Co-Designing XML-based Languages and Classes with *Pontifex*

Robert Kosara Klaus Hammermüller
Silvia Miksch

Vienna University of Technology, Institute of Software Technology,
Favoritenstraße 9/E188, A-1040 Vienna, Austria, Europe
<robert,klaus,silvia>@ifs.tuwien.ac.at

Abstract

While XML provides a number of advantages for the storage of data, there is little support for the application programmer in building data structures that mirror such data. Document Object Model (DOM) is very easy to use for small applications, but it lacks a number of features (type safety, validity checks for CDATA fields, scalability). XML/Schema, on the other hand, provides a number of interesting features, and probably will eventually lead to tools that can create classes based on such a specification — but it is extremely complex, and simply not existant yet.

Pontifex (latin for *bridge builder*) tries to fill this void by creating both a DTD and a Java class library (including a SAX based parser) from a simple specification written in XML, and very similar in structure to DTDs. The classes provide means of attaching listeners to every object, and also to associate any kind of data with them, so the application does not have to mirror the structure of the data.

1 Introduction

When developing an application that uses data from XML [1] files, one has to do at least two things: design the language, and create the corresponding classes. While the first part can simply be done using an editor, the second can become tedious if the language has many different tags, or if it changes. With Document Object Model (DOM, [4]), a very generic and flexible solution exists for the classes problem, but it is by no means perfect. Especially for large applications and complex languages, programming becomes difficult and error-prone. DOM elements are stored in generic nodes containing all information as strings without; there is also no possibility to validate the content of CDATA fields.

Pontifex is based on the idea of co-designing a Document Type Description (DTD, [2]) and corresponding classes at once. XML/Schema [5, 6] also seems to be going into this direction, but unfortunately is not available yet (and also rather complex). Unlike DOM, *Pontifex* does not provide a generic structure for unknown XML documents, but creates a specialized (and static) structure from a specification; and unlike XML/Schema a close connection between the functionality of a specialized XML application to a based XML-structure is intended.

In section 2, we discuss the differences between data-oriented languages and classes. In section 3, we give an overview of *Pontifex*, followed by the description of its input language in section 4, and its output in section 5. A comparison to DOM and XML/Schema is given in section 6.

2 Language vs. Classes

There are a few differences between a class definition and a language that need to be considered before going into the details of how to create both from one specification.

While the order of elements does (or can) play a role in a language, it does not in a class. The order of elements simply has no meaning at all.

A class is also "flat", i.e., it cannot contain any substructures (at least not in Java — only references to other objects are possible). But a language (especially when it is an XML application) can contain nested statements of any depth. Such parts can also be repeated, which is inherently not possible in a class definition.

A disjunction of elements can also not be expressed in a class definition, except if the disjunction is at the top level (which then amounts to the classes in the disjunction being derived from the class containing it, see Figure 2). So it has to be expressed as a number of optional elements (i.e., references that can be `null`, with the additional condition that exactly one of them be non-`null` at any given time).

3 Pontifex

Pontifex (latin for *bridge builder*) is a program that creates a DTD and classes from the specification of a language (which is slightly more powerful in some repects than a simple DTD). One advantage of this language (called HSL, see section 4) is that it is itself an XML application, and therefore, XML editors can be used to write such a specification. Figure 1 gives an overview of how *Pontifex* works.

Input to *Pontifex* is written in HSL, which basically mirrors the definitions in a DTD by providing tags for the definition of elements, attributes and elements' children (see section 4).

Pontifex' output consists of (see section 5):

- a DTD for the specified document
- classes and listeners, related to the defined elements in the DTD;
- a SAX-based [3] parser that will create instances of the created classes when parsing a file in the specified language
- an HTML file containing documentation of the language

4 The Harmless Specification Language (HSL)

HSL[1] is itself an XML application, and can therefore be edited with any data-oriented XML editor. It consists of only a handful of tags that mirror many of the elements found in DTDs, and include some information that is needed for generating useful classes.

The root tag of any HSL file is `<hslspec>`. It can contain a number of `<option>` tags, and must contain at least one `<element>` definition. Every element definition contains zero or more attributes, and zero or more children (see below). Usually, a class is generated for every element. This can be overridden by setting the attribute `makeclass` to "no". The superclass of the created class can be given using the `super` attribute. If the class's name should be different from the name of the attribute (with the first letter converted to upper case), it can be specified with the `classname` attribute.

