
Department of Informatics and Mathematics
Programming Group

# Diploma Thesis

## Feature-Oriented Composition of XML Artifacts

Author:

Jens Dörre

March 11, 2009

Advisors:

Dr.–Ing. Sven Apel,

Prof. Christian Lengauer, Ph.D.

University of Passau
Department of Informatics and Mathematics
94030 Passau, Germany

**Dörre, Jens:**
*Feature-Oriented Composition of XML Artifacts*
Diploma Thesis, University of Passau, 2009.

To my parents.

# Abstract

One of the most important aims of software engineering is to provide programmers with simple views and abstractions to describe and understand the ever more complex systems they create. Since these abstractions should integrate artifacts from all steps of the software-engineering process, language-independent paradigms better suit this purpose. One such paradigm is Feature-Oriented Software Design (FOSD). The central concept of FOSD is the feature, a semantically cohesive unit of behavior. Using FOSD, you are is able to modularize these units, which normally crosscut the boundaries of modules written in modern programming languages, for instance.

Today, many of those artifacts are written using XML. XML, the eXtensible Markup Language, is the standard language for modeling semi-structured data, which means that it can represent virtually any data. This is unlike more formal models, which are only suited for structured data. Because XML facilitates automatic information interchange and because it is supported by an infrastructure with many tools, XML is widely used in software engineering.

The FeatureHouse approach only applies to grammar-based languages. For many XML-based languages, there are no grammars available, so they cannot be easily used with FeatureHouse. Therefore, we need a theory and tools to extend FOSD with support for XML (and in this way also XML with support for FOSD). This work approaches this goal by analyzing the theoretical backgrounds of FOSD and XML, designing and implementing a solution, and validating that solution in case studies.

We use the superimposition of Feature Structure Trees (FSTs) as the composition method for implementing FOSD. FSTs abstract away the details of artifacts and only model the hierarchical structure of these artifacts. So we first map artifacts from different languages to FSTs; for grammar-based languages, this has been done in prior work by processing special annotations in the language grammars in order to generate the mappers needed. As XML-based languages are typically processed using general XML parsers, there are no grammars for them; instead, the structural properties of XML-based languages are often expressed using an XML schema, for instance.

We have therefore created an annotation mechanism for XML schemas that enables us to specify how to compose different XML-based languages. We make use of two high-level XML libraries, TrAX (the Java API for tree processors written in the XSLT language), to transform XML artifacts to and from an XML

representation of FSTs, and JAXB, a Java-to-XML binding tool, to read and write those representations from within Java. Consequently, a solution has been designed based on previous work (FeatureHouse to implement superimposition) and these XML facilities. The user of our tool-set can annotate XML schemas to customize superimposition, and then compose XML (instance) artifacts using our tool. In this way, support for XML-based languages is available in a general tool-set for feature composition.

We have also conducted case studies in different XML languages that show the generality of the XML extension. In the Graph Product Line example, we have composed XHTML documentation alongside with the Java code described by it. In our second case study, we have superimposed diagrams in the visual language UML. We have also composed build scripts in the dynamic language of the rule-based `ant` tool. A lesson learned from our case studies is that schemas for XML languages are oftentimes not available, but usually they are not really needed either.

In conclusion, we can say that we have thoroughly analyzed the applicability of superimposition to XML Artifacts. We have for the first time integrated grammar-based and XML Schema-based languages in one tool to use one implementation of superimposition. We furthermore have successfully conducted case studies in three different XML languages which show the suitability of this approach taken.

# Contents

# List of Figures

# Glossary

**AST** . . . . . abstract syntax tree

**CIDE** . . . . . Colored Integrated Development Environment

**DTD** . . . . . Document Type Declaration

**FOSD** . . . . . Feature-Oriented Software Design

**FST** . . . . . Feature Structure Tree

**gCIDE** . . . . . generalized CIDE

**GPL** . . . . . Graph Product Line

**HTML** . . . . . HyperText Markup Language

**Infoset** . . . . . XML Information Set

**OMG** . . . . . Object Management Group

**PSVI** . . . . . post-schema-validation Infoset

**SE** . . . . . software engineering

**SGML** . . . . . Standard Generalized Markup Language

**UML** . . . . . Unified Modeling Language

**W3C** . . . . . World Wide Web Consortium

**XDM** . . . . . XPath Data Model

**XHTML** . . . . . eXtensible HyperText Markup Language

**XML** . . . . . eXtensible Markup Language

**XPath** . . . . . XML Path Language

**XSL** . . . . . eXtensible Stylesheet Language Family

**XSLT** . . . . . XSL Transformations

# Chapter 1

# Introduction

The work presented here is in the area of programming languages and software engineering. It tries to advance the very methods of computer science.

## 1.1 Motivation

Programming has never been an easy task, and this is all the more true for programming in the large.

Decades of programming language research have led to languages that allow to model programs after conceptions deeply grounded in the human brain, thus avoiding misconceptions. They have also had much success in creating languages that curb errors by not letting the programmer use "harmful" structures (that is, structures that lend themselves to creating errors).

Yet these prohibitions are sometimes *too* strict. This is the case, for example, for configuring program families at compile-time.

In many cases the prohibitions can be circumvented by other, unstructured mechanisms like preprocessors, ultimately leading to error-prone code. If the use of preprocessors is not possible, there is always the programmer's last resort of copying and pasting source code. The consequence is unmaintainable code.

So there is a need for an easy-to-grasp and safe mechanism for defining product families.

## 1.2 Problem Statement

In prior work [AKL09], major modern programming languages have already been examined and integrated. Batory et al. [BSR04] have shown that considering only program source code is not sufficient to compose software. You also need

to compose accompanying artifacts like documentation, visual models, and build scripts.

Because many of these are often written in XML languages today, we will cover this special class of languages in this work. Our goal is to research the possibilities of feature-oriented composition for XML artifacts.

We implement our ideas as a generic extension of FeatureHouse for XML languages. This enables us to test their applicability in practice.

## 1.3 Outline

The rest of this work is structured as follows:

**Section 2** lays out the foundations of Feature Orientation and XML. We explain the theoretical concepts of feature-oriented composition, which have been established in prior work. We also introduce XML, and shortly describe the existing XML languages that we will need to be able to apply superimposition to XML artifacts.

**Section 3** uses these foundations for the engineering of a solution. In this section, we analyse the challenges posed by XML languages, design a solution that responds to them, and discuss how we have implemented this solution.

**Section 4** is an evaluation of this solution by means of case studies. They show that the FeatureHouse extension implemented is applicable to different kinds of XML languages.

**Section 5** relates this work to other work done in this area, summarizes it, and gives some prospects on future work.

# Chapter 2

# Foundations

We first introduce the notion of *Feature-Oriented Software Design (FOSD)* used in this work. Then we give a short overview of XML. This will allow us to apply the concept of FOSD to XML artifacts in the next chapter.

## 2.1 Feature-Oriented Software Design

*generality of FOSD*

FOSD is a relatively new, language-independent paradigm for *software engineering (SE)*. As its name suggests, it is based on features: units of behavior (of a software system) that a user can perceive and distinguish between. These features are manifest in different kinds of software artifacts: in requirements descriptions as well as in implementation code, and in design documents as well as in user documentation.

*separate feature units*

We here use the FeatureHouse approach [AKL09]. In this approach, we try to separate the features into different files. This is in contrast to the *Colored Integrated Development Environment (CIDE)* [Käs07] approach, for example, which only provides separate views of a system already "composed" by hand. In this way, you have the possibility to design and program each feature in separation, and use any appropriate tool for it. But there is also the necessity for a composition method.

### 2.1.1 Feature Structure Trees (FSTs)

*abstract artifact model*

As our concept of a feature is so basic for our work, there is a need to formalize it. We use a hierarchical structure that we call *Feature Structure Tree (FST)*[AKL09][1]. An FST is a rooted ordered tree, where each node has two String-valued labels, type and name. Together, they serve as a two-part naming scheme.

[1] Because we want to extend the FeatureHouse tool implementation, we describe the more concrete FST version of this tool instead of the more conceptual version in this paper.

There are two kinds of nodes:

**NonTerminals** additionally have a list of their children nodes, therefore making up the structure of the FST.

**Terminals** can only be leaves of an FST; they additionally have three String-valued labels, a prefix (for special tokens), the body (content) and the name of a special composition method.

In this section, we will use Java examples to illustrate the concepts. We will simplify them to some degree to better show their most important characteristics. The first example is that of an FST for a Java Stack class, shown in figure 2.1.

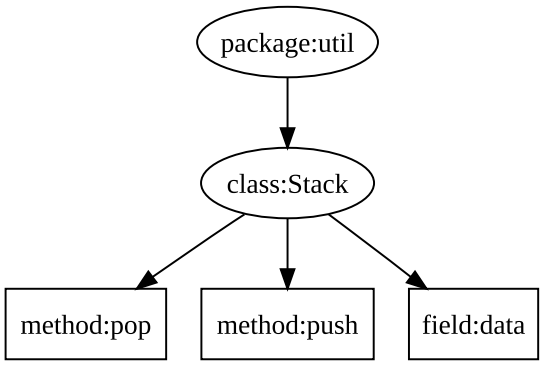


Figure 2.1: An FST for a Java class

Nodes are represented as `type:name` pairs inside ovals for NonTerminals, and inside boxes for Terminals.[2] Children are shown in clockwise order.

