AUTOMATIC CONSTRUCTION OF XML-BASED TOOLS SEEN AS META-PROGRAMMING

Baltasar Trancón y Widemann
Markus Lepper
Jacob Wieland
bt, lepper, ugh@cs.tu-berlin.de
Technische Universität Berlin

Abstract

This article presents XML-based tools for *parser generation* and *data binding generation*. The underlying concept is that of transformation between formal languages, which is a form of meta-programming. We discuss the benefits of such a declarative approach with well-defined semantics: productivity, maintainability, verifiability, performance and safety.

Keywords: XML, SAX, DOM, TDOM, ANTLR, XANTLR, *compiler construction*, *meta-programming*, *parser generation*, *data binding generation*

1. Introduction

1.1 A Possible Land-map of XML Applications

The simple fact that the bandwidth of communication channels and processors has increased rapidly during the last decade allows the revitalization of an old concept known from ancient UNIX days: the use of a simple thing like *text* as central medium for information interchange.

This is reflected by the common agreement upon the necessity of standardization, and the resulting acceptance and vivid participation in the XML standardization processes.

At first glance, these XML-related standards seem quite poor: The basic layer of the specifications ([5]) just regulates the encoding of arbitrarily formed trees.

The next layer should give a notion of "type". Here we find a diversity of different concepts: the ancient DTD, which is not expressive enough, the recently codified W3C-Schema, and more than half a dozen

ISSN 0928-8910

markuslepper.eu

very interesting competitor schema languages, — each of them containing brilliant ideas and different nice features everyone would like to use (cf. the survey given in [10], even newer is [2], based on theoretical considerations in [13]).

But it is just *because* of the most simple notion of "text" the XML kernel imposes on its objects, just because of the absence of almost all typing restrictions and semantic implications, that XML based notations and tools potentially can (and probably will) infiltrate *all areas* of software engineering.

So the notions "XML-based encoding" and "XML-based architecture" will more and more get a significance like "ASCII-based". Indeed the area of "XML-applications" does contain objects from totally divergent disciplines of software engineering, — each with very different underlying mathematical models, different traditions and ways of speaking, different grades of abstractions, etc. This total area could be described by a map with three landmarks in triangular position:

- One vertex of the triangle is made up by instances of XML used as a merely technically determined *coding format* for e.g.
  - tool configuration data, as in [16],
  - network *protocol data units* in client/server architectures, as in SOAP [17], XLANG [12] etc.[1].
  - database interfaces, representing both data and queries[2].
  - definition of meta-models, e.g. business items, e-commerce transaction objects,
- Most remote from that first vertex is the second one, defined by the needs of *authoring*, especially of "compound document". This is a very complex and inherently generic concept: (1) the integration of most heterogenous materials (sound, business objects, graphics etc.) into a well-defined context must be supported, (2) for sake of re-usability and convenience mechanisms for parameterization of types as well as of documents seems highly desirable, and (3) such divergent applications as technical documentations (e.g. using the DOCBOOK DTD), scientific articles (in LaTeX manner), cool web pages, table-oriented data base views for web information services (HTML-like or ECMA-script backend) must be representable.

[1] Having a look at the thousands of pages produced by international standardization boards just talking about *coding*, one feels the relief that hundreds of hours of valuable human labor will be saved, since now we can argue about *content*, and put at the end of each project's agenda "... and further shall there be a standard XML encoding.".

[2] There are numerous approaches, in industrial practice as well as in academic research, e.g. [9] and [1], — the latter providing further references.

ISSN 0928-8910

markuslepper.eu

This is the original field of SGML, and is covered in the authors' toolkit by DDD(see fig. 1, [11]).

- Somehow related to both other areas, but requiring special treatment w.r.t. the correctness of transformations, is the usage of XML tree structures for the representation of *"terms" of a given formal language*. Some self-applications, like schema languages, use these kinds of semantics. Also the more elaborated species of the "coding formats" mentioned above are defined using simple grammars, and re-appear in this group.
  A promising approach on the level of system architecture for sake of inter-operability is the representation of "abstract syntax trees" ("ASTs", which are used as output format of most parser codes) by XML structures.

1.2 Automatic Parser Generation by XANTLR and TDOM

While the authors have worked in all these three areas (cf. figure 1), the following presentation will concentrate on their work on this very last topic, the automated generation of an XML-based AST representation and its further processing by semantic transformations. The main focus is on two tools: XANTLR transforms annotated grammars to DTD and parser code (see section 2), and TDOM constructs a typed document model for further processing (see section 3).

