Prospects of encoding Java source code in XML

Hrvoje Simic and Marko Topolnik
Department of Telecommunications, FER
University of Zagreb
Zagreb, Croatia
{hrvoje.simic, marko.topolnik}@fer.hr

***Abstract*—Currently, the only standard format for representing Java source code is plain text-based. This paper explores the prospects of using Extensible Markup Language (XML) for this purpose. XML enables the leverage of tools and standards more powerful than those available for plain-text formats, while retaining broad accessibility. The paper outlines the potential benefits of future XML grammars that would allow for improved code structure and querying possibilities; code extensions, construction, and formatting; and referencing parts of code. It also introduces the concept of grammar levels and argues for the inclusion of several grammar levels into a common framework. It discusses conversions between grammars and some practical grammar design issues.**

Keywords—XML, Java, source code, parsing, code formatting

I. Introduction

The goal of this paper is to explore the opportunities gained by representing Java program in XML format instead of plain text. We shall focus on an alternate format for Java source code, although most of the issues brought up are relevant to a wide spectrum of classical programming languages, such as C, C++, Pascal, or Smalltalk.

This paper regards Java as a programming language syntactically and semantically defined by the Java Language Specification, 2nd edition [1]. However, there are additional syntax and semantics specifications in wide use which are not defined in this document. Some of them are only informal and are referred to as code conventions—like rules for indentation and identifiers. Others are quite formal and precise, like rules for Javadoc comments and Java Beans method identifiers. Though commonplace and fused with the language, they are in fact extensions to it, defining semantics where it was left undefined. These extensions will not be regarded as part of the Java language in the context of this paper.

This paper continues with the overview of benefits of future XML grammars that would allow for improved code structure and querying possibilities; code extensions, construction, and formatting; and referencing parts of code. The third section discusses some issues encountered when designing such grammars. A section on related work follows. The paper ends with the conclusions and ideas for future work.

II. Benefits of Java in XML

A. Structure

The basic shortcoming of the plain text format is its "flatness", the absence of almost any explicit structure. A free-form plain text document represents a series of tokens, where every token is a simple character string. Any structure required by the programming language has to be coded into the relationships between such tokens. This structure becomes apparent only after a rather sophisticated process of parsing. The XML document model has inherent hierarchical structure easily designed to accommodate Java source code constructs. XML Schema [2], [3] is a standard language for defining rules that an XML document has to follow. Using a schema definition, any general tool that understands XML Schema can check the syntax of such a document, which may cover a significant portion of the language definition.

XML's hierarchical data model may be used to build a tree-like representation of code. The complexities of low-level source code constructs can then be hidden under a general header or description. Structured and appropriately marked-up code may also be easily presented with certain aspects of code hidden, e.g. code without comments, or comments without code; only the public interface, or only method headings without implementation.

XML grammar facilitates the design of a *tree-formed GUI editor* of code, instead of blank-paper-like textual document. These editors may improve developer's efficiency, reduce the necessity for remembering syntax rules, and enhance code quality. Still, such an approach is often associated with a rigid, compact user interface that does not allow writing code in a free-flow, syntactically unrestricted way the developers are accustomed to. It is our belief that such programs can be as unrestricting as any of today's popular code editors when used for normal code editing. However, unusual and less meaningful operations on code (e.g. deleting a single end-of-comment mark with the result of reshuffling the entire code structure) may be difficult to do in such an editor.

Code structure can be used to distinguish major from minor steps in an algorithm. This information can be used to automate logging and improve debugging. With traditional text files, this kind of information is added as redundant comments and log statements, making the code tedious to write and error-prone.

B. Queries

The quality of source code querying is essential to developers. Good queries should reflect the structure, be intuitive to construct and easy to read. They should also be able to return entire Java structures, not just character strings or primitive values.

Prospects of querying code in plain text format are quite limited. Simple textual search is not very powerful and other text-based technologies like regular expressions are too unintuitive and complex. Popular IDEs include Java-specific

search tools that usually allow searching for classes, field definitions, variable references, etc. Although useful, these features present only a small portion of queries needed by programmers or other programming tools.