## 2.1.2 Superimposition of FSTs

We now need a composition mechanism to compose FSTs. Apel et al. [AL08] propose superimposition (of rooted trees). Superimposition can best be described as a recursive top-down algorithm. It composes two trees by merging their matching sub-trees, and appending any non-matching subtrees. The result is a new tree containing one node for each node that occurs in either input tree.

*top-down algorithm*

More formally, composition of two trees is defined as composition of their root nodes. Two nodes can only be composed if they are of the same kind (that is, they are either both NonTerminals, or both Terminals), and if their names and their types are also the same. This way, the kind, name, and type of a composed tree node can easily be defined as the common kind, name, and type, respectively, of the input nodes.

[2] This is a slightly adapted presentation of the examples in [AL08].

**Composition of NonTerminals** To compose two NonTerminals, their lists of children are merged in the following way: We walk the list of the right input, and for each entry we search for an entry in the list of the left input that has the same kind, name and type. If we find a matching entry, then we apply this algorithm recursively to it and the current right entry, and add the result to the children list of the output NonTerminal. Else, if we do not find a matching entry in the left input, we simply add the current right entry unchanged to the output. Finally, we add all entries from the left input which have never been matched.

**Composition of Terminals** To compose Terminals, there are different possibilities. The composition method to be used can be specified using the Terminal's compose field. The default composition method is called the `CompositionError` method, which simply raises an error when called.[3]

*algebraic properties*

Superimposition has some useful algebraic properties. It is associative, which allows you to choose the composition order when composing more than two trees in a row (and even to optimize execution time from linear to logarithmic by creating a balanced tree of function applications). In general, it is not commutative, as the order of the output matters. There are left and right identities to composition, namely the empty trees. Finally, as defined above, superimposition is idempotent (in the sense that you only have to take into account the *rightmost* of a number of the same features). Of course, these properties also have to be guaranteed by special composition methods for Terminals.

*Terminal composition methods*

There are many alternative composition methods (or rules) for Terminals. In general, they depend on the language to be composed, as Terminals with complex bodies may contain (and thus hide) much of a language's structure; therefore, for very coarse granularities, one has to simulate NonTerminal composition to some degree when writing new Terminal composition methods. But there are also simple composition methods, which are therefore more reusable. For example, `StringConcatenation` simply appends the left Terminal body to the right body. Although this composition method somewhat resembles NonTerminal composition, it differs from the latter in that it is not idempotent. The most invasive choice is `Replacement`, as it copies the left Terminal over the right one.

We will denote superimposition by "•". As an example, figure 2.2 shows the superimposition of the BasicStack feature from figure 2.1 with a TopOfStack feature. This feature adds a method to access the top of the stack.

[3] This is a safe default for two reasons: First, the user is forced to think about and choose an appropriate method. Second, if we use a granularity of FSTs that does not generate Terminal nodes (or at least no matching ones), then this algorithm is "pure" in the sense that it is fully described by this formalization.

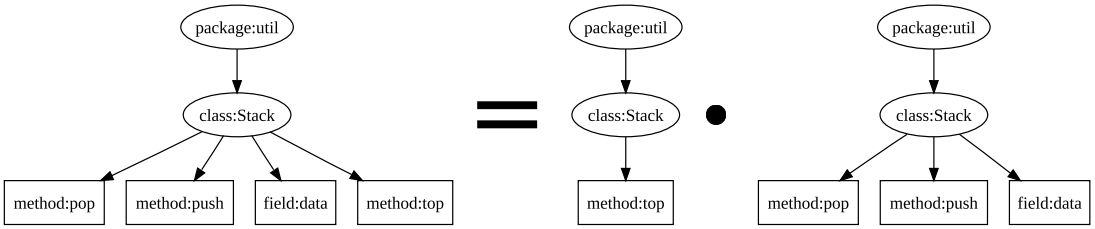


Figure 2.2: Composition of (Java) FSTs

### 2.1.3 Mappings to FSTs

Now we have to map the different kinds of artifacts to FSTs (and later map the FSTs back to their source formats). This mapping has to be done for every kind of artifact or language. At the inception of FSTs, *abstract syntax trees (ASTs)* generated by (existing) parsers have been used as an intermediate step [ALMK08]. You had to write (or generate) the parser, write an adapter from the resulting AST to FST, and also write a pretty-printer.

*inter-mediate ASTs*

```
1  package util;
2  //
3  import java.util.LinkedList;
4  class Stack {
5    LinkedList data = new LinkedList();
6    void push (Object obj) {
7      data.addFirst (obj);
8    }
9    Object pop() {
10     return data.removeFirst();
11   }
12 }
```

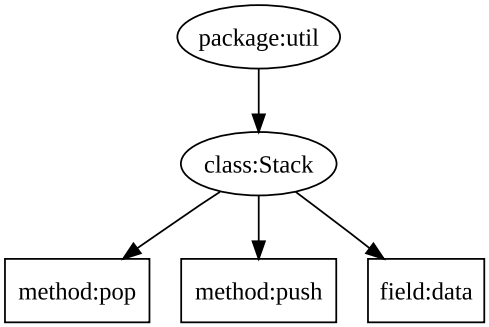


Figure 2.3: Java code and corresponding FST

In our example, the parser and adapter transform the Java code on the left of figure 2.3 into the FST on the right (which is the same as the FST shown in figure 2.1). The pretty-printer transforms the FST back to Java source code. As its name suggests, the pretty-printer is supposed to beautify the input it creates. Therefore, it is allowed to apply syntactic changes, producing a result that may not be equal to the original input, but only semantically equivalent to it.[4]

[4] A real *unparser* would produce the exact same result; however, this requirement complicates FST representation as well as composition, as all information from the input has to be stored in the FST, further preserved during composition, and taken into account by composition, to be finally unparsed back.

### 2.1.4 Generation of Mappings

With time, it has become clear that much of this work can be reused for all other kinds of artifacts, and Apel et al. [AKL09] have accordingly introduced a new technique which generates the parser and a matched pretty-printer from an FBNF specification. FBNF, short for Feature BNF (Backus-Naur-Form), is a language for expressing BNF grammars. It is a slightly adapted version of the parser input format used by JavaCC.

*customization by annotations*

FBNF heavily relies on annotations, a meta-programming technique provided by the underlying JavaCC, which in turn has also introduced annotations to be able to support full Java 1.5.

### 2.1.5 FeatureHouse

Everything described so far has previously been implemented (in Java) in the FeatureHouse (FH) tool-set. You can download[5] and use FeatureHouse to compose files written in Java, C#, JavaCC and other languages, or invest slightly more time and extend it for new languages which are ready for superimposition. The requirements which have to be fulfilled by the language in consideration are listed in section 3.1 of [AKL09]. An existing parser specification of the target language, written for a modern parser generator, significantly helps in avoiding errors and misconceptions during the extension process.

## 2.2 XML

*the XML Universe*

In this work, we aim to integrate XML-based languages into FeatureHouse. The *eXtensible Markup Language (XML)*[6] is at the base of an ever-growing family of markup languages conceived in the last decade. It has first been standardized in 1998 by the *World Wide Web Consortium (W3C)*[7]. The W3C publishes its standards as "recommendations" on the Internet. Nevertheless, they are true standards, and we will cite the permanent web addresses where they can be found like any other reference.

Markup languages are used especially on the Internet to tag parts of a text or of a multimedia document with predefined semantics. Put in another way, artifacts written in an XML language are self-describing in the sense that they consist of data marked up by tags defined by this XML language. Therefore not only

[5] http://www.infosun.fim.uni-passau.de/cl/staff/apel/FH/
[6] http://www.w3.org/TR/2008/REC-xml-20081126/
[7] http://www.w3.org/

is the data as obvious as in other textual encodings, but the tree structure is also visible; and the human reader may even guess the semantics, as the tags are also represented textually. XML therefore has to be able to also express semi-structured data, in contrast to fully structured database data or programming language code.

*semi-structured data*

We will use version 1.0 of the XML standard; there is also a newer version 1.1. Because that version is not as widely supported in combination with other standards, using it would unnecessarily complicate this work.

### 2.2.1 Syntax

The XML standard on its own specifies a "surface grammar" only, for example, how tags are syntactically written, how text is encoded and escaped, and that the structure always has to be that of a rooted (ordered) tree. Notably, it does not specify the logical (inner) structure of XML documents in any way.

