JavaML 2.0: Enriching the Markup Language for Java Source Code

Ademar Aguiar[1], Gabriel David[1], and Greg Badros[2]

[1] Faculdade de Engenharia da Universidade do Porto and INESC Porto
{aaguiar,gtd}@fe.up.pt
[2] Google Inc., 2400 Bayshore Parkway, Mountain View, CA 94043
badros@cs.washington.edu

Abstract. Although the representation of source code in plain text format is convenient for manipulation by programmers, it is not an effective format for processing by software engineering tools at an abstraction level suitable for source code analysis, reverse-engineering, or refactoring. Textual source code files require language-specific parsing to uncover program structure, a task undertaken by all compilers but by only a few software engineering tools. JavaML is an alternative and complementary XML representation of Java source code that adds structural and semantic information into source code, and is easy to manipulate, query, and transform using general purpose XML tools and techniques. This paper presents an evolved version of JavaML, dubbed JavaML 2.0, and the enhancements made to the schema and respective converters: DTD and XML Schema support, cross-linking of all program symbols, and full preservation of original formatting and comments. The application of JavaML 2.0 is illustrated with concrete examples taken from the software documentation tool that motivated the enhancements.

1 Introduction

Since the first computer programming languages, programmers have used a plain-text format for encoding software structure and computation. The immediate users of this format include the compiler, which converts sequences of characters into a data structure that closely reflects the program structure — an *abstract syntax tree* (AST).

Despite the advances in compilers and document management tools, software engineers manipulate source code using the plain text format, and often employ superficial tools based on regular expressions.

Although the plain-text representation of source code is convenient for programmers and has nice properties, pure text is not the most effective format for manipulating source code at an abstraction level suitable for software engineering tools. Plain-text files are good for lexical analysis, but they are not suitable for structural and semantic analysis before being parsed and translated to higher level formats. There is thus a need for a standard format capable of directly representing program structure and semantics, one readable and widely supported format that tools can easily analyze and manipulate.

JavaML is a markup language for Java source code proposed by Greg Badros [1,2] that provides an alternative representation for Java source code. JavaML enriches source code text files with structural and semantic information typically present in a compiler annotated ASTs. Because the representation is XML-based, it results easy to manipulate, query, and transform using general purpose and widely available tools and techniques.

This paper presents an evolved version of JavaML, dubbed 2.0, that incorporates enhancements and new features to the original JavaML [1]: XML Schema support, updated converters, cross-referencing of all program symbols, and full preservation of original lexical information (formatting and comments). These enhancements were motivated by the implementation of XSDoc, an extensible infrastructure for documenting object-oriented frameworks [3].

The next section briefly discusses the benefits of using an XML format for representing source code. The sections that follow describe JavaML 2.0 and present concrete examples of application taken from the XSDoc infrastructure. The paper then reviews related work and concludes pointing ideas for future work.

2 Why represent source code in XML?

The plain-text representation of source code is convenient to express programmers ideas. It is concise, easy to read by programmers, and very simple to exchange and manipulate using a wide variety of tools, such as text editors, version control systems, and file system utilities. But the plain-text representation of source code also has many limitations. Perhaps the most significant is that the program structure is not directly represented in the format and thus require specific language processing to uncover it.

2.1 XML

XML is an universal format widely used to represent structured information in text-based files that was designed to be lightweight and simple. An XML document consists of text marked up with tags enclosed in angle braces.

XML documents have an inherent hierarchical structure and are therefore very convenient for representing source code constructs. XML-based representations have many qualities: easy to understand, easy to manipulate by tools, very flexible, extensible, and widely supported. Although XML documents are primarily intended for automatic processing by tools, the format is human readable and understandable.

XML documents are therefore an empowering complementary representation for source code, more abstract, which enables the usage of tools at an higher level of abstraction.

2.2 Limitations of plain text for representing source code

Despite its nice properties, plain text source code files require parsing to uncover the most important information they contain about a program: the structural and semantic information. This severe limitation forces the inclusion of language-specific parsers in each tool that needs to manipulate programs at an abstraction level higher than the lexical. While sophisticated software engineering tools can afford to embed such parsers, unfortunately, the majority of utility tools cannot and are thus limited to simple tasks.