[1] HSL's name stems from a time when *Pontifex* was still called Heimdall, but it has retained its name, which is now open to speculation about its true meaning. But because it is mostly harmless, it shall be called Harmless Specification Language for the time being.

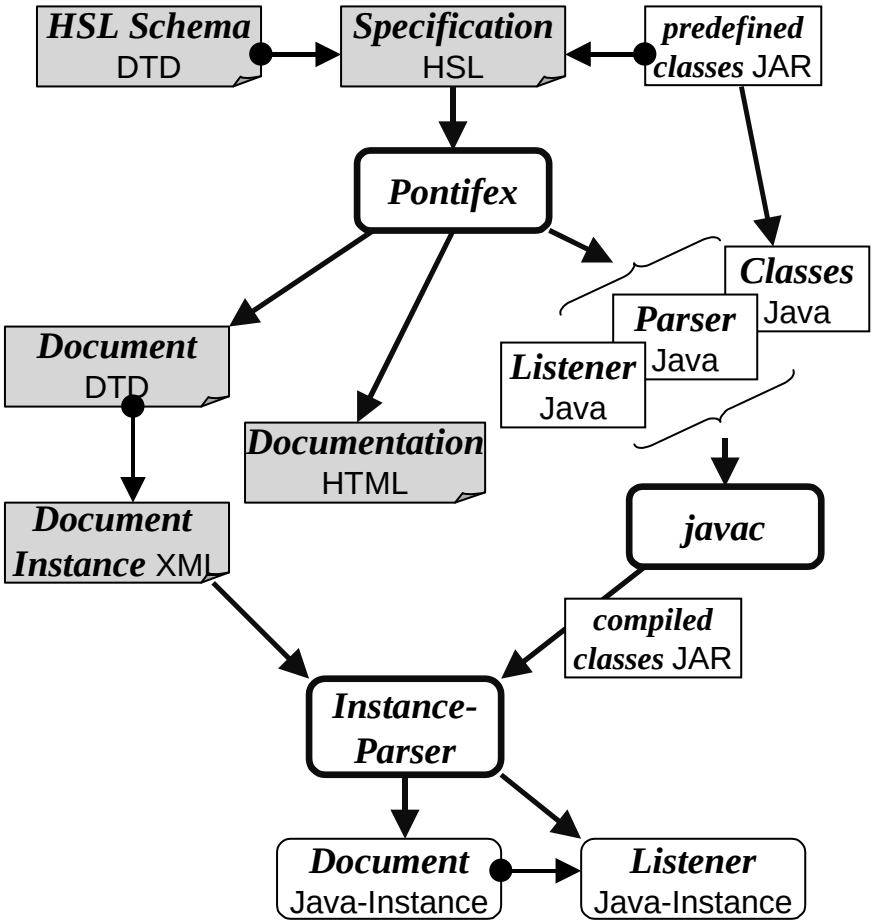

Figure 1: What *Pontifex* does. Grayed boxes are XML documents, white boxes are Java sources, and rounded boxes are object instances in a Java virtual machine. Thick arrows represent data flow, thin arrows with dots denote template-document type relationships.

An attribute is specified by its name and type (see table 1). In addition, a default value can be given (using the `default` attribute), and it can be specified as an optional attribute (by setting `optional` to "yes").

A child is simply a reference to another element. It can be repeated exactly once, zero or once (i.e., optional), zero or more times, or one or more times — this is specified using the `repeat` attribute. If the name of the member variable to be generated for the child should be different from its name, it can be specified using the `impname` attribute. Figure 2 shows two simple examples of element definitions.

Groups are HSL's analogue to parentheses in XML content specifications. Every group can contain other groups and children, and have a `type` (conjunction ("andgroup") or disjunction ("orgroup")), and a `repeat` (which is the same as for children) attribute. Groups can contain the special child tag `pcdata` (which simply specifies a child of XML type PCDATA) as their first element.

