JavaML: A Markup Language for Java Source Code

Greg J. Badros*

Dept. of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA

Abstract

The classical plain-text representation of source code is convenient for programmers but requires parsing to uncover the deep structure of the program. While sophisticated software tools parse source code to gain access to the program's structure, many lightweight programming aids such as `grep` rely instead on only the lexical structure of source code. I describe a new XML application that provides an alternative representation of Java source code. This XML-based representation, called JavaML, is more natural for tools and permits easy specification of numerous software-engineering analyses by leveraging the abundance of XML tools and techniques. A robust converter built with the Jikes Java compiler framework translates from the classical Java source code representation to JavaML, and an XSLT stylesheet converts from JavaML back into the classical textual form.

Keywords: Java, XML, abstract syntax tree representation, software-engineering analysis, Jikes compiler.

1 Introduction

Since the first computer programming languages, programmers have used a text representation as the medium for encoding software structure and computation. Over the years, techniques have been developed that largely mechanize the front-end of compilers—the part that performs the lexical analysis and parsing necessary to uncover the structure of programming language constructs represented as plain text. Tools such as Lex/Flex and Yacc/Bison [42] automate these tedious tasks by using well-founded concepts of regular expressions and grammars. Regular expressions describe how individual characters combine to form tokens, and the grammar enumerates how higher-level constructs are composed recursively of other constructs and primitive tokens. Together, these procedures convert a sequence of characters into a data structure called an *abstract syntax tree* (AST) which more directly reflects the structure of the program.

The textual representation of source code has several nice properties. It is fairly concise and is similar to natural languages, often making it easy to read. Text is a universal data format thus making source code easy to exchange and manipulate using a wide variety of tools including text editors, version control systems, and command pipeline utilities such as `grep`, `awk`, and `wc`.

Nevertheless, the classical source representation has numerous problems. The syntax of popular contemporary languages such as C++ and Perl push the limits of parsing capabilities. Constructing a front-end for such languages is difficult despite the support from tools. Perhaps more disconcerting is that evolving the syntax of the language often requires manipulating a fragile grammar. This limitation complicates handling an evolving language.

1.1 Text representation and software tools

The most significant limitation of the classical source representation is that the structure of the program is made manifest only after parsing. This shortcoming forces language-specific parsing functionality to be duplicated in every tool that needs to reason about the program beyond its lexical nature. Compilers, by necessity, must work with the AST, and numerous other software-engineering tools would benefit from access to a structured representation of the source code. Unfortunately, many software-engineering tools do not embed a parser and thus are limited to lexical tasks.

There are several reasons why tool developers often avoid embedding a parser in tools. As mentioned previously, building a complete front-end is challenging for syntactically-complex languages. Although re-use (e.g., of the grammar definition) simplifies the implementation, the resulting AST is not always intuitive. An AST typically reflects quirky artifacts of the grammar rather than representing the programming-level constructs directly. Additionally, embedding the front-end of a compiler may be deemed overkill when targeting a simple analysis that can do "well enough" with lexical information.

Other complications arise if a transformation of the source code is desired: a change in the AST must ultimately be reflected in the classical source representation because that is the primary long-term storage format. Recreating a text representation from an AST is most straightforwardly done using an unparsing approach that can create undesired lexical side effects (e.g., changes in indentation or whites-

*Corresponding author: `gjb@cs.washington.edu`

pace). Such changes can confuse the other lexical tools that a developer relies upon. For example, a version control system is unable to disambiguate between a meaningful change and a gratuitous one effected unintentionally.

Finally, using a parser in a tool necessarily targets that tool to a specific language, thus reducing its applicability and generality. Worse, because there is no standard structured external representation of a source program, supporting inter-operability of independent tools even targeting the same programming language is very difficult.

The end result of these complications is that developers often use simple, lexically-oriented tools such as `grep` or search-and-replace within an editor. This approach sacrifices accuracy: imagine wanting to rename a local variable from `result` to `answer`. With simple search-and-replace, all occurrences of the word will be changed, even if they refer to characters inside comments, literal strings, or an unrelated instance field.

An alternate route taken by some developers is to rely instead on a fixed set of tools provided within an integrated development environment (IDE) that has access to the structure of their source program via an integrated language-specific parser. This approach sacrifices flexibility. IDEs generally provide only a limited set of capabilities and extending those is hard. Additionally, analyses and transformation on source code are often hard to automate or perform in batch using existing interactive environments. Some more advanced IDEs, such as IBM VisualAge for C++ [48], expose an application programming interface to the representation of the program. Although an improvement, this technique still suffers from an inability to separate simple tools from a complex environment and additionally creates a dependency on proprietary technology that may be undesirable.

1.2 A solution

One of the fundamental issues underlying the above problems is the lack of a canonical structured representation of the source code. We need a universal format for directly representing program structure that software tools can easily analyze and manipulate. The key observation is that XML, the eXtensible Markup Language [9], provides exactly this capability and is an incredibly empowering complementary representation for source code.

In this paper, I introduce the Java Markup Language, JavaML—an XML application for describing Java source programs. The JavaML document type definition (DTD) specifies the various elements of a valid JavaML document and how they may be combined. There is a natural correspondence between the elements and their attributes and the programming language constructs they model. The structure of the source program is reflected in the nesting of elements in the JavaML document. With this representation, we can then leverage the wealth of tools for manipulating and querying XML and SGML documents to provide a rich, open infrastructure for software engineering transformations and analyses on Java source code.

JavaML is well-suited to be used as a canonical representation of Java source code for tools. It shares most of the strengths of the classical representation and overcomes many weaknesses. The next section describes relevant features of Java and XML and section 3 details the markup language and the implementations of converters between the classical representation and JavaML. Section 4 gives numerous examples of how existing XML and SGML tools can be exploited to perform source code analyses and transformations on the richer representation provided by JavaML. Sections 5 and 6 describe related work and suggest avenues for exciting future work, and section 7 concludes. The full document type definition (DTD) for JavaML appears in appendix A and further examples of converted source code are available from the author's JavaML web page [4].

2 Background

The Java Markup Language is influenced by and benefits from numerous features of the two technologies it builds a bridge between: Java and XML.

2.1 Java

Although the XML-based representation of programming language constructs is language independent, Java is an excellent candidate for experimenting with these ideas and techniques.

Java is a popular object-oriented programming language developed by Sun Microsystems in the mid-1990s [3, 25]. It features a platform-independent execution model based on the Java Virtual Machine (JVM) and owes its quick acceptance to its use as a programming language for World Wide Web applications. Java combines a simple object model reminiscent of Smalltalk [26] with Algol block structure, a C++-like [49] syntax, a static type system, and a package system inspired by Modula-2 [10].

As in most other object-oriented (OO) languages, the primary unit of decomposition in Java is a *class* which specifies the behaviour of a set of objects. Each class can define several *methods*, or behaviours, similar to functions or procedures. A class can also define *fields*, or state variables, that are associated with *instances* of the class called *objects*. Classes can inherit behaviour and state from *superclasses*, thus forming a hierarchy of inter-related classes that permits factoring related code into classes at the top of the hierarchy, and encourages re-use. Behaviours are invoked by sending a *message* to a target receiver object that is a request to execute a method defined for that class. Choosing what method to execute in response to a message is called *dynamic dispatch* and is based on the run-time class of the object receiving the message. For example, an instance of the `ColoredBall` class may respond to the `draw` message differently than an instance of a `Ball` class. This ability to behave differently upon receipt of the same message is largely responsible for the extensibility benefits touted by the OO community.