While being usable independently, XANTLR and TDOM are designed to work in a pipeline. Indeed this combination has already been successfully used in one medium scale industrial project[3]: Our "TTthree" TTCN-3 compiler. The production process and the involved personnel is depicted in figure 2, the dashed lines containing the automated processing. For the front-end of our TTCN-3 compiler only a single XANTLR-grammar of about 2,400 lines is given.

XANTLR is an extension to the well-known ANTLR parser generator. ANTLR very comfortably and reliably generates LL(k) parsers with user definable semantic actions and explicit backtracking.

XANTLR can be thought of as a "ANTLR with a preprocessor". It reads a grammar file, where nonterminals are enriched with special annotations. These annotations control the automatic definition and generation of an XML-based representation of the resulting parse tree: (1) Semantic actions emitting SAX events are automatically inserted into

[3]While all tools in figure 1 have at least succeeded with test applications.

ISSN 0928-8910

markuslepper.eu

the grammar definition, and (2) a DTD is derived which exactly describes the abstract syntax.

The compilation of the generated JAVA sources yields a parser which analyzes a text in the language under implementation and delivers the recognized AST by emitting the corresponding SAX events.

On the other hand, TDOM is realized as a DTD-to-JAVA compiler. Each `ELEMENT` definition from the DTD is translated to a JAVA `class` definition. Each of these classes provides: parser methods, which consume XML (SAX or W3C-DOM, respectively) and create a corresponding *typed document object model*, validated against the element's content model, and a set of validity preserving modification methods.

The resulting code is connected to the parser code via a SAX interface, preserving the original locator information. The resulting AST object is further processed by "hand-written" visitor style code, which inherits from the visitor prototypes also automatically generated by TDOM.