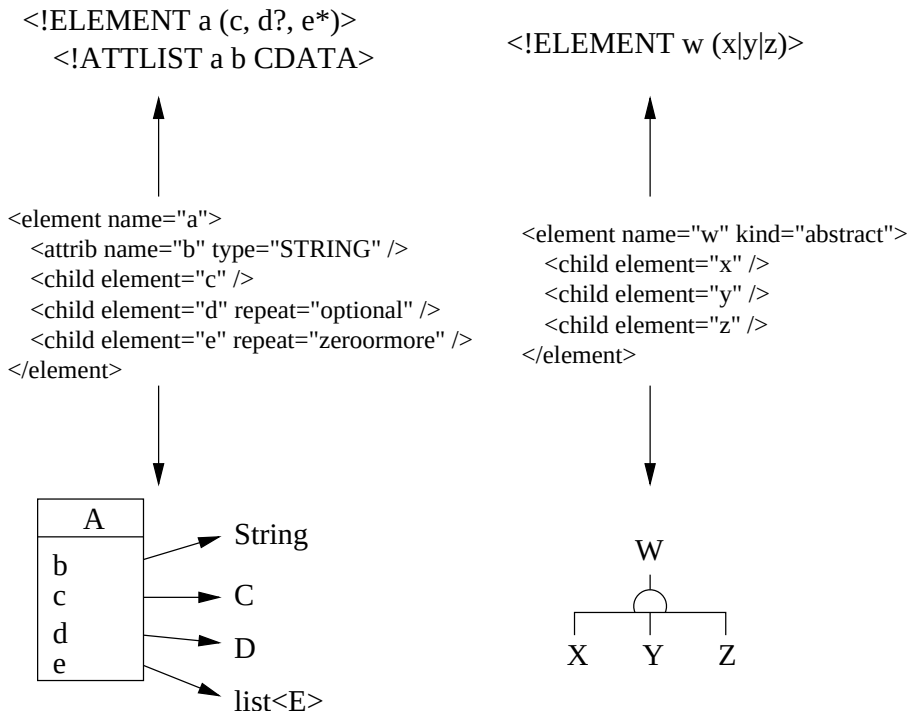

Figure 2: Translation of two different kinds of elements (`tree` and `abstract`) into classes (classes are written using capital letters, member variables using lower cas). The semicircle on the right side denotes class inheritance.

5 Pontifex' Output

5.1 Document Type Description (DTD)

The document type description (DTD) is basically a simple translation of HSL tags into DTD syntax. Content specifications are generated from group and child lists; this also requires expanding of abstract elements. No documentation at all is written to the DTD. This is not necessary, because of the generated documentation (see section 5.4), and because changes should never be made to the DTD, but to the HSL file.

5.2 Generated Classes

The classes generated follow the basic structure of the language. That means, a class is generated for every element, and an instance variable is defined for every attribute.

5.2.1 Elements

There are four different types of class that can be generated (specified by the `kind` attribute in the element definition), which determine the way an element's children are handled. The most common case is that of a tree, i.e., a member variable is created for every child, with the child's class as its type. If a child can be repeated, the type of the variable is a list. In the case of an optional child, that variable can be *null*.

The second type of element is an abstract element. An abstract element acts like a macro (or an entity definition) for the structure of the language. In terms of classes, an abstract element is the superclass of its children (see Figure 2, right side).

Three more types are designed to more easily build text-oriented XML applications: `list`, `flat` and `string`. A node of type `list` only contains one member variable (in addition to the member variables for its attributes): A list that contains all its immediate children (including PCDATA sections), in the order they appeared. This is useful for providing a means of including documentation that is written in XHTML [7], for example. The objects contained in this list can contain children.

An element of type `flat` also contains a list, but this list is a flat list of all the elements contained between the starting and closing tags of this element, no matter which hierarchy level they appear on. This means, that member variables of objects in such a list that point to children will contain the value `null`, even if they would otherwise point to obligatory children (and therefore be guaranteed to be non-null after a successful parse).

The last type is `string`, which means that the object contains a string which contains a simple string representation (with resolved character entities) of everything in between its start- and end-tag.