Java is being widely used both in industry and in education, and it remains popular as a programming language on the web. Unlike C++, a Java class definition exists in a single, self-contained file. There are no separate header files and implementation files, and Java is largely free from order-dependencies of definitions. A method body (when present) is always defined immediately following the declaration of the method signature. Additionally, Java lacks an integrated preprocessor. These features combine to make Java source programs syntactically very clean and make Java an ideal language for representing using XML.[1]

2.2 XML: Extensible Markup Language

XML is a standardized eXtensible Markup Language [9] that is a subset of SGML, the Standard Generalized Markup Language [37]. The World Wide Web Consortium (W3C) designed XML to be lightweight and simple, while retaining compatibility with SGML. Although HTML (HyperText Markup Language) is currently the standard web document language, the W3C is positioning XML to be its replacement. While HTML permits authors to use only a predetermined fixed set of tags in marking up their document, XML allows easy specification of user-defined markup tags adapted to the document and data at hand [28, 27].

An XML document consists simply of text marked up with tags enclosed in angle braces. A simple example is:

```
<?xml version="1.0"?>
<!DOCTYPE email SYSTEM "email.dtd">
<email>
  <head>
    <to>Mom</to>
    <to>Dad</to>
    <from>Greg</from>
    <subject>My trip</subject>
  </head>
  <body encoding="ascii">
   The weather is terrific!
  </body>
</email>
```

The `<email>` is an open tag for the `email` element. The `</email>` at the end of the example is the corresponding close tag. Text and other nested tags can appear between the open and close constructs. Empty elements are allowed and can be abbreviated with a specialized form that combines the open and close tags: `<tag-name/>`. In the above document, the `email` element contains two immediate children elements: a `head` and a `body`. Additionally, an XML open tag can associate attribute/value pairs with an element. For example, the `body` element above has the value `ascii` for its `encoding` attribute. For an XML document to be *well-formed*, the document must simply conform to the numerous syntactic rules required of XML documents (e.g., tags must be balanced and properly nested, attribute values must be of the proper form and enclosed in quotes, etc.).

A more stringent characterization of an XML document is *validity*. An XML document is valid if and only if it both is well-formed and adheres to its specified *document type definition*, or *DTD*. A document type definition is a formal description of the grammar of the specific language to be used by a class of XML documents. It defines all the permitted element names and describes the attributes that each kind of element may possess. It also restricts the structure of the nesting within a valid XML document. The preceding XML example is valid with respect to the following DTD:

```
<!-- email DTD -->
<!ENTITY % encoding-attribute
  "encoding (ascii|mime) #REQUIRED">
<!ELEMENT email (head,body)>
<!ELEMENT head (to+,from,subject?)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT body (#PCDATA)>
<!ATTLIST body
    encrypted (yes|no) #IMPLIED
    %encoding-attribute;>
```

According to this DTD, there are six element types. The `email` element must contain exactly one `head` followed by exactly one `body` element. The `head`, in turn, must contain one or more `to` elements and then a `from` element, followed by an optional `subject` element. The order of the elements must be as specified. Each of those elements may contain text (also know as *parsed character data* or `PCDATA`). The single `ATTLIST` declaration in the DTD specifies that the `body` element *may* specify a value for the `encrypted` attribute, and *must* specify either `ascii` or `mime` for the `encoding` attribute. The `ENTITY` declaration of the `encoding-attribute` (at the top of the DTD) is a simple way to factor out redundant text—the text given between the quotes is substituted as is into the following `ATTLIST` declaration (and, importantly, can be used in multiple `ATTLISTs`).

An XML document that is declared to adhere to this DTD is not valid if any of the above criteria are not met. For example, if the `from` element is missing from an `email`

[1]The applicability of this approach to other languages is discussed further in section 6.

document, that document is not valid, though it may still be well-formed.

When modeling data in XML, a primary design decision is choosing whether to nest elements or to use attributes. In the above example, we could have folded all of the information contained in the `head` into attributes of the `email` element if we chose. There are several important differences between using attributes and nesting elements:

- attributes/value pairs are unordered, while nested children have a specific order;
- values for attributes may contain only character data, and may not include other markup, while nested children can arbitrarily nest further; and
- only one value for an attribute can be given, while multiple elements of the same class can be included by a parent element (e.g., we can have multiple `to` elements contained by the `head`).

Although the above distinctions sometimes mandate using one technique or the other, the decision is often initially a matter of taste. However, later experiences using the resulting documents may suggest revisiting the decision in order to facilitate or simplify some desired manipulation of the document.

Another useful data modeling feature of XML is the ability to attach unique identifiers to elements via an `id` attribute. These elements can then be referred to by `idref` attributes of other elements. A well-formed XML document must have every `idref` value match an `id` given in the document. The `id`–`idref` links describe edges that enable XML to represent generalized directed graphs, not just trees.

XML, in part due to its SGML heritage, is very well supported by tools such as Emacs editing modes, structure-based editors, DTD parsers and editors, validation utilities, querying systems, transformation and style languages, and many more tools. Numerous other W3C recommendations relate to XML including Cascading Style Sheets [8], XSL (Extensible Stylesheet Language) [19], XSLT (XSL for Transformations) [14], XPath [16], and DOM (Document Object Model) [2].

3 Java Markup Language (JavaML)

The Java Markup Language provides a complete self-describing representation of Java source code. Unlike the conventional character-based representation of programs, JavaML reflects the structure of the software artifact directly in the nesting of elements in the XML-based syntax. Additionally, it represents extra edges in the program graph using the `id` and `idref` linking capabilities of XML.

Because XML is a text-based representation, many of the advantages of the classical source representation remain. Because JavaML is an XML application, it is easy to parse, and all existing tools for working with XML can be applied to Java source code in its JavaML representation. JavaML tools can leverage the existing infrastructure and exploit the canonical representation to improve their inter-operability.

3.1 Possible approaches

Although the basic approach of using an XML application to model source code is fairly straightforward, there is a large design space for possible markup languages. The most obvious possibility is to simply use XML as a textual dump format of a typical abstract syntax tree derived from parsing source code. Consider the simple Java program:

```
import java.applet.*;
import java.awt.*;

public class FirstApplet
                   extends Applet {
  public void paint(Graphics g) {
    g.drawString("FirstApplet", 25, 50);
  }
}
```

Performing the obvious (but very unsatisfying) translation from the AST of the above might result in the below XML *for just the first line of code*:

```
<compilation-unit>
 <ImportDeclarationsopt>
  <ImportDeclarations>
   <ImportDeclaration>
    <TypeImportOnDemandDeclaration>
      import
      <Name>
       <QualifiedName>
        <Name>
         <SimpleName>java</SimpleName>
        </Name>
        .
        <Name>
         <SimpleName>applet</SimpleName>
        </Name>
       <QualifiedName>
      </Name>
       . * ;
    </TypeImportOnDemandDeclaration>
   </ImportDeclaration>
  </ImportDeclarations>
 </ImportDeclarationsopt>
...
</compilation-unit>
```

Certainly this translation is far from ideal: it is unacceptably verbose and exposes numerous uninteresting details of the underlying grammar that was used to parse the classical source representation.

