Automized Generation of Abstract Syntax Trees Represented as Typed DOM XML

Extended Abstract

Baltasar Trancón y Widemann, Markus Lepper, Jacob Wieland
{bt,lepper,ugh}@cs.tu-berlin.de

Institute of Computer Science
Technische Universität Berlin

Abstract. The XANTLR/TDOM project is an implementation of a "typed" XML[1] Document Object Model initially used to represent abstract syntax trees in a compiler project. Class files, SAX event receivers, visitor classes and DTD are automatically derived from a sparsely annotated ANTLR grammar. Mapping tag values onto the type system of the target language yields large increases in performance, automized generation is a must for safety and maintainability. The currently supported target language is JAVA.
Keywords: XML, SAX, DOM, ANTLR, AST, *compiler construction*, *meta-compiling*

1 Introduction

Modern development of language analyzing applications, e.g. document processing or compiler construction, is often based on *declarative* approaches: type inference, context checking etc. are most naturally described as sets of rules. Modern *programming* should deal with high-level objects, like sets, maps, patterns, and predicates. In advanced languages these are supported as "first class residents", in traditional languages like C++ and JAVA they can be implemented using libraries of container classes and traversal (visitor or rewriter) patterns.
To benefit from these advanced features of the underlying programming language the structure of any data model has to be reasonably related to the type system of the language. It is a fact of practical matter that many tools, though mature and well-proven in doing their domain job, do not support a strongly typed output model. Instead they use e.g. *homogenous* trees, in which all nodes are of the same type. So the structural properties are rather *described* than *implemented*. Examples of this design are the abstract syntax trees of the popular ANTLR[8] parser generator and the element trees of the DOM[5] format of XML documents.
This comes from the fact that a mapping to the target type system would mostly be *application dependant*. Standard tools with typed output models can often only be realized in a generic way, using techniques of *meta-programming*.
But analysis of homogenous trees is rather tedious and error-prone: Rule matching has to be explicitly coded, at least with a test of the node type and structure, and all the higher level features of the underlying language, such as inheritance, genericity, dynamic binding and overloading cannot be used. So it should not surprise that mapping structural information to the type system of the target language yields significant profit in maintainability, safety and performance.

2 XML in a Compiler Pipeline

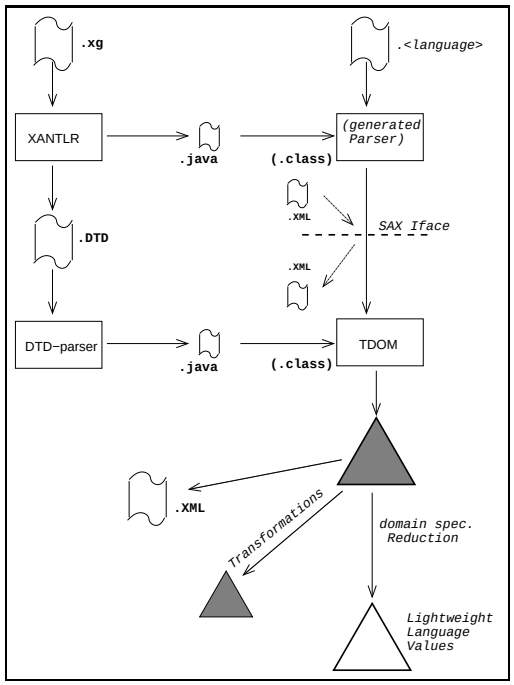
The XANTLR project arose from a given project situation involving two teams in two different instituions, technically connected by a CVS source code repository. One team developed a parser, the other team the back-end of the compiler, and we chose XML as an interchange format.
XML is a convenient way of encoding grammar information (abstract syntax trees, or ASTs) collected by a parser. An AST can be represented as a tree of XML elements (possibly containing literal character data from the input), with the abstract syntax rules specified in the corresponding document type definition (DTD). However, efficient and verifiable generation, transformation and analysis of the XML tree require some support for embedding the XML producer and consumer processes into a given target language, and for exploiting the target language's type system for validation.
In this project setting the grammar and thus also the DTD were fixed, so we could have imple-

mented the heterogenous tree representation by hand. For a grammar of 500+ production rules, however, this would have been a scary task. The XANTLR/TDOM tool chain is an automatic, clean and reusable solution to the problem.