XML provides a variety of useful standards and tools that facilitate construction of quality queries on Java code in XML. To illustrate, we present several typical examples on a hypothetical grammar. The query "find all public fields of the class *C*" could look like this in XPath [4]:

```
//class[ @name='C' ]/field[ @access='public' ]
```

The query "find all *if* statements in which the test contains a reference to the variable *x*" could look like this:

```
//if[ test//var-ref/@name='x' ]
```

The query "list names of all classes that contain calls to method named *m*" could look like this:

```
//class[ .//call-method/@name="m" ]/@name
```

The XQuery technology [5] can be used to generate reports from code in XML. The query "count all literal values in code by type" could look like this:

```
for $type in distinct-values( //literal/@type )
return
  <literal-count>
  {
    $type,
    count( //literal[ @type = $type ] )
  }
  </literal-count>
```

C. Extensions

In plain-text source, structure is deduced from the sequential relationship between tokens. Inserting a token anywhere in the sequence disrupts this structure. This makes it impossible to simply embed additional data about the code that the parser can process normally, but the compiler can ignore. In most cases, the only way is to specifically mark such data as a comment, ignored by the parser. In XML, since all structure is explicitly marked up, inserting additional elements into the tree does not disrupt it. XML facilitates this approach further through the technology of *XML Namespaces*. Every element and attribute can be marked as belonging to a certain namespace. On the other hand, every tool defines its unique namespace and processes only the nodes belonging to it. This makes it easy to introduce *orthogonal extensions* to any XML document type—extensions invisible to the primary tool using the document.

The idea of orthogonal extensions is very important in the context of software development and has been applied to the traditional source file format even though it had to cope with its severe limitations. This section will give an overview of useful orthogonal extensions.

There are many orthogonal extensions which are not application-specific. For example, information about the document: time of creation and last modification, original author, authors of modifications, copyright information etc. In XML, a vocabulary for this kind of information has already been defined and is called Dublin Core. It can be applied with no further work to any XML document and it will be processed by general tools handling Dublin Core data.

1) Annotations, clarifications and other comments

As their name indicates, the original intention behind the support for comments in plain-text source was to allow the embedding of annotations, clarifications and other types of comments about the code. Any new coding format should retain such a general facility, but can add more semantics. For example, the part of code to which a comment pertains can be explicitly specified. Also, a comment could be typed, indicating what kind of role it plays—is it a programmer's clarification, a to-do item, an annotation from a reviewer etc.

2) Versioning and revision tracking

Version information pertains to the whole document, whereas revision tracking can be made more fine-grained and changes can be tracked for each node separately. Such an approach could enable a single file to contain any number of code versions. This would reduce file clutter, simplify code maintenance and enable more powerful analysis of the evolution of a code module.

3) Access control

When a module is being edited by several developers, they usually have different roles in the development. For example, one developer could design object interfaces and another could develop interface implementations. The development process would benefit if information about such roles and their associated access rights is embedded into source code.

A special role in code development is played by a reviewer. The support for such a role involves the support for access control, annotations, and tracking of changes. A reviewer is not allowed to edit any part of code, but can add annotations and propose changes to the code. The primary developer can later analyze the annotations and accept or reject the proposed changes.

4) Documentation

The placement of the documentation about a code module's functionality and internal details together with its code has many advantages. The documentation always accompanies the code and is therefore always accessible. Some aspects of consistency between code and documentation are automatically maintained because there is always a unique location where the documentation has to be placed. The code element that the documentation refers to is guaranteed to exist and if the element is relocated, the documentation follows it.

5) Temporarily inactive code

Sometimes a part of code becomes redundant, obsolete, or broken so it becomes necessary to exclude it from compilation without erasing it completely, but rather marking it as inactive. This has been traditionally done by surrounding that section of code with block comment delimiters or prefixing each line with a line comment marker. This technique is known as "commenting-out", and it is a source of many complications, such as distinguishing it from other types of comments and treating nested comments. Commented-out code is not checked for syntax, which may be occasionally desirable.

Another technique using the idea of inactive code is called conditional compiling. Typically, parts of code are provided for

testing and debugging purposes and have to be deactivated in the release version, but are kept for reuse in the next development cycle. The conditionally-compiled blocks are usually scattered throughout the code and they are all activated or deactivated at once. This calls for a richer model in which the code section's active/inactive status is computed before compilation.

