/element 'authors'.");
    }
    if (authors.size() < 2) {
        throw new IllegalStateException("Number of elements
            'authors' is less than expected minimum 2.");
    }
    if (authors.size() > 4) {
```

```

        throw new IllegalStateException("Number of elements
            'authors' is greater than expected maximum 4.");
    }
    for (Author current: authors) {
        current.validate();
    }
    /* ... */
}

```

Listing 6.19: An Example for a `validate` Method Implementation

The implementation of the Template Interface Generation process (see Section 7.1.4) also supports the immediate compilation of the Java code model. In order to enable unit testing of the implementation, the implementation also supports the direct introduction of JAXB annotations into the generated object model. This allows the direct use of XML documents as instantiation data source for the generated `ObjectModelRootPHP`, which in turn greatly simplifies the test process (see Section 7.2).

## 6.4. Conclusion

In this chapter, the components involved during the instantiation time of the proposed process have been discussed: the Instantiation Data Evaluation, the Template Instantiation and the Instantiation Data Validation components.

The Instantiation Data Evaluation component is invoked by the Template Instantiation components to retrieve the instantiation data in order to instantiate a particular template. A plugin mechanism has been introduced that allows different query languages to be used in conjunction with the proposed approach. This plugin mechanism distinguishes the approach from existing techniques, which typically use a fixed special or general purpose language (like XPath or Java). Several Instantiation Data Evaluation plugins have been presented.

The Template Instantiation is a core component within the proposed approach. In order to create an implementation which can compete with existing approaches like JSP and XSL-T, major effort has been invested into the design of this component. The best-suited XML access technology, StAX, and its advantages have been described. An operational model has been developed that has been used to implement the component. This implementation as a pipeline of components has been described in detail. An estimation for the memory and time complexity of this component has been given.

The Instantiation Data Validation component is responsible for validating the instantiation data retrieved from the Instantiation Data Evaluation component against the instantiation data constraints emitted by the Constraint Separation component. The design and implementation of this component turned out to be straightforward.

An interesting alternative approach to the Instantiation Data Validation component has been introduced: the Template Interface Generation. This is a slight modification of the proposed architecture that elevates the process of the instantiation data validation into the application using the Template Instantiation component. This has been achieved by generating an interface

## *6. Flexible, Efficient and Safe Template Instantiation*

for a particular template, which guarantees the correctness of the instantiation data. This approach for guaranteeing the instantiation data's types has never been used in conjunction with a template approach before.

# 7

## Validation

Es ist leicht, Vorschriften über die Theorie des Beweises aufzustellen, aber der Beweis selbst ist schwer zu führen.

Giordano Bruno [32]

In order to verify the design decisions and to scrutinize the statements which have been made in the previous chapters, a number of validation steps have been executed. Most notably, the implementation of a prototype, illustrating most of the concepts developed in this thesis, delivered a proof of concept for many design decisions.

The prototype implementation is revisited with respect to validation in Section 7.1. The developed prototype has been used and improved in various research projects. These applications are described in Section 7.3. Furthermore, the formal proof given in Section 5.1.5 for the fulfillment of the preservation requirement referenced in Section 7.4 is also a validation means. Measurements have been made in order to evaluate the performance of the template engine in comparison to other, established techniques. The results of the measurements are described in Section 7.5.

Figure 7.1 shows which goals (as defined in Section 3.1) are addressed by the particular validation means described in this chapter.

### 7.1. Implementation of the Prototype

The most important validation tool is the implementation of a prototype called *XTLEngine*. This prototype implements the proposed approach as far as possible within the restrictions of the

## 7. Validation

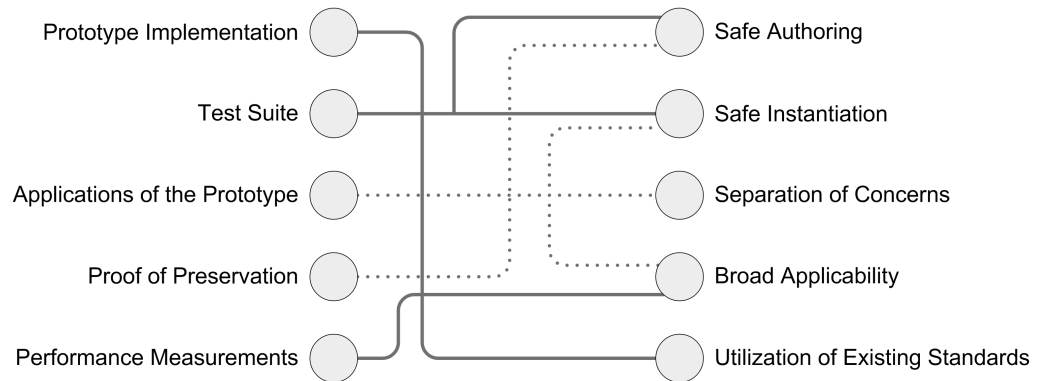


Figure 7.1.: Relations between Validation Means and Goals

underlying base technologies. In the following, the prototype version 2.0, build 607 is described. The prototype consists of approximately 15.500 lines of code comprising 195 classes organized in 35 packages. The prototype includes the following artifacts:

- The `XTLEngine.jar` and all required libraries.
- Command line tools for Windows and Mac OS X (described below).
- Test suites containing examples for the use of the included tools (see Section 7.2).
- The XTL, CXSD and IDC schemas.
- The documentation of the source code.

An overview of the tools supplied with the prototype is given in Figure 7.2. The figure should be compared with the Figures 3.5 and 6.14, since it closely resembles and aggregates their structures. The four tools shown are implementations of the Constraint Separation process (`xtlsc`, see Section 7.1.1), the Template Validation process (`cxsdvalidate`, see Section 7.1.2), the Template Instantiation process (`xtlinstantiate`, see Section 7.1.3) and the Template Interface Generation process (`xtltc`, see Section 7.1.4).

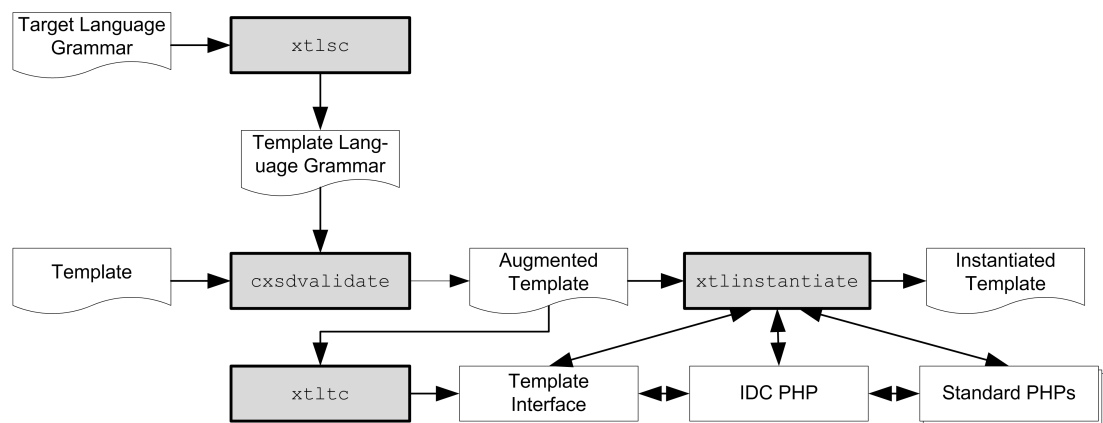
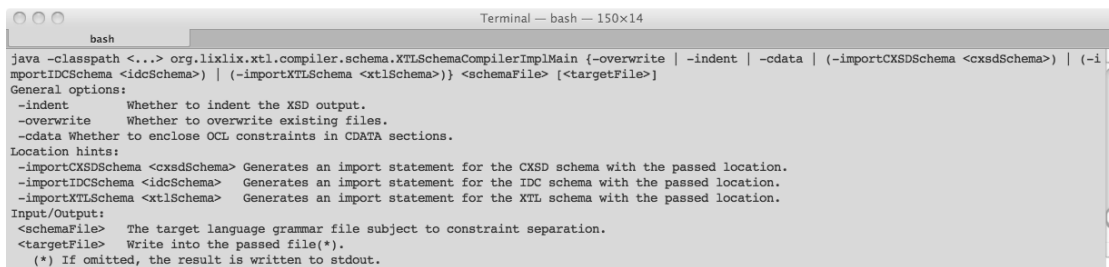


Figure 7.2.: The Prototype's Tool Architecture

### 7.1.1. The Constraint Separation Tool `xtlsc`

The Constraint Separation component is supplied as a command line tool named `xtlsc` (XML Schema Compiler, as shell script and Windows batch file) in the `bin` directory of the prototype. The tool is also available as an ANT task via the class `org.lixlix.xml.compiler.schema.XTLSchemaCompilerTask` or via its API class `org.lixlix.xml.compiler.schema.XTLSchemaCompilerImpl`. Figure 7.3 shows the command line options of the `xtlsc` tool.



```

bash
Terminal - bash - 150x14
java -classpath <...> org.lixlix.xml.compiler.schema.XTLSchemaCompilerImplMain {-overwrite | -indent | -cdata | (-importCXSDSchema <cxsdSchema>) | (-importIDCSchema <idcSchema>) | (-importXTLSchema <xtlSchema>)} <schemaFile> [<targetFile>]
General options:
-indent          Whether to indent the XSD output.
-overwrite       Whether to overwrite existing files.
-cdata          Whether to enclose OCL constraints in CDATA sections.
Location hints:
-importCXSDSchema <cxsdSchema> Generates an import statement for the CXSD schema with the passed location.
-importIDCSchema <idcSchema>   Generates an import statement for the IDC schema with the passed location.
-importXTLSchema <xtlSchema>   Generates an import statement for the XTL schema with the passed location.
Input/Output:
<schemaFile>    The target language grammar file subject to constraint separation.
<targetFile>    Write into the passed file(*).
(*) If omitted, the result is written to stdout.

```

Figure 7.3.: Console Help of the `xtlsc.sh` Command

The `xtlsc` arguments possible here fall into three categories: general options, location hints and input/output arguments.

The first category contains the option `-indent`, which defines whether the generated target language grammar should be indented, the option `-overwrite`, which defines whether an already existing target file should be overwritten, and `-cdata`, which defines whether the generated CXSD constraints should be wrapped in CDATA sections to improve their readability.

The location hints category allows to specify locations for the CXSD, the IDC and the XTL schema, which will be imported using `xsd:import` in the generated schema. The names of the options are `-importCXSDSchema`, `-importIDCSchema`, and `-importXTLSchema`, respectively. If such options are given, their arguments will be used for the `schemaLocation` attribute of the `xsd:import` statements, which will allow other tools to locate the imported schemas.

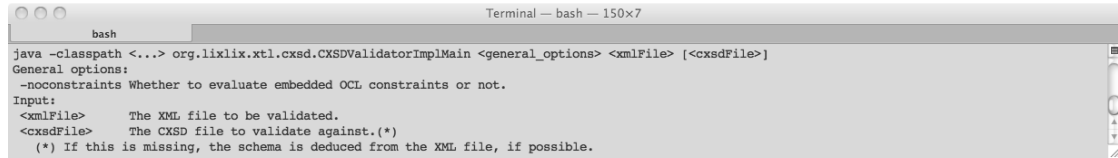
The final category is formed by the arguments, pointing to the XML Schema file to be processed, and an optional target file. If the target file is omitted, the result of the Constraint Separation process is written to the standard output.

### 7.1.2. The Template Validation Tool `cxsdvalidate`

The Template Validation component is supplied as a command line tool named `cxsdvalidate` (as shell script and Windows batch file) in the `bin` directory of the prototype. The tool is also available as an ANT task via the class `org.lixlix.xml.cxsd.CXSDValidatorTask` or via its API class `org.lixlix.xml.cxsd.CXSDValidatorImpl`. Figure 7.4 shows the command line options of the `cxsdvalidate` tool.

`cxsdvalidate` knows only one option: `-noconstraints` can be used to validate against a CXSD schema as if it would be an XML Schema, i.e., all embedded OCL constraints are ignored during validation. The only required argument is the name of the XML document to be validated

## 7. Validation



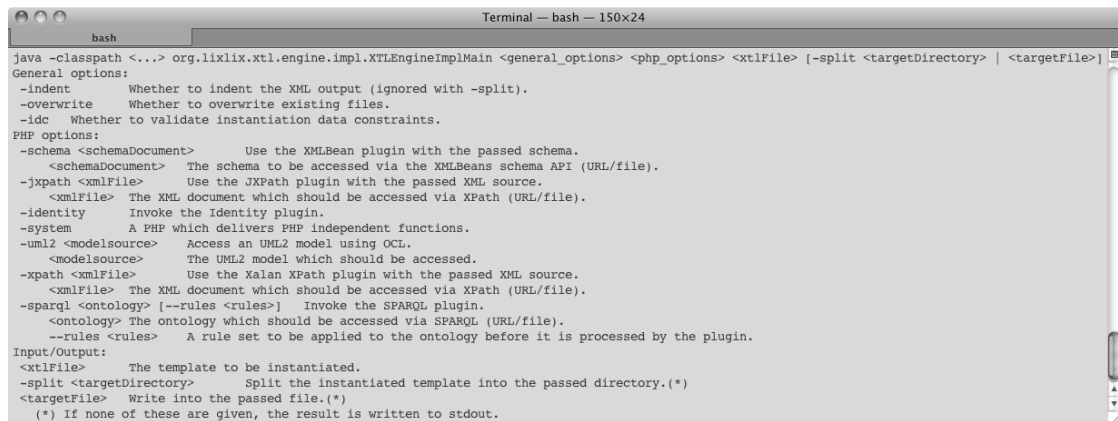
```
Terminal — bash — 150x7
bash
java -classpath <...> org.lxlxl.xml.cxsd.CXSDValidatorImplMain <general_options> <xmlFile> [<cxsdFile>]
General options:
  -noconstraints Whether to evaluate embedded OCL constraints or not.
Input:
  <xmlFile>      The XML file to be validated.
  <cxsdFile>      The CXSD file to validate against.(*)
  (*) If this is missing, the schema is deduced from the XML file, if possible.
```

Figure 7.4.: Console Help of the `cxsdvalidate.sh` Command

against the CXSD schema, which can be passed as second argument. If the second argument is missing, `cxsdvalidate` tries to find the CXSD schema using the `xsd:schemaLocation` or `xsd:noNamespaceSchemaLocation` attributes from within the XML document.

### 7.1.3. The Template Instantiation Tool `xtlinstantiate`

The Template Instantiation component is accessible as a command line tool named `xtlinstantiate` (as shell script and Windows batch file) in the `bin` directory of the prototype. The tool is also available as an ANT task via the class `org.lxlxl.xml.engine.impl.XTLEngineTask` or via its API class `org.lxlxl.xml.engine.impl.XTLEngineImpl`. The command line options of the `xtlinstantiate` tool are shown in Figure 7.5.



```
Terminal — bash — 150x24
bash
java -classpath <...> org.lxlxl.xml.engine.impl.XTLEngineImplMain <general_options> <php_options> <xtlFile> [-split <targetDirectory>] [<targetFile>]
General options:
  -indent          Whether to indent the XML output (ignored with -split).
  -overwrite       Whether to overwrite existing files.
  -idc             Whether to validate instantiation data constraints.
PHP options:
  -schema <schemaDocument> Use the XMLBean plugin with the passed schema.
  <schemaDocument> The schema to be accessed via the XMLBeans schema API (URL/file).
  -jxpath <xmlFile>       Use the JXPath plugin with the passed XML source.
  <xmlFile> The XML document which should be accessed via XPath (URL/file).
  -identity         Invoke the Identity plugin.
  -system           A PHP which delivers PHP independent functions.
  -uml2 <modelSource>    Access an UML2 model using OCL.
  <modelSource> The UML2 model which should be accessed.
  -xpath <xmlFile>       Use the Xalan XPath plugin with the passed XML source.
  <xmlFile> The XML document which should be accessed via XPath (URL/file).
  -sparql <ontology> [--rules <rules>] Invoke the SPARQL plugin.
  <ontology> The ontology which should be accessed via SPARQL (URL/file).
  --rules <rules> A rule set to be applied to the ontology before it is processed by the plugin.
Input/Output:
  <xtlFile> The template to be instantiated.
  -split <targetDirectory> Split the instantiated template into the passed directory.(*)
  <targetFile> Write into the passed file.(*)
  (*) If none of these are given, the result is written to stdout.
```

Figure 7.5.: Console Help of the `xtlinstantiate.sh` Command

The arguments for `xtlinstantiate` fall in three categories: general options, options to enable a PHP and to pass arguments to it, and input/output arguments.

The first category contains the option `-indent`, which defines whether the instantiated template should be indented, the option `-overwrite`, which defines whether an existing target file should be overwritten, and the option `-idc`, which defines whether the instantiation data constraints should be evaluated. The evaluation of the IDC constraints is obviously only possible if the template correctly links to an CXSD schema with embedded IDC constraints.

The second category contains options that can be used to enable and configure a particular PHP. As opposed to what is possible with the ANT task or via the API, the command line tool only allows to activate one plugin.

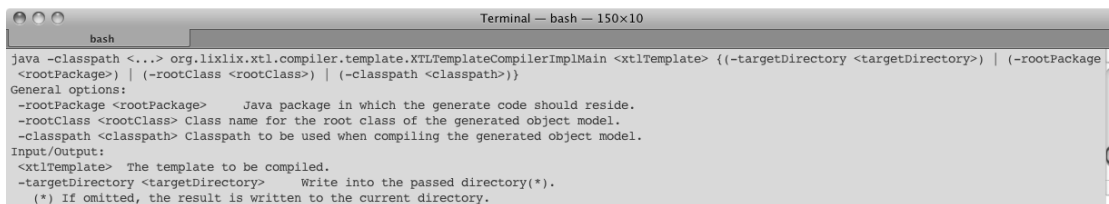


The `-schema` option with a file parameter activates the XMLBean PHP which parses the passed file as XML Schema. The `-jXPath` option activates the JXPath PHP with the passed file being parsed as an XML document. The `-identity` option activates the Identity PHP. The `-system` option activates the System PHP. The `-uml2` option enables the UML PHP, which loads the passed file as XML representation of an UML model. The `-xpath` option activates the XPath PHP, which interprets the passed file as an XML document. Finally, the `-sparql` option activates the SPARQL PHP with the passed file as an ontology. `-sparql` allows the sub option `--rules` with a file parameter: if such a rule file is present, it is applied to the ontology before the first query is executed on it.

The final category is formed by the input/output arguments. `xtlinstantiate` requires a file argument which denotes the XTL template to be instantiated. As the output argument, a file or the option `-split` followed by a directory are allowed. In the latter case, the result is split as described in Section 6.2.3 and the results of the splitting are written to the passed directory. If none of the output arguments are given, the output is written to the standard output.

#### 7.1.4. The Template Interface Generation Tool `xtltc`

The Template Interface Generation Tool `xtltc` implements the ideas described in Section 6.3.2. The tool is available as shell script and Windows batch file in the `bin` directory of the prototype, as an ANT task class `org.lixlil.xtl.compiler.template.XTLTemplateCompilerTask` or via its API class `org.lixlil.xtl.compiler.template.XTLTemplateCompilerImpl`. The command line options of the `xtltc` tool are shown in Figure 7.6.