For every member variable (regardless of whether it is for an attribute or a child), functions are created that make accessing it possible (all member variables are declared `protected`). For a simple attribute (i.e., one that is not a list), a get and a set method are generated. The get method simply returns the value of the variable, the set method sets the variable to a new value, and calls any listeners attached to this object. It is important to note that these methods are declared with the right return and variable types, and not simply pass objects of class `java.lang.Object`.

For list attributes, three methods are generated: an add- and a remove method for adding and removing elements to and from the list, and a method that returns an iterator pointing to the first element in the list.

In addition, a method called `toXML()` is generated for every class, so that if the structure of the data is changed, it can be written to a file again. This method returns a string that represents itself and all its sub-nodes.

5.2.2 Listener Interfaces

For every element, a listener interface is generated that contains listener methods for every class member. A class that wants to listen for changes in an object must implement the corresponding listener interface.

Listeners are a standard method in many Java class libraries. If an object wants to be notified of changes in another object, its class has to implement a special interface. It can then register itself as a listener to that other object. When the observed object changes, it calls one of the methods specified in the listener interface for all the registered listener objects. This makes it easier to separate the underlying data from application specific parts (e.g., a user interface).

5.2.3 Attributes

Attributes are treated depending on their type — the translation of types between their HSL specification and Java and XML types is given in Table 1.

HSL type	Java Type	XML Type
userdefined	userdefined	CDATA
STRING	java.lang.String	CDATA
INTEGER	int	CDATA
LONG	long	CDATA
FLOAT	float	CDATA
DOUBLE	double	CDATA
ID	java.lang.String	ID
IDREF	java.lang.String	IDREF
$a\|b\|c$	int	$a\|b\|c$

Table 1: Mapping of HSL, Java and XML types.

For enumeration types (i.e., $a|b|c$), a constant is defined for every item in the list (its name is converted to all upper case letters for this purpose). The value 0 (zero) is never used for such a constant, but is reserved for optional enumerated attributes that were not encountered in a parse (for float and double attributes, NaN ("Not A Number") is used in such a case). All constants of the same name have the same value in the whole class library (they are declared in every class that needs them, however). This was done to prevent problems with abstract types (where, if the concrete type of the object is not known, the wrong constant could be used).

5.3 Parser

The classes alone are not enough to build an application on, of course, there needs to be a parser that will create objects from an input file. This parser is also created by *Pontifex*. It is based on a SAX compliant parser (at the moment, IBM's XML4J [8] is used, but any other parser could be substituted very easily), and contains just the code needed to create objects and do type checks for integers, etc.

The parser class also contains a method called `toXML()` which converts a string to a valid XML string (i.e., replaces all non-UTF8 characters by their entity definitions).

5.4 Documentation

Pontifex creates an HTML file that contains the DTD in a format that is easier to read than the plain ASCII file (Figure 3). It also makes use of HTML by providing links between the uses of elements and their definition (each use of a child is a link to its definition), and a cross-reference section, which for every element lists the elements it is used in.

Additionally, elements contained in abstract elements are grouped and indented, so that they are easier to recognize. When an abstract element is used, that element is not resolved (not replaced by its parts), but a link is provided to that abstract group.

For every attribute and child, a short description can be given in the HSL file, which is written to the HTML file as well. Element definitions can also contain documentation in XHTML, which is also written to the HTML file. This allows for a slightly higher level of documentation (the tag level, not the individual attribute level). These facilities cannot replace a higher level documentation of such a language (and the application). They make the maintenance of the basic language documentation easier, however.