D. Format

Majority of program languages, including Java, ignore the semantic value of whitespace, identifiers, and several other human-oriented features of code, leaving the programmer to invent or adopt various code formatting conventions. Whitespace conventions are vital to visualize the inherent structure of the code. Semantically, this formatting is completely decoupled from the true structure of the code. Thus, an error in source code formatting can result in bugs that are very hard to locate because of the suggestive nature of formatting, which is irrelevant to the parser.

It is difficult to enforce and maintain a consistent coding style across the organization and over longer periods of time. Using an XML grammar which abstracts at least some aspects of coding style effectively eliminates that problem. When reconstructing plain text source code from XML, any formatting style preferred by the reader can be applied.

XSLT [6] may be used to transform source code in XML into popular XML display formats such as XHTML [8] (for on-screen presentation) or XSL/FO [7] (for printing on paper). These rich formats may improve readability by introducing various layouts, fonts, colours, linking, etc.

E. Referencing parts of code

In plain text, a part of code is usually referenced by its position inside the file. For example, the Java compiler reports a compile-time error with a reference to the line and column number. The referenced position in the source file indicates a token relevant to the error. The problem of this approach is that the programmer has to deduce from the single token which code construct is in error.

Since a Java program is a hierarchical structure of code constructs, they may be referenced through the XML nodes representing them. This works equally well for low-level (such as literals and method parameters) and high-level constructs (like methods and classes).

F. Construction

Advanced code editors have tools for quick code construction in the form of templates. Templates are usually just fragments of code with context-specific parts left out. The user inserts the predefined fragment in its code and fills in the blanks. Once inserted, the constructed code is detached from the template definition. The template itself cannot be checked for syntax because it is usually incomplete code. Also, template code format does not usually match the format of the destination code.

XML code may be formed using templates in XSLT, providing greater power and flexibility. XSLT template transforms data specific to that template into Java code in some XML grammar. The constructed code may be linked to the template definition, so that the destination code may be reconstructed every time the template definition is updated. Template could be checked for validity, since every placeholder has declared type, as could the input data. And, with separation of code formatting, transformation may be generally invariant to code styles.

To illustrate, we present an example with a typical template for the *equals* method. Template's input data consists of one Boolean expression and the name of the variable used in the expression containing the other object. Target class name is extracted from the context (the class where the method is inserted). Variable parts of the template—the placeholders—are shown in italic:

```
public boolean equals( Object o )
{
  if ( !( o instanceof targetClass ) )
    return false;
  targetClass targetVar = (targetClass) o;
  return booleanExpression;
}
```

Input data is also formatted in XML. In our example, the class *Complex* represents complex numbers, in which equality is calculated as equality between the real and imaginary parts. The data is contained in an appropriately named element placed in the class definition.

```
<equals-method varname="z">
  <and>
    <equals>
      <field-ref name="re" />
      <field-ref name="re">
        <var-ref value="z" />
      </field-ref>
    </equals>
    <equals>
      <field-ref name="im" />
      <field-ref name="im">
        <var-ref value="z" />
      </field-ref>
    </equals>
  </and>
</equals-method>
```

The transformation produces the XML code equivalent to:

```
public boolean equals( Object o )
{
  if ( !( o instanceof Complex ) )
    return false;
  Complex z = (Complex) o;
  return (re == z.re) && (im == z.im);
}
```

More complicated templates may be used to enforce Java design patterns (e.g. a *typesafe enum* pattern [7]).

III. Design Issues

This section tries to give some insight into the process of designing a successful XML grammar for Java. We introduce important concepts of grammar levels and grammar and format conversions. Issues of describing type and code inactivity are given to illustrate more detailed design.

A. Grammar levels

XML had sprung from the essential need in complex information systems for separating content from presentation. The principal problem of designing a useful grammar is this: where to draw the line between content and presentation in

Java source code? Certain aspects of source code can be considered as content in some applications and as presentation in others. In this paper, we call XML grammars following the latter approach higher level grammars.