```

bash
java -classpath <...> org.lixlil.xtl.compiler.template.XTLTemplateCompilerImplMain <xtlTemplate> {(-targetDirectory <targetDirectory>) | (-rootPackage
<rootPackage>)} | (-rootClass <rootClass>) | (-classpath <classpath>)}
General options:
-rootPackage <rootPackage>      Java package in which the generate code should reside.
-rootClass <rootClass>          Class name for the root class of the generated object model.
-classpath <classpath>          Classpath to be used when compiling the generated object model.
Input/Output:
<xtlTemplate>                  The template to be compiled.
-targetDirectory <targetDirectory> Write into the passed directory(*).
(*) If omitted, the result is written to the current directory.

```

Figure 7.6.: Console Help of the `xtltc.sh` Command

The `xtltc` tool accepts options from two categories: general options and input/output arguments. In the first category, the option `-rootPackage` can be used to define the package into which the generated Java source code should be placed. The option `-rootClass` defines the name of the root class within the generated object model, since this name can not be inferred from the `select` attribute expressions in the compiled XTL template. The option `-classpath` supplies the Java compiler used to compile the generated Java source files with a class path to compile against.

The input/output argument category contains the name of the XTL template to be compiled as a required argument and an optional `-targetDirectory` option with an argument denoting the directory to which the created Java sources should be written.

## 7.2. Test Suites

The test suites are the main tools to validate the fulfillment of the goals Safe Authoring and Safe Instantiation. All important aspects of the prototype as well as statements made in Chapter 4 are subject to test suites. There are five test suites, which are described in detail below.

All test suites either test a single tool or a particular combination of tools and operate on a number of input documents like schemas or templates, and produce other documents from them. After the tool under test has been executed and results have been produced, the results are compared to the expected results. This is done either textually or via an XML comparison tool (XMLUnit, [194]). The use of XMLUnit allows to compare XML documents with respect to the XML specification [28]. For example, XMLUnit ignores the order of attributes during comparison.

The test fixture, i.e., the input documents for the various test suites, and the expected results, i.e., the instantiated templates, are also part of the prototype. The execution of all test suites is done via ANT.

### 7.2.1. Constraint Separation Test Suite

The Constraint Separation test suite tests the Constraint Separation step (see Section 5.1) by invoking the `xtlsc` tool described in Section 7.1. The test suite operates over an input set of 18 different XML Schema documents, which test the XML Schema features supported by the Constraint Separation process like choices, sequences, required and optional attributes. The test process, which is illustrated in Figure 7.7, consists of the following steps:

- ① The `xtlsc` tool is invoked with each of the target language grammars as input. The generated result template language grammar is saved.
- ② The generated result is compared against the stored *expected* template language grammar. The comparison is done as described above, i.e., semantically irrelevant differences like whitespaces are ignored.

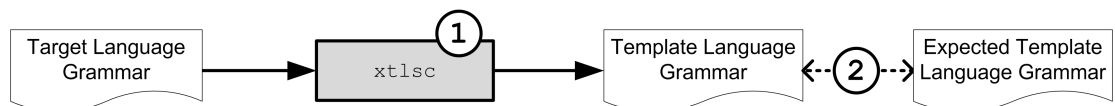


Figure 7.7.: Constraint Separation Test Suite

### 7.2.2. Template Validation Test Suite

The Template Validation test suite tests the Template Validation process by checking the validity of XTL documents with respect to CXSD schemas. The Template Validation tool `cxsdvali-date` produces one of two possible results: it either outputs an augmented (in the sense of Section 5.2) XTL template if the validation has succeeded, or it outputs a validation report with

a list of detected errors if the validation has failed. The test process is illustrated in Figure 7.8 and consists of the following steps:

- ① The `cxsdvalidate` tool is invoked with a pair of an XML document (which is in most cases an XTL template as well, see below) and a corresponding template language grammar (which is a valid CXSD document). The result, which is either an augmented XTL template or a validation report, is stored.
- ② For each test case, either an *expected* augmented XTL template or an *expected* validation report has been stored, which is compared to the actual output of the `cxsdvalidate` invocation. It is an error if the types of the actual and the expected document differ, since this means that the overall validation result is wrong.

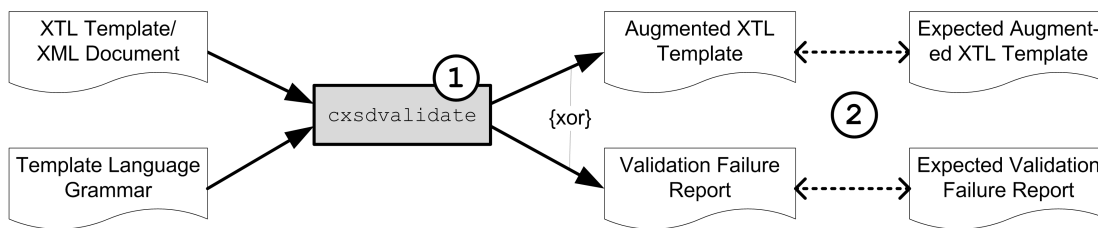


Figure 7.8.: Template Validation Test Suite

The set of input documents is divided into two categories. First, the test suite checks special documents against CXSD schemas in order to check the CXSD validation as such (i.e., the XML Schema validation as well as the construction of the XML model underlying the OCL constraint evaluation). Second, the test suite tests instance documents against the template language grammars produced by the Constraint Separation Test Suite described above. This latter test ensures that the Constraint Separation process works together with the Template Validation process in order to allow the Safe Authoring of templates. In both cases, valid and invalid documents are tested.

### 7.2.3. Template Instantiation Test Suite

The Template Instantiation test suite tests the Template Instantiation component described in Section 6.2 as well as the Instantiation Data Evaluation components described in Section 6.1.

The test suite tests the `xtlinstantiate` tool with 72 templates as input documents. The instantiation data comes from 44 documents. The PHPs for the evaluation of XPath, OCL, and SPARQL as well as the Identity PHP are tested. All XTL instructions and XTL features like bypassing and realms are included in the tests. For 8 augmented templates, the instantiation data validation feature is enabled to also check the Instantiation Data Validation component described in Section 6.3.1.

The test suite also tests that the XTL Engine, i.e., the Java implementation of the Template Instantiation component, adheres to the denotational XTL semantics given in Chapter 4. For this

## 7. Validation

reason, the XTL semantics (which is described in Haskell) has been compiled into an executable using the Glasgow Haskell Compiler (GHC). This compiled version is also part of the prototype and is named `hsxtl`. As the Haskell version of the Template Instantiation component only supports instantiation data sources that are accessible using XPath and is not supporting multiple realms, this test only checks 30 XTL templates against 39 instantiation data files. This restriction aside, all XTL instructions have been tested.

Finally, the test suite tests that the XTL Engine and the translational semantics described in Section 4.6 yield equal results. This test is a two-stage process, as the XTL templates are first transformed into XSL-T stylesheets using the `xtl-to-xslt` stylesheet, and then used to transform the instantiation data XML documents. Again, the set of instantiation data sources had to be restricted, since the XSL-T stylesheets generated from the XTL documents can only evaluate XPath expressions. Thus, the set of input documents is the same as used for the `hsxtl` test.

The test process is illustrated in Figure 7.9 and consists of the followings steps:

- ① The instantiation tool, i.e., either `xtlinstantiate`, `hsxtl`, or `xsl-to-xslt`, is invoked with a combination of an XTL template and single or multiple instantiation data sources. The result, be it an instantiation result or a failure report (if an instantiation data constraint has been violated), is saved.
- ② For each test case, either an expected instantiation result or an expected instantiation failure report has been stored, which is compared to the actual output of the instantiation tool. It is an error if the types of the actual and the expected document differ, since this means that the evaluation of the instantiation data constraints failed.

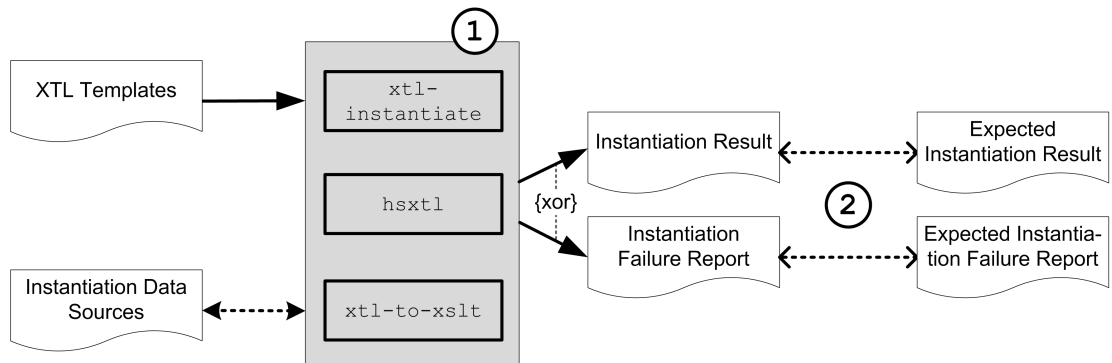


Figure 7.9.: Template Instantiation Test Suite

### 7.2.4. Template Interface Generation Test Suite

This test suite checks the Template Interface Generation component described in Section 6.3.2 via invocation of the `xtl2c` tool described in Section 7.1. The test case compiles 27 XTL templates into Java sources.

An invocation of the `xtltc` tool not only produces Java sources, but also compiled class files and a Java Archive (JAR)–file [177] containing the compiled files. This JAR file would afterwards typically be used to programmatically construct an instantiation data source to be used in conjunction with the template engine itself. To allow for testing the generated Java classes without having to manually code individual test cases for each XTL template compiled within this test suite, the `xtltc` adds JAXB annotations [105] to the classes within the generated object model. This allows the whole object model to be created from a single XML document without having to deal with that particular model in the code. Therefore, the ANT-based variant of the `xtlinstantiate` tool has been extended to accept a JAR file as generated by the `xtltc` tool and a single XML document, which is in turn used to create and initiate an `ObjectModelPHP` (as described in Section 6.3.2) to be used to instantiate the passed XTL template.

Using this mechanism, the test suite illustrated in Figure 7.10 could be constructed, which consists of the following steps:

- ① The `xtltc` tool is invoked with XTL templates containing `select` attribute values that comply to the restrictions introduced in Section 6.3.2. The resulting artifacts, namely the Java source code, the compiled classes and the JAR-file are stored.
- ② The generated Java source code is compared textually to the stored *expected* source code.
- ③ The generated JAR-file is used with a stored XML document (acting as instantiation data source) to instantiate the template originally processed by the `xtltc` tool into an XML document, which is stored as the instantiation result.
- ④ The instantiation result is compared to the stored *expected* instantiation result, which assures that the generated object model is indeed suitable for and working with the input XTL template.

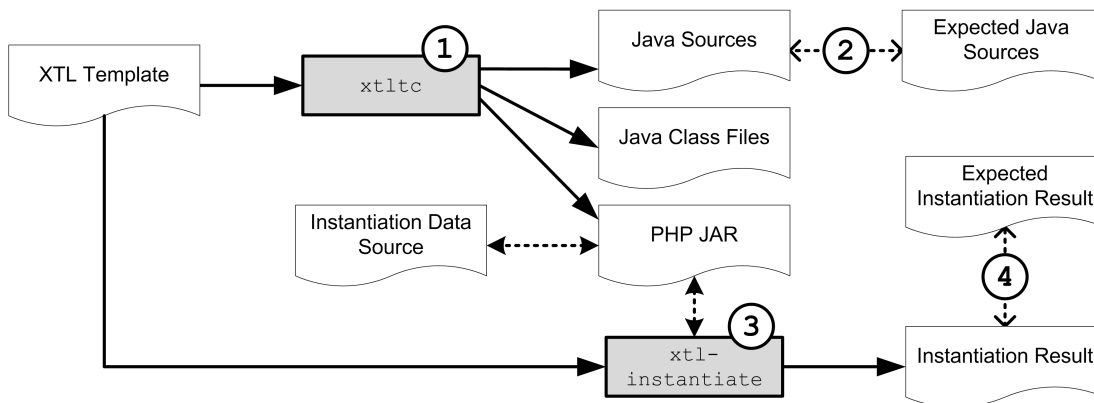


Figure 7.10.: Template Interface Generation Test Suite

### 7.2.5. Round-trip Test Suite

The last test suite, named Round-trip Test Suite, tests the overall template authoring and instantiation process as a whole. In other words, it tests whether the conclusion stated to be enabled by the Constraint Separation process (see Figure 5.1) is valid. This test suite calls three tools and checks their collaboration as illustrated in Figure 7.11, by executing the following steps:

- ① In the first step, the `xtlsc` tool is invoked on a particular target language grammar. The generated template language grammar is stored.
- ② The stored template language grammar is used to validate XTL templates associated with the target language grammar currently processed. The results, each being either an augmented XTL template or a validation report, are stored for the final comparison step within this test suite.
- ③ Each XTL template is also instantiated using the `xtlinstantiate` tool with an associated instantiation data source (which fulfills all instantiation data constraints).
- ④ The instantiation result from the last step is validated using the `cxsdvalidate` tool against the original target language grammar, resulting in either an augmented instantiation result (which equals to the instantiation result, as the original target language grammar contains no authoring or instantiation data constraints) or a validation failure report.
- ⑤ The last step compares the output of both invocations of the `cxsdvalidate` tool: the test succeeds if *either* both invocations report validity of its input document and schema *or* both invocations report invalidity, thereby validating the conclusion which is proposed to be enabled by the Constraint Separation process.

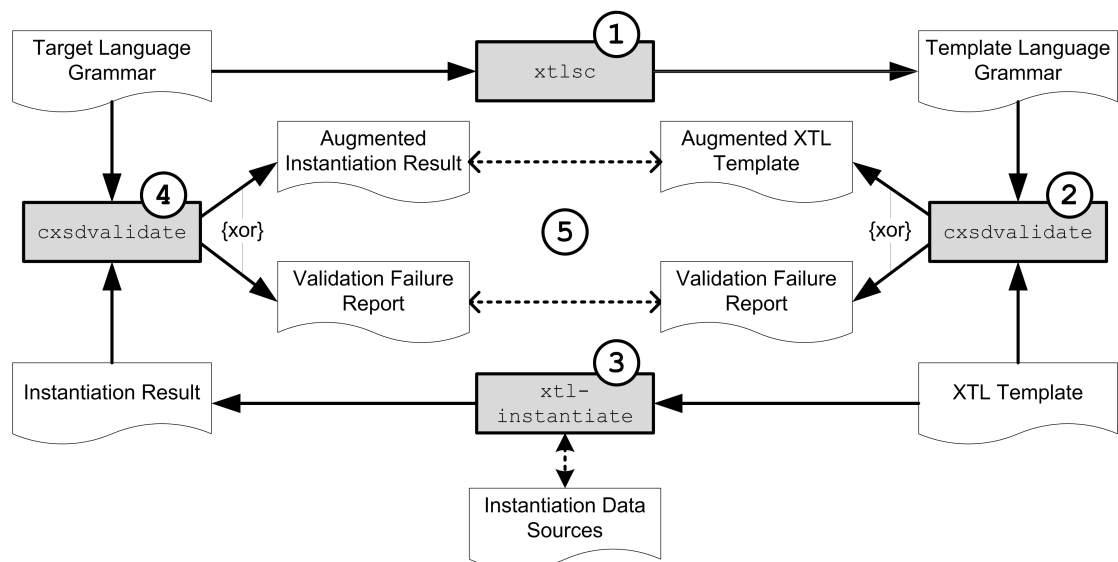


Figure 7.11.: Round-trip Test Suite

### 7.3. Applications of the Prototype

The prototype has been in use in three scenarios: first, in the SNOW project (as described in Section 7.3.1), second, in the EMODE project (described in Section 7.3.2), and, finally, in the FeasiPLE project (as described in Section 7.3.3).

#### 7.3.1. SNOW: Use of XTL in a Staged Architecture

The XTL template engine has been developed along with the XTL language in the EU project SNOW. In order to understand the motivations that lead to the described language and to understand the validation results SNOW delivered, the project is introduced in short.

SNOW [179] was an EU-founded two-year project executed by seven partners, namely ACV [1], FIRST [70], EADS [58], Loquendo [121], Siemens Business Services' C-LAB [41], SAP Research Dresden [159] and TU Graz [78]. SNOW's main goal was the large-scale industrial diffusion of multimodal mobile documentation for maintenance operations.

SNOW was primarily intended to solve a real-world problem in today's aircraft maintenance as described by the partner EADS. The current maintenance scenario is entirely paper-based, i.e., a maintenance worker executes instructions from a so-called maintenance *procedure* printed out on paper. Unexpected situations may force the worker to return to an office and print out a different procedure. Furthermore, in some situations the worker needs a co-worker who reads the procedure if the first worker is unable to look at the printed procedure himself. Both facts slow down maintenance and increase the maintenance costs.

The idea of replacing this access to the procedures by an electronic device like a PDA was obvious. Unfortunately, the situation in the aircraft to be maintained complicates the scenario. First, there is no permanent network connection in the aircraft. Second, the worker sometimes needs to have both hands available to perform a procedure, which makes it necessary to enable the use of speech commands to scroll within the procedure. Since the use of speech as input modality is sometimes prevented by the situation in the aircraft (as the environment can be too noisy), an additional gesture recognition coupled to a head-mounted camera became necessary. Finally, the need for a co-worker described above can be removed by using speech synthesis to read the procedure.

From the main goal, two research directions have been derived. First, it has been questioned how to author multimodal mobile maintenance documentation. Second, methods and techniques for the exploitation of the authored documentation through robust interaction modalities had to be developed.

SNOW made a number of contributions in both research directions. For the first direction, the development of the XML Topic Maps for Procedures (XTM-P) [103], a topic-map based format for the storage of maintenance procedures has to be mentioned. With respect to the second research direction, two languages, the Device-Independent Multimodal Mark-up Language (D3ML) [75] and XTL, as well as an architecture [146; 147] along with a prototype have been developed.

### The SNOW Architecture

The SNOW architecture has been developed with respect to a number of requirements outlined in a number of deliverables [179]. As already mentioned, the first major requirement of the SNOW project was that the resulting software had to be accessible in a multimodal fashion. In the standard use case, this includes speech input and output as well as gesture recognition as input. But beyond that, the architecture should not restrict the number or type of usable modalities.

The second major requirement was to design an architecture which is as domain-neutral as possible, i.e., the number of parts to be exchanged when switching to another domain had to be minimized. A second domain which has been considered during the design of the SNOW architecture was the area of healthcare, where hands-free operation also plays an important role.

In addition to these major requirements, some minor issues had to be considered. First, the number of available devices that are usable in a harsh environment and capable of delivering input for gesture recognition (via built-in or extra camera) were limited. Moreover, the processing power of available devices is restricted, forcing gesture and voice recognition components to be located on a server with extensive processing capabilities.

Lastly, it was required that the documentation is always *at least as good as paper*, which means that even with interruptions of the network connection, the application's user must have access to (prefetched) procedures. The missing network connection may thereby affect accessibility of the application by restricting the use of modalities due to their server-based processing.

The architecture finally designed and implemented by the SNOW consortium is shown in Figure 7.12 as an FMC block diagram. This type of diagram allows a concise high-level view even at sophisticated software architectures. The SNOW architecture is subdivided into multiple servers and a client part. The most important server is the *application server* which is responsible for implementing the modality-independent processing of maintenance procedures. This server is explained in more detail below. Another important server-side component is the *modality server* consisting of a text-to-speech engine and gesture- and speech-recognition components. The number of contained components can be different in other scenarios—the subset shown here represents the set used within the SNOW project.

The application server accesses three data sources: first, a set of maintenance procedures stored as XTM-P documents, second, a set of XTL templates and finally a number of annotations created by maintenance workers and stored in a relational database.

On the client side, a multimodality-enabled browser application has been designed and implemented. This application aggregates a standard XHTML browser with components for the control of the keyboard, the speakers, the microphone and the camera as well as, most importantly, the *integration manager*, which is responsible for the synchronization and composition of the input and output modalities. The components within the client-side application communicate using standard protocols and data formats like XML Remote Procedure Call (XML-RPC), the Media Resource Control Protocol (MRCP), the Extensible MultiModal Annotation Markup Language (EMMA), and the Speech Synthesis Markup Language (SSML).



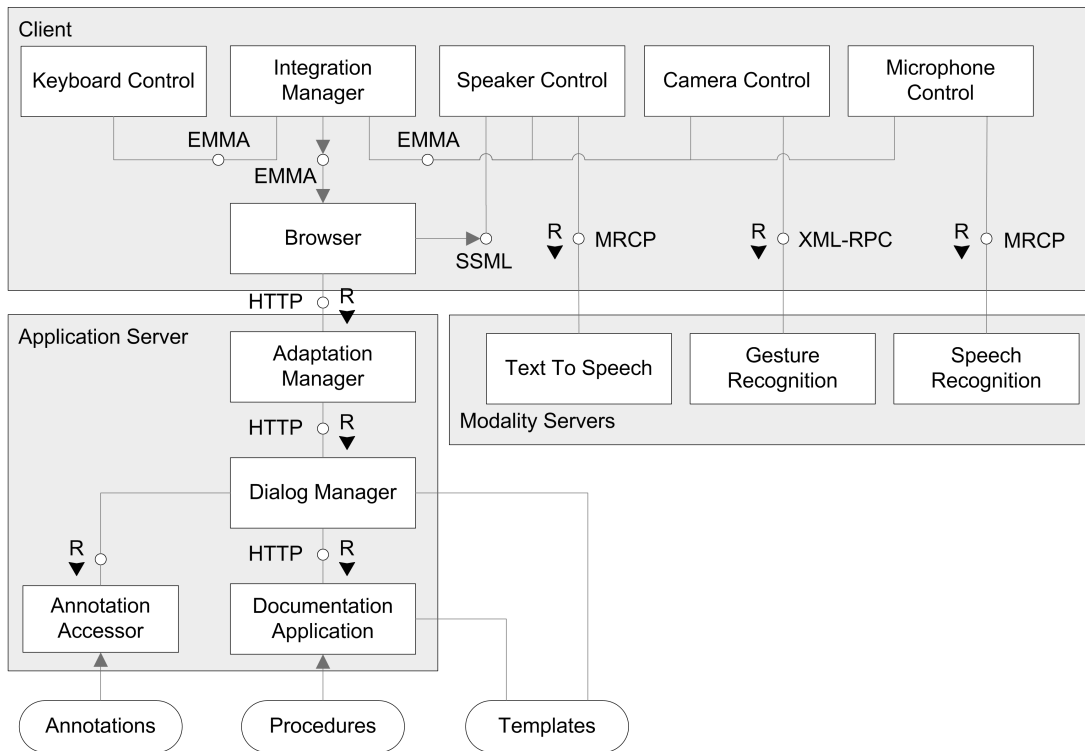


Figure 7.12.: The SNOW Architecture

As described in [146], the SNOW architecture is an instantiation of the Multimodal Interaction Framework (MMI-F), or—precisely—an implementation of the role model proposed by the MMI-F specification [191]. Details about the implementation of the SNOW architecture can be found in [147], a more detailed look into a particular issue of the implementation of the dialog manager can be found in [5].

### Template Processing in Staged Architectures

The XTL template engine is used in two components. The documentation application uses the template engine just once in order to instantiate a domain-specific template with data from the procedures stored as XTM-P files. The utilization of the template engine by the dialog manager is more interesting because it is used two times: first, a presentation template is transformed, augmenting the output from the documentation application with presentational content (like links for navigation); second, the obtained intermediate document still has some evaluateable XTL tags (which bypassed the first transformation) for evaluation with data from the annotation accessor.

It has also been verified that it would be possible to use the XTL engine in the adaptation manager, as the processing performed by this component is basically an XML transformation that could be expressed as an XTL template as well. Unfortunately, because of technical preferences,

## 7. Validation

the SNOW partner responsible for implementing the adaptation manager decided against the use of the XTL template engine.

Both the concepts of realms (see Section 4.5.1) and bypassing (see Section 4.5.2) have been developed as a reaction to actual requirements from the SNOW architecture. The concept of realms representing instantiation data sources that must be accessed using different query languages has been motivated by the multiple data sources in SNOW's Dialog Manager: XTM-P documents, which had to be accessed using an XPath-like path language, and annotations, which had to be accessed by simple string-based identifiers.

The bypassing feature is motivated by a special dependency between components in the SNOW architecture. The inclusion of annotations in the D3ML output is done in two components: in the first, the documentation application, only a placeholder for the rendering of annotations can be generated. Only the second component, the dialog manager, has actually access to the annotation content. It was impossible to move the processing of the annotations to one component without violating the contracts defined for the interaction of the components. Using the bypassing feature, this division of responsibilities in the annotation processing could easily be implemented: the D3ML template contains an `xtl:attribute` instruction from the bypassing namespace that is augmented via an `xtl:attribute` expression from the XTL namespace. The latter instruction generated the `select` statement for the first statement, thereby letting the participating components cooperate.

Generally, it can be stated that bypassing is a valuable feature in all kinds of staged architectures, as it allows to elegantly aggregate data accessible only during particular stages.

The relatively short runtime of the project made it impossible to research the interaction between the proposed Safe Authoring process and the staged architecture within SNOW. The process as described here is only capable of handling the initial stage of the architecture. The main reason for this limitation is the fact that XML Schema is not closed under the composition with the XTL schema, which causes the introduction of CXSD. CXSD-described languages are, however, not considered as input for the described authoring process. This situation can only be changed by using regular tree languages to describe the target language grammar, since those languages are closed under the composition with the XTL schema. Unfortunately, the low dissemination of languages like RelaxNG violates the stated goal of Broad Applicability.

### Usability of the XTL

The most valuable result of the SNOW project with respect to the development of the XTL template language was the feedback from the users of the SNOW architecture, which had to author XTL templates for rendering maintenance procedures into D3ML.

These users were experts from the maintenance department of EADS, with a strong technical background in terms of aircraft maintenance processes and mid-level computer skills, which doesn't include a deep knowledge of markup languages. Therefore, these users had first been introduced into the concepts of the XML dialect D3ML.

After understanding the concepts of a markup language like D3ML, the learning curve of a second, overlying concept like XTL was amazingly flat. The users were immediately capable of dynamically creating attributes or text in D3ML templates using `xtl:attribute` and `xtl:`

text. The same is true for the use of `xtl:if` and `xtl:for-each`, which were also understood within days.

Two additional observations had been made. First, the structure of the query language embedded in the `select` attributes of the XTL instructions plays a very important role in the learning process and can cause the authoring process to become error-prone and very hard to understand. Unfortunately, this was the case with the proprietary query language used to access the XTM-P documents.

Second, it has been observed that the concept of reuse, which is well-understood by computer scientists, has not been accepted by the users of the SNOW architecture. Instead of using the `xtl:macro` and `xtl:call-macro` mechanism supplied by XTL, the users tended to copy and paste repeated parts of the templates. The question on the motivation of this approach was typically answered by a hint to small modifications made to different copies of the reused material. The argument that the maintenance of multiple copies of almost identical document parts is expensive had been rejected—the users were of the opinion that the effort of learning a reuse concept is much higher than to maintain different copies.

#### 7.3.2. EMODE: Use of XTL for Model-to-Text Transformations

The EMODE project was a BMBF-funded project which tried to solve some of the issues occurring when trying to build large-scale multimodal applications by applying a Model Driven Software Development (MDSD) approach. EMODE defined a stack of models that describe the interaction with a multimodel system in increasing levels of detail, starting with a goal model, which is refined into an abstract user interface, and a functional core adapter model [44]. The transformation of models within the model stack are implemented as QVT transformations.

EMODE implemented two target platforms. EMODE reused the D3ML language developed within SNOW and additionally supported Java Abstract Window Toolkit (AWT) as a second target. Different M2C transformation techniques have been used for both targets: D3ML has been generated using the XTL template engine, whereas AWT has been generated using JET. Unfortunately, no comparison between these very different techniques has been published.

#### 7.3.3. FeasiPLe: Use of XTL for Code Generation from Ontologies

The FeasiPLe project was another BMBF-funded project which tried to eventually implement software product lines as a cost-efficient mean for industrial software development. In order to do so, FeasiPLe tried to evaluate the existing approaches and to combine them with promising new techniques like AOP and MDSD. The central validation case of FeasiPLe was a large-scale *SalesScenario*, an example for a Web application dealing with sales processes and including a product, customer and customer order management as well as payment and communication features.

As a part of this project, the Hybrid MDSD approach has been developed—an approach which tries to facilitate the use of multiple DSLs. This is done by using ontologies to capture the semantics of the DSLs [120]. The XTL template engine has been used to generate code from these ontologies using its SPARQL PHP [119]. As a transitive closure was needed, which is beyond

## 7. Validation

the expressive power of SPARQL, the possibility to execute rule sets on the ontology before it is queried using SPARQL was added.

During experiments with the querying of ontologies, it became also apparent that a transitive closure missing from a query language could also easily be added by using multiple template instantiations. For this emulation, the reintroduction of XTL markup via XTL instructions had to be allowed. Afterwards, a query could easily create a further query which performs a query based on the results of the first, which results, if an appropriate stop condition is applied, in the calculation of a transitive closure.

The development of the SPARQL plugin within FeasiPLe also motivated the introduction of `xtl:init`, since SPARQL queries typically involve a lot of XML namespaces. `xtl:init` can be used to refactor the XML namespace assignments into a single, central part of the template. The queries themselves then only use the prefixes assigned to the namespaces.

The PHP mechanism made it easy to extend the XTL template engine to support SPARQL for the querying of instantiation data sources. This unique extension mechanism of XTL enables the Broad Applicability of the approach.

### 7.4. Proof of the Preservation of the Target Language Constraints

The proof demonstrating that the target language constraints are preserved by the Constraint Separation process given in Section 5.1.5 addresses the Safe Authoring goal introduced in Section 3.1.1. Therefore, it is a very important validation means, but due to its central importance in the Safe Template Processing approach, it has been placed in Chapter 5.

### 7.5. Runtime and Memory Usage Measurements

Runtime and memory usage measurements have been conducted to validate the broad usability of the approach. All solution elements proposed in Section 3.3 could in principle be subject to runtime and memory usage measurement. Here, only the most important components, for which runtime and memory usage are crucial for the broad applicability of the approach, will be considered. These most important components are the components active during the authoring phase and the instantiation phase.

During the authoring phase, the acceptance of the approach is determined by the time needed for a complete validation of a document. If a validation takes too much time, the validation will not be used, causing most of the advantages of the approach to vanish. The faster the validation completes, the more often it will be invoked by the author, making the validation a real benefit. For this reason, Section 7.5.1 compares the runtime of a validation against a CXSD schema with the runtime of the validation against a comparable plain XML Schema document.

In the instantiation phase, both runtime and memory usage are of importance to the acceptance of the approach. Long lasting instantiations or exhaustive memory consumption are unacceptable, especially if the template technique should be used within Web applications. In this area, XTL must keep up with competitors like JSP and XSL-T. Therefore, a comparison between XTL and these competitors has been made. Furthermore, the memory and time complexity statements from Section 6.2 have been subject to corresponding measurements in order

to prove their correctness. The runtime measurements concerning the Template Instantiation component itself are described in Section 7.5.2. The corresponding memory usage measurements are described in Section 7.5.3.

All measurements have taken place on the same hardware and software: an Intel-based MacBook Pro with a 2.8 GHz Intel Core Duo CPU and 4 GB RAM. The operating system was Mac OS X 10.6.3, the Java version used to execute the components was Java 1.6.

### 7.5.1. Runtime Measurement of Validation against a CXSD Schema

The process of validating a template against a CXSD schema in order to determine whether the template is going to produce a valid result in terms of the target language schema is of crucial importance to the template author. Unfortunately, an analysis of the evaluation complexity of OCL constraints in terms of runtime and memory usage does not seem to exist.

For this reason, a mere benchmark comparison of validating a template against a CXSD schema with validating against an XML Schema has been produced. As the comparison tries to determine the extra effort caused on the author's side by the more sophisticated validation, documents with a parameterizable size indicated by the parameter  $n$  have been created.

An example document is shown in Listing 7.1. The document starts with a number  $n$  of perfectly valid `content` elements with an attribute named `attribute`. The document further contains a number of  $n - 1$  valid elements (which are different in that they don't carry the attribute `attribute`, but rather contain a further element with the name `attribute`), followed by a `content` element that is neither carrying an attribute nor containing an element. Only this last element is causing a violation of the OCL constraints in the CXSD document.