We will illustrate the concepts in this section with examples in the language *eXtensible HyperText Markup Language (XHTML)*, as many people are familiar with that language. It is, for example, possible to render these XHTML artifacts using your favorite web browser. Figure 2.4 shows a simple XHTML file with a title, a heading, and a one-entry list.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE html
3        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
4        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
5  <html xmlns="http://www.w3.org/1999/xhtml">
6    <head>
7      <title>
8        A simple document (conforming to XHTML 1.0 strict).
9      </title>
10   </head>
11   <body>
12     <h1>
13       Here is an unordered list:
14     </h1>
15     <ul>
16       <li>
17         A first, basic entry.
18       </li>
19     </ul>
20   </body>
21 </html>
```

Figure 2.4: A simple XHTML artifact

The file begins with an *XML declaration*; this can be omitted if it is the default declaration, as is the case here; you can also use other text encodings or version 1.1 of the XML standard if you specify this. In fact, the XML declaration is an *XML Processing Instruction* that is interpreted in a special way: Processing Instructions are enclosed in "`<?`" and "`?>`", they start with their name, and they may contain arbitrary `key="value"` pairs, separated by whitespace. You can also employ *XML Comments*, which are written using "`<!--`" and "`-->`".

Next follows the *Document Type Declaration (DTD)* statement. In the example it specifies that the file is meant to be a document conforming to the W3C standard XHTML 1.0 Strict; it also points to a location where to find the corresponding DTD. Stemming from XML's more complex precursor SGML, DTDs define in essence how Elements may be nested. Although DTDs are included in the XML standard, and even if the validity of XML documents is defined with respect to them, their use is no longer recommended to specify new languages.

*DTD*

On line 5, there is start tag (`<html>`) of an `html` *Element*. The corresponding end tag (`</html>`) is on the last line, so the Element spans the whole XML document, and therefore it is the necessary single root Element. There is also an abbreviated syntax for empty Elements: `<html/>`.

*Elements*

Elements are the most important part of XML syntax. This has several reasons.

- They allow *XML Attributes* inside their start tags. Attributes are also written using the above-mentioned `key="value"` syntax; so, in principle, `xmlns="http://www.w3.org/1999/xhtml"` on line 5 is an example of an attribute.

  *Attributes*

- More importantly, they do not only allow Comments and Processing Instructions, but also (other) Elements and *XML Text* between the start and the end tag.

An example of Text is `A simple document` on line 8; we could also take as an example the whole content of the `title` Element from after the "`>`" character on line 7 to right before the "`<`" character on line 9 (including all whitespace). Examples of nested Elements are `head` and `body` inside `html`. You can use DTDs, for example, to define which Elements may be nested inside which Elements (and also, which Elements may contain Text).

### 2.2.2 Namespaces

On top of XML, there are many other standards, ranging from small, general-purpose extensions to big, specialized languages. One of the most basic extensions

are *Namespaces in XML*[8]. Namespaces have been introduced to allow for easy combination of data from different XML languages.

The names of Elements and Attributes are therefore restricted to a format following the `namespace-prefix:local-name` pattern. Here, no part may contain a "`:`" character. A stand-alone `local-name` is also allowed; then its namespace is the default namespace (for Elements), or the enclosing Element's namespace (for Attributes).

*namespace declarations*

Namespace-prefixes are references to URLs; to establish them, we need *namespace declarations*.Namespace declarations use attribute syntax, too, but they employ they special key `xmlns:namespace-prefix`. You can see this in line 5 of figure 2.4: `xmlns="http://www.w3.org/1999/xhtml"` This declares a new default namespace (which will then be used by Elements without prefixes).

The scope of a namespace declaration begins with the start tag of the declaring Element, and extends up to its end tag; namespace declarations may be overridden by other namespace declarations for the same prefix. You can also undeclare the default namespace (with the declaration `xmlns=""`). There is one implicit namespace present in all Elements, too; its explicit declaration would read `xmlns:xml="http://www.w3.org/XML/1998/namespace"`.

The Namespaces Standard states that the choice of prefix may not matter, so the only meaning of a prefix is the URL that it references. Furthermore, namespace prefixes are only interpreted for Element names and Attribute keys, but not inside Attribute values (or even inside Text).

We will use Namespaces version 1.0; there is also a (not as widely supported) version 1.1 of this standard, that allows to undeclare namespaces other than the default one.

In short, Namespaces are nothing but a two-part naming scheme that uses an additional indirection through namespace prefixes.

### 2.2.3 XPath Data Model (XDM)

We have just defined the syntax of XML, that is, which strings belong to the language defined by the XML grammar (or, to use XML speak, which strings are well-formed XML). Now we need a data model to represent the strings containing nested opening and closing tags as trees made up of nodes. This transformation from strings to trees is called parsing, and will be discussed in section 2.2.7.

[8] `http://www.w3.org/TR/2006/REC-xml-names-20060816/`

There are several different data models for XML, which are used by different XML processing tools. The *XML Information Set (Infoset)*[9] defines a logical view of an XML artifact. It contains, for example, information about the DTD, and a separate tree node for every character of XML Text.

*XML Infoset*

The *XPath Data Model (XDM)*[10], in contrast, does not contain any information about the DTD, and it concatenates adjacent strings to form XML Text nodes with maximum-length content. In fact, XDM can be considered a subset of the Infoset, as it can be created from the information contained therein (see the non-normative appendix[11]: XML Information Set Mapping).

*XDM*

In the XDM, there are seven different kinds of tree nodes. Text nodes—which we have already described—Element, Attribute, Namespace, Processing Instruction and Comment Nodes, and a single Document (or Root) Node. This extra node is needed, for example, to contain not only the root Element Node, but also possible top-level Comment and Processing Instruction Nodes in its list of child nodes.

The inner tree structure is made up of XML Element Nodes, each with a list of all its subtrees in input order. All the other five kinds of tree nodes may only appear as leaves. Yet Attributes and Namespace Nodes are special: their order does not matter, and there may be no duplicate keys; as a result, they form a set (of key-value pairs[12]) instead of a list. Note that Namespace Nodes represent in-scope namespaces, and not merely namespace declarations.

*Element Nodes*

### 2.2.4 XPath

*XML Path Language (XPath)*[13] is an expression language (that is, a functional language) for addressing parts of an XML document. We have already seen its data model in section 2.2.3.

Currently, there are two standardized versions of XPath. XPath 1.0 and XPath 2.0 may be considered to be different languages: XPath 2.0 is a large, not always backwards-compatible extension of XPath 1.0. For example, XPath 2.0 uses a richer data model with sequences in the place of node-sets. Since tool support for XPath 2.0 is substantially poorer that for XPath 1.0, we will use XPath 1.0 in this work.

*XPath 1.0*

The most important feature of XPath are its path expressions. They follow the example set by UNIX path expressions with wildcards. Paths expressions may be either relative to a context node or absolute. They consist of a series of steps,

*path expressions*

[9] `http://www.w3.org/TR/xml-infoset/`
[10] `http://www.w3.org/TR/xpath#data-model`
[11] `http://www.w3.org/TR/xpath#infoset`
[12] This means the set represents a map.
[13] `http://www.w3.org/TR/xpath`

separated by "/". Absolute path expressions start at the (document) root, which is why they have to begin with "/".

Path expressions are matched against the XDM of the input document. The set of matching nodes is then returned as the result.

### 2.2.5 XML Schema

schema languages

XPath can, for example, be used to specify integrity constraints in *schema languages* [MS06]. However, the principal use of schema languages is to further specify in a *schema* [MS06] the tree structure (mainly made up of XML Elements, as already discussed) of a class of XML documents. Such a schema extends what we have called the "surface grammar" of XML to something comparable in expressive power with a traditional grammar. In this way, we speak of an XML language as the class of documents that conform to a given schema.

(W3C) XML Schema[14], which is sometimes abbreviated WXS, is the W3C standard for schema languages today. It is meant to be a replacement for XML DTDs. XML Schema reproduces and extends the capabilities offered by DTDs. For example, it does not only allow to define the structure of XML documents, but also allows to constrain their contents by user-defined datatypes[15] (see [BNV04] for a detailed comparison of DTDs and XML Schema).

XML schemas are normally represented in XML 1.0 syntax; this means that we have to distinguish between schema and instance documents from now on.[16] XML Schema supports Namespaces 1.0 (unlike DTDs), and it uses them, for example, when combining schemas. At the time of writing, XML Schema version 1.1 is not yet finished. This is why we use version 1.0.

XML Schema uses the Infoset as its tree data model of instance documents. This guarantees their well-formedness. The validation of instance documents against a given schema does not only produce a boolean result (whether the instance is *schema valid*), but also enriches ("augments") the Infoset representation of this instance with type annotations and other information. The resulting Infoset is called the *post-schema-validation Infoset (PSVI)*.

Since XML Schema uses XML syntax, it is possible to use any XML tool to manipulate schema information. For example, it is easy to process schemas using *XSL Transformations (XSLT)*. (The *eXtensible Stylesheet Language Family (XSL)* "is a family of recommendations for defining XML document transforma-

[14] `http://www.w3.org/TR/xmlschema-1/`
[15] `http://www.w3.org/TR/xmlschema-2/`
[16] There is even a meta-level: schema documents are instances of the "Schema for Schemas", which is therefore a meta-schema.

tion and presentation."[17] It consists of XPath, XSLT, and XSL-FO, which is "an XML vocabulary for specifying formatting semantics".)

### 2.2.6 XSL Transformations (XSLT)

XSLT[18] is a declarative language for XML tree-to-tree transformations. The trees are XDM trees. XSLT uses XPath as its expression language to select parts of an XML document, for example, to specify matching parts of the document or to extract them.