An alternate possibility is to literally mark-up the Java source program without changing the text of the program (i.e., to only add tags). This approach might convert the `FirstApplet.java` implementation to:

```
<java-source-program>
<import-declaration>import java.applet.*;
   </import-declaration>
<import-declaration>import java.awt.*;
   </import-declaration>

<class-declaration>
<modifiers>public</modifiers> class
   <class-name>FirstApplet</class-name>
     extends
      <superclass>Applet</superclass> {
 <method-definition>
  <modifiers>public</modifiers>
        <return-type>void</return-type>
    <method-name>paint</method-name>
      (<formal-arguments>
          <type>Graphics</type>
          <name>g</name>
        </formal-arguments>)
    <statements>{
    g.drawString("FirstApplet", 25, 50);
  } </statements>
 </method-definition>
}
</class-declaration>
</java-source-program>
```

This format is a huge step towards a more useful markup language. We have definitely added value to the source code and it is trivial to convert back to the classical representation: we simply remove all tags and leave the content of the elements behind (this removal of markup is exactly what the `stripsgml` [31] utility does). Although this representation seems useful for many tasks, it still has some problems. First, many of the details of the code are included in the textual content of elements. If we want to determine what packages are being imported, our XML query would need to lexically analyze the content of the import-declaration elements. Such analysis is inconvenient and does not take advantage of the capabilities that XML provides. Perhaps more significantly, the above XML representation retains artifacts from the classical source code that another representation might permit us to abstract away from and free ourselves of those syntactic burdens altogether.

3.2 The chosen representation

The prototype JavaML representation I have chosen aims to model the programming language constructs of Java (and, indeed, similar object-oriented programming languages) independently of the specific syntax of the language. One can easily imagine a SmalltalkML that would be very similar, and even an OOML that could be converted into both classical Java source code or Smalltalk file-out format. With this goal in mind, JavaML was designed from first principles of the constructs and then iteratively refined to improve the usefulness and readability of the resulting markup language.

JavaML is defined by the document type definition (DTD) in appendix A, but is best illustrated by example. For the `FirstApplet.java` source code listed above, we represent the program in JavaML as shown in figure 1.

In JavaML, concepts such as methods, superclasses, message sends, and literal numbers are all directly represented in the elements and attributes of the document contents. The representation reflects the structure of the programming language in the nesting of the elements. For example, the literal string `"FirstApplet"` is a part of the message send, thus the `literal-string` element is nested inside the `send` element. This nesting is even more apparent when presented visually as in figure 2. See the author's JavaML web page [4] for further examples.

The careful reader will observe that the JavaML representation is about three times longer than the classical source code. That expansion is a fundamental tradeoff of moving to a self-describing data format such as XML. It is important that the terse classical representation can be employed by programmers in certain tasks including ordinary development and program editing (though perhaps JavaML may be the underlying representation). JavaML is complementary to classical source code and is especially appropriate for tools while remaining accessible to and directly readable by developers.

3.3 Design decisions

JavaML provides more than just the structure of the source program. In figure 1, notice the use of the formal-argument `g` in line 17 as the target of the message send. The `idref` attribute of that `var-ref` tag points back at the referenced `formal-argument` element (through its `id` attribute). (The `id` value chosen for a to-be-referenced element must be unique within a document so each identifier is branded with an integer to keep the values distinct.) This linking is standard XML, thus XML tools are able to trace from a variable use to its definition to, e.g., obtain the type of the variable. Similar linking is done for local (block-declared) variables, and more could be done for other edges in the program structure graph. Although a single `var-use` tag would suffice for denoting any mention of a variable, JavaML instead disam-

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE java-source-program SYSTEM "java-ml.dtd">