Common manipulations of source code, such as generation, refactoring, reformatting, or reverse-engineering, usually require the manipulation of more abstract source code representations equivalent to ASTs.

Although modern integrated development environments (IDE) provide application programming interfaces (API) to manipulate in-memory AST representations of source code, these APIs are still not appropriate for simple tools, since they require integration in a complex environment — the IDE — that creates a strong and undesirable dependency.

2.3 Benefits of representing source code in XML

The representation of source code in XML has several benefits [4,5] and enables the use of more powerful software engineering and document management methods and tools.

Explicit code structure. XML documents are structured by nature, and can be used to build tree-like code representations and sophisticated manipulation of source code using general purpose XML tools. Some good examples are automatic synchronization of generated code and documentation, or quick code generation and transformation using predefined templates. With source code in XML it is possible to annotate generated code with its template definition so that the constructed code can be regenerated every time its template is updated. Complex templates for custom instantiation of design patterns would benefit from such mechanisms.

Powerful querying capabilities. In addition to textual searches using regular expressions, modern IDEs usually include also tools specific for source code that allow searching for common programming language constructs, such as classes, methods, fields, etc. These features are useful but are only a small subset of what is possible to query with XML standards and tools, such as XPath [6] and XQuery [7].

Extensible representation. Plain-text source code is not easy to extend with new code constructs or annotations because they would disrupt the code structure and require parser modifications. Because of this, such extensions are often embedded as comments. In a XML document, the addition of new elements is much easier, because the structure is explicitly marked up, and we can separate different kinds of elements (e.g. language specific elements and user-defined elements) using XML namespaces. Distinct tools can define and insert their own elements in the code structure and then process only those that are relevant for them. Examples of common extensions are code annotations, comments, meta-information, authoring and version information, revisions, documentation and conditional code.

Flexible formatting. Most programming languages, including Java, ignore the semantic value of whitespace and enable programmers to adopt diversified formatting conventions, which, despite its usefulness, are not easy to enforce and maintain in consistency. Mistakes in following formatting conventions sometimes result in an increased difficulty to locate structural or semantic errors, due to the suggestive nature of formatting. In an XML representation of code, the structure can be abstracted from the coding style. XML standards and tools, such as XSLT [8] facilitate the (re)formatting of source code using different styles which can enrich its readability through an appropriate usage of layouts, colors, fonts and links.

Cross-referencing. Source code fragments in plain text are usually referenced by their file position, i.e. line and column numbers. On an XML tree-like structure of a program, it is possible to reference code fragments directly to code constructs, thus enabling the relocation of code fragments without disrupting references.

Wide support. A program representation must be widely supported in a wide variety of platforms, otherwise it can't succeed. XML tools are available in all major platforms and thus completely satisfy this requirement.

3 JavaML 2.0

The JavaML markup language provides a complete self-describing representation in XML for Java source code. Unlike the classical plain-text representation of programs, JavaML reflects the structure of Java programs directly in the hierarchical structure of XML documents.

Because JavaML uses the XML format, which is a text-based representation, many of the advantages of the classical source representation remain. In addition, the JavaML representation is easy to parse and manipulate with general purpose XML tools, not requiring language-specific tools difficult to implement and maintain. Therefore, JavaML leverages the development of XML tools for the manipulation of Java source code in JavaML.

Although the immediate users of JavaML format are tools, not being intended to be written directly by hand, the format is easily readable and understandable, enabling its direct inspection by developers.

JavaML 2.0 enriches the original JavaML [1] with more information at several levels of abstracting ranging from the lexical level to the semantic level. The enhanced representation of JavaML 2.0 now includes full lexical information about tokens, comments and formatting, only small enhancements in terms of structural information, and much richer semantic information related with symbol definitions, references and type information.