The typical approach present in related projects (JavaML, cppML, srcML being the clearest examples) is to propose a single XML grammar for source code and argue its benefits. In our view, it makes sense to specify several grammar levels and include them all in a common development framework. This is especially true in the context of transition from traditional plain text to XML.

1) Preserving original code

In one extreme, we can postulate that source code is all content and that the exact sequence of characters making up the source code is to be preserved in the XML document. This approach does not force the author of code to change their routine and still brings important improvements over the plain-text source (like in srcML [15]). The most straightforward grammar design following this approach uses XML tags to markup existing code. For example, the Java statement:

```
public static String s;
```

may be marked-up like this:

```
<field-declaration><access>public</access>
<static>static</static> <type>String</type>
<name>s</name>;</field-declaration>
```

Each relevant syntax element is marked-up with appropriate element. Transforming the XML to original source code involves just the simple matter of removing markup. The disadvantage of this approach is that all whitespace in such an XML document is part of content, which leads to problems with presenting and editing it. To remedy the situation, a special element (e.g. "text") may be introduced to cover all otherwise unmarked-up characters forming the original source code, but with the cost of code clutter:

```
<field-declaration>
  <access>public</access>
  <text> </text>
  <static>static</static>
  <text> </text>
  <type>String</type>
  <text> </text>
  <name>s</name>
  <text>;</text>
</field-declaration>
```

Taking a different approach, some grammar designs may allow losing some information about whitespace, such as line-trailing whitespace and tab/space differences, which are frequently discarded in code editing software. Further relaxation of criteria leads to preserving only the line and column at which the statement or other Java structure starts in the source code, possibly losing some whitespace between words, like in CppML [11].

2) Enforcing coding conventions

The second, higher-level Java XML grammars are those that abstract the original Java source syntax, leaving out the aspects usually regulated by coding conventions. They may therefore lack the information about code presentation, i.e. formatting styles such as whitespace or identifier construction (as described in section II.D). They may also exclude some bad or unnecessary coding options, thus improving the overall code quality.

3) Semantic enhancements

Further abstracting the Java code details and approaching the developer's view of program leads to adding information to the XML formatted code rather than removing it. This information enhances the code semantic, allowing it to be better understood and manipulated on a higher level, especially on the level of human reasoning.

One simple enhancement may be grouping statements in logical clusters usually indicated with vertical spacing. The comment at the beginning of such a cluster usually refers to the whole cluster and not just the following statement. A complex enhancement may be marking a class as immutable.

B. Format conversions

1) Converting existing plain text code to XML

Low-level XML grammars for Java source code are closer to Java plain text format, and the conversion is quite straightforward. However, conversion to higher-level grammars poses a lot of questions concerning the author's intent [10], [15]. If the code is not carefully formatted, the answers to these questions may not be apparent even to a human reader. Therefore, direct conversion from plain text format to higher-grammar XML format is challenging and in most cases not worthwhile. The target use for higher-level grammar Java code is to acquire it directly from the code author and not via lower-level formats.

2) Converting source code in XML to plain text

While low-level grammar XML code (which preserves original formatting) is easy to convert to plain text format by definition, transforming higher grammar XML code requires a code-style template. Definitions of such templates should not include the rules for constructing Java syntax because they are style-invariant. However, such decoupling the style rules from syntax rules may be challenging.

Standard code formatting features like indenting, identifier case formatting and line spacing that are common to all code-style templates and yet challenging to implement in XSLT should also be omitted from the individual code-style templates. A possible solution is to design a special language-independent, code formatting XML grammar to use as a mediator when transforming code from XML formats to plain text. This is an example of a piece of code marked-up with such grammar:

```
<section before="1" after="1">
  <line>private Suit(</line>
  <section indent="1">
    <line>String name</line>
  </section>
  <line>)</line>
  <line>{</line>
  <section indent="1">
    <line>this.name = name;</line>
  </section>
  <line>}</line>
</section>
```

The resulting plain text source should be:

```
private Suit(
```

```
  String name
)
{
  this.name = name;
}
```

3) Conversion between XML grammars

An overview of conversions between various formats and grammars mentioned in this paper is given in Fig. 1. Natural course of conversions is from higher-level to lower-level grammars.