<java-source-program name="FirstApplet.java">
  <import module="java.applet.*"/>
  <import module="java.awt.*"/>
  <class name="FirstApplet" visibility="public">
    <superclass class="Applet"/>
    <method name="paint" visibility="public" id="meth-15">
      <type name="void" primitive="true"/>
      <formal-arguments>
         <formal-argument name="g" id="frmarg-13">
             <type name="Graphics"/></formal-argument>
      </formal-arguments>
      <block>
        <send message="drawString">
          <target><var-ref name="g" idref="frmarg-13"/></target>
            <arguments>
              <literal-string value="FirstApplet"/>
              <literal-number kind="integer" value="25"/>
              <literal-number kind="integer" value="50"/>
            </arguments>
        </send>
      </block>
    </method>
  </class>
</java-source-program>
```

Figure 1: FirstApplet.java converted to JavaML.

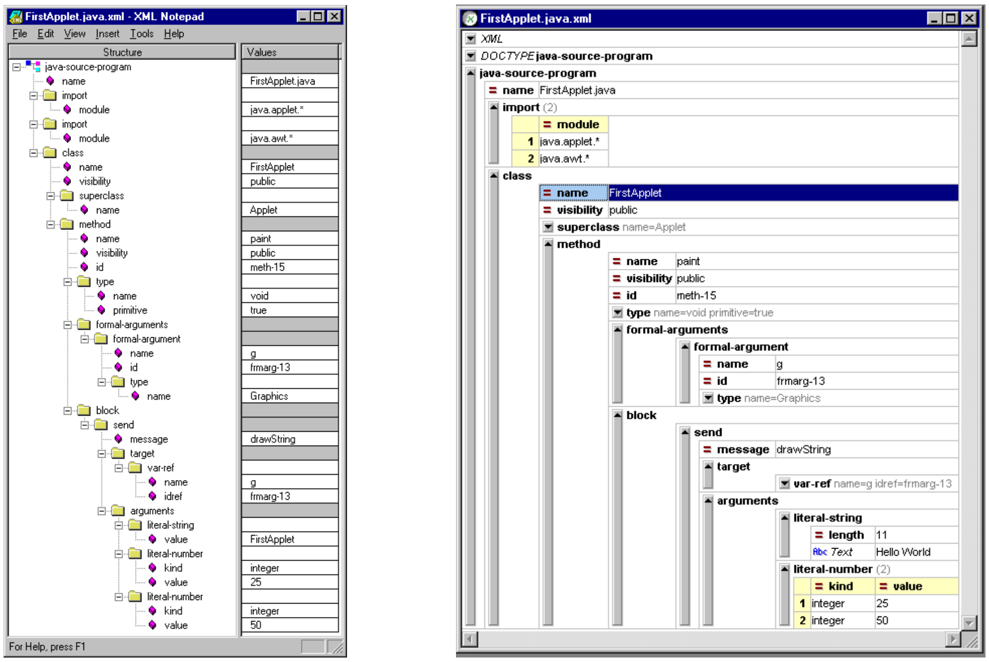

Figure 2: Tree views of the JavaML representation of the `FirstApplet` example as displayed by the XML Notepad utility [44] (on the left) and XML Spy [36] (on the right).

biguates between references to variable values and variables used as lvalues: `var-ref` elements are used for the former, `var-set` for the latter.

Throughout JavaML, attributes of elements are used whenever the structure of the value can never be more complex than a simple text string. Attributes are used for modifiers such as `synchronized` and `final` and for visibility settings such as `public` or `private`. Attributes are not used for properties such as types because types have some structure: a type can consist of a base name and a number of dimensions, and it could also reference the definition of the class that implements the type, if desired. If, say, a return type were just the value of an attribute on the method element, the end user would unacceptably have to do string processing on the attribute's value "`int[][]`" to determine that the base type of that two-dimensional array was the primitive type `int`. Instead, types are modeled as explicit child elements such as `<type name="int" dimensions="2">`.

JavaML generalizes related concepts to simplify some analyses but also preserves distinctions that may be needed for other tasks. For example, `45` and `1.9` are represented as: `<literal-number kind="integer" value="45">` and `<literal-number kind="float" value="1.9">`, respectively. An alternate possible markup is: `<literal-integer value="45">` and `<literal-float value="1.9">` but using separate element classes eliminates the tight relationship that both values are numbers and can complicate using the representation. Instead, we use a single element tag and disambiguate these literals based on a `kind` attribute. Thus, we can still tell the difference between a floating point literal and an integer literal, but in the common case we gain the same flexibility of numeric types that the Java language has.

Another place where JavaML generalizes language constructs is loops. Both `for` and `while` loops can be viewed as general looping constructs with 0 or more initializers, a guarding test that occurs before each iteration, 0 or more update operations, and a body of statements that comprise the looped-over instructions. Thus, instead of using two classes of elements, `for-loop` and `while-loop`, JavaML uses a single `loop` element that has a `kind` attribute with value either `for` or `while`. When a `while` loop is converted, it will have neither `initializer` nor `update` children, yet a `for` loop could potentially contain many of each. In contrast, distinct `do-loop` elements are used for `do` loops because they have their test performed at the end of the loop, instead of at the start.

As yet another example, we represent both instance and class (i.e., static) fields as `field` elements with a `static` attribute used to disambiguate. Although there are more substantial differences between these two concepts than between `while` and `for` loops, it still seems beneficial to use a single kind of element for both kinds of fields.

Local variable declarations provide a syntactic shorthand that raises an interesting question about their underlying representation. The code segment `int dx, dy;` defines two variables both of type `int`, but with perhaps a subtle additional intention: that the two variables have the same type. For contrast, consider `int weight, i;`. Here, there probably is *not* the implicit desire that the two variables have the same type, but instead the shorthand syntax is being used simply for brevity. Because it is hard to automate distinguishing these cases, JavaML simply preserves this syntactic feature by using a `continued="true"` attribute on variable declarations that exploit this shorthand.

Comments in source code are especially troublesome to deal with in JavaML. At present, the DTD permits certain "important" elements (including `class`, `anonymous-class`, `interface`, `method`, `field`, `block`, `loop`) to specify a `comment` attribute. Determining which comments to attach to which elements is challenging; the current implementation simply queues up comments and includes all that appear since the last "important" element in the `comment` attribute of the current such element.

An alternate possibility for comments is to just insert them in the JavaML representation as parsed character data interspersed with the normal structure, thus leaving the semantic inference problem to another tool. Unfortunately, this would force various elements to have "mixed content" which reduces the validation capabilities when checking for DTD conformance. Using XML Schema [51] instead of DTDs may make this approach more useful.

3.4 Implementation of converters

To experiment with the design of JavaML and gain experience in using the representation, it was essential to implement a converter from the Java classical source representation to JavaML. Within the IBM Jikes Java compiler framework [32], I added an `XMLUnparse` method to each of the AST nodes. This change, along with some small additional code for managing the options to request the XML output, results in a robust and fast JavaML converter. In total, I added about 1650 non-comment-non-blank lines of C++ code to the Jikes framework to support JavaML.

The converter has been tested by converting 15,000 lines of numerous sample programs including the 4300 line Cassowary Constraint Solving Toolkit [5] and over twenty diverse applets [50]. Each of the files converted was then validated with respect to the JavaML DTD using James Clark's Jade package's `nsgmls` tool [12]. The processing of the entire regression test takes only about twelve seconds on the author's RedHat6-based dual Pentium III-450 machine.

Also implemented is an XSLT stylesheet that outputs the classical source representation given the JavaML representation. The style sheet consists of 65 template rules and just under 600 lines of code. It was tested (using both Saxon [39] and XT [15]) on numerous programs by processing a file to JavaML, back-converting it, and then re-converting to JavaML: no differences should exist between the result and the originally-converted JavaML file.

All of the source code is available from the JavaML home page [4].

4 Leveraging XML

JavaML uses XML as an alternate, structured representation of a Java source program. Although the abstraction away from syntactic details of Java is convenient, the more important benefit is that JavaML enables the use of the rich infrastructure developed to support SGML and XML. Instead of building analysis and transformation tools from scratch to work on a proprietary binary structured format for a program, existing SGML and XML tools can be used, combined, and extended. XML tools encompass a broad range of features that include querying and transformation, document differencing and merging [33], and simple APIs for working with the document directly. In this paper, I will (for space reasons) limit discussion to uses of only three tool groups:

- the XML toolbox (ltxml) from Edinburgh University [52] which contains `sgcount`, `sgrpg`, `sggrep`, and more;
- XSLT [14] processors (e.g., XT [15] and Saxon [39]) and the XML parser XP [13];
- the Perl XML::DOM package [20] which exposes a DOM level 1 [2] interface to an XML tree.

These are just a very small subset of the tools that prove useful when working with JavaML. In the following examples, we will query `Hangman.java.xml`, the JavaML representation of the Hangman applet available at Sun Microsystems' applet page [50] and also at the JavaML home page [4]. Although these examples are small by real-world standards, XML and SGML tools target documents ranging up through lengthy books so the implementations are designed to scale well.

One common software engineering task (for better or for worse) is to accumulate metrics about a source code artifact. With JavaML, the SGML utility `sgcount` does an excellent job of summarizing the constructs in a Java program:[2]

```
% sgcount Hangman.java.xml
```

outputs:

```
arguments              103
array-initializer      4
assignment-expr        60
catch                  3
class                  1
if                     27
true-case              27
false-case             7
field                  28
field-access           18
import                 5
java-source-program    1
literal-char           5
literal-boolean        5
literal-null           5
literal-number         127
literal-string         61
local-variable         23
loop                   13
method                 18
new                    4
new-array              5
return                 5
send                   99
type                   96
var-ref                262
var-set                52
...
```

In the above output, each row lists an element class and the number of times that that element appeared in the document. Thus, we can easily see that there are 18 `method` elements, thus there are 18 method definitions. Similarly, we can see that there is 1 class definition, 262 variable references, 99 message sends, and 61 string literals. This summary is far more indicative of the content of a program than a typical lexical measure such as the number of lines of code.

Suppose we wish to see all the string literals that a program contains. We can do this trivially using `sggrep` on the JavaML representation of the program:

```
% sggrep '.*/literal-string' \
            < Hangman.java.xml
```

outputs:

```
<literal-string value='audio/dance.au'/>
<literal-string value='img/dancing-duke/T'/>
<literal-string value='.gif'/>
<literal-string value='img/hanging-duke/h'/>
<literal-string value='.gif'/>
<literal-string value='Courier'/>
<literal-string value='Courier'/>
...
```

Notice that the output of `sggrep` is also a (not necessarily valid nor even well-formed) XML document. Thus we can string together SGML and XML tools in a Unix pipeline to combine tools in novel and useful ways. For example, it is sometimes worthwhile to convert results back into ordinary Java source representation to aid the human software-engineer. We can do this using `results-to-plain-`

[2] The output of commands has been pruned and slightly edited for presentation.

`source` which is a wrapper around an XSLT stylesheet that converts JavaML back into plain source code:

```
% sggrep '.*/literal-string' \
      < Hangman.java.xml \
      | results-to-plain-source
```

outputs:

```
"audio/dance.au"
"img/dancing-duke/T"
".gif"
"img/hanging-duke/h"
".gif"
"Courier"
"Courier"
...
```

We can also query the JavaML source for elements based on values of their attributes. For example, if we wish to find all sends of the message `setFont` we can do so easily and precisely:

```
% sggrep '.*/send[message=setFont]' \
                        < Hangman.java.xml
```

outputs:

```
<send message='setFont'>
  <target>
   <var-ref name='g' idref='frmarg-212'/>
  </target>
  <arguments>
   <var-ref name='font' idref='locvar-611'/>
  </arguments>
</send>
<send message='setFont'>
  <target>
   <var-ref name='g' idref='frmarg-212'/>
  </target>
  <arguments>
   <var-ref name='wordFont'/>
  </arguments>
</send>
```

Because of the structural markup, places where the seven characters "setFont" appear in a comment, a literal string, or a variable name will *not* be reported by this query. A similar attempt to retrieve this information using lexical tools would likely contain those false positives. Imagine trying to find all type cast expressions using only lexical tools—the over-use of parentheses in Java expressions make that task very difficult, while it is trivial with JavaML thanks to the `cast-expr` element.

Another class of common analyses is the semantic checks done by the compiler prior to translation. For example, in Java code, only abstract classes may have abstract methods. When compiling, a semantic error will be flagged if this rule is violated. We can query a JavaML document for concrete (i.e., not abstract) classes that contain an abstract method:

```
% sggrep -q '.*/class[abstract!=true]/\
                method[abstract=true]' \
                      < Hangman.java.xml
```

and the output will be empty because this semantic restriction is not violated in our target document (i.e., the analyzed program).

A common error for novice Java programmers is to accidentally use the assignment operator, =, instead of using the equality test operator, ==. Although the Java type checker will catch most of these errors at compile time, it will miss the problem if the assigned-to variable is a `boolean`. If we wish to find these questionable constructs, `sggrep` makes this analysis trivial thanks to the JavaML representation:

```
% sggrep -q '.*/if/test/assignment-expr' \
                        < Hangman.java.xml
```

The `sgrpg` (SGML RePort Generator) program permits combining a top-level query with a restriction on the children and an output format for the results (a common paradigm for querying tools [24]). For example:

```
% sgrpg '.*/method' \
        '.*/send[message=drawLine]' \
        '' '%s %s
' visibility name < Hangman.java.xml
```

outputs:

```
public paint
```

searches for method definitions that contain message sends of the message `drawLine`. It then outputs the `visibility` and `name` attributes of the matched elements as shown above, confirming our intuition that the `paint()` method is the only function that invoked `drawLine`.

A wide variety of analyses are possible just using the querying capabilities provided by standard XML tools. Other things we can find are returns from inside for loops, all definitions of integer variables, string variables that do not conform to our project's naming convention, and much more.

The preceding queries illustrate a shortcoming of current XML querying tools: most respond only with the matched elements—they do not provide any context information about where in the document the results were found. Although this behaviour is appropriate when treating an XML file strictly as a database, the software engineer may want to know where the results were in the JavaML file to then map them back into positions in the source document for editing or viewing by hand. I address this difficulty in JavaML by attaching information about the original source-code location of constructs as attributes of various elements.

The location information includes the starting and ending line and column numbers of the construct (the filename is found in the ancestor `java-class-file` element).

In addition to queries, transformations on source code are very useful when modifying and evolving software artifacts. Querying tools generally only prune elements from the source document or combine elements from multiple documents. More powerful transformations are possible using XSLT [14], DSSSL [38], or directly manipulating the document using a DOM (Document Object Model) [2] interface accessible from numerous languages including Perl, Python, Java, and C++. For example, we can rename all methods named `isBall` to `FIsBall` using a straightforward XSLT stylesheet:

```
<xsl:stylesheet ......>

<xsl:param name="oldname"/>
<xsl:param name="newname"/>

<!-- mostly do an identity transform -->
<xsl:template match="*|@*|text()">
 <xsl:copy>
  <xsl:apply-templates
       select="*|@*|text()"/>
 </xsl:copy>
</xsl:template>

<xsl:template
     match="method[@name=$oldname]">
 <method name="{$newname}">
  <xsl:apply-templates/>
 </method>
</xsl:template>

<xsl:template
     match="send[@message=$oldname]">
 <send message="{$newname}">
  <xsl:apply-templates/>
 </send>
</xsl:template>

</xsl:stylesheet>
```

and executing it like so:

```
xt source.java.xml method-rename.xsl \
       oldname=isBall newname=FIsBall
```

While a similar textual transformation could be performed using a text editor or Sed, those tools will over-aggressively change all occurrences of the six character sequence `isBall`. Variable names, literal strings, comments, and packages might incorrectly be affected by the text-based transformation. This is a key benefit of the JavaML representation: we have more fine-grained, semantically-based control over the affected constructs.

Other possibilities for transformations include using a style sheet to output a browse-able HTML representation of the program (see figure 3) or syntax-highlighted PostScript. Adding debug or instrumentation code at entry and exit to and from functions is also straightforward (see figure 4).

5 Related work

A key benefit of JavaML is its ability to leverage the growing infrastructure of SGML and XML related tools and techniques as described in the previous section. Various researchers have similarly approached the problem of improving software engineering and development tools with varying degrees of success.

TAWK [29] extends the AWK [21] paradigm by matching patterns in the AST of a C program. Numerous XML querying tools provide this same functionality for JavaML, and the event-action framework is similar to that used by SAX (Simple API for XML) [43].

ASTLog [18] extends the Prolog [17] logic programming language with the ability to reason about an external database that models the AST. Unlike Prolog, ASTLog statements are evaluated with respect to a current object. The approach that Crew uses may be interesting to apply to the XML world, but the numerous XML tools already provide comparable functionality through a more conventional (if perhaps less convenient) framework.

GRAS [40] is a graph-oriented database system for software engineering environments. Front-ends are available for integrating source-code from ordinary representation of C, Modula-3, and Modula-2 into the database. The database approach may prove useful for storing XML, especially in the context of the software-engineering applications for which JavaML is designed.

The Software Development Foundation [46] is an open architecture based on XML designed for developing tools for programming environments. An XML database format called CSF—code structure format—stores relationships, but includes no details about the computation performed. Chava [41] takes a similar approach, but is based on the C Program Database [11]. Chava also permits interrogating Java code via reverse engineering the byte-codes.

CCEL [22] provides a metalanguage for expressing non-linguistic intentions (i.e., ones that cannot be expressed in the language) about software artifacts written in C++. JavaML can provide a similar capability by simply writing queries that search for violations of the intended invariants and reporting them as part of the edit, build, or regression-test procedure throughout the development cycle.

Microsoft's Intentional Programming group [47] has long been working on a more abstract representation of computation that is syntax-independent. Their goal appears to be to permit developers to describe new abstractions along

Figure 3: A view of Hangman.java.xml processed by an XSLT HTML pretty-printer and indexer. The method index links to the start of the definition of each method, and syntax highlighting is done using color-coding and italics.