```
<?xml version="1.0" encoding="UTF-8"?>
<test>
  <!--  $n$  valid elements. -->
  <content a="text"/>
  <!-- ... -->
  <!--  $n - 1$  elements. -->
  <content>
    <attribute name="a">text</attribute>
  </content>
  <!-- ... -->
  <!-- 1 invalid element. -->
  <content/>
</test>
```

Listing 7.1: An Example Instance Document for Runtime Measurements

Figure 7.13 shows the comparison between the validation time of the example document with the parameter  $n$  against the CXSD schema and the corresponding XML Schema. The measurements have been executed by a Perl script which executed the CXSD validation tool described in Section 7.1 with and without the `-noconstraints` parameter. For each size of the document to be validated, the validation time has been measured 300 times. The figure shows the average validation time over these 300 measurements.

## 7. Validation

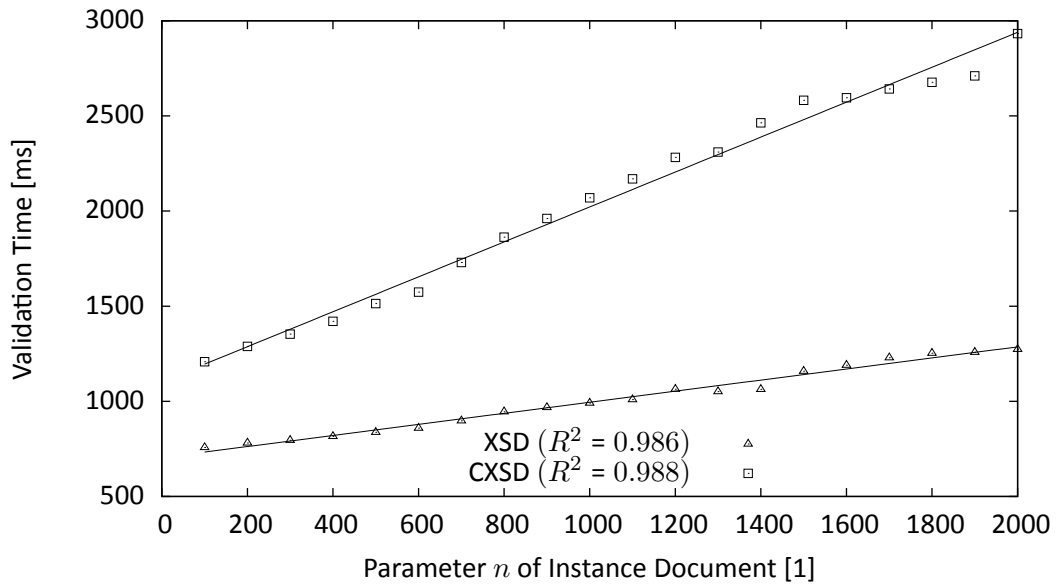


Figure 7.13.: Time Consumption during Document Validation

It can easily be seen that the evaluation of the OCL constraints embedded within the CXSD schema contributes significantly to the validation time. However, the time complexity order is not changed: validation is still completed in linear time. The additional time needed could be decreased by several measures, most of all by an incremental validation approach as it would be enabled by an impact analysis as proposed in [3].

Even without optimizations like this, the validation speed is still acceptable, because there is no reason to interrupt the user in his workflow, since the validation feedback can be given in the background while the user is continuing his work. Even if the user is forced to wait for the validation result, he is probably accepting a delay of up to 10 seconds before he is going to perform other tasks [134]. Within this 10 seconds interval, it will be possible to validate even complex documents against CXSD schemas.

### 7.5.2. Runtime Measurements of the Template Instantiation

For all runtime measurements of the Template Instantiation component, the XTL Engine has been embedded within a servlet in an Apache Tomcat 6.0.26 Servlet Container [12], in order to enable the comparison of the instantiation times of XTL documents with the competing approaches JSP and XSL-T.

For the mere measurement of the time complexity of the instantiation process, i.e., to validate the complexity expression stated in Section 6.2, an XTL servlet has been implemented which renders a simple HTML representation from the XML representation of the plays of Shakespeare [27]. The servlet accesses the XML representation of a play using the JXPath PHP. A Perl script invokes this servlet using its Uniform Resource Locator (URL) and passing the name of the play

to be rendered. For each play, the URL is first invoked once, only to prepare the ground for the following instantiations, then 1000 times in order to determine the average instantiation time.

Figure 7.14 shows the result of this measurement of instantiation times, plotted against the size of the instantiation result. The linear dependency is easy to see and is confirmed by the coefficient of determination  $R^2$ . The measurement proves the time complexity proposed in Section 6.2 very well.

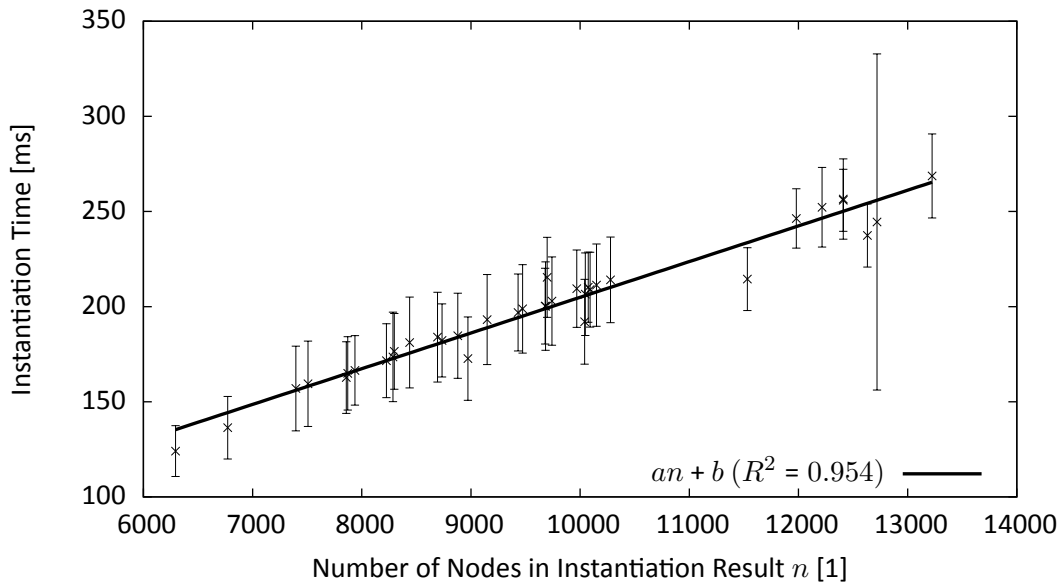


Figure 7.14.: Time Consumption during Template Instantiation

Unfortunately, the instantiation time measurement just presented does not say very much about the applicability of the Template Instantiation component in general. In order to validate this aspect, a comparison of the instantiation time with its most popular competitors, JSP and XSL-T, has been made. For this measurement, an XSL-T servlet and a JSP page have been implemented, which create the same HTML presentation from the plays of Shakespeare like the XTL servlet. In order to keep the influence of the Instantiation Data Evaluation process as low as possible, the JSP page closely resembles the XML access performed by the XPath PHP. In XSL-T, the XPath expression used to access the XML representation from the XTL servlet and the JSP page has been reused.

The JSP engine is the engine built into Tomcat 6.0.26, whereas the XSL-T engine used relies on the transformer API of the underlying Java Development Kit (JDK), which is a version of the Xalan XSL-T transformer.

A Perl script has been used to invoke the URL of each of the three rendering mechanisms. For each mechanism, the first retrieval of the HTML representation is only made to prepare the system for further measurements, followed by 1500 invocations to determine an average instantiation time.

## 7. Validation

Figure 7.15 plots the instantiation times needed by JSP, XSL-T and XTL against the size (in kbyte<sup>1</sup>) of the XML representation of a particular play. Again, the plot shows a linear dependency, which is simply caused by the fact that the dependency between the number of nodes of the instantiated template used as X axis in Figure 7.14 and the size in kbyte of the XML representation used as X axis here is itself linear.

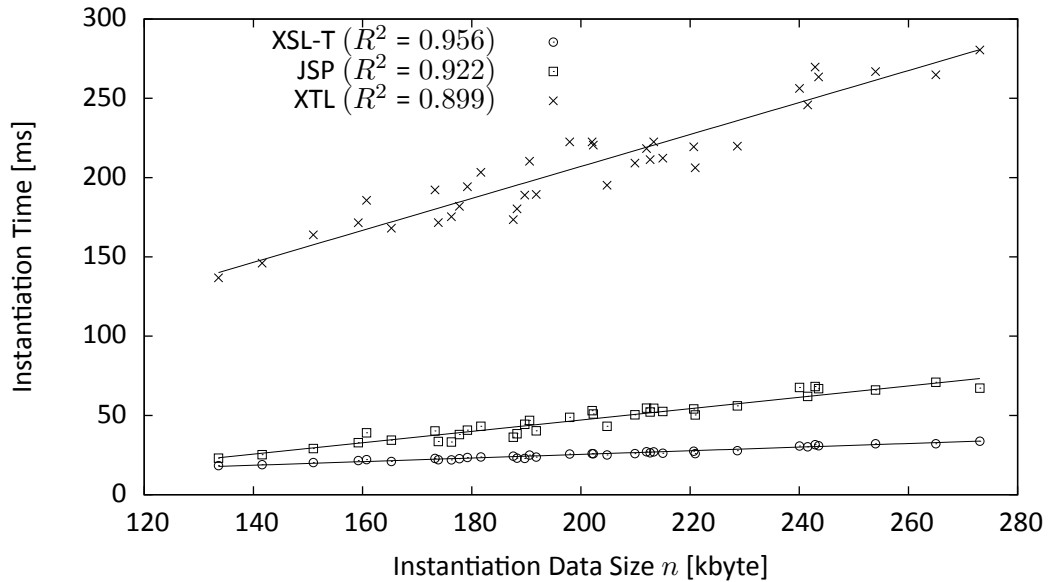


Figure 7.15.: Time Consumption Comparison between XTL, JSP, and XSL-T

The comparison shows that the XTL instantiation is in the same order of magnitude as the ones of JSP and XSL-T, even if XTL is obviously the slowest engine. The main reason for the difference in the instantiation time is that XTL templates are interpreted, whereas JSP pages as well as XSL-T stylesheets are compiled. The implementation of a compiling XTL engine would help closing this gap to the competing techniques.

### 7.5.3. Memory Usage Measurements of the Template Instantiation

For the memory usage, a special command line application has been constructed which outputs the maximum memory consumption during the instantiation. For this purpose, the application uses a thread which samples the heap memory usage every time a configurable amount of the instantiated template has been created. Both maximum and minimum memory usage are recorded. For each instantiation, the difference between maximum and minimum memory consumption is calculated. The heap memory usage is recorded using the `MemoryMXBean` mechanism of the JVM. Before the memory usage value is requested, the method `System.gc()` is invoked twice to give the JVM the chance to run the garbage collector in order to reclaim heap space that is not longer used and would severely influence the measurement results.

<sup>1</sup>The unit *kbyte* denotes 1024 bytes here and in the following.



In order to validate the memory complexity proposed in Section 6.2, special XTL templates have been constructed in a way that they can be parametrized in two ways. Each template consists of an `xtl:for-each` statement, executing three times and containing a parametrizable size  $n$  of `xtl:text` statements, which are creating a random text of 1024 characters using the Identity PHP. This `xtl:for-each` statement is prefixed and postfixed by a number  $p$  of elements, each of them containing a random text of between 0 and 1024 characters. An example template with the values  $n = 3$  and  $p = 2$  is shown in Listing 7.2 – the random text has been replaced by ... for better readability.