The XSLT version corresponds to the XPath version it uses; it follows that there are even less implementations of XSLT 2.0 than of XPath 2.0. Although XSLT 2.0 has some real advantages over XSLT 1.0, these are merely theoretical without enough tool support. Therefore we will restrict ourselves to version 1.0 of XSLT.

*XSLT 1.0*

### 2.2.7 XML Parsers

XML languages pose a special problem: We have seen in section 2.2.1 and section 2.2.5 that they are not defined by grammars, but rather use a two-layer approach with a "surface grammar" and an additional schema. The powerful tool infrastructure for grammars (like parser generators, for example) can only be used in conjunction with traditional grammars. This means that it only has access to the "surface grammar" of XML languages, and not to the schema.

*traditional infrastructure not available*

Of course, you it would be perfectly possible to create a complete grammar for an XML language, but as these grammars can become very complex, this requires much extra effort. In consequence, there are hardly any grammars for XML languages.

The XML infrastructure is, of course, general in the sense that it can be employed with any XML language (and there are many of them!). From a general formal language perspective, this tool infrastructure is very special and restricted to only XML languages.

At the base of the XML infrastructure lie *XML parsers*. Like all parsers, XML parsers transform a string of characters into a tree representation. XML Parsers therefore have to build a tree, for example, an XDM tree, from the sequence of opening and closing tags found in an XML artifact.

*XML parsers*

Most parsers for widely-used modern languages are generated using a specification. This is the state of the art in compiler construction [GBJL02]. In contrast,

[17] `http://www.w3.org/Style/XSL/`
[18] `http://www.w3.org/TR/xslt`

existing XML parsers like Xerces-Java, for example, are written by hand. This means that you cannot easily modify the grammar and then simply re-generate the XML parser. Instead, you have to modify the complete parser by hand. Since XML parsers are very complex, you are likely to introduce errors in this process. Therefore we need to use XML parsers as they are.

*integration with FeatureHouse*

Because of all this, it is difficult to integrate an XML language into the grammar-based FeatureHouse tool-set. Neither of the two most obvious options can be considered to be a good approach, because:

- Writing a parser by hand for each language does not make much sense, as many XML languages are quite complex.
- Generating a parser from a grammar for an XML language is often impossible: Because there is not so much use for them, BNF grammars have rarely been developed for XML languages, The few existing grammars are often incomplete, because their correctness is not of utmost importance, given that XML documents need not be validated against them. This additionally restricts their use in a tool like FeatureHouse, which needs to preserve semantics while it transforms, composes and transforms back artifacts.

In both cases, the user would have to specify the generic XML parts *for every XML language to be integrated*, and would likely have to copy (and adapt) these parts, as there is no possibility to reuse them.

*integrate off-the-shelf XML parser*

It is not an option to write (and debug, and test ...) a new XML parser, as this is too complex a task; even adapting an existing XML parser does not appear to be a good idea. So we will have to integrate an existing (off-the-shelf) parser. It should be written in the same language as FeatureHouse to minimize incompatibilities; this means that we will later have to select the right Java parser for XML. We will have to use at least some information from a possibly supplied XML Schema to customize the mapping from artifacts written in some XML language to FSTs. This information may have originally been present in the input, or it may consist of user-supplied "annotations". In the latter case, we will need to allow and process these annotations.

# Chapter 3

# Extending FeatureHouse for XML Languages

In this section, we describe the implementation of superimposition for XML languages within the FeatureHouse tool-set. We build upon the foundations of these two areas set out in section 2.1 and section 2.2. We will first analyze whether and how superimposition can be applied to XML artifacts. After a short digression concerning the APIs for XML processing in Java, we use them to design a solution. In the end, we give an overview of its implementation.

## 3.1 Analysis

In this section, we will explore whether superimposition is applicable to XML artifacts.

XML trees and FSTs seem to be very similar, at least at first sight. Yet there are some intricacies, so we have to take an in-depth look at XML trees, how they can be annotated, and finally, how their structure can be specified.

### 3.1.1 XML Trees

XML models rooted ordered trees. As we have seen in section 2.2.3, there are different data models for XML with different tree representations of an XML artifact.

*XDM Trees*

Because of its frequent reuse in other XML standards, the XDM is the XML tree model best supported by XML processors. Although the XDM is the most limited one, it is also the most useful of the available data models, as it allows us to use any XML processing facility we may need. So we will use the XDM and therefore disregard all information which cannot be represented in this data model.

The XML Standard stipulates that the root of an XML document is the only XML Document (Root) Node in a tree, only XML Elements may be inner tree

nodes, and all five other kinds of XDM nodes, namely XML Text, Comment and Processing Instruction Nodes and XML Attribute and Namespace Nodes must be leaves in the tree. A document has to fulfill these requirements to be called *well-formed*.

*unique child names*

To apply superimposition, all *children* of a NonTerminal with the *same type* must have unique names. The reason for this is the following: The superimposition algorithm does not take into account the position of a node in the list of children when searching for a matching node. Therefore superimposition cannot distinguish between two nodes with the same name (and the same type and kind). As a result, a match will be established with *all* the children with the same name. This "wildcard matching" indeed amounts to quantification. Although the quantification is only local, as it is restricted to the (direct) children of a node, any quantification is contrary to the compositional approach we have chosen. A modification of the algorithm to use only the first (or perhaps the last?) match would also be rather arbitrary.

This "'unique child names' constraint" is not generally fulfilled in XML languages. For example, in XHTML documents, it is normal to have more than one second-level heading (`<h2>Some Title</h2>`) inside the same `body` Element; the same holds true for XHTML list elements (`<li>Some Item</li>`) inside of an `ul` Element representing an unordered list, for instance.[1]

*need for instance annotations*

Therefore we will have to resort to additional meta-data to be able to compose in XML instance documents in the correct way. We will directly embed this meta-data in annotations of the Elements in XML instance documents. An alternative is to put external meta-data in a separate file, but then you have to provide a mechanism to link the meta-data to the data they describe; these links tend to be little robust when the data documents are composed. Another problem is that they also need to be composed (and therefore (made) composable in the first place).

From a technical point of view, it is only necessary to annotate those nodes that are matched during superimposition; however, from a theoretical and engineering perspective, it is advisable to annotate all children that cannot be otherwise guaranteed to have a unique name. This allows for easy extension by superimposition without the need to first annotate the base instance documents.

This does not mean that you have to give names to *all* Elements of an XML artifact. For example, an existing schema may already guarantee that there is at most one child Element of the same type. In this case, there will be at most one matching Element during superimposition, so there is no need for annotating Elements of this type.

[1] You cannot even generate unique names from the content of a *whole subtree*, since it is perfectly legal to have, for example, two identical list items in XHTML.

### 3.1.2 Annotations for XML

We will now look into how XML artifacts can be annotated.

*mature grammar annotations*

Annotations in grammars and in programming languages (where the grammar annotations stem from) have in each case been introduced *after* the implementation of the corresponding base formalism, which means that the annotations are purely optional and use an independent syntax. They have matured enough now to be supported by many standard compilers and programming language tools, so you can easily use them at present.

*no XML annotations*

By contrast, there is no standard annotation mechanism for XML, at least not yet. This has several reasons:

- There is no pressing need for (additional) annotations, since XML Elements are themselves a kind of meta-data.[2] This is all the more true for XML Attributes.
- It is not easy to extend an XML parser to support (your own kind of) annotations because of the complexity of XML.
- There are several possibilities to use (and misuse) the standard XML syntax to add annotations.

So let us consider the possibilities for annotations available without modifying XML parsers.

*extra Elements*

The simplest way for annotations is to add extra Elements inside the XML structure, which, of course, will have to be automatically ignored by the application using the XML artifact. This is difficult to achieve, even for languages without a schema.

*Comments, Text*

Another possibility (available in almost all formal languages) is to use Comments, which provide for a totally unstructured way of adding information. In XML, you could—at least in principle—even use Text for annotations, if you can distinguish annotation content from normal data.

*Processing Instructions*

A more structured way offered by XML is to employ Processing Instructions. They are designed for machine use and will be automatically ignored by other applications. The downside to this is that you have to especially prepare your application for using them.

*Attributes*

Finally, you can use Attributes. This approach is in some ways comparable to adding extra Elements. Yet it is less invasive, because Attributes are often not as

[2] This is most obvious in the case of XHTML Elements structuring a text to be read by humans.

strictly specified as Elements, and because the path from an annotated Element to the Document Root Node remains the same. As a consequence, extra Attributes will often be ignored. The other advantage is that Attributes can be specified using schema languages like XML Schema. This allows for controlled annotation and for verification of the *annotations*.

There are some minor disadvantages to the use of Attributes as annotations: The most important one is the fact that an Element's attributes have set semantics. (This prohibits two attributes with the same key). Besides, XML poses restrictions on the possible values of Attributes.

*Namespace declarations*