```
#!/usr/bin/perl -w
use XML::DOM;
use IO::Handle;

my $filename = shift @ARGV;

my $parser = new XML::DOM::Parser;
my $doc = $parser->parsefile ($filename);

my $nodes = $doc->getElementsByTagName("method");

for (my $i = 0; $i < $nodes->getLength(); $i++) {
  my $method = $nodes->item($i);
  my $block = $method->
    getElementsByTagName("block")->item(0);
  my $name = $method->getAttribute("name");

  my $start_code
    = SendMessageBlock($doc,"Tracer","StartMethod",
                       $name);
  my $exit_code
    = SendMessageBlock($doc,"Tracer","ExitMethod",
                       $name);

  $block->insertBefore($start_code,
                       $block->getFirstChild());
  $block->appendChild($exit_code);
}
print $doc->toString;

sub SendMessageBlock {
  my ($doc,$target_var,$method_name,$data) = (@_);
  # insert, e.g:  Tracer.StartMethod("paint");
 return parseXMLFragment($doc,<<"__END_FRAGMENT__"
<send message="$method_name">
  <target><var-ref name="$target_var"/></target>
  <arguments>
    <literal-string value="$data"/>
  </arguments>
</send>
__END_FRAGMENT__
 );
}

sub parseXMLFragment {
  my ($doc,$code) = (@_);
  my $newdoc = $parser->parse($code);
  my $subtree = $newdoc->getDocumentElement();
  $newdoc->removeChild($subtree);
  $subtree->setOwnerDocument($doc);
  return $subtree;
}
```

Figure 4: Perl program to instrument every method of a Java class with invocations of `Tracer.StartMethod(`*method_name*`)` and `Tracer.ExitMethod(`*method_name*`)`. The program uses the Document Object Model (DOM) [2] Perl module [20].

with techniques to reduce those abstractions down to known primitives. In essence, they are interested in permitting the developer to grow a domain-specific language as they build their software. JavaML is especially exciting as a representation for this approach. We can view new abstractions as incremental extensions to DTDs. In order for the new document type, call it Java++ML, to still be compilable by a stock Java compiler, the developer must simply write a transformation from Java++ML to JavaML. Because DTDs are exceptionally easy to extend, this approach is tenable and likely a fruitful avenue for future work. There are several utilities for documenting and comparing DTDs (e.g., `dtd2html` and `dtddiff` of the perlSGML package [31]) that would be helpful when applying this technique.

6 Future work

Although this paper has presented a markup language for Java, the same basic approach can be applied to other programming languages, or even to translate among languages. To the extent that the representation abstracts away syntax, JavaML may also prove useful in permitting the import of visual representations such as Unified Modeling Language diagrams [1, 35]. Certainly generating visual representations of important properties of software artifacts is on the immediate horizon given the capabilities of XSL and DSSSL.

One significant complication in applying this approach to C++, another popular conventional object-oriented-programming language, is the C preprocessor. The C preprocessor provides a first pass of textual processing to permit abstractions that cannot otherwise be expressed in the core C++ language. These abstractions are often very important to the understandability and maintainability of the code, but do not interact well with parsing techniques [23, 6].

A useful extension to the current transformation system is to do more cross-linking of elements. Type elements could reference their defining classes in other JavaML documents. Import declarations could reference the top-level documentation for the imported package. Many more possibilities are feasible.

The current converter that translates back from JavaML to the classical source representation is based on XSLT. Adding a Jikes front-end would permit the compiler to read JavaML directly. Such an implementation would use an XML parser (e.g., XML4C++ [34]) to construct the XML DOM from the JavaML source, then simply recursively build the Jikes internal AST using the DOM API. Back-conversion to the plain source code could then be done using Jikes's pre-existing conventional unparser.

Using JavaML as the primary source representation has the potential to simplify the compiler beyond just eliminating its classical front-end. Some semantic analyses can be removed from the compiler once it knows that the input is a valid JavaML document. It will be useful to characterize which semantic errors are provably impossible to encounter given that precondition. Because XML Schema [51, 7] provide an even finer-grained specification of validity for XML documents, it is likely beneficial to migrate JavaML to use an them instead of a DTD after the working drafts are finalized. Additionally, more semantic analyses can be moved into the editing environment reasonably painlessly in the form of straightforward queries (such as some of those described earlier in section 4).

Because the concise textual representation of source code is nicely suited to human programmers, it is unlikely that they will be interested in discarding their favourite text editor anytime soon. We must investigate better ways to convert interactively and incrementally between the classical source representation and JavaML. The capabilities could then be used to transparently support interactive editing of XML representations using plain-text format to which human engineers are accustomed. The considerable work on structured text editors [30, 45] is highly relevant and may finally achieve acceptance give the incredible resources that will now be thrown at the problem given the growing commercial importance of XML technology.

7 Conclusion

JavaML is an alternate representation of Java source programs that is based on XML. Unlike the classical textual source representation, the JavaML representation makes it easy for software tools to reason about programming-level constructs in a Java program. This benefit results from the ability of JavaML to more directly represent the structure of the program.

Given JavaML, the wealth of pre-existing XML and SGML tools can perform numerous interesting and useful analyses and transformations of Java source programs. XML tools are improving continually to support the growing infrastructure of XML-based documents. Ultimately, JavaML could replace the classical source representation of Java programs as the storage format for programs, relegating text-parsing to just one of many possible ways of interacting directly with the structured representation of the software artifact throughout the development process.

Acknowledgments

I enthusiastically thank Zack Ives for his comments, discussion, and input. I thank Corin Anderson and Alan Borning for much-appreciated comments on a draft of this paper. I also thank Miguel Figueroa, Karl-Trygve Kalleberg, Craig Kaplan, Todd Millstein, Stig E. Sandø, and Stefan Bjarni Sigurdsson for their helpful discussions. Thanks to IBM for constructing the Jikes compiler framework, and for making it publicly available, and thanks to Mike Ernst for helpful pointers on using it. This work was supported by the University of Washington Computer Science and Engineering Wilma Bradley fellowship and by NSF Grant No. IIS-9975990.

A JavaML DTD

```
<!-- java-ml.dtd 0.96 -->

<!ENTITY % visibility-attribute
    "visibility (public|private|\
     protected) #IMPLIED">
<!ENTITY % interface-visibility-attribute
    "visibility (public) #IMPLIED">
<!ENTITY % kind-attribute
    "kind (integer|long|float|\
     double) #IMPLIED">
<!ENTITY % mod-final
    "final CDATA #IMPLIED">
<!ENTITY % mod-static
    "static CDATA #IMPLIED">
<!ENTITY % mod-volatile
    "volatile CDATA #IMPLIED">
<!ENTITY % mod-transient
    "transient CDATA #IMPLIED">
<!ENTITY % mod-native
    "native CDATA #IMPLIED">
<!ENTITY % mod-abstract
    "abstract CDATA #IMPLIED">
<!ENTITY % mod-synchronized
    "synchronized CDATA #IMPLIED">
<!ENTITY % location-info
"line CDATA #IMPLIED col CDATA #IMPLIED
         end-line CDATA #IMPLIED end-col CD
     ATA #IMPLIED
 comment CDATA #IMPLIED">
<!ENTITY % expr-elems
    "send|new|new-array|var-ref|\
     field-access|array-ref|paren|\
     assignment-expr|conditional-expr|\
     binary-expr|unary-expr|cast-expr|\
     instanceof-test|literal-number|\
     literal-string|literal-char|\
     literal-boolean|literal-null|this|\
     super">
<!ENTITY % stmt-elems
    "block|local-variable|try|throw|if|\
     switch|loop|do-loop|return|continue|\
     break|synchronized|%expr-elems;">