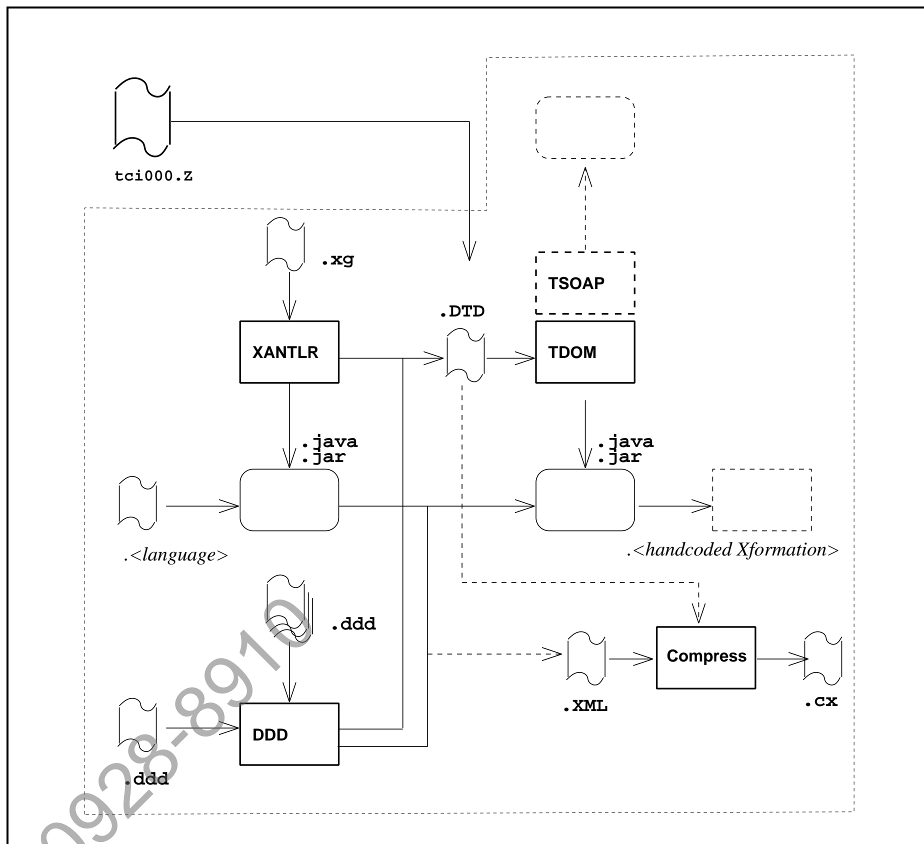

Figure 1: Current Status of the Authors' XML Related Toolkit

ISSN 0928-8910

markuslepper.eu

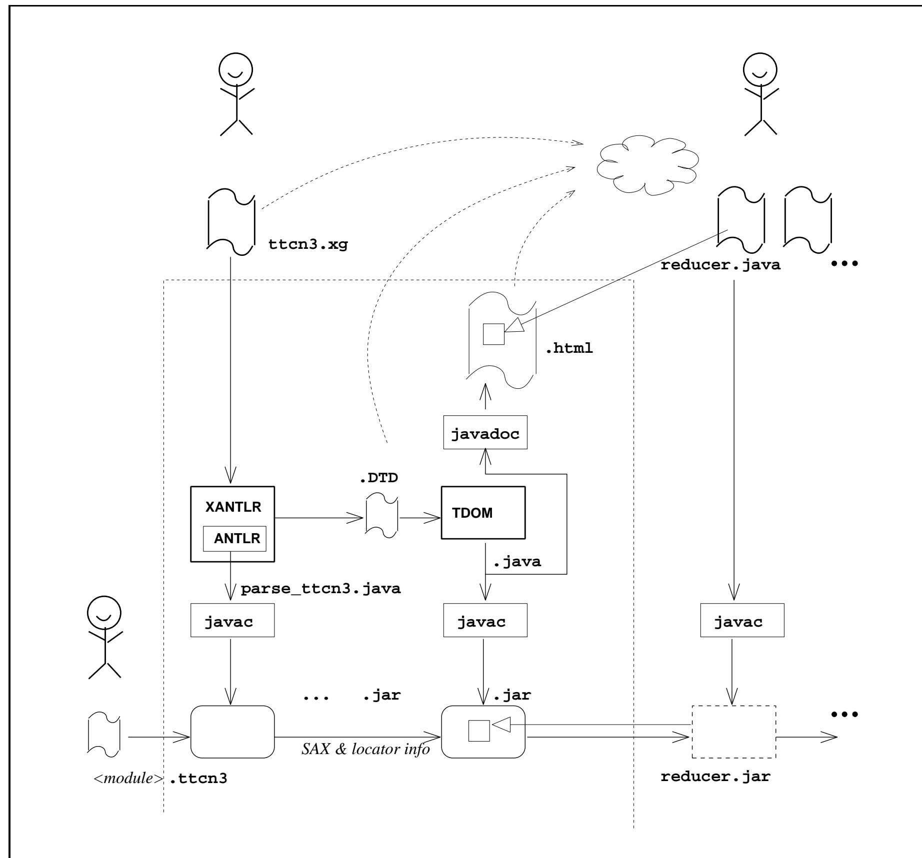

Figure 2: The parser and object model generation process

1.3 Trees and Transformations

1.3.1 Typed Trees vs. Homogenous Trees. As soon as talking about XML and AST, one comes across two (de-facto-) standard object models: W3C-DOM and the ANTLR default trees, respectively. Both are *homogenous* tree models, i.e. the tag classes of the elements are not mapped onto (/not known to) the type system of the hosting language.

But obviously there are many arguments for mapping the "grammar" of the documents as closely as possible onto the hosting type system:

- Expressiveness: Powerful mechanisms of the hosting language (inheritance, genericity, built-in container classes) may be only accessible via the type system.
- More programming errors may be detected statically at compile time.

ISSN 0928-8910

markuslepper.eu

- Better code performance: compile time type information is exploited by any intelligent compiler for severe optimizations.
- Built-in documentation mechanisms related to the type system can be used.
- Maintainability is improved by type checking of hosting language.

For these reasons, our TDOM realizes a *typed* document object model, in contrast to W3C-DOM and genuine ANTLR trees.

1.3.2 Text Transformation seen as Meta-Programming.

At first glance, the fact that XANTLR as well as TDOM produce *source text* for a target language, could be considered to be just an implementation "hack", — in our case: to compensate the lack of genericity in JAVA.

The picture changes immediately, as soon as the transformations can be *completely described as morphisms between two term algebras with exactly definable semantics*. Suddenly well-known results from "academic" research and corresponding standard techniques are applicable, from most diverse fields like grammar morphisms, graph transformations, language translation, logic, context analysis, etc.

On the output side this condition is doubtlessly met when transforming into a "denotational" format, like XANTLR transforming grammar rules into DTD content models. It *may* even be met when generating code of an "imperative" language, since code generation can (and should!) of course always be planned in such way that its output obeys strict limitations, e.g. behaves strictly functionally, or has only local side effects etc.