3.1 Background on JavaML

In order to represent Java source code in XML there are many possible approaches [1]. Consider the following source file for the class `FirstTest`.

```
package junit.samples;
import junit.framework.*;

/**
 * My first unit test.
 */
public class FirstTest extends TestCase {
   double value = 2.0;

   public void testAdd() {
      double result = value + 3;
      // forced failure result == 5
      assertTrue(result == 6);
   }
}
```

The most obvious approach is to dump the derived AST to a XML format, but this would result very verbose and uninteresting due to the irrelevant grammar details it would reveal.

Another possibility is to markup the Java source program without changing the original text. The result would be a richer representation, easier to convert back to the original source file, but the retrieval of specific information from the representation would require undesired lexical analysis of element contents.

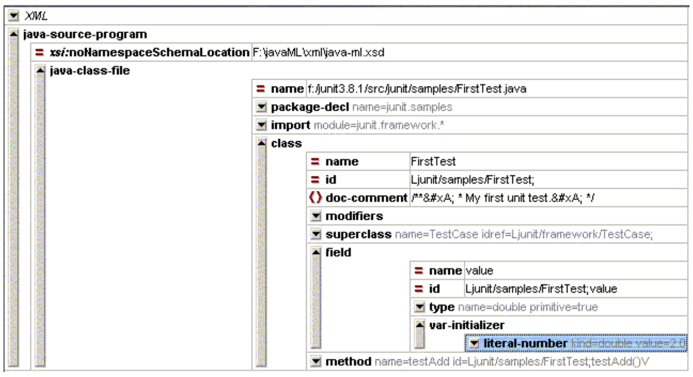
The representation chosen for JavaML aimed to model Java programming language constructs without binding to the specificities of the language syntax [1]. As a result of this design principle, JavaML can be used as a base for the design of a generalized markup language supporting other object-oriented programming languages, such as C#, C++ or Smalltalk.

To illustrate the approach followed by JavaML, consider the source file `FirstTest.java` presented above and its corresponding basic JavaML representation (`FirstTest.java.xml`). The major design decisions are enumerated below.

```
<java-class-file name="f:/junit3.8.1/src/junit/samples/FirstTest.java">
 <package-decl name="junit.samples"/>
 <import module="junit.framework.*"/>
 <class name="FirstTest" id="Ljunit/samples/FirstTest;">
  <doc-comment>/**&#xA; * My first unit test.&#xA; */</doc-comment>
  <modifiers>
   <modifier name="public"/>
  </modifiers>
  <superclass name="TestCase" idref="Ljunit/framework/TestCase;"/>
  <field name="value" id="Ljunit/samples/FirstTest;value">
   <type name="double" primitive="true"/>
   <var-initializer>
    <literal-number kind="double" value="2.0"/>
   </var-initializer>
  </field>
  <method name="testAdd" id="Ljunit/samples/FirstTest;testAdd()V">
   <modifiers>
    <modifier name="public"/>
   </modifiers>
   <type name="void" primitive="true"/>
   <formal-arguments/>
   <block  >
    <local-variable-decl>
     <type name="double" primitive="true"/>
     <local-variable name="result" id="Ljunit/samples/FirstTest;var1946">
      <var-initializer>
       <binary-expr op="+">
        <field-ref name="value" idref="Ljunit/samples/FirstTest;value"/>
        <literal-number kind="integer" value="3"/>
       </binary-expr>
      </var-initializer>
     </local-variable>
    </local-variable-decl>
    <send message="assertTrue" idref="Ljunit/framework/Assert;assertTrue(Z)V">
     <arguments>
      <binary-expr op="==">
       <var-ref name="result" idref="Ljunit/samples/FirstTest;var1946"/>
       <literal-number kind="integer" value="6"/>
      </binary-expr>
     </arguments>
    </send>
   </block>
  </method>
 </class>
</java-class-file>
```

Representation of code constructs. JavaML represents the important concepts of the Java language, namely classes, superclasses, fields, methods, variables, message sends, and literals, directly in document elements and attributes.