We have not yet mentioned Namespace declarations for two reasons: Syntactically, they are equivalent to attributes, which we have already discussed above. Even worse, their semantics as declarations implies that they an XML processor may remove them if they are not used. This eliminates their use as annotations.

None of all these possibilities can be used to directly annotate anything but Elements. This is because the possible annotations are only allowed inside of Elements, and they therefore can only annotate these Elements in an easy way. Attributes are most closely tied to their parent Element (as they are written inside its start tag), which makes them ideal for annotating Elements.[3] As Elements are the most important constituent of a structured XML document, this advantage outweighs their disadvantages compared to Processing Instructions.

In conclusion, Attributes appear to be the most viable means of annotating XML. Still, annotation Attributes should not be over-used.

### 3.1.3 Specifications for XML Languages

Schema (definition) languages are rather restricted languages with the special purpose of specifying the structure of *instance documents* (or, simpler *instances*). Of course, you could use languages with a wider purpose, such as transformation or even query languages. However, they are not as adapted for this purpose as schema languages.

In the case of XML, schema languages specify the possible ways of nesting different XML Elements, and whether these Elements may contain Text or different Attributes. So schema languages internally use an XML data model consisting only of Elements, Attributes and Text. This is even more restricted than the XDM.

*alternatives*

The XML Standard defines the DTD as its schema language, and XML documents matching their (included or referenced) DTD are officially called *valid*.

[3] With some extra effort, you may use them to annotate Attributes, too.

Nevertheless, DTD declarations originate from XML's precursor SGML, and since they do not use XML syntax themselves, you need an extra tool-set to manipulate them. This is why other schema languages such as (W3C) XML Schema (sometimes abbreviated WXS), RELAX NG (RNG), and Schematron have been designed. RELAX NG[4] is standardized by the Organization for the Advancement of Structured Information Standards (OASIS), while Schematron is an ISO standard (ISO/IEC 19757 - Part 3). Of these alternatives, XML Schema is the most complex schema definition language. There are many reports on XML Schema being "too complicated" (see [MS06] for an example).

*XML Schema*

XML Schema is not only the official W3C standard, but it also appears to be the most widely used schema language [BNV04], and it will probably be the most important schema language in the future. We will use XML Schema as our schema language to be able to support many current and most future XML languages.

*schema-less languages*

Nevertheless, we have to take into account that even the Element structure of many XML languages cannot be *completely* specified using schema languages. Two examples of XML languages without schemas are:

**XSLT** allows embedded template content: transformations may include literal result Elements from the output language.

**ant** defines the names of Elements in the same build script in which they are used; in effect, it includes an extra schema language.

So we need to be aware that the use of schema languages is limited, and that there are some cases where a schema simply cannot be defined (and even more cases where a *simple* schema cannot be defined).

Let us now take a look at the facilities for processing XML artifacts available in the Java programming language. This will allow us in section 3.3 to design a solution as an extension of the Java-based FeatureHouse.

## 3.2 JAXP: XML Processing for Java

The Java API for XML Processing (JAXP) is the standard API (application programming interface) to do any XML-related processing in Java. It is available in recent versions[5] of Suns Java 2 Standard Edition (J2SE), which are used by most Java programmers. It defines Java interfaces, which may in turn be implemented by third-party software providers; there is also an implementation included with the JDK. Newer versions of these reference implementations may be obtained as

[4] `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`
[5] We use JDK1.6.

open-source software from the Internet. JAXP consists of several different APIs. We will describe in short the facilities they provide. Chapter 7 of [MS06] treats each of them in more detail.

### 3.2.1 Lower-Level APIs

To begin with, there are the rather low-level APIs for parsing XML (that is, converting the string of characters of an artifact into an XML tree).

*SAX2*

With the Simple API for XML processing, version 2, (SAX2), an event is pushed for each token in the XML input stream. The API user then has to write and register appropriate event listeners to be called upon reception of such an event. From a traditional compiler construction perspective, this merely corresponds to the first, scanner phase in compilation; yet the SAX2 API also allows for symbol table creation (which is needed for the namespaces feature) and even for validation of the input against a schema (which implies creating an AST).

*StAX*

The Streaming API for XML (StAX) is comparable to SAX2 in many ways, the major difference being its pull-based interface: The programmer requests the tokens she needs. This allows StAX to skip portions of the input.

*DOM3*

In contrast, the Document Object Model, version 3, (DOM3) is a tree-based API.[6] Its in-memory representation of an XML tree can be manipulated in-place, for example by modifying Elements or adding Attributes. As DOM3 is a language-independent standard of the *Object Management Group (OMG)* with very similar bindings to multiple programming languages, DOM3 is not very natural to use in Java.

All previously mentioned APIs have been conceived to be used as parsers for generic XML; therefore any well-formed XML file can be read with them. However, when it comes to *writing* XML files or streams (or *unparsing*, serializing, or pretty-printing, as it is called in different contexts), they do not propose an easy-to-use and standard way.

It is true that parsing (in general and of XML as a special case) is much more complex a task than unparsing, as parsing means deriving a tree structure by combining and nesting the elements in a list of tokens; nevertheless, unparsing can also be a nontrivial task, even though the structural transformation is a simple run through a tree that produces a list; the nontrivial part here stems from the prefix "un-" in "unparsing": unparsing is supposed to be the *inverse* of parsing, and it should therefore reverse all changes made during the parsing process. This algebraic property is of practical importance as it enables *round-*

[6] Of course, this does not mean that implementations may not use other representations such as tables internally, as is done for example, by some versions of the Apache Xerces parser.

*tripping*. Round-tripping means that you can start editing where you want, be it a tree or a list of tokens, apply any changes you like, and then all your changes will appear in both representations.

### 3.2.2 JAXB: Binding Java Classes to XML Schemas

This *round-tripping* capability is available with the Java Architecture for XML Binding (JAXB). To make it possible, JAXB has to restrict its data model of XML to allow only Elements, Attributes and Text. JAXB uses XML Schema to define the XML side of the binding (or mapping), and Java classes for the Java side. The specification of these structures also works either way: You provide one side, and JAXB automatically generates the code of the other side. Bindings may be customized using a separate[7] XML file (for schema-to-java binding) or annotations in the Java source files (for java-to-schema binding).

### 3.2.3 TrAX: Using XSLT from Java

In section 2.2.6, we have described XSLT, the standard for XML processing. The Transformation API For XML (TrAX) makes this language available in Java, as a scripting language with the special purpose of processing XML data. TrAX has the option of pre-compiling XSLT transformations to Java byte code (via Java source code), providing all the advantages of a compiler. Furthermore, the use of TrAX is the only way to convert XML representations created by one JAXP API into those used by another one, and TrAX provides an easy way of writing (unparsing) them to streams or files.[8] Unfortunately, the XDM used by XSLT is the most restricted data model for processing XML, so you might be losing information if the data model that you normally define the semantics on is more detailed.

## 3.3 Design

With all the background and tools needed available, we can now proceed to designing a solution for composing XML artifacts.

As we have learned, our desired input and output formats are different XML languages (which can be specified by XML schemas). So we will also have to unify the processing of the different XML languages to be supported. Because the

[7] This underlines the above-mentioned difficulty of annotating XML.

[8] To this end, you use the XSLT identity transform, which is created if you call an additional object constructor provided by TrAX.

composition shall be done by FeatureHouse, which uses a Java FST representation of the input, at some point we will have to adapt XML input to Java, providing for conversions to and from Java.

In this section, we will first choose the principal programming language to be used; We can then fix the data structures we will use, and determine what content they will have. Finally, we decide on how to represent the two kinds of annotations we need.

### 3.3.1 XSLT as the Primary Programming Language

We have to choose the programming languages to be used first, as this will have a considerable impact on the design. There are two natural options:

*domain-specific languages*

- domain-specific XML processing languages like XSLT (because the data uses this representation)
- the general purpose programming language Java (because it is already used by FeatureHouse)

XSLT offers some advantages in being especially designed for processing XML trees. It offers high-level treatment of XML transformations. As a language, it may not be as mature as Java, but there appear to be several implementations available, which are easily usable from Java within the TrAX API, as we have seen in section 3.2.3. Hence we will in general prefer XSLT to Java in this work.

The consequence of this choice is that we will do large parts of the processing using XML data.

### 3.3.2 XML-FSTs as the Principal Data Structure

*structure*

In order to have a clear separation between XML and Java processing, we introduce a new intermediate representation: XML-FSTs. They have the same structure as FSTs, yet they are not represented in Java, but in XML.

*content*

We specify their *content* by defining how XML-FSTs are filled with data from an XML tree. In essence, we need to lift XML up one level: We are effectively creating a generic (meta-)representation of XML in XML. There is only one kind of data to be treated in a special way: These are the annotations we will look at further below.

The next step is to decide how to represent (all) the information from the XDM, our data model, in (XML-)FSTs. Let us recall from section 2.1.1 that FSTs (and