On the input side of the transformations there is justified hope that more and more standard interface definitions will migrate to an XML-based encoding. The corresponding DTD- or schema-based declarations always induce a "canonical term algebra", which may be suitable as a basis for specifying the semantic transformations. To be really seen as an instance of "meta-programming", such a mathematical specification seems indispensable.

The automatic generation of source code is well known from IDL-compilers, — a topic which can easily be treated exactly, since no dynamic behavior is involved. But in the future more and more CASE-tools will be controllable by XML-based interface languages. A much more challenging task will be the automated generation of gluing software between these kinds of tools, because the dynamic semantics of the meta-models involved have to be translated correctly.

ISSN 0928-8910

markuslepper.eu

1.3.3 Type-safe Transformations.

The transformation encoding in our practical application projects are (still) encoded "by hand" and use an "imperative" language, — but not an imperative style of programming: algorithms are encoded by pattern matching, realized through visitor classes derived from the automatically generated prototypes. In a majority of cases a "pure functional style" of coding evolves quite naturally.

Please note, that — dialectically — XSLT, while being specially designed for "declarative" definitions of transformations, has full Turing-power, with all known consequences. Potentially TDOM-based visitors have nearly the same degree in polymorphism as XSLT, e.g. changes in the definition of the document's structure do not necessarily affect the existing visitor code.

2. Parser Generation and XML

Since parsing techniques were developed in a time when memory was expensive and each allocated data chunk had to be justified, classical parsers (and thus the output of classical parser generators) are designed for *single-pass* translation. In such a parser, the state transitions associated with the matching of terminal symbols or the construction of nonterminals are labeled with side effects (sometimes called semantic actions). The output of the parser consists of the trace of side effects produced by the consumption of a whole input unit.

Action parsers are very efficient for simple input languages that can be handled in such a linear way. However, they have severe deficiencies: First, they cannot handle forward references gracefully, resulting in a *definition before use* rule, which is often undesirable in abstract descriptive languages. Second, and more important, the structure of the parser output is given only implicitly by the set of possible action traces. Since semantic actions may invoke arbitrary code, there is no general way of specifying the results of parsing.

The first issue is remedied by means of *abstract syntax trees* (ASTs), i.e., data structures that capture the relevant content of input documents and abstract from redundancies and peculiarities of the input language, such as keywords and layout constraints. Information that could not be obtained in the first parsing pass can then be collected by iterated (and hopefully efficient) traversal over the AST data. However, as long as ASTs are created imperatively by semantic actions, the second problem remains unsolved.

If a parser is seen as the implementation of a formal transformation, the output language must be as well-defined as the input language. I.e.,

ISSN 0928-8910

markuslepper.eu

the definition given to the parser generator must not only specify an input grammar decorated with stray semantic actions, but rather a *grammar morphism* giving exact mappings of rules of the input grammar to rules of the output grammar. Then, transformations can be safely composed by applying a second parser to the results of the first one, since the critical transition from the domain of formal languages to the domain of side effects is deferred.

Once such a transformational parser exists, it can be shared by different applications that process the given input language. For example, a compiler and a formatter might generate code and documentation for a literate programming language out of the same intermediate representation; or a browser, a database and a XSLT processor might use the same XML to SAX parser.

The structured constituents of XML, i.e., elements and attributes, allow for a standardized universal representation of ASTs. But some thought has to be given to efficiency: Textual XML, though simple to process, is extremely redundant, so parse trees may grow by several orders of magnitude compared to the corresponding input (see Appendix A). Whether a data (DOM) based or an event (SAX) based interface is preferable, depends on the application. The advantage of event-driven implementations in terms of memory efficiency may be compensated by the need to split complex transformations into several linear passes, whereas a data structure in memory provides random access.

2.1 The XANTLR Algorithm

Parsers generated by the powerful ANTLR[14] $LL(k)$ parser generator produce generic homogeneous ASTs that support serialization to plain XML text. The output can then be processed by a standard XML parser, such as the Xerces[3] Parser to generate SAX events or to construct a DOM tree.

We have developed XANTLR, an XML-aware extension to ANTLR. XANTLR generated parsers directly emit SAX events, which can be configured for each nonterminal individually, using an extension of the genuine ANTLR `option` mechanism. Since the output format is not determined by hand-coded semantic actions, but by rule-by-rule annotations controlling the grammar mapping, XANTLR is a genuinely declarative transformation formalism as postulated in 1.3.1.