Code structure is reflected in the nesting of elements. Program structure is reflected in the nesting of elements. As an example, it is presented below a visual presentation of the document tree, which shows the nesting of the literal number `3` in the initializer part of the variable declaration it appears.



```
XML
java-source-program
    = xsi:noNamespaceSchemaLocation F:\javaML\xml\java-ml.xsd
    java-class-file
        = name f:/junit3.8.1/src/junit/samples/FirstTest.java
        package-decl name=junit.samples
        import module=junit.framework.*
        class
            = name          FirstTest
            = id            Ljunit/samples/FirstTest;
            () doc-comment  /**&#xA; * My first unit test.&#xA; */
            modifiers
            superclass name=TestCase idref=Ljunit/framework/TestCase;
            field
                = name value
                = id   Ljunit/samples/FirstTest;value
                type name=double primitive=true
                var-initializer
                    literal-number kind=double value=2.0
            method name=testAdd id=Ljunit/samples/FirstTest;testAdd()V
```

Fig. 1. Tree view of the JavaML representation[**?**].

Generic elements. In order to reduce the size and complexity of the schema, JavaML generalizes related concepts and represents them using generic elements with different attribute values. Loops and literal numbers are just two examples of such generalizations. In `FirstTest.java.xml`, the lines 15 and 31 contain `literal-number` elements with different `kind` attribute values to disambiguate the representation of a `double` and an `integer`. In addition, unstructured information is stored in attribute elements as illustrated in the `modifier` element, in line 20.

3.2 The new JavaML 2.0 schema

The major enhancements of JavaML 2.0 consisted on improving the schema to accommodate richer lexical and semantic information. The goal is to enable full preservation of original source files and complete cross-linking of program symbols. The corresponding converters were re-implemented to cope with these new schema requirements.

These enhancements were mainly motivated by the implementation of XSDoc, an infrastructure for framework documentation that uses JavaML to dynamically integrate source code in the documentation.

All JavaML and JavaML 2.0 artifacts referred in this paper are available online in [2,9].

DTD and XML Schema support. Both XML Schema and DTD provide a basic grammar for defining XML documents in terms of the metadata that comprise the shape of the document. XML Schemas are themselves XML documents. XML Schemas provide a more powerful means to define a XML document structure and validations than DTDs, because provide an object-oriented approach, with all the benefits this entails, namely the ability to reuse and extend type definitions. Because both DTD and XML Schema have their own advantages, JavaML 2.0 is primarily designed for XML Schema but still support DTDs. The JavaML 2.0 XML Schema has around 90 elements. The original DTD was four times shorter in terms of number of lines.

Cross-referencing of program symbols. JavaML representation includes information to link symbol references to their definitions. This linking is achieved through the standard XML mechanism using `id` attributes in definitions and `idref` attributes in references. Although not strictly necessary, JavaML considers variables, fields, and arguments as distinct kinds of symbols. In line 39, we can see a reference to the local variable named `result` that points back to its definition in line 27.

```
27        <local-variable name="result" id="Ljunit/samples/FirstTest;var1946">

39          <var-ref name="result" idref="Ljunit/samples/FirstTest;var1946"/>
```

Because these references are defined in the XML Schema, they are automatically checked when the document is validated. The listing below shows the lines that define the keys and references for `local-variable` elements in the JavaML 2.0 schema (`java-ml.xsd`). The key is defined as a value appearing in the `id` attribute (line 213) of `local-variable` elements or `formal-argument` elements that are immediate children of `catch` elements (line 212). This key is referenced by values of the `idref` attribute (line 217) of `var-ref` elements (line 216). The values used in these keys and references are typical to symbol tables and are automatically generated by the Java compiler.

```
211         <xs:key name="KeyLocalVariable">
212          <xs:selector xpath=".//local-variable|.//catch/formal-argument"/>
213          <xs:field xpath="@id"/>
214         </xs:key>
215         <xs:keyref name="RefLocalVariable" refer="KeyLocalVariable">
216          <xs:selector xpath=".//var-ref"/>
217          <xs:field xpath="@idref"/>

218         </xs:keyref>
```