```
<?xml version='1.0'?>
<template xmlns:xtl='http://research.sap.com/xtl/1.0'>
  <text>...</text>
  <text>...</text>
  <xtl:for-each select='3'>
    <xtl:text select='...' />
    <xtl:text select='...' />
    <xtl:text select='...' />
  </xtl:for-each>
  <text>...</text>
  <text>...</text>
</template>
```

Listing 7.2: An Example Template for Memory Measurement ( $n = 3, p = 2$ )

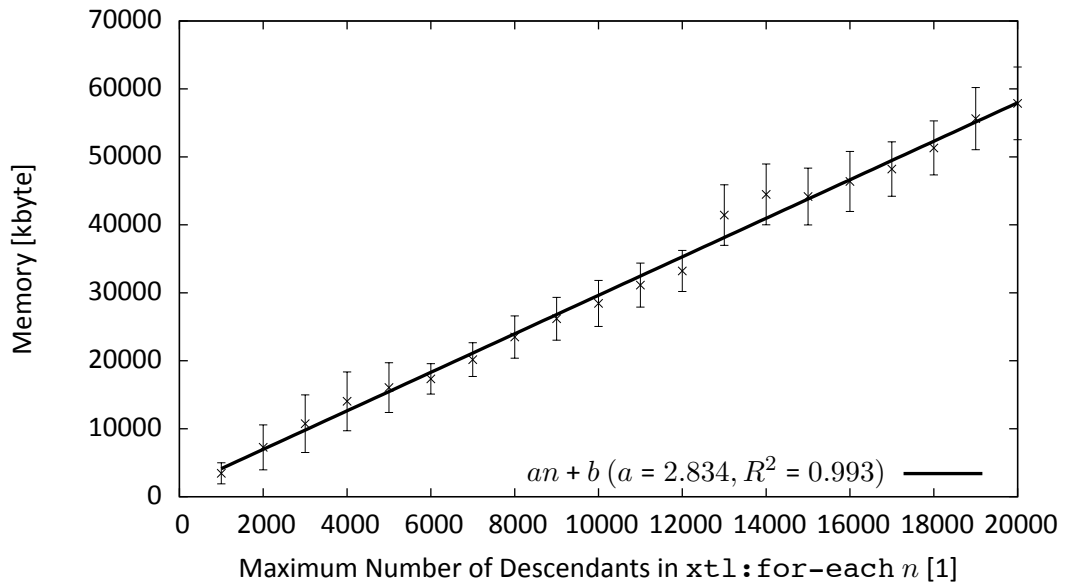
Using these parametrizable templates, the memory complexity is measured using a Perl script which calls the application 100 times for different values of  $n$  and 100 times for different values of  $p$ . The first invocation is only made to prepare the ground for the following instantiations, its memory consumption is ignored. The average of the following three instantiations is considered the memory consumption of this template.

In order to record only the amount of memory needed by the Template Instantiation itself, the template creation as well as the target of the Template Instantiation process had to be implemented in special ways. For the template creation, a special implementation of the `javax.xml.stream.XMLEventReader` interface has been created. This implementation delivers the next `XMLEvent` of the template without constructing the entire template upfront, thus saving the memory which would otherwise be needed to hold the template. For the target of the Template Instantiation process, a special subclass of the `java.io.OutputStream` has been used which immediately discards all bytes making up the instantiated template.

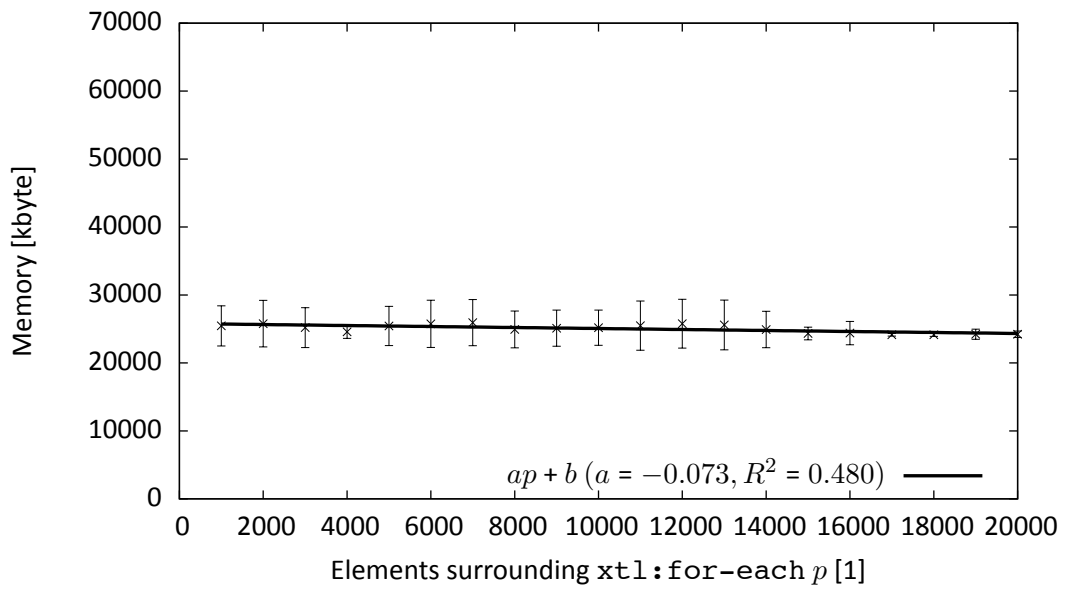
First, a measurement has been made with a fixed value of  $p = 10000$  and  $n$  ranging from 1000 to 20000. The result of this measurement is shown in Figure 7.16(a). The coefficient of determination  $R^2$  clearly shows the linear dependency of the memory usage from  $n$ . Therefore, the implementation follows the memory complexity expression proposed in Section 6.2, with  $n$  being the maximum number of nodes in an `xtl:for-each` in the template  $t^\circ$ , or, formally

$$n = \max_{x \in t^\circ // \text{xtl:for-each}} |x // \text{node}()|$$

## 7. Validation



(a) Variable `xtl:for-each` Size



(b) Variable Prefix and Postfix Size

Figure 7.16.: Results of the Memory Consumption Measurements

As a crosscheck, it has been measured how the memory consumption depends on the value of  $p$ , i.e., the number of elements containing random text prefixing and postfixing the `xtl:for-each` statement in the constructed templates. Figure 7.16(b)<sup>2</sup> shows a measurement for  $n = 10000$  and  $p$  varying from 1000 to 20000. There is no linear correlation between  $p$  and the memory consumption, as indicated by the slope close to 0. In other words, the memory usage does not depend on the number of elements surrounding `xtl:for-each` in the template: it is rather constant.

Taken together, these measurements empirically prove the correctness of the memory usage estimation given in Section 6.2.

## 7.6. Conclusion

This chapter introduced the means used for validating the results of this thesis. First, the validation means have been put into relation to the goals. Each goal has been validated using one or two validation means. The most basic validation means was the implementation of a prototype, which has been described in detail. Based on the prototype, a test suite has been presented which validates the particular components of the prototype as well as the overall process. Applications of the prototype in various research projects has been discussed. The correctness of the Constraint Separation process has been proved. Finally, a set of measurements has been made in order to validate the correctness of the time and memory complexity estimations given in Section 6.2.4.

---

<sup>2</sup>The scale in this figure has been set to the scale used in Figure 7.16(a) to achieve a better comparability. The reason for the (negligible) negative slope is that the memory consumption has been measured with a fixed sample rate for technical reasons. As the highest memory consumption occurs for a relatively smaller amount of time in larger templates, the peak is less closely approximated with larger  $p$ .

## 7. *Validation*

# 8

## Summary, Conclusion, and Outlook

Die Zukunft soll man nicht voraussehen wollen, sondern möglich machen.

Antoine de Saint-Exupery [45]

This chapter summarizes the results of this thesis in Section 8.1. A conclusion of the thesis follows in Section 8.2: this conclusion summarizes the main contributions made by this thesis. Section 8.3 contains suggestions for improvements which could be applied to the XML technological space. The final Section 8.4 shows future research directions, i.e., minor and major research questions or standardization efforts which could help advance the proposed approach into a state-of-the-art technology.

### 8.1. Summary

The main objective of this thesis was to create a technique, a process and tools which allow the use of templates to generate documents that belong to a particular, predefined target language. The results of this thesis are of use for all areas of applications where templates are used today, especially for code generation and for Web applications.

The preface in Chapter 1 is basically an outline of the thesis. It relates the use of templates to the SoC principle and discusses the problems of using templates in the way we use them today. These problems are illustrated using a motivating example. Based on this, the goals of the thesis are outlined. The contributions made by the thesis are described comprehensively. The chapter concludes with a short summary of related work and a description of typographic conventions used within the thesis.

## 8. Summary, Conclusion, and Outlook

Chapter 2 sets the foundations for the thesis: it defines the notion of a *template*, discusses the areas where templates in the defined sense are typically used as well as alternatives to using templates, and lists related research areas. Finally, a classification of template approaches is given. Chapter 3 proposes an approach to the problems found when using templates today. It defines the goals of the thesis and derives requirements from them. Afterwards, a proposal for an architecture and a process fulfilling the requirements is made. Based on this proposal, the following part of the thesis has been structured into three main chapters, dealing with the design of a suitable template language, with the support that can be given to the template author and with the components which are involved during the instantiation of a template.

Chapter 4 discusses the design of the universal, syntax- and semantics-preserving Slot Markup Language XTL. In its first part, general design discussion are described, followed by a description of the features of the XTL. Each instruction of the XTL is described by showing its syntax, its semantics and giving an example. The chapter concludes with some words about a translational semantics definition of the XTL and the use of the XTL as schema language.

Chapter 5 introduces the two components in the proposed architecture and process which support the template author. The first of these components is the Constraint Separation process, which combines the target language grammar with the slot markup language grammar and separates the constraints from this combined template language grammar into constraints which can be verified during authoring time of the template and constraints which must be checked during the instantiation time of the template. The second component is the Template Validation component, which actually validates a template against the template language grammar and verifies the authoring time constraints.

Chapter 6 discusses the three components involved in the actual instantiation of a template. The first component is the Instantiation Data Evaluation component which is responsible for evaluating the instantiation data referred to by the template. The second component is the Template Instantiation component itself, which is described in detail and for which a complexity estimation is given. Finally, the Instantiation Data Validation component is discussed, which is responsible for verifying the instantiation data constraints with the actual instantiation data.

Chapter 7 validates the proposed architecture and process using several means. First, the prototype implemented in order to demonstrate the feasibility of the approach is described. Furthermore, the test suite established for validating the prototype with respect to its various subcomponents is described. Applications of the prototype in various research projects are shown. The chapter concludes with a discussion of the correctness of the Constraint Separation process and with a summary of the measurements which have been conducted in order to verify the complexity statements given in Chapter 6 and to compare the XTL instantiation with competing approaches like XSL-T and JSP.

### 8.2. Conclusion

This thesis made several major contributions to today's use of the template approach. The first contribution is the definition of the XTL slot markup language itself. The language is universally usable to create templates for all XML dialects. It is syntax-preserving, i.e., it refrains from introducing a special slot markup syntax. It is also semantics-preserving, as it does not redefine

the semantics of its target language in any way. The preservation of the syntax as well as the semantics is achieved by relying on XML namespaces for the slot markup. The denotationally defined semantics of the XTL itself is also a novelty in the area of template languages, which are typically only described informally. By its clean design, the XTL already eliminates a typical problem occurring when XML documents are created using typical template approaches: as opposed to the existing approaches, an XTL template will always produce well-formed XML documents.

The Safe Authoring goal has been achieved with the thesis. The author of a template gets the highest possible safety that its template will actually instantiate into the target language. This is inherent to the design of the XTL and the design of the Constraint Separation component. The Constraint Separation process can also be parameterized to facilitate a Partial Templatization, which allows achieving an entanglement index of 0, which has been stated impossible in [143].

It has been shown that the Safe Instantiation goal can be achieved in two different ways. First, the Instantiation Data Validation can be executed as part of the template instantiation, i.e., by checking the instantiation data constraints after the instantiation data has been fetched from an instantiation data source. Second, a modification of the proposed architecture is possible that allows for creating interfaces for templates, which makes it basically impossible to pass invalid data into the templates.

Furthermore, the thesis formulated a new and concrete definition of the notion of a template and gave a new classification of template techniques. The definition is different from existing definitions and captures the intuitive use of the template term in the areas of code generation and Web applications more closely. This conformity with the intuitive meaning is primarily achieved by basing the definition on the prototypical nature of templates. The classification of template techniques is unique in the orthogonality of the introduced classification properties while it still exhaustively classifies every approach which is captured by the introduced template definition.

### 8.3. Suggested Improvements for XML Technologies

As this thesis has been set up with the goal of utilizing existing standards, some parts of the design and the implementation of the approach become very sophisticated. In order to make things easier, some improvements to the existing XML technology stack should be made. In the following, such suggestions are described shortly.

An XML schema language that supports regular languages is absolutely necessary in order to implement the proposed approach. There are two candidates for such a language: RelaxNG and—as has been shown through the work in Section 5.2—XML Schema itself. RelaxNG is a well-designed language that would fit nicely with the proposed approach, but its tool support is still (after 9 years of standardization) very poor. Furthermore, the complex transformation rules that have to be applied to validate documents against a RelaxNG schema make the construction of a JAXB-like binding framework difficult [88]. For these reasons, a further dissemination seems unlikely.

The approach to develop XML Schema into a full-featured regular tree grammar language is preferable. XML Schema has already a sufficient syntax to capture regular tree grammars. Only the UPA rule (inherited from SGML) prevents it to be used in such a way. The main motiva-

## 8. Summary, Conclusion, and Outlook

tion for the UPA—simple validation because of the minimal lookahead required—could still be considered in an advanced XML Schema version by introducing language profiles. Such profiles could easily allow distinct expressive powers of XML Schema based on the well-known syntax. The profiles suggested here are a legacy profile which leaves the UPA rule as is, a more powerful profile which allows deferring the particle attribution to extensions established using the `xsd:appinfo` mechanism, and a full profile that abandons the UPA. The second profile would be well-suited for XML Schema extensions like the CXSD or Schematron.

A problem which caused the exclusion of the substitution group feature from the list of XML schema features supported in target language schemas is the impossibility to extend content models during an extension of a complex type at the *beginning* of the content model. This problem occurs when a complex type without attributes is extended to a complex type with attributes: it is impossible to allow the use of the `xtl:attribute` instruction directly after the element which is declared using the inheriting complex type.

A minor improvement which could be made to XML Schema would be a feature that allows it to express attribute/element or content/attribute relationships as they are described in Section 5.1.2. While these relationships can be expressed well in CXSD, their description as a constraint does not allow a syntax-aware editor to offer features like code completion. A declarative solution within XML Schema would fix this problem and allow better editors to be built.

### 8.4. Future Research Directions

The developed tool chain supporting the introduced approach for Safe XML Processing could be extended to support a larger subset of XML Schema, e.g., the `all` content model and substitution groups (as far as this is not prevented by the fact that content models could not be extended at the beginning, see above). Such an extension should be considered a minor research issue, as no further general insights are to be expected from these extensions.

A more challenging and promising research question is the unification of document validation and generation using a single language, like it could be done using XTL (see Section 4.7). Interesting questions involve the expressive power achievable by such a combined language and the acceptance by users for both validating and generating documents.

There also exist similarities between (electronic) form processing and templates. A form can be considered a template to be filled by a human. It would be interesting to elaborate on the requirements a generic slot markup language has to fulfill in order to allow it to be used to express forms and whether the implemented prototype could easily be extended into a form processing engine. A possible approach would be to allow XTL instructions in XHTML documents to be rendered as input elements (within a Web browser or by server-side processing).

A generic slot markup language like the XTL should be an integral part of the XML technological space. A standardization by the W3C would be the method of choice to achieve this. This process would also allow to review the related specifications like XML Schema with respect to the requirements of Safe Template Processing, which is the only way to elegantly and lastingly implement the proposed approach and to help its dissemination as a state-of-the-art technology.





# Referenced XML Schemata and Instances

Umgangssprachlich wird von Schema F gesprochen, wenn etwas bürokratisch-routinemäßig, stereotyp, mechanisch oder gedankenlos abläuft. Der Ausdruck geht zurück auf die Vordrucke für die im preußischen Heer seit 1861 vorgeschriebenen so genannten Frontrapporte, auszufüllende Berichte über den Bestandsnachweis der vollen Kriegsstärke. Diese Vordrucke waren mit dem Buchstaben F gekennzeichnet. Bei der Kontrolle der Truppenstärke musste diese genau mit den Angaben im Vordruck übereinstimmen.

Wikipedia [190]

## A.1. XML Schema of XTL

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  targetNamespace="http://research.sap.com/xtl/1.0"
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xtl="http://research.sap.com/xtl/1.0"
>
  <xsd:annotation>
    <xsd:documentation>
      <p>
        This schema describes the <em>X</em>ML <em>T</em>emplate
        <em>L</em>anguage (abbreviated <em>XTL</em>), a collection
        of elements which can be used to markup slots in an XML document,
```