Any SAX event handler can be connected to the XANTLR-generated parser, thus having access to the XML structure of the parsed input without constructing intermediate data structures. A locator implementation is provided, so the point of origin of each syntax element can

ISSN 0928-8910

markuslepper.eu

be tracked down to the input. A persistent XML document can still be obtained (without going through ANTLR-AST construction and serialization) by attaching a SAX serializer. Thus, every language that is feasibly $LL(k)$-parsable can be assigned a canonical XML representation and automatically translated into XML documents.[4] For nontrivial post-processing of the AST data, XANTLR can be combined with our typed DOM generator TDOM (cf. section 3).

The XANTLR preprocessing algorithm is specified by the following transformation rules. Parts of the grammar definition (XANTLR's input) that are matched literally are printed in black, meta-variables are grey, with Latin and Greek letters denoting single nonterminals and regular expressions, respectively. When all possible transformations are applied, the result is a plain ANTLR grammar with semantic actions (written in curly braces) that fire appropriate SAX events:

- The default is to represent each nonterminal as an element of the same name:

$$\frac{a\texttt{: } \alpha \texttt{ ;}}{a\texttt{: \{startElement("}a\texttt{");\} } \alpha \texttt{ \{endElement("}a\texttt{");\} ;}}$$

- A different name can be given:

$$\frac{a \texttt{ options\{xmlNodeName=b;\}: } \alpha \texttt{ ;}}{a\texttt{: \{startElement("b");\} } \alpha \texttt{ \{endElement("b");\} ;}}$$

- A nonterminal can be marked as a *lexical category*, containing literal data:

$$\frac{a \texttt{ options\{xmlNodeType=pcdata;\}: } \alpha \texttt{ ;}}{a\texttt{: } \alpha \texttt{ \{characters(}\alpha\texttt{.getText());\} ;}}$$

- A nonterminal may become an anonymous content building block:

$$\frac{a \texttt{ options\{xmlNodeType=entity;\}: } \alpha \texttt{ ;}}{a\texttt{: } \alpha \texttt{ ;}}$$

In parallel to parser generation, a DTD is produced, specifying the grammar of the parser's output for XML consumers. Since input and output languages are homomorphic, the DTD derivation algorithm can be defined inductively by the following rules:

- Per default, rules are mapped to elements (with optional renaming):

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

[4] We expect this to be a major application of XANTLR.

ISSN 0928-8910

markuslepper.eu

- Lexical categories become leaf elements:

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

- Rules may be mapped to content building blocks (parameter entities):

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

The prime operator on meta-variables denotes the transformation of regular expressions from ANTLR to DTD notation and is recursively defined as follows:

- Sequence and choice combinators are preserved:

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

- So are the modifying combinators:

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

- Empty sub-alternatives (not expressible in DTD) are mapped to `?`:

$$\frac{\texttt{(}\alpha_1\ \texttt{|}\ \dots\ \texttt{|}\ \alpha_n\ \texttt{|}\ \texttt{)}}{\texttt{(}\alpha'_1\ \texttt{|}\ \dots\ \texttt{|}\ \alpha'_n\texttt{)?}}$$

Validity of all possible parser output with respect to the so derived DTD is *provable* by the same induction patterns, thus giving the production tool XANTLR formal semantics.

The ANTLR-generated top-down parsers, together with an event-based output format, have the valuable property that the AST-producing parser and the AST consumer can be pipelined. Start tag events do not interfere with the backtracking algorithm used for disambiguation, and are emitted as soon as the parser enters a rule in deterministic mode. This early-response behavior cannot be emulated by LR bottom-up parsers, where a nonterminal becomes manifest only after the reduction step of the corresponding grammar rule. On the other hand, though LL parsing can be extended by backtracking to emulate LR recognition capabilities (as supported by ANTLR), performance for typical LR phenomena (e.g., for left-associative binary operators) is rather poor.

ISSN 0928-8910

markuslepper.eu

2.2 Related Work

ANTLR allows the definition of AST consumers (called tree parsers) by a grammar. That grammar, however, has to be written by hand, and its appropriateness for the actual output of an AST-producing parser cannot be verified automatically.

The technique of specification in terms of homomorphisms, which is the formal principle behind XANTLR, is well known and heavily used in the area of algebraic specification (see [15]) and program transformation (see [4]), but to our knowledge, it has not been coupled with parser generation before.