therefore also XML-FSTs) use a two-part naming scheme, consisting of a type and a name for each node. Types are used as a classification mechanism for FST nodes; they could, for example, be used to select appropriate composition mechanisms for Terminals. Names, however, are only used as identifiers; together with the types they are compared to find a matching subtree during the composition process.

*names and types of FST nodes*

How should we now map the XDM node types and node names (possibly belonging to namespaces) to FST types and names? For XML nodes, it is clear that nodes belonging to different XDM node types are not interchangeable in a document. At first sight, you would expect that the names of XML Elements also correspond to FST types, as they can be defined in a schema, similar to the AST node types defined in a grammar. Surprisingly, XML Element names, for example, are in general *not* suited as a classification mechanism: XML Schema contains many concepts more targeted at data modeling than pure structural language definition. For example, an XML schema may directly allow two alternative kinds of Elements in a given context; it may specify a subtype which can take its place, (there are two different ways of deriving subtypes in XML Schema), or it may even allow substitution of one Element for another wherever it may occur; these possibilities even exist for Elements from different XML namespaces. To be able to extend this work with support for any of these XML Schema concepts, we therefore cannot make FST types of XML names (not even of the namespace part).

*XML names as FST types?*

So we propose the following approach: Each of the seven node types of the XDM will be represented as a separate (XML-)FST node type, while the names in the FST will be derived from the names of the XML nodes. For the anonymous node types Comment and Text, a new name will have to be generated. XML namespace prefixes will be additionally preserved, as they are the first part of the XML names of Elements and Attributes. (Namespace *declarations* will be included as all in-scope namespaces from the XDM will simply be added as nodes, in this way the fully qualified names can be reconstructed.) In this way, there will be only seven different types of FST nodes for all XML languages. They correspond to the node types of the XDM.

*mapping names and types*

### 3.3.3 Attribute Declarations as Schema Annotations

FeatureHouse uses annotations in grammars to specify how to parse documents into FSTs (and how to pretty-print FSTs). Likewise, we will now try to allow for the annotation of XML schemas.

To decide on the form of the annotations, we can use the guidelines set up in section 3.1.2 for general annotations in XML. They would normally lead us to choose XML Attributes (or Processing Instructions) as the representation of an-

notations.

Yet in the case of *schema* annotations, there is an additional problem: When parsing an instance document, how do we get information from its XML schema through the XML parser into the FST? If we cannot modify the XML Parser, we need to also put the information from its schema annotations into the instances.

On the other hand, the use of XML Schema also offers an additional opportunity to realize annotations: We can add *declarations* for XML Attributes in the schema, and then *use* these extra Attribute in the instances.

This choice will offer us the possibility to automatically pass on the annotations to the instance.[9] As a consequence, an (XML-)FST will contain *every information*[10] needed for superimposition as well as pretty-printing/unparsing.

### 3.3.4 Attributes as Instance Annotations

In FeatureHouse, we need to annotate only grammar documents to specify the mapping to FSTs (see section 2.1.4). The programming languages previously integrated into FeatureHouse provide enough structure (in particular, enough named structural elements) at the granularity chosen in each case to successfully map their instances to FSTs. For theses languages, there is no need to provide any annotations in the instances. Therefore, the grammars did not have to be extended with respect to the language accepted: The language defined by the annotated grammar remains unchanged in comparison to the language defined by the non-annotated grammar.[11]

In the case of XML, where documents often contain many anonymous tree nodes, it is necessary to also allow for additional instance annotations, at least to specify unique child names[12] (see section 3.1.1). Fortunately, our choice of passing the information from the schema annotations through the instances into the FST allows us to also take into account annotations in the *instance* at no extra cost.

## 3.4 Implementation

Figure 3.1 summarizes what FeatureHouse with our XML extension does in order to compose artifacts written in different XML languages. Artifacts created during

[9] We will see this in section 3.4.2.

[10] This is necessary as we will not be able to use any information coming directly from the schema or the schema annotations.

[11] The only exception is the `original()` keyword needed for the wrapping of Terminals, for example, of Java methods.

[12] These names will then have to be provided by the user in the instance documents.

processing are represented by boxes, whereas transformations are shown using arrows.

- On the Schema Level
  - □ XML Schema Documents without Schema Annotations
  - ↕ semi-automatic Schema annotation
  - □ XML Schema Documents with Schema Annotations
  - ↕ Schema validation of Instances with augmentation
- If there is no Schema
  - ↕ XSLT to directly add Instance Annotations
- On the Instance Level
  - □ XML Instance Documents without Instance Annotations
  - ↕ semi-automatic Instance annotation (see above), automatic Instance de-annotation
  - □ XML Instance Documents with Instance Annotations
  - ↕ XSLT for transforming back and forth
  - □ XML FSTs
  - ↕ JAXB for two-way Java-XML mapping of FSTs
  - □ FSTs
  - ↔ FSTComposer for superimposition of Directory Hierarchies

Figure 3.1: Schematic view of the FeatureHouse XML Extension

We will explain in the following how the transformations in figure 3.1 are implemented. For this, we will respect the order chosen in this figure, and proceed from top to bottom.

### 3.4.1 Annotating XML Schemas

Schema annotation does not have to be done very often; normally, this will only happen once for each XML language to be integrated into FeatureHouse. Nevertheless, since XML schemas for languages like XHTML are rather big artifacts, it would be convenient to have tool support for this process.

As we implement schema annotations using schema declarations of default values for Attributes (see section 3.3.3), it is possible to add them using XSLT. This way, you can automatically add all possible schema annotations with predefined values. This part of the process is common to all XML schemas. Afterwards, you need to check the annotations and customize them, which requires understanding

of the schema in question. This means in summary that schema annotation is a semi-automatic process.

It is easy to achieve de-annotation of all schemas using XSLT. However, this is not really necessary, as you can always simply store the original XML schema instead.

Next, we will have to process these schema annotations. As we have seen in section 3.3.3, it is necessary to carry these meta-data over to the instance.

### 3.4.2 Pushing Schema Information to the Instance

There is one mechanism in the XML universe that is perfectly suited for this purpose: Schema Validation of an XML instance document. As we have mentioned in section 2.2.5, this does not only yield a yes/no answer, but it also allows us to retrieve a PSVI.

*Schema Validation with Augmentation*

The PSVI contains the schema type for each Element and Attribute in the instance. It includes attributes not represented in the instance, but for which the schema specifies default values. Thus the instance information is augmented by schema-supplied information. In the PSVI, there are also tables to manage the references for example, from `IDREF`- to `ID`- attributes. The tables, which correspond to the symbol tables created by traditional parsers, complement the parsed tree structure to make up the graph of the input, adding non-tree edges.

As XML Schema is so complex, the schema types cannot be represented as simple names, but rather have to be references into the schema(s) used for validation. Having these links accessible through the PSVI of the instance would in principle allow us to physically store all annotations in the schema, and yet to retrieve them through the instance after parsing and validation.

Unfortunately, the format of the PSVI is not specified in the XML Schema standard, although it would be both natural and convenient to serialize and represent the PSVI in XML using an augmented version of the input instance document.

As a consequence, it is not easily possible to retrieve schema information from the instance with the currently available XML tools for Java. The only data which can be retrieved rather easily are in fact the attributes defaulted from the schema.

Surprisingly, is not easy to find a tool to implement attribute defaulting from an XML Schema. An extensive search has revealed the following two possibilities: `xsv` and SAX2. So, you can either use `xsv` or a simple Java tool we have written using SAX2. This will push the information from the schema annotated with attribute declarations (with default values) to the instance documents, from

where they can be easily retrieved later.

### 3.4.3 If There Is No Schema

It is clear that schema defaulting can only work for languages defined by an XML schema. We have seen in section 3.1.3 that this is not always the case. For these languages, we need another mechanism to specify the meta-data normally represented in schema annotations.

This problem has been simplified through our decision to push schema annotations to the instance for languages with XML schemas, as described in section 3.4.2. Therefore we are now able to directly represent the *schema* annotations in the *instance*.

Regarding schema-less languages, this means that we can find a replacement for an XML schema in combination with the process of schema defaulting. This replacement will, in general, be specific to the schema-less XML language in question. It can be implemented using XSLT.[13]

Now that we have explained the implementation of the transformations at the schema level, we can "descend" one level of abstraction and thus reach the instance level. In the following, all transformations will apply to instance documents only.

We start with non-annotated instance documents and follow their transformation until we reach the Java FST representation needed by FeatureHouse. In each step, we will also shortly describe how the corresponding back-transformation is realized.

### 3.4.4 Annotating XML Instances

The input to be superimposed consists of non-annotated instance artifacts. To begin, we will use one of the two ways of annotating instances described in section 3.4.2 and section 3.4.3. For languages with an XML Schema, it is preferable to use the former way (that is, schema defaulting) here. The result of this step are annotated instance artifacts.

De-annotating instances cannot be implemented using schema validation, because there is no possibility to remove (annotation) Attributes during validation.[14] We