<!ELEMENT code-fragment ANY>
<!ELEMENT result ANY>
<!ELEMENT java-source-program
    (java-class-file+)>
<!ELEMENT java-class-file
    (package-decl?,import*,
     (class|interface)+)>
<!ATTLIST java-class-file
    name CDATA #IMPLIED
    version CDATA #IMPLIED>
<!ELEMENT import EMPTY>
<!ATTLIST import
    module CDATA #REQUIRED>
<!ELEMENT class
    (superclass?, implement*,
      (class|interface|constructor|method|\
     field|static-initializer|\
     instance-initializer)*)>
<!ATTLIST class
    name CDATA #REQUIRED
    %visibility-attribute;
    %mod-static;
    %mod-abstract;
    %mod-final;
    %mod-synchronized;
    %location-info;>
<!ELEMENT anonymous-class
    (superclass?, implement*,
      (constructor|method|field|\
     instance-initializer)*)>
<!ATTLIST anonymous-class
    %mod-abstract;
    %mod-final;
    %mod-synchronized;
    %location-info;>
<!ELEMENT superclass EMPTY>
<!ATTLIST superclass
    name CDATA #REQUIRED>
<!ELEMENT interface
    (extend*, (method|field)*)>
<!ATTLIST interface
    name CDATA #REQUIRED
    %interface-visibility-attribute;
    %location-info;>
<!ELEMENT implement EMPTY>
<!ATTLIST implement
    interface CDATA #REQUIRED>
<!ELEMENT extend EMPTY>
<!ATTLIST extend
    interface CDATA #REQUIRED>
<!ELEMENT field
    (type,
     (array-initializer|%expr-elems;)?)>
<!ATTLIST field
    name CDATA #REQUIRED
    %visibility-attribute;
    %mod-final;
    %mod-static;
    %mod-volatile;
    %mod-transient;
    %location-info;>
<!ELEMENT constructor
    (formal-arguments,throws*,
     (super-call|this-call)?,
     (%stmt-elems;)?)>
```

```
<!ATTLIST constructor
    name CDATA #REQUIRED
    id ID #REQUIRED
    %visibility-attribute;
    %mod-final;
    %mod-static;
    %mod-synchronized;
    %mod-volatile;
    %mod-transient;
    %mod-native;
    %location-info;>
<!ELEMENT method
    (type,formal-arguments,throws*,
     (%stmt-elems;)?)>
<!ATTLIST method
    name CDATA #REQUIRED
    id ID #REQUIRED
    %visibility-attribute;
    %mod-abstract;
    %mod-final;
    %mod-static;
    %mod-synchronized;
    %mod-volatile;
    %mod-transient;
    %mod-native;
    %location-info;>
<!ELEMENT formal-arguments
    (formal-argument)*>
<!ELEMENT formal-argument (type)>
<!ATTLIST formal-argument
    name CDATA #REQUIRED
    id ID #REQUIRED
    %mod-final;>
<!ELEMENT send (target?,arguments)>
<!ATTLIST send
    message CDATA #REQUIRED
    idref IDREF #IMPLIED>
<!ELEMENT block (label*,(%stmt-elems;)*)>
<!ATTLIST block
    %location-info;>
<!ELEMENT label EMPTY>
<!ATTLIST label
    name CDATA #REQUIRED>
<!ELEMENT target (%expr-elems;)>
<!ELEMENT return (%expr-elems;)?>
<!ELEMENT throw (%expr-elems;)>
<!ELEMENT throws EMPTY>
<!ATTLIST throws
    exception CDATA #REQUIRED>
<!ELEMENT new
    (type,arguments,anonymous-class?)>
<!ELEMENT type EMPTY>
<!ATTLIST type
    primitive CDATA #IMPLIED
    name CDATA #REQUIRED
    dimensions CDATA #IMPLIED
    idref IDREF #IMPLIED>
<!ELEMENT new-array
    (type,dim-expr*,array-initializer?)>
<!ATTLIST new-array
    dimensions CDATA #REQUIRED>
<!ELEMENT dim-expr (%expr-elems;)>
<!ELEMENT local-variable
    (type,
     (static-initializer|array-initializer|\
     %expr-elems;)?)>
<!ATTLIST local-variable
    name CDATA #REQUIRED
    id ID #REQUIRED
    continued CDATA #IMPLIED
    %mod-final;>
<!ELEMENT array-initializer
    (array-initializer|%expr-elems;)*>
<!ATTLIST array-initializer
    length CDATA #REQUIRED>
<!ELEMENT arguments (%expr-elems;)*>
<!ELEMENT literal-string EMPTY>
<!ATTLIST literal-string
    value CDATA #REQUIRED>
<!ELEMENT literal-char EMPTY>
<!ATTLIST literal-char
    value CDATA #REQUIRED>
<!ELEMENT literal-number EMPTY>
<!ATTLIST literal-number
    value CDATA #REQUIRED
    %kind-attribute;
    base CDATA "10">
<!ELEMENT var-ref EMPTY>
<!ATTLIST var-ref
    name CDATA #REQUIRED
    idref IDREF #IMPLIED>
<!ELEMENT field-access (%expr-elems;)>
<!ATTLIST field-access
    field CDATA #REQUIRED>
<!ELEMENT var-set EMPTY>
<!ATTLIST var-set
    name CDATA #REQUIRED>
<!ELEMENT field-set (%expr-elems;)>
<!ATTLIST field-set
    field CDATA #REQUIRED>
<!ELEMENT package-decl EMPTY>
<!ATTLIST package-decl
    name CDATA #REQUIRED>
<!ELEMENT assignment-expr
    (lvalue,(%expr-elems;))>
<!ATTLIST assignment-expr
    op CDATA #REQUIRED>
<!ELEMENT lvalue
    (var-set|field-set|%expr-elems;)>
<!ELEMENT instanceof-test
    ((%expr-elems;),type)>
<!ELEMENT binary-expr
    ((%expr-elems;),(%expr-elems;))>
<!ATTLIST binary-expr
    op CDATA #REQUIRED>
<!ELEMENT paren (%expr-elems;)>
<!ELEMENT unary-expr (%expr-elems;)>
<!ATTLIST unary-expr
```