## A. Referenced XML Schemata and Instances

```
        thereby making the XML document a template.
    </p>
    <p>
        Besides relying on XML, XTL is language independent: the same
        constructs can be used to markup slots in an XHTML, an SVG or some
        other XML-based document.
    </p>
    <p>
        XTL is independent of particular mechanisms used to fetch the data into
        a template. XTL engines typically achieve this independence by
        implementing a plugin mechanism: for each mechanism used to fetch data,
        a corresponding plugin is needed. These plugins are called
        <em>placeholder processors</em>. In order to allow multiple plugins to
        be used within a single template, the
        <a href="attribute/realn.html">realn</a> attribute might be used.
    </p>
    <p>
        XTL supports a mechanism called <em>bypassing</em> which allows
        deferring the evaluation of XTL language constructs.
    </p>
    <p>
        The XTL language is intended to serve as a proof for the statement that
        a generic slot markup language is not only usable as a template
        language, but is also useful for schema validation, semi-static
        API-based generators and as part of an abstract UI language. Some of
        these use cases might redefine the semantics of XTL language elements.
        For example, in case of a semi-static API-based generator, the
        <a href="attribute/select.html">select</a> attribute is no longer
        interpreted as the hint where to get the data for the slot from, but
        rather as a hint on how to structure the API for the template.
    </p>
</xsd:documentation>
</xsd:annotation>

<xsd:element name="attribute">
    <xsd:annotation>
        <xsd:documentation>
            <p>
                This element can be used to create or overwrite an attribute at an
                element.
            </p>
            <p>
                The attribute is created or overwritten at the next element along
                the ancestor axis which does not belong to the XTL namespace.
                Attributes can be created at elements that are assigned to an
                XTL-bypass namespace.
            </p>
            <p>
                Between an attribute element and its direct parent element, only
                whitespaces, comments or other
                <a href=" ../element/attribute.html">attribute</a>
                elements are allowed.
            </p>
            <p>
                The name of the element to be created is taken from the
                <a href="#attr_name">name</a> attribute of this element. The
                value is fetched from the placeholder processor which is designated
                by the <a href="#attr_realn">realn</a> attribute by passing it the
                value of the <a href="#attr_select">select</a> attribute.
            </p>
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>
```

```

<xsd:complexType>
  <xsd:attribute name="name" type="xsd:QName" use="required">
    <xsd:annotation>
      <xsd:documentation>
        <p>
          The value of this attribute defines the name of the
          attribute to be created or overwritten. The attribute name
          might be prefixed in order to create a qualified attribute.
        </p>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attributeGroup ref="xtl:selectAttributeGroup" />
  <xsd:attributeGroup ref="xtl:realmAttributeGroup" />
  <xsd:attributeGroup ref="xtl:typeAttributeGroup" />
  <xsd:attribute name="mode" type="xtl:attributeModeType">
    <xsd:annotation>
      <xsd:documentation>
        <p>
          The mode defines the behaviour of the xtl:attribute, if
          the attribute to be created already exists or if multiple
          xtl:attribute commands with the same value of the name
          attribute exist.
        </p>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="result" type="xsd:string">
    <xsd:annotation>
      <xsd:documentation>
        <p>
          The result attribute exists for technical reasons and must
          not be used in an XTL template.
        </p>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>
</xsd:element>

<xsd:element name="text">
  <xsd:annotation>
    <xsd:documentation>
      <p>
        This element can be used to create text.
      </p>
      <p>
        If the element is used in a template, the element is replaced
        by the value which is returned by the placeholder processor
        for the value of its <a href="../attribute/select.html">select</a>
        attribute. The returned value is encoded, i.e., if markup is
        returned by the placeholder processor, it will be converted to text
        in the template. It is therefore (by intention) not possible to
        change the template structure using this element.
      </p>
    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType>
    <xsd:attributeGroup ref="xtl:selectAttributeGroup" />
    <xsd:attributeGroup ref="xtl:realmAttributeGroup" />

```

## A. Referenced XML Schemata and Instances

```
<xsd:attributeGroup ref="xsl:typeAttributeGroup" />
</xsd:complexType>
</xsd:element>

<xsd:element name="if">
  <xsd:annotation>
    <xsd:documentation>
      <p>
        This element allows the conditional inclusion of template parts.
      </p>
      <p>
        During expansion of the template, the
        <a href="../attribute/select.html">select</a> attribute is
        evaluated. For this element, the evaluation <strong>
        MUST</strong> return a boolean value. If the value is true, the
        content of the if element is included, if it is false, the content
        is not expanded.
      </p>
    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:any minOccurs="1" maxOccurs="1" processContents="strict"/>
    </xsd:sequence>
    <xsd:attributeGroup ref="xsl:selectAttributeGroup" />
    <xsd:attributeGroup ref="xsl:realmAttributeGroup" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="for-each">
  <xsd:annotation>
    <xsd:documentation>
      <p>
        This element allows the repeated inclusion of a part of the
        template.
      </p>
      <p>
        During the expansion process, the
        <a href="../attribute/select.html">select</a> attribute is
        evaluated. If the evaluation of this attribute yields null or an
        empty collection, the content of the for-each element is not
        expanded. If the evaluation yields a single object, the content
        of the for-each element is evaluated once. if the evaluation yields
        a non-empty collection, the content of this element is evaluated
        once for each element in the collection.
      </p>
      <p>
        It is important to note that placeholder processors might implement
        a context which captures the position of
        <a href="../attribute/select.html">select</a> expressions with
        respect to surrounding for-each elements. This way, the expansion
        of the content of the for-each element might yield different
        results for different elements in the collection mentioned above.
        An example for this behaviour is the XPath placeholder processor
        which implements a context similar to the XSL-T context, thereby
        allowing
        <a href="../attribute/select.html">select</a> expressions to be
        absolute or relative to the current position within the for-each
        collection.
      </p>
    </xsd:documentation>
  </xsd:annotation>
  <xsd:sequence>
    <xsd:any minOccurs="1" maxOccurs="1" processContents="strict"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="xsl:selectAttributeGroup" />
  <xsd:attributeGroup ref="xsl:realmAttributeGroup" />
</xsd:element>
```

```

</xsd:annotation>

<xsd:complexType mixed="true">
  <xsd:sequence>
    <xsd:any minOccurs="1" maxOccurs="1" processContents="strict"/>
  </xsd:sequence>
  <xsd:attributeGroup ref="xtl:selectAttributeGroup" />
  <xsd:attributeGroup ref="xtl:realmAttributeGroup" />
  <xsd:attributeGroup ref="xtl:minMaxAttributeGroup" />
  <xsd:attribute name="order-by" type="xsd:string" use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <p>
          This attribute can be used to specify a subquery which is
          used to sort the result of the query.
        </p>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="order" type="xtl:orderType" use="optional">
    <xsd:annotation>
      <xsd:documentation>
        <p>
          This attribute specifies the sorting order
          (ascending/descending).
        </p>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>
</xsd:element>

<xsd:element name="include">
  <xsd:annotation>
    <xsd:documentation>
      <p>
        This element can be used to include arbitrary markup.
      </p>
      <p>
        During the expansion of the template, the
        <a href=" ../attribute/select.html">select</a> expression is
        evaluated. It must return a single DOM node which replaces the
        include element. In contrast to the
        <a href=" ../element/text.html">text</a> element, the result of the
        evaluation is not encoded.
      </p>
    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType>
    <xsd:attributeGroup ref="xtl:selectAttributeGroup" />
    <xsd:attributeGroup ref="xtl:realmAttributeGroup" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="macro">
  <xsd:annotation>
    <xsd:documentation>
      <p>
        This element allows the definition of reusable template parts,
        so-called <em>macros</em>.
      </p>
    </xsd:documentation>
  </xsd:annotation>

```

## A. Referenced XML Schemata and Instances

```
<p>
    During evaluation, the content of the element is associated with
    the value of its <a href="#name">name</a> attribute. No processing
    of the content is performed during this association.
</p>
</xsd:documentation>
</xsd:annotation>

<xsd:complexType>
  <xsd:sequence>
    <xsd:any />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName" use="required">
    <xsd:annotation>
      <xsd:documentation>
        <p>
          This attribute uniquely identifies the macro for later use
          with the
          <a href="../element/call-macro.html">call-macro</a>
          element.
        </p>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:complexType>

<xsd:key name="macroName">
  <xsd:selector xpath="//macro" />
  <xsd:field xpath="@name" />
</xsd:key>
</xsd:element>

<xsd:element name="call-macro">
  <xsd:annotation>
    <xsd:documentation>
      <p>
        This element allows the invocation of reusable template parts
        called macros.
      </p>
      <p>
        The stored content of a <a href="../element/macro.html">macro</a>
        definition is looked up, embedded into the template and afterwards
        expanded by the template engine. Thus, the expansion of the macro
        does not influence the semantics of the instantiation in any way:
        it makes no difference whether a macro is used or whether the
        content of the macro is copied into all the places where it is
        called.
      </p>
    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType>
    <xsd:attribute name="name" type="xsd:NCName" use="required">
      <xsd:annotation>
        <xsd:documentation>
          <p>
            The value of this attribute identifies the macro which
            should replace this call-macro element.
          </p>
        </xsd:documentation>
      </xsd:annotation>
    </xsd:attribute>
```

```

</xsd:complexType>

<xsd:keyref name="macroNameRef" refer="xtl:macroName">
  <xsd:selector xpath="//call-macro" />
  <xsd:field xpath="@name" />
</xsd:keyref>
</xsd:element>

<xsd:element name="init">
  <xsd:annotation>
    <xsd:documentation>
      <p>
        This element allows to initialize a placeholder plugin with
        plugin-dependent data.
      </p>
      <p>
        The evaluation of this element yields nothing.
      </p>
    </xsd:documentation>
  </xsd:annotation>

  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:any />
    </xsd:sequence>
    <xsd:attributeGroup ref="xtl:realmAttributeGroup" />
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="attributeModeType">
  <xsd:list>
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="create" />
        <xsd:enumeration value="append" />
        <xsd:enumeration value="set" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:list>
</xsd:simpleType>

<xsd:simpleType name="orderType">
  <xsd:list>
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="ascending" />
        <xsd:enumeration value="descending" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:list>
</xsd:simpleType>

<xsd:attributeGroup name="selectAttributeGroup">
  <xsd:attribute name="select" type="xsd:string" use="required"
    form="unqualified">
    <xsd:annotation>
      <xsd:documentation>
        <p>
          The value of this attribute is used by a placeholder processor
          to acquire data for an element.
        </p>
      </xsd:documentation>
    </xsd:annotation>
  </xsd:attribute>
</xsd:attributeGroup>

```

## A. Referenced XML Schemata and Instances

```

        This data could either be used to fill a slot (e.g., for the
        <a href="../element/attribute.html">attribute</a> element) or
        to control the expansion of a template (e.g., for the
        <a href="../element/if.html">if</a>
        and the <a href="../element/for-each.html">for-each</a>
        elements).
    </p>
</xsd:documentation>
</xsd:annotation>
</xsd:attribute>
</xsd:attributeGroup>

<xsd:attributeGroup name="realmAttributeGroup">
    <xsd:attribute name="realm" type="xsd:NCName" use="optional"
        form="unqualified">
        <xsd:annotation>
            <xsd:documentation>
                <p>
                    This attribute is used by XTL to distinguish different
                    placeholder processors which might be used within a single
                    template.
                </p>
                <p>
                    <span style="color:red">
                        Unfortunately, no mechanism is defined yet to map from the
                        realm attribute value to a placeholder processor (which
                        might, for example, be defined by a Java class name). In
                        addition to this, the placeholder processors might need
                        some additional configuration (like the XML source for an
                        XPath placeholder processor).
                    </span>
                </p>
            </xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:attributeGroup>

<xsd:attributeGroup name="typeAttributeGroup">
    <xsd:attribute name="type" type="xsd:QName" use="prohibited"
        form="unqualified">
        <xsd:annotation>
            <xsd:documentation>
                <p>
                    This is a helper attribute which may be used to define the
                    type that is needed in order to fill a slot.
                </p>
            </xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:attributeGroup>

<xsd:attributeGroup name="minMaxAttributeGroup">
    <xsd:attribute name="min" type="xsd:nonNegativeInteger" use="prohibited"
        form="unqualified">
        <xsd:annotation>
            <xsd:documentation>
                <p>
                    This is a helper attribute which may be used to classify
                    the number of times a for-each instruction must be executed
                    at minimum.
                </p>
            </xsd:documentation>
        </xsd:annotation>
    </xsd:attribute>
</xsd:attributeGroup>

```



```

        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="max" type="xsd:allNNI" use="prohibited"
        form="unqualified">
        <xsd:annotation>
          <xsd:documentation>
            <p>
              This is a helper attribute which may be used to classify
              the number of times a for-each instruction may be executed
              at maximum.
            </p>
          </xsd:documentation>
        </xsd:annotation>
      </xsd:attribute>
    </xsd:attributeGroup>
  </xsd:schema>

```

Listing A.1: XTL Schema xtl.xsd

## A.2. Purchase Order Schema

```

<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="po.xsd"
  orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>

```

Listing A.2: Purchase order XML instance po.xml

### A.3. Purchase Order Instance

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN"
      fixed="US"/>
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="USPrice" type="xsd:decimal"/>
            <xsd:element ref="comment" minOccurs="0"/>
            <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="partNum" type="SKU" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
```

```
<xsd:restriction base="xsd:string">
  <xsd:pattern value="\d{3}-[A-Z]{2}" />
</xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Listing A.3: Purchase order XML Schema po.xsd

#### *A. Referenced XML Schemata and Instances*



## **Detailed Results of the Runtime and Memory Measurements**

Traue keiner Statistik, die du nicht selbst gefälscht hast.

*(Origin unknown)*

## B. Detailed Results of the Runtime and Memory Measurements

Parameter $n$ of Instance Document [1]	Validation Times [ms]			
	XSD		CXSD	
	$\varnothing$	$\sigma$	$\varnothing$	$\sigma$
100	757,3	35,3	1 208,2	36,5
200	781,1	38,5	1 288,6	28,9
300	795,5	27,5	1 353,0	29,7
400	815,6	32,9	1 420,6	28,0
500	837,2	23,1	1 513,4	56,5
600	858,9	21,8	1 574,3	35,6
700	897,7	31,8	1 729,6	54,5
800	945,3	32,3	1 862,8	73,3
900	967,5	31,7	1 961,3	68,5
1000	990,5	41,3	2 069,9	80,0
1100	1 009,1	32,9	2 169,2	95,0
1200	1 063,5	176,3	2 281,8	97,5
1300	1 051,4	31,9	2 310,1	103,0
1400	1 062,7	33,2	2 463,9	77,5
1500	1 158,6	41,6	2 582,4	82,1
1600	1 189,7	39,5	2 595,6	123,2
1700	1 229,0	45,6	2 641,8	107,9
1800	1 253,2	48,3	2 677,1	94,6
1900	1 257,6	37,0	2 710,7	97,3
2000	1 273,0	40,5	2 932,4	89,8
Number of measurements per line: 300				

Table B.1.: Runtime Measurement of Validation against a CXSD document

IDS Source File	Number of Nodes in the IDS Source File [1]	Instantiation Time [ms]	
		$\varnothing$	$\sigma$
a_and_c.xml	12 720	244,48	88,35
all_well.xml	10 042	192,04	22,30
as_you.xml	8 972	172,67	21,94
com_err.xml	6 294	124,07	13,35
coriolan.xml	12 632	237,39	16,58
cymbelin.xml	11 532	214,47	16,52
dream.xml	6 772	136,35	16,45
hamlet.xml	13 224	268,69	22,10
hen_iv_1.xml	9 680	200,26	19,93
hen_iv_2.xml	10 278	214,04	22,49
hen_v.xml	9 432	196,91	20,21
hen_vi_1.xml	8 736	182,21	19,21
hen_vi_2.xml	10 092	208,92	19,69
hen_vi_3.xml	9 742	202,89	23,20
hen_viii.xml	9 684	200,27	23,25
j_caesar.xml	8 880	184,69	22,36
john.xml	7 858	162,71	18,79
lear.xml	11 980	246,34	15,59
lll.xml	10 074	210,23	18,43
m_for_m.xml	9 698	215,42	21,02
m_wives.xml	9 970	209,41	20,34
macbeth.xml	7 938	166,47	18,27
merchant.xml	8 286	173,56	23,51
much_ado.xml	9 474	198,83	23,22
othello.xml	12 410	255,90	16,30
pericles.xml	7 508	159,45	22,42
r_and_j.xml	10 150	211,30	21,62
rich_ii.xml	8 226	171,59	19,44
rich_iii.xml	12 410	256,53	21,09
t_night.xml	9 148	193,19	23,67
taming.xml	8 438	181,13	23,89
tempest.xml	7 396	156,99	22,26
timon.xml	8 694	183,97	23,61
titus.xml	7 872	164,95	19,28
troilus.xml	12 216	252,22	20,93
two_gent.xml	8 298	176,50	19,93
win_tale.xml	10 048	206,53	21,71
Number of measurements per line: 1000			

Table B.2.: Analysis of the Time Complexity

## B. Detailed Results of the Runtime and Memory Measurements

IDS Source File	Size of IDS Source File [kbyte]	Instantiation Time [ms]		
		Ø JSP	Ø XSL-T	Ø XTL
a_and_c.xml	245,96	57,6	33,3	268,1
all_well.xml	204,78	43,2	25,1	195,2
as_you.xml	187,60	36,2	24,3	173,4
com_err.xml	133,60	23,1	18,3	136,8
coriolan.xml	253,97	66,1	32,1	266,8
cymbelin.xml	241,53	62,0	30,2	245,8
dream.xml	141,61	25,3	18,9	146,0
hamlet.xml	273,07	67,2	33,8	280,5
hen_iv_1.xml	209,90	50,4	25,9	209,2
hen_iv_2.xml	228,66	56,0	27,8	219,8
hen_v.xml	220,93	50,3	25,9	206,2
hen_vi_1.xml	191,82	40,3	23,8	189,4
hen_vi_2.xml	220,68	54,1	27,4	219,4
hen_vi_3.xml	215,01	52,5	26,2	212,3
hen_viii.xml	212,68	52,1	26,4	211,3
j_caesar.xml	179,20	40,7	23,5	194,3
john.xml	173,88	33,6	22,1	171,6
lear.xml	240,05	67,6	30,7	256,3
lll.xml	202,06	53,0	25,9	222,6
m_for_m.xml	197,94	48,8	25,6	222,5
m_wives.xml	202,28	50,9	25,8	220,5
macbeth.xml	159,22	32,8	21,5	171,5
merchant.xml	177,74	37,9	22,7	181,9
much_ado.xml	190,60	46,8	25,0	210,3
othello.xml	242,91	68,3	31,6	269,7
pericles.xml	165,29	34,4	20,9	168,1
r_and_j.xml	213,35	54,5	27,0	222,5
rich_ii.xml	188,29	38,4	23,1	180,3
rich_iii.xml	265,02	70,9	32,1	264,8
t_night.xml	181,68	43,2	23,8	203,3
taming.xml	189,71	44,5	22,9	188,9
tempest.xml	150,99	29,1	20,2	163,9
timon.xml	173,27	40,2	22,9	192,3
titus.xml	176,28	33,2	22,0	175,4
troilus.xml	243,54	66,9	30,9	263,5
two_gent.xml	160,73	39,0	22,1	185,6
win_tale.xml	212,01	54,6	27,1	218,3
Number of measurements per line: 1500				

Table B.3.: Comparison between JSP, XSL-T and XTL



Parameter $n$ of Template [1]	Memory Usage [kbyte]	
	$\varnothing$	$\sigma$
1000	3 437,8	1 550,2
2000	7 254,9	3 315,8
3000	10 734,2	4 245,4
4000	14 026,9	4 325,2
5000	16 056,6	3 660,7
6000	17 333,9	2 230,3
7000	20 167,5	2 479,0
8000	23 489,1	3 113,0
9000	26 176,8	3 152,7
10000	28 438,9	3 388,5
11000	31 134,3	3 230,9
12000	33 222,6	3 023,8
13000	41 440,3	4 455,6
14000	44 474,1	4 473,5
15000	44 174,7	4 180,5
16000	46 378,1	4 415,0
17000	48 214,3	4 008,9
18000	51 319,2	3 981,4
19000	55 630,1	4 577,3
20000	57 872,8	5 350,7
Value of $p$ : 10000		
Number of measurements per line: 100		

Table B.4.: Memory measurement with constant parameter  $p$

*B. Detailed Results of the Runtime and Memory Measurements*

Parameter $p$ of Template [1]	Memory Usage [kbyte]	
	$\varnothing$	$\sigma$
1000	25 450,1	2 952,3
2000	25 783,6	3 420,2
3000	25 199,4	2 932,8
4000	24 574,0	955,9
5000	25 435,9	2 887,1
6000	25 756,8	3 481,0
7000	25 929,7	3 389,2
8000	24 930,2	2 704,6
9000	25 112,7	2 655,6
10000	25 184,4	2 588,3
11000	25 483,8	3 625,3
12000	25 772,9	3 590,0
13000	25 594,1	3 663,7
14000	24 910,8	2 679,5
15000	24 334,7	928,8
16000	24 398,6	1 717,7
17000	24 139,6	196,7
18000	24 140,3	230,2
19000	24 225,1	743,7
20000	24 221,7	471,3
Value of $n$ : 10000		
Number of measurements per line: 100		