Type dependencies. Source code files usually have dependencies to other source files and libraries. During type checking and name resolution, all these files must be analyzed and a type dependency graph can be built. JavaML 2.0 representation includes these dependencies to complement the structural information presented before.

```
48  <type-dependences>
49   <type-dependence filename="f:/junit3.8.1/src/junit/samples/FirstTest.java"
50                                   signature="Ljunit/samples/FirstTest;">
51    <type-ref signature="Ljunit/framework/TestCase;"/>
52   </type-dependence>

53  </type-dependences>
```

This information enables following references to source code and documentation of external types. In line 6 of `FirstTest.java.xml`, we can see the value used to uniquely identify the class `FirstTest` and in line 11 there is a reference to the external type `TestCase`.

```
  <class name="FirstTest" id="Ljunit/samples/FirstTest;">

   <superclass name="TestCase" idref="Ljunit/framework/TestCase;"/>
```

Preservation of formatting and comments. The preservation of comments and whitespace were two issues not completely solved in the original JavaML implementation [1].

Although the comments are easy to store, they are challenging to attach to the correct elements. JavaML 2.0 preserves all comments present in source files and attach the formal ones, i.e. Javadoc comments [10], to their respective code elements (`class`, `method`, `field`, etc.) using the rules defined by Javadoc.

To enable the exact regeneration of original source files, the JavaML 2.0 has new `codeline`, `token`, and `comment` elements to store all the lexical information of a source code file. Below, it is presented the JavaML 2.0 lexical representation of line 1 of `FirstTest.java`.

```
<java-source-code>
 <codeline no="1">
  <token idx="1" line="1" column="1" type="preprocessor" lexeme="package"
             afterEol="true"/>
  <sp/>
  <token idx="2" line="1" column="9"  type="normal" lexeme="junit"/>
  <token idx="3" line="1" column="14" type="normal" lexeme="."/>
  <token idx="4" line="1" column="15" type="normal" lexeme="samples"/>
  <token idx="5" line="1" column="22" type="normal" lexeme=";"/>

 </codeline>
```

Due to the verbosity of this information, it is optionally generated. In the following listing is shown how Javadoc comments are represented.

```
 <codeline no="4">
  <comment idx="1" line="4" column="1" type="formal">/**</comment>
 </codeline>
 <codeline no="5">
  <comment idx="1" continued="true"> * My first unit test.</comment>
 </codeline>
 <codeline no="6">
  <comment idx="1" continued="true"> */</comment>

 </codeline>
```

3.3 Java to JavaML converter

Because JavaML was designed to be primarily written by tools, it is mandatory to have one converter from Java source files to JavaML. The approach followed is presented in [1] and consisted on adding one `XMLUnparse` method for each AST node of the IBM Jikes Java compiler framework (version 1.12). The result is a fast and robust JavaML converter [11].

To implement the schema enhancements of JavaML 2.0, the converter was initially migrated to Jikes 1.18 and then evolved to fit the new requirements. The generation of lexical information is implemented by embedding Jikes scanner results in JavaML elements. JavaML elements are generated by visiting the AST and its associated symbol and type annotations. The generation of semantic information to assign to the `id` and `idref` attributes of each program symbol is definitely the most trickier part to implement because it requires navigation in the AST and lookups in the symbol table and type definitions. The overall code that adds JavaML support to Jikes rounds 2000 lines of C++.

3.4 JavaML converters

Once generated, the JavaML representation can be easily processed using general purpose XML tools. Using an XSLT stylesheet it is possible to convert the JavaML representation to several other formats. To exemplify this, below are presented two converters: one for producing an HTML view of the source code and another for regenerating the original Java source file.

JavaML to HTML. The JavaML tools include an XSLT stylesheet that converts JavaML to HTML, named (`javaml-to-html.xsl`). Because JavaML 2.0 contains more lexical information than the original JavaML, the new stylesheet resulted simpler than the original [1]. It produces HTML that cross-links all symbol references (types, methods, variables, etc.) to their definitions in source code or documentation, depending on what is available. When compared to the original source code, the generated HTML view results more convenient for program understanding, because it enables good navigation from references both to internal and external definitions.