Though our recent work concentrated on eliminating the real ASCII-coded XML files from the tool chain, XML became even more present on the meta-level as a conceptual structure definition language.

3 The XANTLR/TDOM Tool Chain

The tool chain is depicted in the following figure. It integrates $LL(k)$-parser generation, both event-based (SAX) and data-based (DOM) AST communication, validation and light-weight compilable pattern matching, tree reduction and validity-invariant rewriting:



The XANTLR tool chain

3.1 Parsing to XML

Our tool chain starts with XANTLR, an extension to the powerful ANTLR parser generator. The ANTLR-generated parsers produce generic homogenous ASTs that can be serialized to plain XML text. The output can then be processed by a standard XML parser, such as the Xerces[9] Parser to generate SAX events or to construct a DOM tree.

With our extensions, XANTLR can be configured to directly emit SAX events instead of constructing an AST. Any SAX event handler can be connected to the XANTLR-generated parser, thus having access to the XML structure of the parsed input whithout constructing intermediate data structures. A locator implementation is provided, so the point of origin of each syntax element can be tracked down to the input. A persistent XML document can still be obtained (whithout going through AST construction and serialization) by attaching a SAX serializer.

In parallel to parser generation, a DTD equivalent to the input grammar is produced, specifying the parser output for XML consumers:

$$\frac{a\texttt{: }\alpha\texttt{ ;}}{a\texttt{: \{start(}a\texttt{);\} }\alpha\texttt{ \{end(}a\texttt{);\} ;}}$$

$$\frac{a\texttt{ options\{xmlNodeType=pcdata;\}: }\alpha\texttt{ ;}}{a\texttt{: c=}\alpha\texttt{ \{cdata(c);\} ;}}$$

$$\frac{a\texttt{ options\{xmlNodeName=b;\}: }\alpha\texttt{ ;}}{a\texttt{: \{start(}b\texttt{);\} }\alpha\texttt{ \{end(}b\texttt{);\} ;}}$$

XANTLR to ANTLR transformation (schematic)

XANTLR acts as a preprocessor to ANTLR that recognizes special options specifying the desired XML representation and inserts semantic actions which emit the appropriate SAX events. The default is to represent each grammar nonterminal as an element:

$$\frac{a\texttt{: }\alpha\texttt{ ;}}{\texttt{<!ELEMENT }a\ \alpha'\texttt{>}}$$

$$\frac{a\texttt{ options\{xmlNodeType=pcdata;\}: }\alpha\texttt{ ;}}{\texttt{<!ELEMENT }a\texttt{ (\#PCDATA)>}}$$

$$\frac{a\texttt{ options\{xmlNodeType=entity;\}: }\alpha\texttt{ ;}}{\texttt{<!ENTITY \% }a\texttt{ '}\alpha'\texttt{'>}}$$

$$\frac{a\texttt{ options\{xmlNodeName=b;\}: }\alpha\texttt{ ;}}{\texttt{<!ELEMENT }b\ \alpha'\texttt{>}}$$

$$\frac{(\alpha_1\ \ldots\ \alpha_n)}{(\alpha'_1,\ \ldots,\ \alpha'_n)} \qquad \frac{(\alpha_1\ |\ \ldots\ |\ \alpha_n)}{(\alpha'_1\ |\ \ldots\ |\ \alpha'_n)}$$

$$\frac{(\alpha)*}{(\alpha)*} \qquad \frac{(\alpha)+}{(\alpha)+} \qquad \frac{(\alpha)?\ \mid\ (\alpha\ \mid\ )}{(\alpha)?}$$

XANTLR to DTD transformation (schematic)