```
    op CDATA #REQUIRED
    post (true|false) #IMPLIED>
<!ELEMENT cast-expr (type,(%expr-elems;))>
<!ELEMENT literal-boolean EMPTY>
<!ATTLIST literal-boolean
    value (true|false) #REQUIRED>
<!ELEMENT literal-null EMPTY>
<!ELEMENT synchronized (expr,block)>
<!ELEMENT expr (%expr-elems;)>
<!ELEMENT if (test,true-case,false-case?)>
<!ELEMENT test (%expr-elems;)>
<!ELEMENT true-case (%stmt-elems;)?>
<!ELEMENT false-case (%stmt-elems;)?>
<!ELEMENT array-ref (base,offset)>
<!ELEMENT base (%expr-elems;)>
<!ELEMENT offset (%expr-elems;)>
<!ELEMENT static-initializer
    (%stmt-elems;)*>
<!ELEMENT instance-initializer
    (%stmt-elems;)*>
<!ELEMENT super-call (arguments)>
<!ELEMENT this-call (arguments)>
<!ELEMENT super EMPTY>
<!ELEMENT this EMPTY>
<!ELEMENT loop
    (init*,test?,update*,(%stmt-elems;)?)>
<!ATTLIST loop
    kind (for|while) #IMPLIED
    %location-info;>
<!ELEMENT init
    (local-variable|%expr-elems;)*>
<!ELEMENT update (%expr-elems;)>
<!ELEMENT do-loop ((%stmt-elems;)?,test?)>
<!ELEMENT try
    ((%stmt-elems;),catch*,finally?)>
<!ELEMENT catch
    (formal-argument,(%stmt-elems;)?)>
<!ELEMENT finally (%stmt-elems;)>
<!ELEMENT continue EMPTY>
<!ATTLIST continue
    targetname CDATA #IMPLIED>
<!ELEMENT break EMPTY>
<!ATTLIST break
    targetname CDATA #IMPLIED>
<!ELEMENT conditional-expr
    ((%expr-elems;),(%expr-elems;),
     (%expr-elems;))>
<!ELEMENT switch
    ((%expr-elems;),switch-block+)>
<!ELEMENT switch-block
    ((case|default-case)+,
     (%stmt-elems;)*)>
<!ELEMENT case (%expr-elems;)>
<!ELEMENT default-case EMPTY>
```

References

[1] S. S. Alhir. *UML in a Nutshell*. O'Reilly & Associates, Inc., 1998.

[2] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model (DOM) level 1. W3C Recommendation, October 1998. http://www.w3.org/TR/REC-DOM-Level-1.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1998.

[4] G. J. Badros. JavaML Home Page. http://www.cs.washington.edu/homes/gjb/JavaML.

[5] G. J. Badros and A. Borning. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington, June 1998. http://www.cs.washington.edu/research/constraints/cassowary/cassowary-tr.pdf.

[6] G. J. Badros and D. Notkin. A framework for preprocessor-aware C source code analyses. *Software—Practice and Experience*, 2000. To appear.

[7] P. V. Biron and A. Malhotra. Xml scheme part 2: Datatypes. W3C Working Draft, November 1999. http://www.w3.org/TR/xmlschema-2.

[8] B. Bos, H. W. Lie, C. Lilley, and I. Jacobs. Cascading style sheets, level 2. W3C Working Draft, Jan. 1998. http://www.w3.org/TR/WD-css2/.

[9] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation, February 1998. http://www.w3.org/TR/REC-xml.

[10] British Standards Institution. Modula-2 draft international standard, iso-94, June 1994.

[11] Y.-F. Chen. The c program database and its applications. In *Proceedings of the Summer 1989 USENIX Conference*, pages 157–171, Baltimore, 1989.

[12] J. Clark. James' DSSSL engine (JADE). http://www.jclark.com/jade.

[13] J. Clark. XP version 0.5, 1998. http://www.jclark.com/xml/xp.

[14] J. Clark. XSL transformations. W3C Recommendation, November 1999. http://www.w3.org/TR/xslt.

[15] J. Clark. XT version 19991105, November 1999. http://www.jclark.com/xml/xt.html.

[16] J. Clark and S. DeRose. XML path language (xpath) version 1.0. W3C Recommendation, November 1999. http://www.w3.org/TR/xpath.

[17] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, fourth edition, 1994.

[18] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

[19] S. Deach. Extensible stylesheet language (xsl) specification. W3C Working Draft, January 2000. http://www.w3.org/TR/WD-xsl.

[20] E. Derksen and C. Cooper. Perl XML::DOM module. `http://users.erols.com/enno/dom`.

[21] D. Dougherty. *Sed and Awk*. O'Reilly & Associates, Inc., Sebastopol, California, 1990.

[22] C. K. Duby, S. Meyers, and S. P. Reiss. CCEL: A metalanguage for C++. In *Proceedings of the USENIX 1992 C++ Conference*, Portland, Oregon, August 1992.

[23] M. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 2000. To appear.

[24] M. Fernandez, J. Siméon, and P. Wadler. XML query language: Experiences and exemplars, 1999. `http://www-db.research.bell-labs.com/user/simeon/xquery.html`.

[25] D. Flanagan. *Java in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, California, 2nd edition, 1997.

[26] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.

[27] C. F. Goldfarb and P. Prescod. *The XML Handbook*. Prentice Hall PTR, 1998.

[28] M. Goosens and S. Rahtz. *The LaTeXWeb Companion*. Addison Wesley Longman, 1999.

[29] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the IEEE 1996 Workshop on Program Comprehension*, March 1996.

[30] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, pages pp. 1117–1127, 1986.

[31] E. Hood. perlSGML library. `http://www.oac.uci.edu/indiv/ehood/perlSGML.html`.

[32] IBM. Jikes java compiler. `http://www.alphaworks.ibm.com/tech/Jikes`.

[33] IBM AlphaWorks. XML diff and merge tool. `http://www.alphaworks.ibm.com/tech/xmldiffmerge`.

[34] IBM AlphaWorks. XML for C++. `http://www.alphaworks.ibm.com/tech/xml4c`.

[35] IBM AlphaWorks. XML metadata interchange (XMI) toolkit. `http://www.alphaworks.ibm.com/tech/xmitoolkit`.

[36] Icon I.S. XML spy 3.0beta2. `http://www.xmlspy.com`.

[37] ISO. Standard generalized markup language (SGML). ISO 8879, 1986. `http://www.iso.ch/cate/d16387.html`.

[38] ISO/IEC. Document style semantics and specification language (DSSSL). ISO/IEC 10179, 1996.

[39] M. H. Kay. *SAXON*. `http://users.iclway.co.uk/mhkay/saxon/`.

[40] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Information Systems*, 20(1):21–52, 1995. Oxford: Pergamon Press.

[41] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 314–325, October 1999.

[42] J. R. Levine. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, California, 2nd edition, 1992.

[43] Megginson Technologies. SAX 1.0: The simple API for XML. Web document, 1999. `http://www.megginson.com/SAX`.

[44] Microsoft. XML Notepad Beta 1.5. `http://msdn.microsoft.com/xml/notepad`.

[45] R. C. Miller and B. A. Myers. Lightweight structured text processing. In *Proceedings of USENIX 1999*, Monterey, CA, 1999.

[46] S. E. Sandøand K.-T. Kalleberg. Software development foundation, February 2000. `http://sds.yi.org`.

[47] C. Simonyi. Intentional programming - innovation in the legacy age. International Federation for Information Processing WG 2.1 meeting, June 1996.

[48] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension mechanisms in montana. In *Proceedings of 8th Israeli Conference on Computer-Based Systems and Software Engineering*, June 1997.

[49] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.

[50] Sun Microsystems. Applet resources, February 2000. `http://java.sun.com/applets/`.

[51] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. Xml scheme part 1: Structures. W3C Working Draft, November 1999. `http://www.w3.org/TR/xmlschema-1`.

[52] University of Edinburgh Language Technology Group. LT XML vesion 1.1. `http://www.ltg.ed.ac.uk/software/xml`.

Vitae

Greg J. Badros is a final-year Ph.D. candidate at the University of Washington in Seattle, USA where he earned his M.Sc. degree in 1998. He graduated *magna cum laude* with a B.S. degree in Mathematics and Computer Science from Duke University in 1995. He is the primary author of the Scheme Constraints Window Manager and the Cassowary Constraint Solving Toolkit. His research interests include constraint technology, software engineering, languages, and the Internet.