Table B.5.: Memory measurement with constant parameter  $n$

# List of Acronyms

<b>ACV</b>	Advanced Computer Vision [1]
<b>AOP</b>	Aspect-oriented Programming [63]
<b>API</b>	Application Programming Interface
<b>AST</b>	Abstract Syntax Tree [2]
<b>AWT</b>	Abstract Window Toolkit, a Java GUI framework
<b>BMBF</b>	Bundesministerium für Bildung und Forschung
<b>CMS</b>	Content Management System
<b>CST</b>	Concrete Syntax Tree (see <i>parse tree</i> in [2])
<b>CXSD</b>	Constraint XML Schema Definition Language [83]
<b>DOM</b>	Document Object Model [13]
<b>DSL</b>	Domain Specific Language
<b>DTD</b>	Document Type Definition [28]
<b>D3ML</b>	Device-Independent Multimodal Mark-up Language [75]
<b>DTML</b>	Document Template Markup Language [114]
<b>EADS</b>	European Aeronautic Defence and Space Company [58]
<b>EMF</b>	Eclipse Modeling Framework [57]
<b>EMMA</b>	Extensible MultiModal Annotation Markup Language [193]
<b>EMODE</b>	Enabling Model Transformation-Based Cost Efficient Adaptive Multi-modal User Interfaces
<b>ERB</b>	Embedded Ruby [50]
<b>EU</b>	European Union

## *List of Acronyms*

- FeasiPLe** Feature-getriebene, aspektorientierte und modellgetriebene Produktlinienentwicklung (German for Feature-driven, Aspect-oriented Product Line Development) [60]
- FIRST** Fraunhofer Institut für Rechnerarchitektur und Softwaretechnik [70]
- FMC** Fundamental Modeling Concepts [72; 109]
- GHC** Glasgow Haskell Compiler [84]
- GUI** Graphical User Interface
- HTML** Extensible Hypertext Markup Language [153]
- HTTP** Hypertext Transfer Protocol [62]
- IDC** Instantiation Data Constraint language
- IDE** Integrated Development Environment
- IDS** Instantiation Data Source (see Section 4.1)
- ISC** Invasive Software Composition [15]
- JAR** Java ARchive[177]
- JAXB** Java Architecture for XML Binding [155; 105]
- JDK** Java Development Kit
- JET** Java Emitter Templates [52]
- JSP** Java Server Pages [176]
- JSR** Java Specification Request
- JVM** Java Virtual Machine
- LISP** LISt Processing [170]
- MDA** Model Driven Architecture [128]
- MDSD** Model Driven Software Development
- MDT** Model Development Tools [54]
- MMI-F** Multimodal Interaction Framework [191]
- MOF** Meta-Object Facility [138]
- MRCP** Media Resource Control Protocol [91]
- MSL** Model Schema Language [31; 30]

**M2C** Model-to-Code transformation [43]  
**M2M** Model-to-Model transformation [43]  
**MVC** Model-View-Controller [154]  
**OCL** Object Constraint Language [136]  
**PDA** Personal Digital Assistant  
**PHP** Placeholder Plugin (see Section 6.1)  
**PSVI** Post-Schema-Validation Infoset [180]  
**QVT** Query View Transformation [140]  
**RAP** Rich Application Platform [55]  
**RelaxNG** Regular Language for XML Next Generation [39]  
**RSF** Reasonable Server Faces [186]  
**RWT** Rich Application Platform (RAP) Widget Toolkit  
**SAX** Simple API for XML [161]  
**SGML** Standard Generalized Markup Language [76]  
**SNOW** Services for Nomadic Workers [179]  
**SoC** Separation of Concerns [48]  
**SPARQL** SPARQL Protocol and RDF Query Language [151]  
**SPath** Path Language for XML Schema [126]  
**SQL** Structured Query Language  
**SSM** Simplified Stylesheet Module [107] (called *simplified syntax* in [36])  
**SSML** Speech Synthesis Markup Language [192]  
**ST** StringTemplate [143]  
**StAX** Streaming API for XML [95; 145]  
**SVG** Scalable Vector Graphics [61]  
**TAL** Template Attribute Language [196]  
**UML** Unified Modeling Language [139]  
**UPA** Unique Particle Attribution [180, Section 3.8.6]

### *List of Acronyms*

- URI** Uniform Resource Identifier [23]
- URL** Uniform Resource Locator [22]
- WML** Wireless Markup Language [90]
- W3C** World Wide Web Consortium <http://www.w3.org>
- XHTML** Extensible Hypertext Markup Language [4]
- XMI** XML Metadata Interchange [137]
- XML** Extensible Markup Language [28]
- XML-RPC** XML Remote Procedure Call [164]
- XPath** XML Path Language [38]
- XSD** XML Schema Definition [59; 180; 26]
- XSL** Extensible Stylesheet Language [19]
- XSL-T** Extensible Stylesheet Language (XSL) Transformations [36]
- XTL** XML Template Language
- XTM-P** XML Topic Maps for Procedures [103]
- XVCL** XML-based Variant Configuration Language [93]

# List of Figures

1.1. A typical Web Application can produce both valid and invalid XHTML Documents	10
1.2. The current Development Process for Templates . . . . .	11
2.1. Comparison of the Scopes of the Definitions of the Term <i>Template</i> . . . . .	22
2.2. Relations between Template and Target Language . . . . .	23
2.3. Template Technique and Template Life Cycle . . . . .	24
2.4. Formalization of the XML document in Listing 2.2 . . . . .	26
2.5. Classification of Schema Languages [simplified, based on 131] . . . . .	27
2.6. Comparison of the Alternatives with Templates . . . . .	34
2.7. Target Language Awareness of Slot Markup . . . . .	39
2.8. Sequence Diagrams of Push resp. Pull Strategy . . . . .	42
2.9. Categories of Query Languages . . . . .	43
3.1. Error Messages caused by JSP Pages . . . . .	51
3.2. Separation of Concerns in Different Scenarios . . . . .	52
3.3. Relations between Goals and Requirements . . . . .	54
3.4. Consequences of Insufficient or Exaggerated Expressiveness . . . . .	56
3.5. The Proposed Architecture . . . . .	58
3.6. Relations between Requirements and Solution Elements . . . . .	59
3.7. Relations between the Solution Elements and the Following Chapters . . . . .	60
4.1. Types of XML Transformation Pipelines . . . . .	80
4.2. Using a Vertical XSL-T Pipeline to Emulate the XTL Engine . . . . .	84
4.3. Schema Validation and Template Instantiation . . . . .	84
4.4. Similarity between Schema/Template and Instance . . . . .	85
5.1. Conclusion Enabled by the Constraint Separation Process . . . . .	88
5.2. Meta-model for the CXSD constraints . . . . .	95
5.3. Set Relations between Template and Target Language . . . . .	100
5.4. The Constraint Separation Processing Steps . . . . .	108
5.5. The Proposed Development Process for Templates . . . . .	118
6.1. Accessing Multiple Instantiation Data Sources Using Multiple PHPs . . . . .	124
6.2. Push- and Pull-Parser . . . . .	130
6.3. XTL Engine with Input and Output Streams . . . . .	131

## List of Figures

6.4. Examples of Read Window Operations' Execution . . . . .	133
6.5. The XTLEngine's Processing Pipeline . . . . .	136
6.6. The XTLEvent Hierarchy . . . . .	137
6.7. Activities during a Call to XTLEventReader.getNextEvent . . . . .	138
6.8. Activities during a Call to BypassProcessingReader.getNextEvent . . . . .	139
6.9. Activities during a Call to XTLPProcessingReader.getNextEvent . . . . .	140
6.10. Indentation Parts of the XTL Processing Pipeline . . . . .	147
6.11. State Chart of the IndentingXMLEventWriter . . . . .	148
6.12. State Chart of the SplittingOutputStream . . . . .	149
6.13. XTL Instantiation with enabled Instantiation Data Validation . . . . .	151
6.14. Architecture with Template Interface Generation . . . . .	153
6.15. The Object Model Deduced from the Template in Listing 6.16 . . . . .	155
6.16. The XPath Syntax Accepted by the Template Interface Generation Process . . . . .	156
6.17. The Tree of Property Descriptors Built from the Template Shown in Listing 6.16 . . . . .	159
7.1. Relations between Validation Means and Goals . . . . .	166
7.2. The Prototype's Tool Architecture . . . . .	166
7.3. Console Help of the <code>xtlsc.sh</code> Command . . . . .	167
7.4. Console Help of the <code>cxsdvalidate.sh</code> Command . . . . .	168
7.5. Console Help of the <code>xtlinstantiate.sh</code> Command . . . . .	168
7.6. Console Help of the <code>xtltc.sh</code> Command . . . . .	169
7.7. Constraint Separation Test Suite . . . . .	170
7.8. Template Validation Test Suite . . . . .	171
7.9. Template Instantiation Test Suite . . . . .	172
7.10. Template Interface Generation Test Suite . . . . .	173
7.11. Round-trip Test Suite . . . . .	174
7.12. The SNOW Architecture . . . . .	177
7.13. Time Consumption during Document Validation . . . . .	182
7.14. Time Consumption during Template Instantiation . . . . .	183
7.15. Time Consumption Comparison between XTL, JSP, and XSL-T . . . . .	184
7.16. Results of the Memory Consumption Measurements . . . . .	186



# List of Listings

1.1. A JSP Document failing to produce wellformed XHTML Documents . . . . .	11
1.2. A JSP Document producing a Document that is not XHTML (1) . . . . .	12
1.3. A JSP document producing a Document that is not XHTML (2) . . . . .	13
2.1. Origins of Fragments in a Template . . . . .	23
2.2. A simple XML file . . . . .	25
2.3. A BETA Form . . . . .	36
2.4. Frame Processing Example with XVCL . . . . .	38
2.5. Suppression of Newlines in XPAND . . . . .	45
4.1. Representation of XML documents in the Instantiation Semantics . . . . .	63
4.2. Definition of the IDS class . . . . .	63
4.3. Preamble of the Denotational Instantiation Semantics . . . . .	64
4.4. Semantics for Text, Comment and Element Nodes . . . . .	65
4.5. Semantics of <code>xtl:text</code> . . . . .	67
4.6. Example Use of <code>xtl:text</code> . . . . .	67
4.7. Semantics of <code>xtl:attribute</code> . . . . .	69
4.8. Example Use of <code>xtl:attribute</code> . . . . .	69
4.9. Semantics of <code>xtl:include</code> . . . . .	70
4.10. Example Use of <code>xtl:include</code> . . . . .	70
4.11. Semantics of <code>xtl:if</code> . . . . .	72
4.12. Example Use of <code>xtl:if</code> . . . . .	73
4.13. Semantics of <code>xtl:for-each</code> . . . . .	75
4.14. Example Use of <code>xtl:for-each</code> . . . . .	75
4.15. Semantics of <code>xtl:macro</code> . . . . .	76
4.16. Semantics of <code>xtl:call-macro</code> . . . . .	77
4.17. Example Use of <code>xtl:macro</code> and <code>xtl:call-macro</code> . . . . .	78
4.18. Example Use of Realms . . . . .	80
4.19. Bypassing Semantics . . . . .	82
4.20. Bypassing Example . . . . .	83
5.1. A Purchase Order with Potentially Dynamic Parts Highlighted . . . . .	89
5.2. A Purchase Order XTL Template . . . . .	91
5.3. The PurchaseOrderType from <code>po.xsd</code> . . . . .	91

## List of Listings

5.4. The Modified PurchaseOrderType, Allowing the Use of <code>xtl:attribute</code>	91
5.5. The USAddress Type from <code>po.xsd</code>	92
5.6. The Modified USAddress Type, Allowing the Use of <code>xtl:text</code>	92
5.7. The Modified PurchaseOrderType, Allowing the Use of <code>xtl:if</code>	93
5.8. A complete CXSD Element Declaration with an Embedded OCL Constraint	97
5.9. Expressing a Constraint from the XML Schema Specification with CXSD	97
5.10. Expressing a Constraint from the XSL-T 2.0 Specification with CXSD	98
5.11. An Instantiation Data Constraint in an XML Schema fragment	99
5.12. The ConstraintSeparationContext Interface	109
5.13. The ConstraintFactory Interface	111
5.14. Added <code>xsd:import</code> Statements	112
5.15. Top-level Declaration of a Previously Anonymous Simple Type	112
5.16. Choice between <code>comment</code> and <code>xtl:if</code>	114
5.17. Enabled <code>xtl:attribute</code> with IDC Constraints	115
5.18. A CXSD Constraint for Required Attributes	115
5.19. Enabled <code>xtl:text</code> for the Creation of the Content of the <code>zip</code> Element	116
5.20. A CXSD constraint for Simple Content	116
5.21. A Simple XHTML 1.0 File	117
5.22. Linked Instantiation Data Constraints Compared with Embedded PSVI	120
6.1. The PlaceholderPlugin Interface	125
6.2. The ReadWindow Interface	132
6.3. The LoopStack Interface	134
6.4. The MacroMap Interface	135
6.5. The PlaceholderPluginMap Interface	135
6.6. The InstantiationContext Interface	140
6.7. The <code>process</code> Method in <code>XTLText</code>	141
6.8. The <code>process</code> Method in <code>XTLIfStart</code>	142
6.9. The <code>process</code> Method in <code>XTLForEachStart</code>	143
6.10. The <code>process</code> Method in <code>XTLForEachEnd</code>	144
6.11. The <code>process</code> Method in <code>XTLMacroStart</code>	144
6.12. The <code>process</code> Method in <code>XTLCallMacro</code>	145
6.13. The <code>process</code> Method in <code>XTLInclude</code>	145
6.14. The <code>process</code> Method in <code>XTLInit</code>	146
6.15. A Template Instantiation Result before Splitting	149
6.16. Example Template for Template Interface Generation	154
6.17. The <code>retrievePropertyDescriptor</code> Method in the <code>AnalyzerPHP</code>	161
6.18. The <code>evaluateForEach</code> Method in the <code>AnalyzerPHP</code>	162
6.19. An Example for a <code>validate</code> Method Implementation	163
7.1. An Example Instance Document for Runtime Measurements	181
7.2. An Example Template for Memory Measurement ( $n = 3, p = 2$ )	185
A.1. XTL Schema <code>xtl.xsd</code>	201

*List of Listings*

A.2. Purchase order XML instance <code>po.xml</code> . . . . .	201
A.3. Purchase order XML Schema <code>po.xsd</code> . . . . .	203

*List of Listings*

# List of Tables

1.1. XML Namespaces and Prefixes . . . . .	17
B.1. Runtime Measurement of Validation against a CXSD document . . . . .	206
B.2. Analysis of the Time Complexity . . . . .	207
B.3. Comparison between JSP, XSL-T and XTL . . . . .	208
B.4. Memory measurement with constant parameter $p$ . . . . .	209
B.5. Memory measurement with constant parameter $n$ . . . . .	210

## *List of Tables*

# Bibliography

- [1] *ACV – Advanced Computer Vision*. Advanced Computer Vision GmbH – ACV, 2007. URL <http://www.acv.ac.at/start.html>.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
- [3] Michael Altenhofen, Thomas Hettel, and Stefan Kusterer. OCL support in an industrial environment. In *MoDELS'06: Proceedings of the 2003 international conference on Models in software engineering*, pages 169–178, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-69488-5.
- [4] Murray Altheim and Shane McCarron, editors. *XHTML™1.1 - Module-based XHTML, W3C Recommendation 31 May 2001*. The World Wide Web Consortium, 2001. URL <http://www.w3.org/TR/2001/REC-xhtml11-20010531/>.
- [5] Frank Anke and Falk Hartmann. Cocoon mit StAX – Pull-Parsing in einem SAX-basierten Framework. *JavaSPEKTRUM*, (3), 2006.
- [6] *ANTLR Parser Generator*. The ANTLR Project, 2008. URL <http://www.antlr.org/>.
- [7] *Apache Cocoon*. Apache Software Foundation, 2003. URL <http://cocoon.apache.org/2.1/>.
- [8] *XMLBeans*. Apache Software Foundation, 2004. URL <http://xmlbeans.apache.org/>.
- [9] *The XPath Component*. Apache Software Foundation, 2007. URL <http://commons.apache.org/jxpath/index.html>.
- [10] *XSLT-processor Xalan*. Apache Software Foundation, 2007. URL <http://xml.apache.org/xalan-j/index.html>.
- [11] *XML parser Xerces*. Apache Software Foundation, 2007. URL <http://xerces.apache.org/>.
- [12] *Apache Tomcat*. Apache Software Foundation, 2010. URL <http://tomcat.apache.org/>.

## Bibliography

- [13] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, and Lauren Wood, editors. *Document Object Model (DOM) Level 1 Specification, Version 1.0, W3C Recommendation 1 October 1998*. The World Wide Web Consortium, 1998. URL <http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/>.
- [14] Jeroen Arnoldus, Jeanot Bijpost, and Mark van den Brand. Repleo: a Syntax-Safe Template Engine. In Charles Consel and Julia L. Lawall, editors, *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007*, pages 25–32, Salzburg, Austria, 2007. ACM. ISBN 978-1-59593-855-8. doi: <http://doi.acm.org/10.1145/1289971.1289977>.
- [15] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. ISBN 3540443851.
- [16] Uwe Aßmann. Architectural styles for active documents. *Science of Computer Programming*, 56(1-2):79–98, 2005. ISSN 0167-6423.
- [17] Paul Bassett. Frame-Based Software Engineering. *IEEE Software*, 4(4):9–16, 1987.
- [18] Paul G. Bassett. *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997. ISBN 0-13-327859-X.
- [19] Anders Berglund, editor. *Extensible Stylesheet Language (XSL) Version 1.1, W3C Recommendation 5 December 2006*. 2006. URL <http://www.w3.org/TR/2006/REC-xsl11-20061205/>.
- [20] Alexandru Berlea and Helmut Seidl. fxt – A Transformation Language for XML Documents. *Journal of Computing and Information Technology (CIT), Special Issue on Domain-Specific Languages*, 2001.
- [21] Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Representing xml schema in uml - a comparison of approaches. In Nora Koch, Piero Fraternali, and Martin Wirsing, editors, *ICWE*, volume 3140 of *Lecture Notes in Computer Science*, pages 440–444. Springer, 2004. ISBN 3-540-22511-0.
- [22] T. Berners-Lee, L. Masinter, and M. McCahill. RFC 1738, Uniform Resource Locators (URL), 1994. URL <http://www.ietf.org/rfc/rfc1738.txt>.
- [23] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic Syntax, 2005. URL <http://www.ietf.org/rfc/rfc3986.txt>.
- [24] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *ASE*, pages 273–280. IEEE Computer Society, 2001. ISBN 0-7695-1426-X.
- [25] Lutz Bichler. Tool support for generating implementations of MOF-based modeling languages. 2003.



- [26] Paul V. Biron and Ashok Malhotra, editors. *XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004*. 2004. URL <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>.
- [27] Jon Bosak. *The Plays of Shakespeare in XML*, 2000. URL <http://xml.coverpages.org/bosakShakespeare200.html>.
- [28] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0, W3C Recommendation 10 February 1998*. The World Wide Web Consortium, 1998. URL <http://www.w3.org/TR/1998/REC-xml-19980210>.
- [29] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin, editors. *Namespaces in XML 1.0 (Second Edition), W3C Recommendation 16 August 2006*. The World Wide Web Consortium, 2006. URL <http://www.w3.org/TR/2006/REC-xml-names-20060816/>.
- [30] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler. MSL - a model for W3C XML schema. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 191–200, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-348-0.
- [31] Allen Brown, Matthew Fuchs, Jonathan Robie, and Philip Wadler, editors. *XML Schema: Formal Description, W3C Working Draft, 25 September 2001*. 2001. URL <http://www.w3.org/TR/xmlschema-formal/>.
- [32] Giordano Bruno. *Über die Ursache, das Prinzip und das Eine*. Reklam, Ditzingen, 1986. ISBN 3150051134.
- [33] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964. ISSN 0004-5411. doi: <http://doi.acm.org/10.1145/321239.321249>.
- [34] David Carlson. *Modeling XML applications with UML: practical e-business applications*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2001. ISBN 0-201-70915-5.
- [35] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- [36] James Clark, editor. *XSL Transformations (XSLT), Version 1.0, W3C Recommendation 16 November 1999*. The World Wide Web Consortium, 1999. URL <http://www.w3.org/TR/1999/REC-xslt-19991116>.
- [37] James Clark. An algorithm for RELAX NG validation. Web page, 2002. URL <http://thaiopensource.com/relaxng/derivative.html>.
- [38] James Clark and Steve DeRose, editors. *XML Path Language (XPath), Version 1.0, W3C Recommendation 16 November 1999*. The World Wide Web Consortium, 1999. URL <http://www.w3.org/TR/1999/REC-xpath-19991116>.