3.2 Typing XML

A XML document type definition constitutes a grammar that can be mapped to the type system of a given programming language, such as JAVA, allowing for an efficient implementation of the element tree structure. Each element declaration becomes a type declaration in the target language. Besides member fields describing the attributes and content of an element, statically type-checked transformation methods can be defined on each element type, such as replacing a child node by another of the same type, adding a type-conform child node to a list, or changing the value of a non-fixed attribute. These work directly on the target-language-level representation, thus providing an efficient lightweight interface to the document structure, as opposed to the high genericity level and overhead of XSL[2] processing, and eliminating the need for revalidation by incrementally constructing an *a priori* valid document.

Of course, every given DTD has to be compiled into a specialized set of type declarations. Our implementation of this phase, the typed DOM (or TDOM) compiler, reads in a DTD, and produces the following JAVA classes:

- A class encapsulating the input DTD providing runtime access to the document type definition. In the way JAVA reflection handles the *interface* of the generated classes, the DTD model specifies their *semantics*.
- An abstract base class for all elements declared in the given DTD.
- A class for each element. Element classes support both validating construction from a DOM tree and fast, valid-by-typecheck construction from TDOM objects. Methods to get and set attributes and content are provided, as well as conversion back to DOM. Validation of contents models in DOM trees and SAX streams is provided by a small parser generator: for deterministic content models, an efficient LL(1) parser is derived, whereas nondeterministic content can still be handled by a fallback nondeterministic validator automaton.
- A container class for each content choice or sequence. A sequence class is just a typed container record for its elements. A choice container is an abstract base class with one subclass for each alternative, and some methods for distinguishing between these alternatives.
- A class for each attribute. This class will have a default constructor if the attribute is not required, providing the default value. Besides, there is a value constructor, a method to get and (if the attribute is not fixed) to set the current value.
- A visitor class implementing generic tree traversal for all nodes covered by the DTD. Applications can subclass the visitor to implement selective actions upon encountering the desired nodes or patterns in the tree.

$$\frac{\texttt{<!ELEMENT}\ a\ \texttt{(}\alpha\texttt{,}\ \beta\texttt{)>}}{\texttt{class Element\_}a\ \texttt{extends Element \{}\ \alpha'\texttt{;}\ \beta'\ \texttt{\}}}$$

$$\frac{\texttt{(}\alpha\texttt{)?}}{\alpha'\texttt{; boolean has\_}\alpha\texttt{();}} \qquad \frac{\texttt{(}\alpha\texttt{)*}\ \mid\ \texttt{(}\alpha\texttt{)+}}{\alpha'\texttt{[]; int count\_}\alpha\texttt{();}}$$

$$\frac{\texttt{(}\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{)}_i}{\texttt{class Seq\_}i\ \texttt{\{}\ \alpha'_1\texttt{;}\ \ldots\texttt{;}\ \alpha'_n\ \texttt{\}}}$$

$$\frac{\texttt{(}\ldots\texttt{,}\ \alpha\texttt{,}\ \ldots\texttt{)}}{\mathbb{T}(\alpha)\ \alpha\texttt{;}\ \mathbb{T}(\alpha)\ \texttt{get\_}\alpha\texttt{();}\ \ \texttt{void set\_}\alpha\texttt{(}\mathbb{T}(\alpha)\texttt{);}}$$

$$\frac{\texttt{(}\alpha_1\ \mid\ \ldots\ \mid\ \alpha_n\texttt{)}_i}{\texttt{class Choice\_}i\ \texttt{\{}\ \alpha'_1\texttt{;}\ \ldots\texttt{;}\ \alpha'_n\ \texttt{\}}}$$

$$\frac{\texttt{(}\ldots\ \mid\ \alpha\ \mid\ \ldots\texttt{)}_i}{\texttt{class Alt\_}\alpha\ \texttt{extends Choice\_}i\ \texttt{\{}\ \alpha'\ \texttt{\}}}$$

DTD to TDOM transformation (schematic)