The conversion consists basically on two tasks: first, on applying a predefined style to each token, based on the kind it was assigned during scanning (literal, keyword, etc.); and, second, on defining anchor elements (`<a name=>`) and reference elements ((`<a href=>`) for each symbol definition and symbol referenced, respectively. The most important part of the linking is the conversion of `id` and `idref` values to anchor names and references, which is done with the help of a Java function that receives the symbol information and computes the link. The new stylesheet (`javaml-to-html.xsl`) has 21 template rules and around 300 lines. Below is listed the line of the template rule `create-link` that creates and formats the anchor name element.

```
  <xsl:value-of select="linker:getLinkFromId($javadoc-url,$filename,
      $type-signature,$kind,key('KeyTypeSignature',$signature)/@filename,$id)"/>
```

JavaML to Java. The regeneration of Java source code is straightforward to implement because the JavaML representation contains low-level lexical information about the tokens, comments and spaces of the original source file. The corresponding XSLT stylesheet has only 30 lines and only 6 simple template rules. The most complex template rule is for processing `token` elements.

```
<xsl:template match="token">
    <xsl:if test="not(@type='TK_EOF')">
        <xsl:value-of select="@lexeme" disable-output-escaping="yes"/>
    </xsl:if>
</xsl:template>
```

4 Applying JavaML

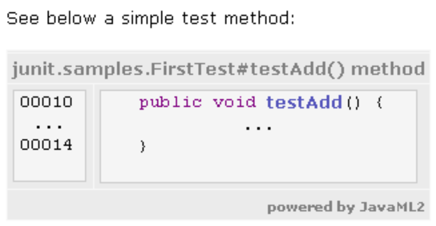
JavaML 2.0 is the result of evolving original JavaML [1] to fit the requirements of XSDoc [3], an extensible infrastructure based on a WikiWikiWeb engine that supports the creation, integration, publishing and presentation of documentation for object-oriented frameworks. XSDoc helps on creating and annotating framework documents, on integrating different kinds of contents (text, models and source code), and provides a simple and economic cooperative web-based documentation environment that can be used standalone in a web-browser, or inside an integrated development environment.

To illustrate the application of JavaML 2.0, it will be presented a concrete example taken from a simple usage of XSDoc for integrating in a web document source code from the file `FirstTest.java` and some text. XSDoc provides two dynamic mechanisms for the integration and synchronization of the possible kinds of document contents (source code, UML diagrams and XML files): inlining of contents, and automatic linking.

The inlining of Java source code is defined with a reference to the specific contents, annotated with the `<javaSource>` tags.

```
See below the method for testing the addition:
[<javaSource>]junit.samples.FirstTest#testAdd(); comments=no;
lines=first, last; [</javaSource>]
```

For example, the text above extracts the code fragment corresponding to the method `testAdd()` of class `junit.samples.FirstTest`, then removes all its comments, and returns its first and last line. This would produce the web page presented in Figure 2.


See below a simple test method:

junit.samples.FirstTest#testAdd() method

```
00010      public void testAdd() {
 ...                  ...
00014      }
```

powered by JavaML2

Fig. 2. Example of a XSDoc page inlining code from `FirstTest.java`.

4.1 Generating JavaML

After parsing the source code reference (in Javadoc format) contained in the `<javaSource>` tag, XSDoc finds that the code to inline must be in the file `FirstTest.java`. If the JavaML representation is outdated, it is updated by invoking the Jikes compiler with the appropriate arguments.

```
$jikes +B +L +c +T=3 +ulx FirstTest.java
```

4.2 Filtering JavaML

The JavaML document is then filtered to get the requested method. Based on the source code reference, XSDoc builds a XPath query and applies it using Saxon [12].

```
*[(name()='class' or name()='interface') and
@name='FirstTest']/method[@name='testAdd' and
./formal-arguments/descendant-or-self::*[last()=1]]
```

After this structural filtering, the document is lexically filtered to remove the comments, and then, the first and last line are extracted.

4.3 Converting to HTML

Finally, the resulting document is converted to HTML using the XSLT stylesheet described before in section 3.4.