[13] This way, we use XSLT as a schema language, which has been discussed in section 3.1.3.

[14] A schema will either contain the corresponding attribute declarations and thus allow the Attributes, or it will not contain the declarations, and reject an instance containing these Attributes. This holds true in the case of a complete schema, when you select strict checking;

have defined a simple transformation in XSLT that is used to de-annotate instances independent of the XML language they are written in. This de-annotation is the final step in the composition process, as it transforms the annotated versions of the composed variants into non-annotated version.

This step is necessary to make the composed variants valid with respect to the original, non-annotated schema of the language they are written in. However, it is a good idea to also keep the annotated variants, as they can be further composed with other features without the need of (manually) re-annotating them.

### 3.4.5 Transforming Annotated XML to XML-FSTs

The next step is to transform the annotated instances to XML-FSTs (see section 3.3.2). Since this transformation uses XML for input and output, an XML transformation language is ideally suited for the processing. As explained in section 3.3.1, we consider XSLT in version 1.0 the transformation language of choice. We have also seen in section 3.2.3 that it is not only possible, but also quite simple to drive an XSLT processor from Java using TrAX. Of course, the exact same argument is also true for the back-conversion from XML-FSTs to XML.

This way, the only process that we do not have tool support for is the verification of the (back-)transformation: Transformation and back-transfromation should be the inverse of each other. As we use XSLT1 to implement them, this can only be proven manually (or evidence could be gathered by testing). A schema-aware XSLT2- or XQuery1-processor would indeed be able to statically type-check the two transformations against supplied input and output schemas.[15] But, as we have seen in section 2.2.6, at the time of writing, there are too many disadvantages to the use of these XML processing languages.

In conclusion, we cannot prove that the transformations are correct; we can only say that our tests show that they work as expected: The (forward) transformation produces the XML-FSTs needed in the next step, and the back-transformation will later on be able represent the XML-FST of the composed variant in its source XML language.

for a partial schema with lax checking, the Attributes will always be allowed.

[15] We would then have to provide a specific XML Schema for the source XML language and a generic one for XML-FSTs as the target; the forward transformation would be type-checked against the source schema as input and the target schema as output, and for the back-transformation we would interchange the two schemas.

### 3.4.6 Transforming XML-FSTs to FSTs

*Binding XML-FSTs to FSTs*

What we are lacking now is just a way to convert between XML-FSTs and "normal" (Java) FSTs. We could think about writing a parser for XML-FSTs (maybe we would even write this parser manually, as this restricted XML format is under our control), and also writing a simple output routine to "pretty-print" FSTs as XML.[16] Alternatively, we could decide to employ as much programming language tool support as possible, and make use of a Java XML parser which also supports the generation of XML output.

Fortunately, we have a tool at hand that is even simpler to use: JAXB (see section 3.2.2) allow us to specify a simple mapping between an XML Schema- and a Java Class-hierarchy; it will then automatically generate code to parse and unparse XML, optionally performing on-the-fly validation.

We have chosen this approach to generate the mapping between Java- and XML-FSTs. As a consequence, the JAXB run-time will automatically perform this two-way mapping. This means that after this step, the XML artifacts from the input will be represented as Java-FSTs, ready to be composed by FeatureHouse.

In figure 3.2, we show the FST of the simple XHTML example whose source code we have shown in figure 2.4. Note the computer-generated names for the XML Text nodes which are the Terminal nodes of this FST. Like the FST for a Java artifact (shown in figure 2.1), this figure of an FST has also been slightly simplified. For instance, it does not show all the XML Text nodes that correspond to the whitespace between the XHTML Elements, which is only used for indentation.

Care has been taken to not lose any information contained in the XDM during all the bidirectional transformations described in all the previous sections.

### 3.4.7 Superimposing FSTs for XML Artifacts

By application of the transformations listed above, FeatureHouse is able to compose XML files by superimposition. FeatureHouse will not only compose single files, but complete directories with all their sub-directories. This means that you can put all the files that a feature consists of in a directory, and FeatureHouse will superimpose them at once when you call it to compose two features.

In conclusion, this XML extension makes FeatureHouse capable of composing artifacts written in many different languages in a unified way. Furthermore, the lan-

[16] A smarter way of doing these conversions manually would be to take advantage of the above-mentioned asymmetry between parsing and unparsing, and to replace the complex XML parser written in Java with a Java "pretty-printer" written in XSLT, thereby reutilizing Java's capability of parsing and subsequently loading its own source code at run-time.

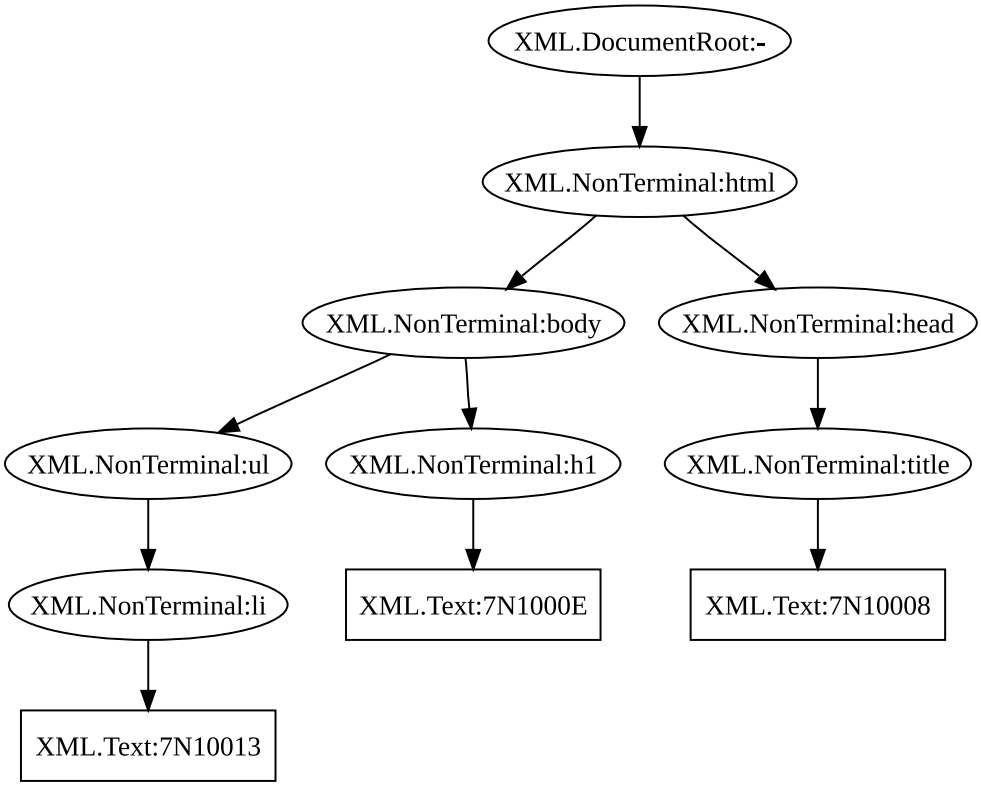


Figure 3.2: An FST for an XHTML artifact

guages can be described using two different specification languages: FeatureBNF for traditional grammar-based languages, and XML Schema (with additional semantics for feature annotations) for XML languages; there is even a possibility to use FeatureHouse for (XML-)languages without a formal language definition.

# Chapter 4

# Case Studies

In this section we present three case studies. We have conducted them to evaluate the FeatureHouse approach for XML languages. Each case study uses a different XML language.

## 4.1 Graph Product Line (XHTML 1.0 strict)

The *Graph Product Line (GPL)* case study [LB01] consists of several features with Java code for different graph implementations, as well as graph algorithms that work on them. It has been extended in prior work with XHTML documentation to describe the capabilities of a composed variant. Fortunately, the documentation has already been annotated with fstname Attributes to guarantee unique child names.

XHTML is an XML-based reformulation of *Standard Generalized Markup Language (SGML)*-based HTML. It has been designed to represent semi-structured data. As it is an official (and recent) W3C standard, and a XML Schema defining XHTML is available.[1] In practice, many of the artifacts claiming to conform to XHTML are invalid with respect to this XML Schema, because they have only been updated to reference the XHTML schema, while their content remains antiquated *HyperText Markup Language (HTML)*. XHTML

Since FeatureHouse is a general superimposition tool, we are able to compose the Java code and the accompanying XHTML artifacts at the same time, using the same implementation of superimposition.

The result is a variant of the GPL with only the data structures and algorithms needed. It contains tailored documentation describing the graph features included.

[1] In fact, there are three different XML schemas for XHTML; we use XHTML 1.0 strict.

## 4.2 Submission (XMI 1.2 with UML 1.4)

In this case study, we compose *Unified Modeling Language (UML)* class diagrams that model the submission process of scientific papers. The case study has been originally presented in [BCRL07]. Its XMI representation has been kindly provided by Florian Janda.

*no schema available*

The OMG standard UML uses XMI as its serialization. We will use version 1.4 of UML here, which is embedded in version 1.2 XMI artifacts. The reason for this choice is that ArgoUML, which appears to be the best freely available UML tool, uses this format. Unfortunately, XML schemas for both XMI and UML have only been created for later versions of these standards.

Although UML is a visual language that uses diagrams as its main construct, there is no standard serialization for them in version 1.4: An XMI artifact for an UML model only describes its structure; (re-)visualizing is left to the processing application.

*graphs in XML*

UML class diagrams model class hierarchies using graphs instead of trees. This is necessary, as the class diagrams do not only allow tree edges for the inheritance relation, but also associations between arbitrarily chosen classes. As a consequence, the serialized representation XMI also needs to be a graph data format. In this way, it uses references from `IDREF` Attributes to `ID` Attributes in order to store graph edges. `ID` Attributes are used to uniquely identify an Element within one XML document. Because UML is a visual language, `IDs` are auto-generated, and there is not even a possibility for the user to visually change them.

*IDs*

Since `IDs` are document-unique identifiers, an additional problem arises when composing separately created UML class diagrams: Each feature is a separate document (from the application's point of view) with separately auto-created `IDs`. Superimposition will match Elements with the same fstname annotation, but different `IDs`, effectively removing at least one of the two `IDs` concerned. References to these `IDs` will not be taken into account by general tree superimposition. This means that a composed variant will have dangling references from `IDREF` Attributes to `ID` Attributes which do no longer exist. This does not only make the variant invalid with respect to its XML Schema or DTD, but also effectively removes these edges from the UML class diagram the variant represents.

It is clear that standard tree superimposition cannot be applied without problems to graphs "modeled" as trees. Therefore the question is, how can we solve this problem for XMI (and perhaps for comparable languages, too)?

In XMI, `IDs` appear to be arbitrarily chosen, and they will not be referenced

from anywhere outside of the composed XMI artifact.[2] It is therefore possible to replace them by more suitably chosen meta-data. Natural candidates are the fstname annotations that you have to add to the instances to make children names unique. Of course, they need to be more carefully chosen for this purpose, because they do not only have to be unique among a node's children now, but also unique within a whole composed artifact.

*reuse fstnames for new IDs*

Therefore, we will additionally use fstname annotations to create new `IDs` (which will then be the same before and after composition). As a consequence, we will also have to change the referencing `IDREF` Attributes. Both operations can be executed simultaneously in a preprocessing step. This is done using XSLT. Note that this transformation employs our knowledge of the XMI language: We need to know which the `ID` Attributes are, and that they can be arbitrarily chosen.

Using this additional preprocessing step, we are able to successfully compose the UML class diagrams for the submission process.