## Bibliography

- [39] James Clark and Murata Makoto, editors. *RELAX NG Specification, Committee Specification 3 December 2001*. Organization for the Advancement of Structured Information Standards, 2001. URL <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/>.
- [40] William Clinger and Jonathan Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM Press. ISBN 0-89791-419-8.
- [41] *C-Lab Homepage*. Cooperative Computing & Communication Laboratory, 2007. URL <http://www.c-lab.de/>.
- [42] John Cowan and Richard Tobin, editors. *XML Information Set (Second Edition), W3C Recommendation 4 February 2004*. The World Wide Web Consortium, 2004. URL <http://www.w3.org/TR/2004/REC-xml-infoset-20040204/>.
- [43] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*, oct 2003.
- [44] Waltenegus Dargie, Anja Strunk, Matthias Winkler, Bernd Mrohs, Sunil Thakar, and Wilfried Enkelmann. A Model-Based Approach for Developing Adaptive Multimodal Interactive Systems. In *Proceedings of ICSOFT 2007, 2nd International Conference on Software and Data Technologies*, pages 73–79, Barcelona, Spain, 2007. INSTICC Press.
- [45] Antoine de Saint-Exupéry. *Die Stadt in der Wüste*. Rauch Verlag, 2009.
- [46] Tom DeMarco and Timothy R. Lister, editors. *Software State of the Art: Selected Papers*. Dorset House Publishing Co., Inc., New York, NY, USA, 2000. ISBN 0932633145.
- [47] Andreas Diel. Lokalisierung internationaler Software am Beispiel der E-Business-Plattform enfinity der INTERSHOP AG: Modellierung der Daten und Geschäftsprozesse. Master's thesis, FH Jena, 2001.
- [48] Edsger W. Dijkstra. On the role of scientific thought. Published as [49], August 1974. URL <http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD447.PDF>.
- [49] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.
- [50] *Class: ERB*. Documenting the Ruby Language, 2008. URL <http://www.ruby-doc.org/stdlib/libdoc/erb/rdoc/classes/ERB.html>.
- [51] Desmond D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998. ISBN 0201310120.
- [52] *Java Emitter Templates*. Eclipse Foundation, 2007. URL <http://www.eclipse.org/modeling/m2t/?project=jet>.

- [53] *Model Development Tools, OCL subproject*. Eclipse Foundation, 2007. URL <http://www.eclipse.org/modeling/mdt/?project=ocl>.
- [54] *Model Development Tools*. Eclipse Foundation, 2007. URL <http://www.eclipse.org/modeling/mdt/>.
- [55] *Rich Application Platform*. Eclipse Foundation, 2007. URL <http://www.eclipse.org/rap/>.
- [56] *Xpand Code Generation Language*. Eclipse Foundation, 2007. URL <http://www.eclipse.org/modeling/m2t/?project=xpand>.
- [57] *Eclipse Modeling Framework Project (EMF)*. Eclipse Foundation, 2010. URL <http://www.eclipse.org/modeling/emf/>.
- [58] EADS. *EADS N.V.* European Aeronautic Defence and Space Company, 2007. URL <http://eads.com>.
- [59] David C. Fallside and Priscilla Walmsley, editors. *XML Schema Part 0: Primer Second Edition, W3C Recommendation 28 October 2004*. 2004. URL <http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/>.
- [60] *Forschungsprojekt FeasiPLe - Feature-getriebene, aspektorientierte und modellgetriebene Produktlinienentwicklung*. FeasiPLe Konsortium. URL <http://www.feasiple.de>.
- [61] Jon Ferraiolo, Jun Fujisawa, and Dean Jackson, editors. *Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation 14 January 2003*. The World Wide Web Consortium, 2003. URL <http://www.w3.org/TR/SVG11/>.
- [62] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616, Hypertext Transfer Protocol – HTTP/1.1, 1999. URL <http://www.ietf.org/rfc/rfc2616.txt>.
- [63] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.
- [64] Daniel Fötsch and Andreas Speck. XTC - The XML Transformation Coordinator for XML Document Transformation Technologies. In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 507–511, Washington, DC, USA, 2006. IEEE Computer Society.
- [65] Daniel Fötsch, Andreas Speck, and Peter Hänsen. The operator hierarchy concept for xml document transformation technologies. In Rainer Eckstein and Robert Tolksdorf, editors, *Berliner XML Tage*, pages 59–70, 2005. ISBN 3-9810105-2-3.
- [66] Daniel Fötsch, Andreas Speck, Wilhelm Rossak, and Jörg Krumbiegel. A concept for modelling and validation of web based presentation templates. In Otto K. Ferstl, Elmar J.

## Bibliography

- Sinz, Sven Eckert, and Tilman Issehorst, editors, *Wirtschaftsinformatik*, pages 391–406. Physica-Verlag, 2005. ISBN 3-7908-1574-8.
- [67] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, Boston, MA, USA, 2002. ISBN 0321127420.
- [68] Martin Fowler. Moving away from XSL-T, 2003. URL <http://www.martinfowler.com/bliki/MovingAwayFromXslt.html>.
- [69] Charles François. *International Encyclopedia of Systems and Cybernetics*. K.G.Saur, München, 1997.
- [70] *Fraunhofer Institut Rechnerarchitektur und Softwaretechnik*. Fraunhofer-Gesellschaft, 2007. URL <http://www.first.fraunhofer.de/>.
- [71] Alan Freedman. *The computer glossary: the complete illustrated desk reference (4th ed.)*. American Management Assoc., Inc., New York, NY, USA, 1989. ISBN 0-8144-7709-7.
- [72] *Fundamental Modeling Concepts*. The Fundamental Modeling Concepts Consortium, 2003. <http://www.f-m-c.org/> (visited 2006, May 29th).
- [73] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707: 406–431, 1993.
- [74] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
- [75] Steffen Göbel, Falk Hartmann, Kay Kadner, and Christoph Pohl. A device-independent multimodal mark-up language. In *INFORMATIK 2006: Informatik für Menschen, Band 2*, pages 170–177, 2006.
- [76] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press, 1990. ISBN 0-198-53737-9.
- [77] Joseph D. Gradecki and Jim Cole. *Mastering Apache Velocity*. Wiley Technology Publishing, New York, 2003.
- [78] *TU Graz*. GRAZ UNIVERSITY OF TECHNOLOGY, 2007. URL <http://www.tugraz.at/>.
- [79] Denise Gürer. Pioneering women in computer science. *SIGCSE Bull.*, 34(2):175–180, 2002. ISSN 0097-8418.
- [80] René Haberland. Transformation von XML-Dokumenten mittels Prolog. Großer Beleg, TU Dresden, 2006.
- [81] René Haberland. Vereinheitlichung von XML-Template-Expansion und Schema-Validierung. Master’s thesis, TU Dresden, July 2007.

- [82] Falk Hartmann. An Architecture for an XML-Template Engine Enabling Safe Authoring. In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 502–507, Washington, DC, USA, 2006. IEEE Computer Society.
- [83] Falk Hartmann. Ensuring the Instantiation Results of XML Templates. In Pedro Isaías, Miguel Nunes, and Joao Barroso, editors, *6th IADIS International Conference WWW/Internet*, pages 269–276, Vila Real, Portugal, 2007. International Association for Development of the Information Society. ISBN 978-972-8924-44-7.
- [84] *The Glasgow Haskell Compiler*. Haskell.org, 2010. URL <http://www.haskell.org/ghc/>.
- [85] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, Christian Wende, and Marcel Böhme. Generating safe template languages. In *Proceedings of the eighth international conference on Generative programming and component engineering*, GPCE '09, pages 99–108, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-494-2. doi: <http://doi.acm.org/10.1145/1621607.1621624>. URL <http://doi.acm.org/10.1145/1621607.1621624>.
- [86] Jakob Henriksson, Jendrik Johannes, Steffen Zschaler, and Uwe Aßmann. Reuseware — Adding Modularity to Your Language of Choice. *Journal of Object Technology*, 6(9): 127–146, October 2007. Special Issue. TOOLS EUROPE 2007.
- [87] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid (20th-anniversary Edition)*. Penguin Books, 1999.
- [88] Carsten Holzmüller. Entwicklung eines Java-XML-Binding-Frameworks auf der Basis alternativer XML-Metasprachen. Master's thesis, TU Dresden, July 2007.
- [89] Don Hopkins. Maximizing Composability and Relax NG Trivia. Blog entry, 2005. URL <http://www.donhopkins.com/drupal/node/117>.
- [90] *Open Mobile Alliance Homepage*. <http://www.openmobilealliance.org>, 2007. URL <http://www.openmobilealliance.org/>.
- [91] IETF. *A Media Resource Control Protocol (MRCP)*. The Internet Engineering Task Force, 2006. <http://www.apps.ietf.org/rfc/rfc4463.html> (visited 2006, October 4th).
- [92] JAlbum. JAlbum - the free web photo album software and photo gallery software, 2007. URL <http://jalbum.net/>.
- [93] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. Xvcl: Xml-based variant configuration language. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 810–811, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X.
- [94] *Java Specification Request JSR 173: Streaming API for XML*. Java Community Process, 2003. URL <http://www.jcp.org/en/jsr/detail?id=173>.

## Bibliography

- [95] *Java Specification Request (JSR) 173: Streaming API for XML*. Java Community Process, 2004. URL <http://www.jcp.org/en/jsr/detail?id=173>.
- [96] java-source.net. Open Source Template Engines in Java, 2007. URL <http://java-source.net/open-source/template-engines>.
- [97] *CodeModel*. java.net, 2010. URL <https://codemodel.dev.java.net/>.
- [98] Rick Jelliffe. Family tree of schema languages for xml. Blog, 2007. URL <http://www.oreillynet.com/xml/blog/images/SchemaFamilyTree.pdf>.
- [99] Rick Jelliffe, editor. *The Schematron Assertion Language 1.6*. 2002. URL <http://xml.ascc.net/resource/schematron/Schematron2000.html>.
- [100] *Jena—A Semantic Web Framework for Java*. The Jena Community, 2010. URL <http://jena.sourceforge.net/>.
- [101] Martin Johns. Towards practical prevention of code injection vulnerabilities on the programming language level, 2007.
- [102] JXP. Jxp introduction, 2006. URL <http://jxp.sourceforge.net>.
- [103] Kay Kadner and David Roussel. Documentation for Aircraft Maintenance based on Topic Maps. In *Leveraging the Semantics of Topic Maps*, pages 56–61, 2007.
- [104] Henning Kagermann. Transcript of SAPPHIRE’06 Keynote *Making IT Strategic to the Business*, 2006. URL [http://www.sap.com/community/pub/webcast/2006\\_05\\_SAPPHIRE\\_US/2006\\_05\\_sapphire\\_us\\_OR1186\\_transcript.pdf](http://www.sap.com/community/pub/webcast/2006_05_SAPPHIRE_US/2006_05_sapphire_us_OR1186_transcript.pdf).
- [105] Kohsuke Kawaguchi, Sekhar Vajjhala, and Joe Fialli, editors. *The Java™ Architecture for XML Binding (JAXB) 2.1*. 2006. URL <http://www.jcp.org/en/jsr/detail?id=222>.
- [106] Michael Kay. SAXON - The XSL-T and XQuery Processor, 2007. URL <http://saxon.sourceforge.net/>.
- [107] Michael Kay, editor. *XSL Transformations (XSLT) Version 2.0, W3C Recommendation 23 January 2007*. The World Wide Web Consortium, 2007. URL <http://www.w3.org/TR/2007/REC-xslt20-20070123/>.
- [108] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.
- [109] Andreas Knöpfel. FMC quick introduction. *FMC Publication*, 2003. URL <http://www.f-m-c.org/>.

- [110] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM Press. ISBN 0-89791-200-4.
- [111] Christian Krauß. Vergleich verschiedener Java/XML Binding Tools im Hinblick auf die Möglichkeit der Erzeugung halbdynamischer Dokumente. Master's thesis, TU Dresden, September 2007.
- [112] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Abstraction mechanisms in the BETA programming language. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 1983. ACM. ISBN 0-89791-090-7. doi: <http://doi.acm.org/10.1145/567067.567094>.
- [113] Ivan Kurtev. *Adaptability of Model Transformations*. PhD thesis, IPA, 2005. ISBN 90-365-2184-X.
- [114] Amos Latteier and Michael Pellatier. *The Zope Book*. New Riders Publishing, Thousand Oaks, CA, USA, 2001. ISBN 0735711372.
- [115] Andreas Laux and Lars Martin. *XUpdate - XML Update Language, Working Draft 14th September 2000*. 2000. URL <http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html>.
- [116] Dongwon Lee and Wesley W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(3):76–87, 2000.
- [117] Christopher Lenz. Push-Strategy Web Templating. Blog entry, 2005. URL [http://www.cmlenz.net/blog/2005/01/pushstrategy\\_we.html](http://www.cmlenz.net/blog/2005/01/pushstrategy_we.html).
- [118] Diego Lo Giudice. The State Of Model-Driven Development (Market Overview). Technical report, Forrester Research, Inc., 2007.
- [119] Henrik Lochmann. Towards Connecting Application Parts for Reduced Effort in Feature Implementations. In *Proceedings of 2nd IFIP Central and East European Conference on Software Engineering Techniques (CEE-SET 2007)*, Posen, Poland, October 2007.
- [120] Henrik Lochmann. *HybridMDSD: Multi-Domain Engineering with Model-Driven Software Development using Ontological Foundations*. PhD thesis, TU Dresden, 2009.
- [121] *Loquendo Vocal Technology and Services*. Loquendo, S.p.A., 2007. URL <http://www.loquendo.com>.
- [122] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62430-3.

## Bibliography

- [123] Murali Mani, Dongwon Lee, and Richard R. Muntz. Semantic data modeling using XML schemas. *Lecture Notes in Computer Science*, 2224:149–163, 2001.
- [124] M. D. McIlroy. Macro instruction extension of compiler languages. *Comm. Assoc. Comp. Mach.*, 3:214–220, April 1960. Reprinted as pp. 560-571 in *Programming Systems and Languages*, ed. S. Rosen, McGraw-Hill, 1967 and as pp. 512-528 in *Compiler Techniques*, ed. Bary W. Pollack, Auerbach, 1972.
- [125] Erik Meijer and Mark Shields. XML: A functional language for constructing and manipulating XML documents. (Draft), 1999.
- [126] Felix Michel. Representation of XML Schema Components. Master’s thesis, Computer Engineering and Networks Laboratory, ETH Zürich, Zürich, Switzerland, March 2007.
- [127] Russell Miles. *An Introduction to the AspectXML Concept*, 2004.
- [128] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group, 2003. URL <http://www.omg.org/docs/omg/03-06-01.pdf>.
- [129] Marvin Minsky. A Framework for Representing Knowledge. Technical report, Cambridge, MA, USA, 1974.
- [130] M. Murata. Hedge Automata: a Formal Model for XML Schemata. Web page, 2000.
- [131] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.
- [132] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of xml schema languages using formal language theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, November 2005. ISSN 1533-5399. doi: <http://dx.doi.org/10.1145/1111627.1111631>. URL <http://dx.doi.org/10.1145/1111627.1111631>.
- [133] Brian S O. Neill and Michael Rathjen. Tea template language. Technical report, Walt Disney Internet Group, 2001.
- [134] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, San Francisco, California, October 1994. ISBN 0125184069.
- [135] Dimitre Novatchev. Functional programming in XSLT using the FXSL library. In *Extreme Markup Languages*. 2003. URL <http://www.mulberrytech.com/Extreme/Proceedings/html/2003/Novatchev01/EML2003Novatchev01.html>.
- [136] *UML 2.0 OCL specification*. Object Management Group, 2003. URL <http://www.omg.org/cgi-bin/doc?ptc/03-10-14>.
- [137] *MOF 2.0/XMI Mapping Specification, v2.1*. Object Management Group, 2005. URL <http://www.omg.org/cgi-bin/apps/doc?formal/05-09-01.pdf>.



- [138] *Meta Object Facility (MOF) Core Specification, v2.0*. Object Management Group, 2006. URL <http://www.omg.org/docs/formal/06-01-01.pdf>.
- [139] *Unified Modeling Language*. Object Management Group, 2007. URL <http://www.uml.org/>.
- [140] *Meta Object Facility (MOF) 2.0 Query/View/ Transformation Specification*. Object Management Group, 2008. URL <http://www.omg.org/spec/QVT/1.0/PDF/>.
- [141] OpenOffice. *The OpenOffice Homepage*. OpenOffice.org, 2007. URL <http://www.openoffice.org/>.
- [142] Terence Parr. *The Complete ANTLR Reference Guide*. Pragmatic, Lewisville, 2007. ISBN 0978739256.
- [143] Terence John Parr. Enforcing strict model-view separation in template engines. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 224–233, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-844-X.
- [144] Terence John Parr. A functional language for generating structured text. Public Draft, 2006. URL <http://www.cs.usfca.edu/~parrt/papers/ST.pdf>.
- [145] Alessandro Costa Pereira and Falk Hartmann. Der Mittelweg - Lesen und Schreiben von XML-Dokumenten mit dem Streaming API For XML (StAX). *Java Magazin*, (7), 2006.
- [146] Alessandro Costa Pereira, Falk Hartmann, and Kay Kadner. A Distributed Staged Architecture for Multimodal Applications (Extended Abstract). In *Software Engineering 2007 (SE 2007). Lecture Notes in Informatics (LNI) 105. Copyright Gesellschaft für Informatik*, pages 255–256. Köllen Verlag, Bonn, March 2007.
- [147] Alessandro Costa Pereira, Falk Hartmann, and Kay Kadner. A Distributed Staged Architecture for Multimodal Applications. In Flávio Oquendo, editor, *ECSCA*, volume 4758 of *Lecture Notes in Computer Science*, pages 195–206. Springer, 2007. ISBN 978-3-540-75131-1.
- [148] Remko Popma. Introduction to JET. 2004. URL [http://www.eclipse.org/articles/Article-JET/jet\\_tutorial1.html](http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html).
- [149] Dirk Preising. Entwurf und Entwicklung eines Systems zur Unterstützung der Lokalisierung von Software für internationale Märkte. Master's thesis, HTWK Leipzig, 2001.
- [150] Oxford University Press. *Dictionary of computing (3rd ed.)*. Oxford University Press, Inc., New York, NY, USA, 1990. ISBN 0-19-853825-1.
- [151] Eric Prud'hommeaux and Andy Seaborne, editors. *SPARQL Query Language for RDF, W3C Recommendation 15 January 2008*. The World Wide Web Consortium, 2008. URL <http://www.w3.org/TR/rdf-sparql-query/>.

## Bibliography

- [152] Giuseppe Psaila. On the Problem of Coupling Java Algorithms and XML Parsers (Invited Paper). In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 487–491, Washington, DC, USA, 2006. IEEE Computer Society.
- [153] Dave Raggett, Arnaud Le Hors, and Ian Jacobs, editors. *HTML 4.01 Specification, W3C Recommendation 24 December 1999*. The World Wide Web Consortium, 1999. URL <http://www.w3.org/TR/1999/REC-html401-19991224/>.
- [154] Trygve M. H. Reenskaug. Models - Views - Controllers. Technical note, Xerox PARC, 1979.
- [155] Mark Reinhold. An XML DataBinding Facility for the Java™Platform, 1999.
- [156] Reuseware. The Reuseware Composition Framework, 2007. URL <http://www.reuseware.org/>.
- [157] R. J. Rodger. Jostraca: a template engine for generative programming, 2002. Position paper for the ECOOP2002 Workshop on Generative Programming.
- [158] Tavis Rudd, Mike Orr, and Ian Bicking. Cheetah: The python-powered template engine. *The Tenth International Python Conference*, 2001.
- [159] SAP - SAP Research Centers: CEC Dresden, Germany. SAP AG, 2007. URL <http://www.sap.com/about/company/research/centers/dresden.epx>.
- [160] Ilie Savga, Charlie Abela, and Uwe Aßmann. Report on the design of component model and composition technology for the Datalog and Prolog variants of the REVERSE languages. Research report IST506779/Linköping/I3-D1/D/PP/a1, Linköping University, 2004. URL <http://rewerse.net/deliverables/i3-d1.pdf>. REVERSE Deliverable.
- [161] *Simple API for XML*. The SAX project, 2007. URL <http://www.saxproject.org/>.
- [162] Nikita Schmidt and Corina Sas. Software usability: a comparison between two tree-structured data transformation languages. In *NordiCHI '04: Proceedings of the third Nordic conference on Human-computer interaction*, pages 145–148, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-857-1.
- [163] Uwe Schmidt et al. *Haskell XML Toolbox 8.5.0*. FH Wedel, 2010. URL <http://www.fh-wedel.de/~si/HXmlToolbox/>.
- [164] *XML-RPC Home Page*. Scripting News, Inc., 2007. URL <http://www.xmlrpc.com/>.
- [165] *Smarty Template Engine*. The Smarty Project, 2008. URL <http://www.smarty.net/>.
- [166] Sparx. MDA Overview – Whitepaper on using Enterprise Architect for MDA. Technical report, Sparx Systems, 2007. URL <http://www.sparxsystems.com.au/bin/MDA~20Tool.pdf>.