3. Data Binding Generation for XML

A XML document type definition (or schema) 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 contents 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 XSLT processing, and eliminating the need for revalidation by incrementally constructing an *a priori* valid document.

3.1 The TDOM Algorithm

Figure 3.1 shows the transformation rules behind the TDOM code generator. These are far from complete and yield only most abstract pseudo-code, in contrast to the exact formal specification of XANTLR. However, they may serve as a brief summary of the following prose descriptions.

Of course, every given DTD has to be compiled into a distinct specialized set of type declarations. Our implementation, the 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 captures the JAVA properties (e.g., fields and methods) of classes, the DTD model specifies their DTD properties (attributes and content model).

ISSN 0928-3910

markuslepper.eu

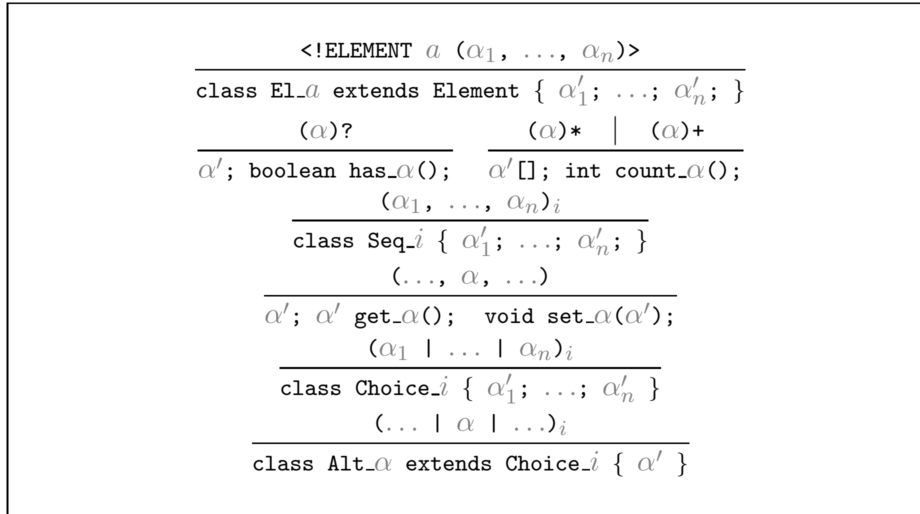

Figure 3: DTD to TDOM transformation (schematic)

- 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 W3C-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 W3C-DOM. Validation, (i.e., type assignment) of content models read from W3C-DOM trees and SAX streams (e.g., output of XANTLR-generated parsers) 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 fall-back breadth-search validator automaton.
- A dedicated 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.

Note that these classes do *not* implement the untyped W3C-DOM interface, because this would enable arbitrary destructive modification of the document structure that may yield an invalid document object.

ISSN 0928-8910

markuslepper.eu

In a typed environment such as JAVA TDOM, these would show up as runtime type errors[5]. If such an access to the document is desired, abandoning the constraints given by the DTD, the TDOM document can still be transformed to W3C-DOM, mangled in every desirable (and undesirable) way, and eventually revalidated from scratch into a new TDOM object.

The measured performance of TDOM at low-level data access is roughly comparable to that of Xerces DOM, which is a full-fledged, highly tuned JAVA implementation (cf. Appendix A). We assume that TDOM will exhibit superior productivity in cases of complex tree pattern matching, easily implementable in the visitor style. Since our development of TDOM saved us from hand-coding complex W3C-DOM analyzers so far, we have no significant benchmarks in this area.

3.2 Related Work

There seem to be several implementations of lightweight DOM variants. The two we have encountered so far, namely DOM light[8] and JDOM[7] provide native JAVA implementations tuned for the most frequently used W3C-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.

4. Comprehensive Example: An Arithmetics Interpreter

As an example for using XANTLR and TDOM, we implemented an interpreter (abstract machine) for a toy language that could be found in any textbook on compiler construction: A program consists of a sequence of assignments to numeric variables, followed by an expression. Only basic arithmetic operations are supported. Each variable that is assigned may be referenced in a subsequent assignment or the final expression. The parser grammar (technical details omitted) is:

```
program   : (def)* expr EOF ;
def       : variable ":=" expr ";" ;
expr      : term (sumOp term)* | sumOp term ;
term      : factor (productOp factor)* ;
factor      options { xmlNodeType = entity ; }
          : number | variable | "(" expr ")" ;
number      options { xmlNodeType = pcdata ; }
          : NUMBER ;
variable    options { xmlNodeType = pcdata ; }
          : VARIABLE ;
```