5 Related Work

There are various research activities involving XML grammars for modelling source code and describing analytical aspects of code. These activities would benefit from having source code already in XML. The work most closely related with JavaML is srcML [13] and cppML [4]. Although not so relevant, in [1,**?**,**?**,**?**] is described other work not so directly related to JavaML.

Source Markup Language, srcML, is an XML format for source code markup that adds a layer on top of the original source code, leaving the source code untouched. The disadvantage of srcML compared to JavaML 2.0 is that in srcML the code is only semi-parsed because it doesn't include type specifications, for example, and in JavaML 2.0 the code is completely parsed and annotated with symbol and type information.

cppML is an XML grammar for C++ code that takes a similar approach to JavaML, but it doesn't provide a standalone cppML generator, requiring the usage of a compiler integrated in the VisualAge for C++ from IBM.

6 Conclusions and Future Work

XML-based representations for source code have several benefits over classical plain-text files that facilitate their manipulation by software engineering tools. They have an explicit structure, they contain information equivalent to an annotated abstract syntax tree, and they don't require language-specific parsing, but only general purpose processing using widely available XML tools. JavaML 2.0 is a rich alternate XML representation for Java source code evolved from the original JavaML that adds more source code information to the representation.

The JavaML 2.0 representation includes source code information at various levels of abstraction, starting from the lexical (tokens, comments, and formatting) and structural levels (abstract-syntax tree) to the semantic level (symbols, types and references). As a result, with JavaML 2.0 is possible to convert from Java source code to XML files, and convert back the representation to the original format without loosing information. Software engineering tools that intend to manipulate Java source code at levels of abstraction higher than the lexical one can now do it directly in JavaML representation without the effort of embedding Java language-specific parsers. JavaML 2.0 is thus an empowering representation that may leverage the development of more sophisticated software engineering tools for manipulating Java source code.

Future work should continue to refine the schema in order to make it more concise and even easier to produce and manipulate. Although the Jikes compiler proved to be a fast and robust JavaML converter, it would be interesting to provide open IDEs, such as Eclipse [14], with JavaML parsing and generation capability. This would promote the usage of JavaML in a wider range of tools and will open new perspectives on JavaML.

References

1. Greg J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):159–177, 2000.
2. Greg J. Badros. JavaML Home Page. http://javaml.sourceforge.net/.
3. Ademar Aguiar, Gabriel David, and Manuel Padilha. XSDoc: an Extensible Wiki-based Infrastructure for Framework Documentation. In Ernesto Pimentel, Nieves R. Brisaboa, and Jaime Gómez, editors, *JISBD*, pages 11–24, 2003.
4. Evan Mamas and Kostas Kontogiannis. Towards Portable Source Code Representations Using XML. In *Proceedings of WCRE'00, Brisbane Australia*, pages 172–182, November 2000.
5. Hrvoje Simic and Marko Topolnik. Prospects of encoding Java source code in XML. In *Proceedings of the ConTel 2003: 7th International Conference on Telecommunications, Zagreb, Croatia*, June 2003.
6. World Wide Web Consortium. XML Path Language (XPath) Version 1.0, November 1999. Available from http://www.w3.org/TR/1999/REC-xpath-19991116.
7. World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model, November 2002. Available from http://www.w3.org/TR/2002/WD-query-datamodel-20021115.
8. World Wide Web Consortium. XSL Transformations (XSLT) Version 1.0, November 1999. Available from http://www.w3.org/TR/xslt.
9. Ademar Aguiar. JavaML 2.0 Home Page. http://www.fe.up.pt/ãaguiar/javaml/.
10. Sun Microsystems. Javadoc Tool Home Page. http://java.sun.com/j2se/javadoc/.
11. IBM. Jikes java compiler. http://www.ibm.com/developerWorks/oss/jikes/.
12. Michael H. Kay. SAXON Home Page. http://users.iclway.co.uk/mhkay/saxon/.
13. Michael L. Collard, Jonathan I. Maletic, and Andrian Marcus. Supporting Document and Data Views of Source Code. November 2002.
14. Eclipse. Eclipse, an open and extensible integrated development environment, 2003. Available from http://www.eclipse.org.