- [167] C. M. Sperberg-McQueen. *Canonical XML forms for post-schema-validation infosets: A preliminary reconnaissance*, 2002. URL <http://www.w3.org/2002/04/xmlschema-psvi-in-xml>.
- [168] C. M. Sperberg-McQueen. Applications of Brzozowski derivatives to XML Schema processing. In *Extreme Markup Languages®*, 2005.
- [169] Thomas Stahl and Markus Völter. *Modellgetriebene Softwareentwicklung*. dpunkt Verlag, March 2005. ISBN 3898643107.
- [170] Guy L. Steele. *COMMON LISP: the language*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1984. ISBN 0-932376-41-X (paperback). With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb.
- [171] James Stichnoth. *Generating Code for High-Level Operations through Code Composition*. PhD thesis, School of Computer Science, Carnegie Mellon University, August 1997.
- [172] Maximilian Stoerzer and Stefan Hanenberg. A classification of pointcut language constructs. In Lodewijk Bergmans, Kris Gybels, Peri Tarr, and Erik Ernst, editors, *Software Engineering Properties of Languages and Aspect Technologies*, March 2005.
- [173] *StringTemplate Template Engine*. The StringTemplate Project, 2008. URL <http://www.stringtemplate.org/>.
- [174] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. ISBN 0201700735.
- [175] Thorsten Sturm, Jesco von Voss, and Marko Boger. Generating code from uml with velocity templates. In Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook, editors, *UML*, volume 2460 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2002. ISBN 3-540-44254-5.
- [176] *JavaServer Pages Technology*. Sun Microsystems, 1999. URL <http://java.sun.com/products/jsp/>.
- [177] *JAR File Specification*. Sun Microsystems, 2008. URL <http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html>.
- [178] Philip Teale, Christopher Etz, and Michael Kiel. *Data Patterns (Patterns & Practices)*. Microsoft Press, Redmond, WA, USA, 2005. ISBN 0735622000.
- [179] SNOW. *SNOW: Services for Nomadic Workers*. The SNOW Consortium, 2007. URL <http://www.snow-project.org/>.
- [180] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn, editors. *XML Schema Part 1: Structures Second Edition, W3C Recommendation 28 October 2004*. 2004. URL <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/>.

## Bibliography

- [181] Henry S. Thompson, John Tebbutt, and Tony Cincotta, editors. *XML Schema Test Suite, Version 20 June 2007*. 2004. URL <http://www.w3.org/XML/2004/xml-schema-test-suite/>.
- [182] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201342758.
- [183] *Dresden OCL – OCL support for your modeling language*. TU Dresden, 2007. URL <http://www.reuseware.org/index.php/DresdenOCL>.
- [184] Kurt Tucholsky. Zur soziologischen Psychologie der Löcher. 1931. URL <http://www.textlog.de/tucholsky-psychologie-1931.html>.
- [185] *What is IKAT?* University of Cambridge, Centre for Applied Research in Educational Technologies, 2007. URL <http://www2.caret.cam.ac.uk/rsfwiki/Wiki.jsp?page=IKAT>.
- [186] *The RSF Framework*. University of Cambridge, Centre for Applied Research in Educational Technologies, 2007. URL <http://www2.caret.cam.ac.uk/rsfwiki/Wiki.jsp?page=Main>.
- [187] Markus Völter. *Jenerator - Generative Programming for Java*. 2001.
- [188] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000. ISBN 0596000278.
- [189] Karsten Wendland. *Der Template-Zyklus: Web-Templates im Spannungsfeld von schöpferischem Gestalten und einschränkender Zumutung*. PhD thesis, TU Darmstadt, Aachen, Germany, 2006.
- [190] *Schema F*. Wikipedia – Die freie Enzyklopädie, 2010. URL [http://de.wikipedia.org/wiki/Schema\\_F](http://de.wikipedia.org/wiki/Schema_F).
- [191] W3C. *W3C Multimodal Interaction Framework*. The World Wide Web Consortium, 2003. <http://www.w3.org/TR/2003/NOTE-mmi-framework-20030506/> (visited 2006, May 29th).
- [192] W3C. *Speech Synthesis Markup Language (SSML) Version 1.0*. The World Wide Web Consortium, 2004. <http://www.w3.org/TR/speech-synthesis/> (visited 2006, October 4th).
- [193] W3C. *EMMA: Extensible MultiModal Annotation markup language*. The World Wide Web Consortium, 2005. <http://www.w3.org/TR/emma/> (visited 2006, June 2nd).
- [194] *XMLUnit - JUnit and NUnit testing for XML*. XMLUnit Community, 2009. URL <http://xmlunit.sourceforge.net/>.
- [195] David H. Young. *Enhydra XMLC Java Presentation Development*. Sams Publishing, Indianapolis, IN, USA, 2002.

- [196] *Template Attribute Language*. The ZOPE Community, 2007. URL <http://wiki.zope.org/ZPT/TAL>.
- [197] Oliver Zschau. Glossar <http://www.contentmanager.de/>, 2007. URL [http://www.contentmanager.de/ressourcen/glossar\\_8\\_template.html](http://www.contentmanager.de/ressourcen/glossar_8_template.html).

## *Bibliography*

# Index

- abstract syntax, 25
- adaptation phase, **24**, 46, 58, 60, 152
- advice, 32
- analysis phase, **24**
- ANLTR, 44
- AOP, 16, 20 f., 32, 39, 179
- approximation, 100
- aspect, 32
- AspectJ, 16, 34
- AspectXML, 33
- AST, 33
- authoring constraint, 103
- authoring phase, **24**, 59 f., 92, 180
- authoring time, **24**, 56, 87 f.
  
- Bassett, Paul, 9
- BETA, 35 f., 240 f.
- bind
  - composition operator, 37
- broad applicability, 14, **53**, 55–59, 178, 180
- Brzozowski derivatives, 98
- build time, 13, 33
- bypassing, 62, 64 f., 78, 80, 82 f.
  
- C, 35
- C+, 9, 35 f., 53
- character data, 28, 88
- Cheetah, 31
- CMS, 16, 31, 52 f.
- Cocoon, 80
- computability, 55 f.
- concerns
  - crosscutting, 32
- concrete syntax, 25
- constraint separation, 58 f., 87–91, 93 f., 100 f., 103 ff., 107–112, 117 ff., 121, 150, 163, 166 f., 170 f., 174, 180, 187, 190 f.
- ContentHandler, 17, 129
- contract, 56 f.
- core, 32
- coverage
  - requirement, 55, 58
- CST, 33
- CXSD, 17, 93–101, 104, 108, 110–113, 115 f., 118–121, 166 ff., 170 f., 178, 180 ff., 192, 206, 215
  
- design phase, **24**, 58
- Dijkstra, Edsger W., 8
- DOM, 95, 119 f., 127, 129
- DTML, 30
  
- ease of use, 10, 15
- EMODE, 15, 78, 175, 179
- entanglement index, 15, 40, 53
- ERB, 30
- evaluateForEach, 74, 79
- evaluateIf, 72, 74
- evaluateInclude, 70, 74
- evaluateText, 66, 74, 79
- expressiveness
  - requirement, 56, 58 f., 62, 71, 129
- extend
  - composition operator, 37
- extensibility
  - ISC, 37
  
- form

## *Bibliography*

- BETA, 36
- fragment
  - ISC, 37
- fragment form, 36
- fragment group, 36
- fragment language, 36
- frame, 37 f.
- frame processing, 37 f.
- generation number, 81
- Haskell, 63, 86, 172
- hook
  - ISC, 37
- HTML, 24, 30 f., 33 f., 39, 43, 66, 182 f.
- HTTP, 41
- IDC, 17, 93, 98 f., 101, 108, 110 f., 114 f., 121, 151, 166 ff.
- IKAT, 41
- implementation phase, **24**
- independence of query language
  - requirement, 57 ff., 65, 69, 71, 124
- instantiation, **22**, 43, 50 f., 54, 56, 62, 69, 72, 81, 84
- instantiation data, 12–15, **22**, 32 f., 41, 43, 45 f., 50 f., 55 ff., 59, 63, 65–68, 70–74, 78, 82, 84, 88, 93, 98 f., 103 f., 114 f., 117 ff., 123, 129, 150 f., 153, 163 f., 172
- instantiation data constraint, 58 f., 87 f., 99, 103 f., 117 ff., 121, 150–153, 160, 163, 172, 174
- instantiation data evaluation, 59, 123, 129, 152, 163, 171, 183, 190
- instantiation data evaluator, 66, 71, 73, 78 f., 119, 123, 133, 135, 150
- instantiation data source, 43 f., 59, 63 f., 79, 83 ff., 125 f., 153, 163, 173 f., 207 f.
- instantiation data type safety
  - requirement, 56 f., 59, 155
- instantiation data validation, 13, 16, 59, 118, 123, 129, 150 ff., 163, 171, 190 f.
- instantiation data validator, 88, 98, 150
- instantiation phase, **25**, 59 f., 92, 150, 152, 180
- instantiation time, 13, 88 f.
- ISC, 35 ff., 239 ff.
- JAR, 173
- Java, 16, 34, 44 f., 86, 127, 131, 154 ff., 159 f., 162 f., 169, 171, 173
- JAXB, 13, 16, 33 f., 50, 53, 108 f., 111, 129, 154, 163, 173, 191
- Jenerator, 33
- JET, 31, 179
- join point, 32
- Jostraca, 45
- JSP, 9–13, 16, 30, 34 f., 39, 42 ff., 50 f., 53 f., 57, 127, 163, 180, 182 ff., 190, 208
- JSR 173, *see* StAX
- Jxp, 39
- JXPath, 57, 125, 127 f., 182 f.
- LISP, 35
- localization, 33
- macro, 15, 35
  - hygienic, 35
- markup, 88
- marshalling, 33
- MDA, 31, 95
- MDT, 94, 120
- meta language, *see* schema language
- mode (attribute), 68
- MOF, 32
- move copy of data, 124, 153
- M2C, 31, 52, 157, 179
- M2M, 31
- MVC, 9 f., 32, 40, 53, 56
- name (attribute), 68, 76 f., 94
- named block
  - frame processing, 37
- NMTOKEN, 12 f.
- obliviousness, 32
- OCL, 46, 94–98, 101, 115 f., 118, 120 f., 124, 167, 171, 181 f.



- order (attribute), 73
- order-by (attribute), 73 f.
- origin, 36
  
- partial templatzation, 15, 18, 41, 94, 116 f., 121, 191
- Perl, 30, 39, 181 ff.
- PHP, 123–128, 130, 134 f., 141 f., 145, 151 f., 154, 160 ff., 168 f., 171, 179 f., 182 f., 185
  - Identity, 126 f., 169, 171, 185
  - JXPath, 127 f., 169
  - SPARQL, 127 f., 169, 179
  - System, 128, 134, 169
  - UML, 169
  - XMLBean, 169
  - XPath, 127, 169
- pipeline, 78, 80 f.
- pointcut, 21, 32 f.
- pointcut languages, 21
- preprocessor, 35
- preservation
  - requirement, 54, 58 f., 66, 68, 71, 100, 165
- PSVI, 118 f.
- pull parser, 130
- pull strategy, 42, 124, 153
- push parser, 130
- push strategy, 42, 124, 153
  
- QName, 68, 95, 119, 157 f.
- query language, **23 f.**, 34, 39, 43 f., 46, 50, 57, 65 f., 69, 71 ff., 78, 82 f., 85, 123 f., 150, 163
  
- realm, 62, 66, 68, 70 f., 73, 78 ff., 124, 133
- realm (attribute), 78 f., 135
- RelaxNG, 28 f., 84 ff., 94, 138, 178, 191
- Repleo, 46
- RSF, 41
- run time, 33
- RWT, 33, 53 f.
  
- safe authoring, 13, 18, 36, **50**, 55 f., 58 f., 62, 67, 70, 87, 104, 116, 118, 170 f., 178, 180, 191
- safe instantiation, 14 ff., 50, **51**, 57, 59, 150, 170, 191
- safe template processing, 7, 13 ff., 18, 49, 53, 57, 180, 192
- SAX, 129 f.
- Saxon, 31
- schema language, 15, 27 f.
  - grammar-based, 27 f.
  - pattern-based, 27 f.
- schema type, 15
- Schematron, 28, 94, 192
- select (attribute), 65 f., 68, 70 f., 73 f., 78 f., 82 f., 85, 124–127, 135, 154, 159, 173
- separation
  - of concerns, 8 f., 14 ff., 31, 34, 43, **52 f.**, 55–59, 61, 65, 68 f., 117, 189
  - of constraints, 12
- separation rules, 40
- SGML, 8, 25, 191
- shell, 39
- slot, **21**, 37, 67
  - BETA, 36
  - ISC, 37
- slot markup, 21 f., 39, 62, 67, 190 f.
- slot markup language, 15, 18, **21 ff.**, 24, 40, 46, 56 ff., 60 ff., 66, 76, 84, 86, 129, 190, 192
  - design, 61
- slot markup language design, 57 f.
- Smarty, 30
- SNOW, 15, 18, 78, 175–179
- SPARQL, 15, 78, 124, 127 f., 171, 180
- SPath, 117
- SQL, 57
- SSM, 15, 24, 32, 34, 40, 42, 44, 55, 86
- ST, 10, 16, 30 f., 37, 41 f., 44 f.
- staged architecture, 80
- StAX, 119 f., 129 f., 138, 146, 163
- stylesheet, 24, 32, 42, 55, 80, 83, 86

## *Bibliography*

- TAL, 39, 71, 73  
target language, 20 f., **22**, 23 ff., 39 ff., 46, 50, 53 ff., 58 f., 61, 66, 71, 87 f., 94, 99 ff., 104, 118, 123, 150, 152 f., 170, 174, 180 f.  
Tea, 39, 44  
technological space, 16, 33, 35, 60, 152  
template, 9, 13, 15, 17, **20 f.**, 24, 32 f., 37, 40 f., 43 f., 46, 50, 53–59, 61 ff., 65 f., 73, 76, 78 f., 83 f., 86 ff., 104 ff., 117, 135, 150, 152 f., 158, 165, 170, 185  
    validation, 59, 87, 117 ff., 121, 151, 154, 166 f., 170 f., 190  
template author, 24, 88, 117  
template engine, 13, **22**, 43 f., 46, 55, 59, 61, 78 f., 81, 87, 123 f., 129 ff., 150, 152 f., 160  
template instantiation, 59, 115, 119, 123, 127, 129 f., 152, 154 f., 162 f., 166, 168, 171 f., 181 ff., 185, 190  
template interface, 152 f.  
template interface generation, 13, 15 f., 18, 46, 59, 98, 124, 152–157, 160, 163, 166, 172  
template language, 21, **21 f.**, 23, 54 ff., 58 f., 61 f., 88, 99 f., 104, 170 f., 174  
template validator, 87, 117 f., 120  
template view, 32  
test suite, 166, 170–174  
transform view, 32  
Tunncliffe, William W., 8, 52  
ttype (attribute), 119, 159  
  
UML, 94 ff., 123 f., 157  
Unix, 39  
unmarshalling, 33  
unparser, 14 ff., 53  
UPA, 28, 93 f., 98, 118 f., 121, 191 f.  
upfront verification, 12, 14 ff., 18  
URI, 17, 25, 63, 81, 112, 139, 149  
URL, 182 f.  
use of existing standards, **54**  
utilization of existing standards, 14, 53 ff., 58 f., 62, 87, 94  
  
validation phase, **25**  
variable  
    frame processing, 37  
variable interpolation, 30  
Velocity, 30, 42, 53  
  
W3C, 25, 27, 192  
weaving, 32  
Web 2.0, 30  
wellformedness, 9–12, 14, 40, 62, 104  
WML, 25, 53  
  
Xalan, 31, 127, 183  
Xerces, 33, 108, 118 f.  
XGrammar, 28, 108  
XHTML, 9–12, 14, 21, 25, 30, 33, 39, 50 f., 53, 88, 116 f., 153, 176, 192  
XMI, 32, 169  
XML, 8 f., 11 ff., 15 ff., 19, 21, 25–29, 31, 33 f., 39 f., 42 ff., 53 f., 57, 60–63, 65 ff., 69 f., 78, 80, 83 f., 88, 93, 96 f., 103 ff., 109, 113, 117 ff., 121, 123 f., 127, 129 f., 138, 140 f., 146, 149, 156 f., 163, 167–173, 176 f., 182 ff., 190 ff., 213 f.  
    dialect, 14, 25, 27, 31, 34, 53, 55, 58, 80, 87, 123, 178  
    namespace, 14, 16 f., 25, 39, 62 f., 65, 81, 110–113, 127, 139 f., 149, 178, 180, 191  
XML $\lambda$ , 13, 33  
XML binding tool, 33, 129, 152  
XML information set, 25  
XML Schema, *see* XSD  
XMLBeans, 16, 33, 108, 129  
XMLEC, 39, 66  
XPAND, 31, 33, 44 f., 57  
XPath, 15, 24, 33, 42, 44, 46, 57, 65, 67, 72, 74, 78, 83, 85, 95, 101, 118, 124, 127 f., 150, 154–160, 163, 171 f., 178, 183  
XSD, 14 f., 17, 19, 27 ff., 33, 40, 55, 58, 62, 68, 85 ff., 89 f., 93–102, 104, 108–112, 114, 117–121, 123, 149, 151, 154 ff.,

- 160, 167, 169 ff., 178, 180 f., 191 ff., 203, 206, 213
- XSL-T, 10, 15 f., 20 f., 24, 31 f., 34, 40, 42, 44, 50, 55, 62, 67, 71–74, 80, 83 f., 86, 89, 97 f., 124 f., 127, 149, 163, 172, 180, 182 ff., 190, 208, 228
- XTL
  - XML Schema, 17, 62, 99, 103
- XTL, 11, 14–18, 28, 44, 61–86, 88, 90, 93 f., 100 f., 105 f., 108, 110–113, 117 ff., 124–127, 130 f., 133–136, 138–141, 146, 151, 153 f., 157–160, 162, 166 f., 169–180, 182–185, 190 ff., 208
  - engine, 123 f., 128, 130 f., 136, 138, 149 f., 171 f., 182
- xtl:attribute, 67 ff., 72, 74, 77, 79, 90–94, 99, 101 ff., 105, 111, 114 f., 119, 125 f., 128, 141, 146, 151, 159, 178, 192
- xtl:call-macro, 76 ff., 86, 101, 135, 144, 179
- xtl:for-each, 66, 68, 71, 73 ff., 77, 79, 90, 93 f., 101 ff., 107, 112 ff., 119, 125 f., 128, 130 f., 133 f., 142 f., 150, 152, 154 f., 158 f., 162, 179, 185 ff.
- xtl:if, 71 ff., 90, 93 f., 101 ff., 107, 112 f., 119, 126, 128, 141, 151, 154, 156, 158, 179
- xtl:include, 69 f., 101, 126, 128, 145
- xtl:init, 76, 78 f., 126 f., 145, 180
- xtl:macro, 76–79, 86, 101, 130, 135, 144, 179
- xtl:text, 66 ff., 70 f., 79, 90, 92 ff., 101 ff., 105 f., 111 f., 115 f., 119, 125 f., 128, 134, 141, 151, 153 ff., 159, 178, 185
- XUpdate, 33
- XVCL, 38