[5] the infamous ClassCastException

ISSN 0928-8910

markuslepper.eu

```
sumOp     : plus | minus ;
plus      : "+" ;
minus     : "-" ;
productOp : times | divide ;
times     : "*" ;
divide    : "/" ;
```

The generated DTD reads as follows:

```
<!ELEMENT program (def*, expr)>
<!ELEMENT def (variable, expr)>
<!ELEMENT expr ((term, (sumOp, term)*) | (sumOp, term))>
<!ENTITY % factor '(number | variable | expr)'>
<!ELEMENT term (%factor;, (productOp, %factor;)*)>
<!ELEMENT number (#PCDATA)>
<!ELEMENT variable (#PCDATA)>
<!ELEMENT sumOp (plus | minus)>
<!ELEMENT plus EMPTY>
<!ELEMENT minus EMPTY>
<!ELEMENT productOp (times | divide)>
<!ELEMENT times EMPTY>
<!ELEMENT divide EMPTY>
```

The visitor that TDOM generates for this DTD can be used as a base class for, e.g., an interpreter. The complete operational semantics of our toy language can be implemented by overriding six out of twelve methods for visiting elements. When invoked on the AST of an input program, the program result is computed in the field `value`:

```
public class ArithExprReducer
  extends arith.Visitor {
  public double value ;
  protected abstract void setVariable(String name, double value) ;
  protected abstract double getVariable(String name) ;
  public void visit(Element_def def) {
    visit(def.getElem_1_expr()) ;
    setVariable(def.getElem_1_variable().getPCData(), value) ;
  }
  public void visit(Element_number number) {
    value = Double.parseDouble(number.getPCData()) ;
  }
  public void visit(Element_variable variable) {
    value = getVariable(variable.getPCData()) ;
  }
  public void visit(Element_expr.Choice_1_Alt_1_Seq_1 sum) {
    final double backupValue = value ;
    super.visit(sum) ;
    switch(sum.getElem_1_sumOp().getChoice_1().getAltIndex()) {
    case Element_sumOp.Choice_1_Alt_1.ALT_INDEX:
      value = backupValue + value ;
      break ;
    case Element_sumOp.Choice_1_Alt_2.ALT_INDEX:
      value = backupValue - value ;
    }
  }
  public void visit(Element_expr.Choice_1_Alt_2 signed) {
```


ISSN 0928-8910

markuslepper.eu

```
    super.visit(signed) ;
    if(signed.getElem_1_sumOp().getChoice_1().isAlt_2())
      value = -value ;
  }
  public void visit(Element_term.Seq_1 product) {
    final double backupValue = value ;
    super.visit(product) ;
    switch(product.getElem_1_productOp().getChoice_1().getAltIndex()) {
    case Element_productOp.Choice_1_Alt_1.ALT_INDEX:
      value = backupValue * value ;
      break ;
    case Element_productOp.Choice_1_Alt_2.ALT_INDEX:
      value = backupValue / value ;
    }
  }
}
```

5. Conclusion and Future Work

We have presented running (albeit prototypic) instances of meta-programming tools, i.e., *program-generating* programs that obey *formal transformation* semantics. Such tools do more than just save the user from tedious source code editing. They also allow formal methods to penetrate deep into the process of software engineering, easing the tasks of *maintenance* and *verification*. With *interoperability* being an indispensable requirement for such tools, XML has been shown to serve well as the common denominator of a tool chain with respect to both data representation and specification.

We consider TDOM a replacement for the generic XSLT approach aimed at domain specific (i.e., schema specific) tasks calling for performance, safety and tight embedding into a hosting language. Possible interaction, especially with the selection language XPath[6] 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 `IDREF` flavor should be translated to *references* to other TDOM nodes. However, DTD is not expressive enough to control the mapping of simple types. Therefore, we postponed our efforts here and look forward to a mature unified instance of the numerous XML Schema type definition languages.[10]

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* (cf. Figure 1). When the DTD of a document is known, all information that can be inferred from the declarations is redundant in the document. E.g., the textual size of element tags (which

ISSN 0928-8910

markuslepper.eu

should be verbose and human-readable in XML text, as opposed to the cryptic HTML nomenclature) plays no role in auto-compressed XML.

The desirable increase in interoperability of CASE tools and transformation systems requires further research on *compositionality* of grammars and re-usability of visitor classes, an issue for theoretic research as well as for practical software architecture design.