## 4.3 Builder (ant build.xml 1.7)

Apache `ant` is the standard build tool for Java-based projects. It takes the place of the classical `make`, which is especially used in C- and C++-projects. The `ant` tool uses XML-based *build scripts* instead of the (sometimes infamous) *Makefiles* [ZK05].

*dynamic language structure*

As we have already discussed in section 3.1.3, it is possible for the user to define in a build script new Elements for use in the *same* build script. This means that the language structure is defined dynamically. Consequently, a schema that completely specifies the structure of this build script will depend on the *content* of this very build script. This means that there cannot be a complete schema for the class of all `ant` build scripts. Actually, not even a partial schema for all build scripts has been created by the `ant` authors. It is only possible to retrieve a (complete) DTD for a *given* build script.

We have created the Builder case study especially for this work. It contains in different features several build targets that depend on one another. As a rule-based tool, `ant` will automatically use these dependencies to determine which other targets it needs to build when the user invokes `ant` to build a specific target.

After composition, the generated variant contains the targets needed by the selected features. It can be successfully used by `ant` to build targets. This shows

[2] If they need to be referenced from other documents, for instance, then there are also other possibilities to keep one of the two `IDs` concerned.

that we have successfully superimposed artifacts written in the language of the `ant` tool.

In summary, the case studies conducted have shown that superimposition is applicable to different classes of XML languages. We have learned that these languages use several concepts that need special attention. Although some languages may require additional preprocessing, there is no principal obstacle that hinders the use of superimposition to compose them.

# Chapter 5

# Summary

We will now end with some concluding remarks. To begin, we discuss related work. Next follows a summary of the work presented. We then give an outlook into possible future research.

## 5.1 Related Work

Since this work is an extension of FeatureHouse, the research leading to FeatureHouse is the most related one. In the beginning, superimposition has been proposed in [AL08] as a general approach to FOSD. This has been implemented in the FSTComposer tool.

Later on, it has been formalized in an algebra [ALMK08]. This algebra furthermore unifies the compositional approach taken with other approaches, like, for example, global modifications. These global modifications are, for example, used by Aspect-Oriented Programming [KLM+97]. The FSTComposer tool has then been redesigned to use code generation for its language plugins; this version, which is extended by the work presented here, is described in [AKL09]. It already contains a grammar-based plugin for the XML language as a whole. Yet this does not allow to take into account the properties of any single XML language that is specified by a schema.

We also have to mention the CIDE approach [Käs07] here, as it is complementary to the compositional approach of FeatureHouse: It researches how features can be mined and extracted from legacy software. The most recent version of this work is implemented in the *generalized CIDE (gCIDE)* tool [KTA08]. With this Eclipse-based tool, you can mark features in source code, or in the corresponding AST. They are highlighted in different colors, so that you can easily see which code fragment belongs to which feature(s). To summarize, CIDE enables you to view a (legacy) system from a feature-oriented perspective. It even allows you to export this decomposition, so that each resulting artifact does not contain more than one feature.

The most closely related work regarding composition of XML artifacts is that on the XAK tool [ADT07]. It defines XML modules that can be combined using quantification. XAK is an interesting approach for feature-oriented XML. It is part of the AHEAD Tool Suite [Bat06], which means that is implemented as a stand-alone tool. It uses its own implementation of XPath-based quantification.

Viewed from a broader perspective, this work is connected to the domain of software merging [Men02]. The most simple approaches in this domain only exploit the line-based representation of textual artifacts to match them. This approach is used to date in successful version control systems like CVS [ZK05]. However, more advanced approaches propose semantic merging to guarantee for semantic correctness of merged artifacts.

Semantic merging has also been proposed for XML artifacts [Lin04]. In this paper, differences between trees are modeled as edits. In a second step, the edits are reconciled. This approach is designed for more complex scenarios, in which different users modify documents in complex ways.

The composition of UML artifacts has been regarded in [BCRL07] as a special case of general model merging. It encompasses an abstract view of UML artifacts that is not restricted by the XMI representation they may have. This enables a semantics-oriented approach to composing UML artifacts compared to the tree-based approach taken by this work.

## 5.2 Conclusions

In this work, we have made the following contributions:

We first have conducted an analysis of the challenges posed by XML languages. We therefore have unified XML Trees and Feature Structure Trees; we have explained the need for unique child names during superimposition. this involves the necessity of instance annotations for XML artifacts. Furthermore, we have discussed different possibilities of annotations for XML; the most promising of them are Processing Instructions and Attributes. To conclude the analysis, we have provided an overview of the languages you can use to specify other XML languages.

This analysis has enabled us to integrate a generic extension for XML languages with the FeatureHouse tool-set. Our implementation has the following properties:

- use of high-level APIs and the special-purpose languages XSLT and XML Schema
- a meta-representation of XML in XML, which is used as the intermediate

data structure between XML and FSTs

- annotations realized by a combination of Attribute Declarations in the schema and Attributes in the instances
- included handling of schema-less languages

With this extension, FeatureHouse is able to compose artifacts written in grammar-based as well as schema-based languages in a unified way.

We have furthermore conducted three case studies with artifacts written in highly different XML languages. These languages have been selected because of their prevalent use in modern software engineering. We estimate that these languages cover a broad area of application.

1. The Graph Product Line example contains XHTML documentation together with the Java code it accompanies. We here demonstrate the integrated composition of XML- and non-XML artifacts that FeatureHouse now is capable of. It also shows how semi-structured data can be superimposed.

2. The Submission example consists of UML class diagrams, which constitute a visual, graph-based language. Our FeatureHouse extension for XML has composed them as well.

3. In the Builder example, we have superimposed `ant` build scripts, which use a dynamically defined XML language. This case study has also proved successful.

We have learned during our case studies that XML schemas are not available for some important XML languages. (Neither UML nor `ant` provide such a schema.) Nevertheless, it is possible to also annotate these schema-less languages with some additional effort.

## 5.3 Future Work

This work has analyzed how XML languages can be superimposed, and how they can be integrated into FeatureHouse. We have only integrated some example languages; several other important XML languages remain for future integration.

There also are many possibilities to automate the especially time-consuming instance annotation process, at least to some degree. The most worthwhile option may be the extension of gCIDE for XML languages; this will require a considerable effort, but it will also be very valuable, as it will allow the semi-automatic de-composition of existing XML artifacts into features.

To make composition of XML artifacts safer, it is desirable to include automatic validation of composed variants against an XML Schema. This may also be an option for the features in the input, depending on whether they are supposed to be valid instances on their own or not.

# Bibliography


[ADT07] Felipe I. Anfurrutia, Oscar Díaz, and Salvador Trujillo. On refining XML artifacts. In *Proceedings of the 7th International Conference on Web Engineering*, volume 4607 of *Lecture Notes in Computer Science*, pages 473–478. Springer, July 2007. 36

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proceedings of the 31th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, May 2009. To appear. 1, 3, 7, 35

[AL08] Sven Apel and Christian Lengauer. Superimposition: A language-independent approach to software composition. In *Proceedings of the ETAPS International Symposium on Software Composition (SC)*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35. Springer, March 2008. 4, 35

[ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebra for features and feature composition. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, July 2008. 6, 35

[Bat06] Don S. Batory. A tutorial on feature oriented programming and the AHEAD Tool Suite. In *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 3–35. Springer, 2006. 36

[BCRL07] Artur Boronat, José Á. Carsí, Isidro Ramos, and Patricio Letelier. Formal model merging applied to class diagram integration. *Electron. Notes Theor. Comput. Sci.*, 166:5–26, 2007. 32, 36

[BNV04] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML Schema: a practical study. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 79–84, New York, NY, USA, 2004. ACM. 12, 19

[BSR04] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004. 1

[GBJL02] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. John Wiley, 2002. 13

[Käs07] Christian Kästner. CIDE: Decomposing legacy applications into features. In *Proceedings of the 11th International Software Product Line Conference (SPLC), second volume (Demonstration)*, pages 149–150, 2007. 3, 35

[KLM+97] 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)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997. 35

[KTA08] Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing software product line variabilities in source code. In *Proceedings of the 2nd International SPLC Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 303–313, September 2008. 35

[LB01] Roberto E. Lopez-Herrejon and Don Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24. Springer, September 2001. 31

[Lin04] Tancred Lindholm. A three-way merge for XML documents. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*, pages 1–10, New York, NY, USA, 2004. ACM. 36

[Men02] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002. 36

[MS06] Anders Møller and Michael I. Schwartzbach. *An Introduction to XML and Web Technologies*. Addison-Wesley, January 2006. 12, 19, 20

[ZK05] Andreas Zeller and Jens Krinke. *Essential Open Source Toolset*. Wiley and Sons, January 2005. 33, 36

# Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Passau, den 11. März 2009

Jens Dörre