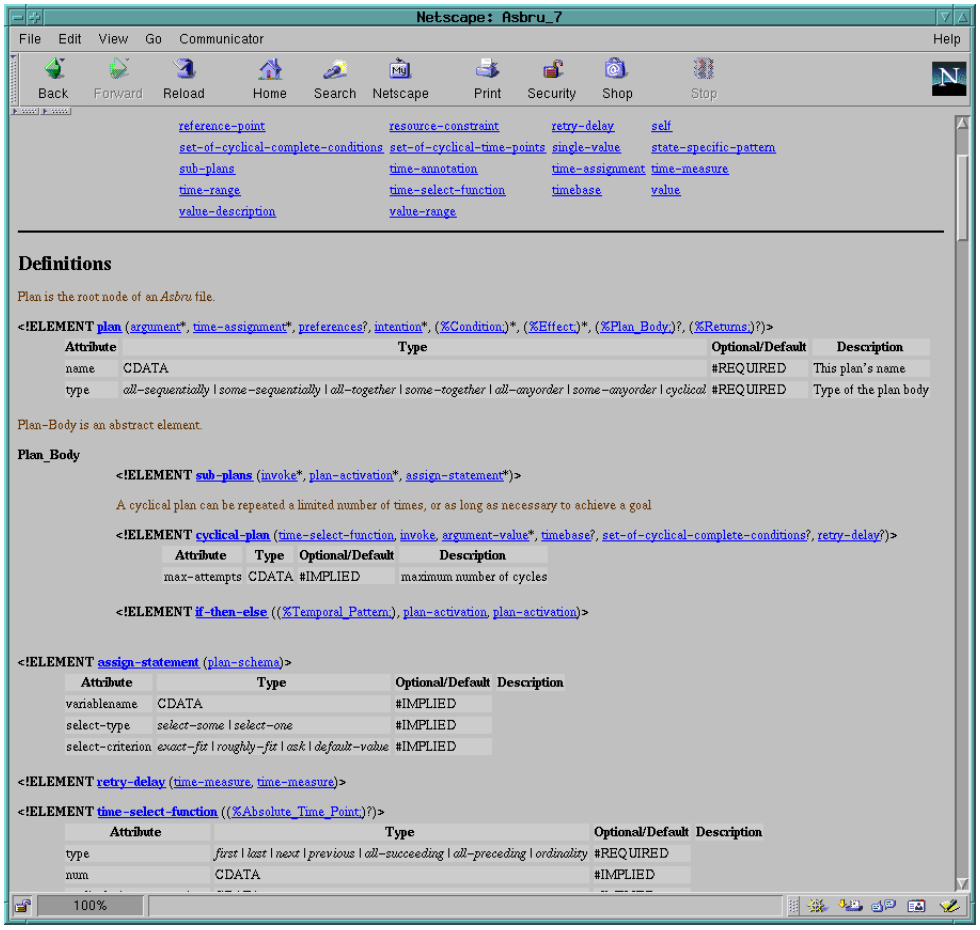

Figure 3: An example of the documentation generated by Pontifex for *Asbru*, the language used in the Asgaard Project.

6 Comparison to DOM and XML/Schema

Depending on the type of application, *Pontifex* has several advantages over DOM and XML Schemas. It was designed for an application that deals with a relatively complex language (which contains medical therapy plans), that is processed in a number of ways.

Specificity. Compared to DOM, the classes generated by *Pontifex* are more specific to the application than DOM classes can be. An object has the type that is associated with the parsed tag, not just a basic "XMLNode" type. All the references to an element's children also have the correct

types. Additionally, enumerated types in attributes lead to the creation of symbolic constants, that are more convenient (and faster) to use than comparing strings. This also applies to different number formats, which are checked at parse time, so that the application can work with the actual numbers instead of having to convert them and check them for validity.

Strict Typing. So using *Pontifex*, it is a lot more difficult to overlook a part of the program when the language is changed, because the different types and method names in such unchanged parts of the program will cause the compiler to complain. With DOM, the types of objects cannot be checked at compile time, nor can the existance of attributes or children that are returned by methods that get their names as strings as parameters.

Listeners. Listeners can be attached to every object, so that different parts of a program can react immediately to changes in the underlying data.

Applicationd Data. An application can also attach their own data to objects using the application data arrays provided. These store one reference per object and user (where "user" means an independent part of the application). Any part of an application can acquire an index into this array that is valid for all objects present in the current application. This way, the application does not have to mirror (and maintain) the structure of the generated objects, but can still have its own data easily accessible (see Figure 4).