Appendix A. Some Numbers

As a kind of informal benchmark, we gathered some statistics about a real-world application of XANTLR and TDOM. In fact, these numbers are taken from the aforementioned compiler construction project that actually enforced the development of such tools by sheer size. The grammar definition that served as input for XANTLR defines a typical domain specific imperative programming language. The resulting DTD was compiled with TDOM. Then, to obtain some performance characteristics, the parser was configured to produce textual XML, and both Xerces-DOM and TDOM ASTs out of the same example program. Finally, all terminals of type `identifier` in the abstract syntax were retrieved from the two AST models.[6] The result indicates several things:

(1) The DTD generated by XANTLR is a concise specification of the parser. It actually reads much easier than the input grammar, because technical details such as disambiguation or error handling are out of the way.

(2) The overhead due to type-safety in TDOM is mainly static. While the generated data binding code is tremendously big compared to the DTD (and does put quite a bit of extra load on the JAVA class loader), runtime access seems to perform similar to a good-quality homogeneous (untyped) DOM.

(3) Not to our surprise, using TDOM yields no significant improvement in runtime performance. The main benefits are presumably found in other phases of the software life-cycle, namely verification and maintenance. This is very much harder to quantify, and should therefore be treated as a conjecture.

Sample Grammar (lines of code)	
XANTLR Grammar	2,400
ANTLR Parser Class	12,600
DTD	673
TDOM Classes	53,800
Sample Instance Document	
Source file	196k
XML AST file	1,389k
DOM vs. TDOM Retrieval	
Total No. of Nodes	30,077
Nodes with Tag “Identifier”	4,213
DOM getByTagName	113ms
TDOM visit	111ms

[6] Accessing all elements of a given tag name is implemented as a builtin function in DOM, and by overwriting one method of the generic visitor in TDOM.

ISSN 0928-8910

markuslepper.eu

References

[1] The lore project. Technical report, Stanford University. http://www-db.stanford.edu/lore/.

[2] . *RELAX (Regular Language description for XML)*. Relaxer Working Group, http://www.xml.gr.jp/relax/.

[3] Apache XML Project. *Xerces Java Parser*. Apache Software Foundation, http://xml.apache.org/xerces-j.

[4] J. Bergstra, T. Dinesh, J. Field, and J. Heering. A complete transformational toolkit for compilers. Technical Report CS-R9601, CWI, January 1996.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C Recommendation, http://www.w3.org/TR/2000/REC-xml.

[6] J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C Recommendation, http://www.w3.org/TR/xpath.

[7] J. Hunter and B. McLaughlin. *JDOM*. JDOM Project, http://www.jdom.org.

[8] P. Kaplan and T. Kormann. *DOM light. A fast and "easy-to-use" DOM-like API*. Koala Project, INRIA, http://www-sop.inria.fr/koala/domlight.

[9] G. Kappel, E. Kapsammer, S. Rausch-Schott, and W. Retschitzegger. X-ray – toward integrating xml and realational database systems. In *Conceptual Modeling – ER 2000*. Springer LNCS 1920, 2000.

[10] D. Lee and W. W. Chu. Comparative analysis of six XML schema languages. In *ACM SIGMOD Record 29(3)*, http://www.cobase.cs.ucla.edu/tech-docs/dongwon/sigmod-record-00.ps, 2000.

[11] M. Lepper, B. Trancón y Widemann, and J. Wieland. Minimze mark-up ! – Natural writing should guide the design of textual modeling frontends. In *Conceptual Modeling — ER2001*, volume 2224 of *LNCS*. Springer, November 2001.

[12] Microsoft Corporation, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm. *XLANG — Web Services for Business Process Design*, 2001.

[13] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, http://www.cobase.cs.ucla.edu/tech-docs/dongwon/mura0619.ps, august 2001.

[14] T. Parr. *ANTLR Reference Manual*. jGuru, http://www.antlr.org/doc.

[15] P. Pepper and M. Wirsing. A Method for the Development of Correct Software. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, LNCS 1009. Springer, 1995.

[16] G. Trausmuth and W. Schneider. Industrial experience with an XML based configuration service for Java applications. In *Proc. of the XSE Workshop*, pages 44–47, 2001.

[17] W3C Note, http://www.w3.org/TR/SOAP. *Simple Object Access Protocol (SOAP) 1.1*.

ISSN 0928-8910

markuslepper.eu