3.3 Reducing from XML

The next phase of our compiler pipeline is going to leave the world of XML: Reducing various front-end structures to expressions of a slim kernel calculus is a powerful technique in compiler construction. Not the ASTs representing those different front-end phenomena are processed by the further steps of the compiler, but the tuned and reduced expressions of the kernel language. This transformation step highly depends on the kind of language: Not only "syntactic sugar" is removed, but also rather different syntactic constructs are mapped to the same term structures, subtrees are flattened and stored in hash sets or sequences, declarations are collected and topologically sorted etc. This is performed by the reduction step at the bottom of the tool chain figure: the nodes of the TDOM are traversed using the automatically constructed visitor classes, and expressions of the kernel language are generated. Complex tree analysis is done here at high efficiency: Almost all pattern matching is handled by the JAVA type system, either statically by overloading resolution or dynamically by method dispatching. It is no surprise that performance proves far superior to DOM-based approaches, where node tags are matched at `String` level, children have to be counted etc. Of course the visitor classes (coming for free with TDOM compilation) can be used for any other processing of the TDOM tree, such as searching, sorting, extracting. The expressions of the kernel language still have to he coded as JAVA classes manually because of the very specific requirements imposed on the compilation process by the object language semantics.

4 Future and Related Work

We consider TDOM a replacement for the generic XSL approach aimed at domain specific (i.e., DTD specific) tasks calling for performance and tight embedding into a programming language. Possible interaction, especially with the selection language XPath[3] is a promising field of research.
The type mapping does not yet extend to the lowest level of access, e.g., XML attributes containing numeric values should be accessible as `int` or `double` on language level, and (even more important) attributes of `ID` flavor should be translated to *references* to other TDOM nodes. However, a DTD is not expressive enough to control low-level data mapping. Therefore, we postponed our efforts here and look forward to a mature version of the XML Schema[4] type definition language.
There seem to be several implementations of lightweight DOM variants. The two we have encountered so far, namely DOM light[7] and JDOM[6] provide native JAVA implementations tuned for the most frequently used DOM features. As far as we know, they do not address the issues of validation, validity-preserving transformation and type-driven analysis that are dominant in the TDOM approach.

A feature that has been prototypically implemented in the current TDOM, but that still needs some research, is the automatic generation of XML *compression codecs*. When the DTD of a document is known, all information that can be inferred from the declarations is redundant in the document. For example, the textual size of element tags (which should be verbose and human-readable in XML text, as opposed to the cryptic HTML nomenclature) plays no role in auto-compressed XML.

5 Acknowledgements

The work presented herein contains the consequences drawn from a common project with Ina Schieferdecker, Theofanis Vassiliou-Gioles and others from GMD Fokus, Berlin. Also thanks to Wolfgang Grieskamp, now at microsoft research, who initiated our TDOM activities. The intense discussions with these collegues have always been a source of inspiration.

References

1. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, `http://www.w3.org/TR/2000/REC-xml`.
2. James Clark. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation, `http://www.w3.org/TR/1999/REC-xslt`.
3. James Clark and Steve DeRose. *XML Path Language (XPath)*. W3C Recommendation, `http://www.w3.org/TR/xpath`.
4. Schema Working Group. *XML Schema*. W3C Candidate Recommendation, `http://www.w3.org/XML/Schema`.
5. Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. *Document Object Model (DOM) Level 2 Core Specification Version 1.0*. W3C Recommendation, `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core`.
6. Jason Hunter and Brett McLaughlin. *JDOM*. JDOM Project, http://www.jdom.org.
7. Philippe Kaplan and Thierry Kormann. *DOM light. A fast and "easy-to-use" DOM-like API*. Koala Project, INRIA, `http://www-sop.inria.fr/koala/domlight`.
8. Terence Parr. *ANTLR Reference Manual*. jGuru, `http://www.antlr.org/doc`.
9. Apache XML Project. *Xerces Java Parser*. Apache Software Foundation, `http://xml.apache.org/xerces-j`.