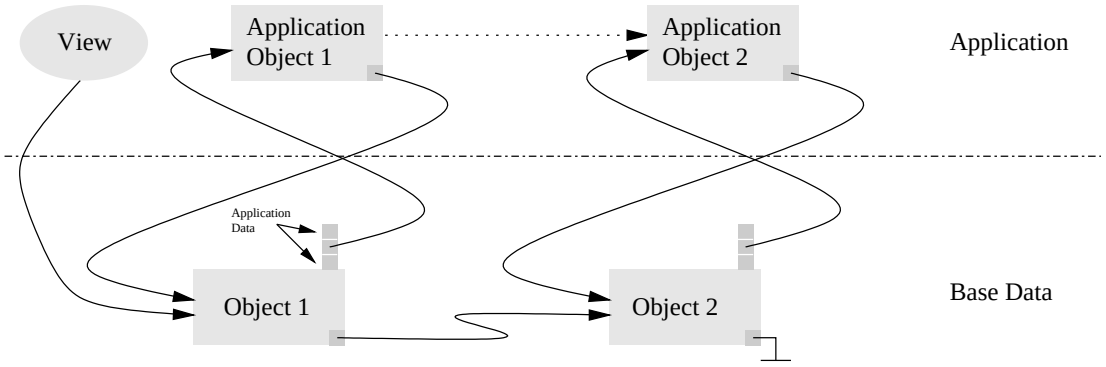

Figure 4: Application data arrays and how they are used.

Node Types. *Pontifex* is also more flexible on the element level than DOM. If one wants a flat list of all the tags and text segments starting from a particular node, that is possible. This can be convenient if one wants to include a node that only contains documentation that is to be written as it is (or with minor changes, e.g., the insertion of links when certain patterns are recognized); it also saves memory for nodes that are not needed if one only needs a string representation of the contents of a node.

Availability. At the time of writing, XML Schema Part I and II are still working drafts. It is conceivable that powerful tools will be built based on this defintion, some of which will be more powerful than *Pontifex*. This is, however, not the case yet, and there will probably be many applications that do not need the whole extent of what Schemas provide, and whose developers will shun the immense complexity of XML Schemas. But Schemas also lack some features, like the possibility of creating a kind of dictionary to validate inputs of numbers with units, where the units can be changed without changing the Schema (so it is not possible to specify what is called a picture in Schema).

7 Conclusion

While DOM is a powerful and flexible interface, it is too general for many applications that are beyond a certain complexity. XML Schemas will hopefully one day provide means to create code,

but do not yet exist, and are also very complex. Creating classes by hand is not only tedious, but also a maintenance nightmare.

Pontifex provides a simple yet flexible solution for these problems. It is more specific than DOM, but also lacks many of the features of XML Schemas — but it is also a lot less complex. For many applications now it should be an alternative to DOM that makes application development easier and faster — especially in the early phases of a project — and that also provides the information the compiler needs to do basic error checking far beyond what is possible with DOM.

As soon as Pontifex is ready for its first release (which mainly means writing useful documentation), it will be available from `http://www.ifs.tuwien.ac.at/~rkosara/pontifex/`.

Acknowledgements

Pontifex was developed as a tool for the Asgaard Project, which is supported by *"Fonds zur Förderung der wissenschaftlichen Forschung" (Austrian Science Fund)*, grant P12797-INF.

References

[1] Extensible Markup Language (XML) 1.0 (Recommendation). *W3C*, February 10, 1998.
`http://www.w3.org/TR/1998/REC-xml-19980210`

[2] W3C XML Specification DTD ("XMLspec", Report). *W3C*, May 11, 1998.
`http://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm`

[3] SAX 1.0: The Simple API for XML. *Megginson Technologies*, May 11, 1998.
`http://www.megginson.com/SAX/index.html`

[4] Document Object Model (DOM) Level 2 Specification, Candidate Recommendation. *W3C*, December 10, 1999.
`http://www.w3.org/TR/REC-DOM-Level-1/`

[5] XML Schema Part 1: Structures (Working Draft). *W3C*, 17 December 1999.
`http://www.w3.org/TR/xmlschema-1/`

[6] XML Schema Part 2: Datatypes (Working Draft). *W3C*, December 17, 1999.
`http://www.w3.org/TR/xmlschema-2/`

[7] XHTML™ 1.0: The Extensible HyperText Markup Language. *W3C*, Januar 26, 2000.
`http://www.w3.org/TR/xhtml1/`

[8] XML4J 3.0: A Validating XML parser. *IBM*, June 12, 1999.
`http://www.alphaworks.ibm.com`