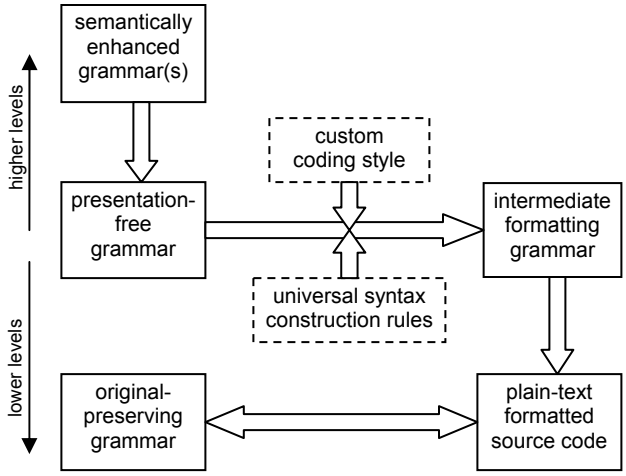

Figure 1. Conversions between XML grammars and plain text format.

C. Describing type

A typical schema design problem is the construction of Java type. Type is referenced in situations such as declaring local variables, methods or class fields. Type in Java may be a simple type (like "int" for integers), a class or an interface. A special case of class types are array types, denoted with the type identifier suffixed with square brackets "[]" (e.g. "int[][]" references an array type with two dimensions of "int" element type).

Table I. shows some possible solutions to the problem. The easiest solution (a) is to describe the type as a string value of a standard attribute, e.g. type="int[][]". Determining the structure of that array type implies string parsing, avoidance of which is the very reason the XML grammar was introduced, so this solution is undesirable. A better solution (b) [JavaML] is to describe the type as a subelement. However, enforcing the use of subelement in each use case complicates the schema and the handling of the document. Our favourite solution (e) is to describe the array type with a special identifier as the value of attribute "type" not used by other types, namely "[]", and additional two attributes to define the array element type and dimensions.

TABLE I. FIVE DESIGN SOLUTIONS FOR DESCRIBING JAVA TYPES "INT" AND "INT[][]" IN XML.

a) attribute with Java type name	`type="int"` `type="int[][]"`

b) subelement with name and optional dimensions attributes	`<type name="int" />`
	
	`<type name="int" dimensions="2" />`

c) recursive subelements containing component types for arrays	`<type name="int" />`
	
	`<type name="[]">`
	`  <type name="[]">`
	`  <type name="int"/>`
	`</type></type>`

d) attribute "is-array" (default false) distinguishes arrays	`type="int"`
	
	`is-array="true" element-type="int" dimensions="2"`

e) special value "[]" of attribute "type" is used to denote arrays	`type="int"`
	
	`type="[]" element-type="int" dimensions="2"`

D. Denoting inactive code

To denote (in)activity (as described in section II.C.5), relevant elements constituting code could simply be assigned an active/inactive status. Each element has an optional *active* attribute defaulting to *true*. Marking an element as inactive automatically inactivates all containing elements, so an element representing an active piece of code must have all super-elements marked active. The value of this attribute would not prevent checking element structure against the schema.

For example, a typical auxiliary statement in Java

```
user.notify( "Pass 1" );
```

can be rendered inactive in XML format with a single attribute value (attribute *active* is defined in a separate, language-independent namespace, denoted here with prefix *x*):

```
<method-call x:active="false" name="notify">
  <var-ref name="user" />
  <arguments>
    <literal type="String" value="Pass 1" />
  </arguments>
</method-call>
```

Note that the content of the inactive element is still required to be well-formed and valid, and that subelements of the inactive element are also considered inactive.

Conditional activation may be implemented by extending the *active* attribute to values such as "always" for "true" and "never" for "false". Standard words for logical operators might be included, and non-reserved words may be used as global Boolean variables. Examples of attributes:

```
x:active="not release"
x:active="release and demo"
```

If inactive code is not to be checked for grammar, it should be enclosed in the special element defined in the schema not to further check its content. This element should not have conditional activation feature.

IV. Related Work

There is much research activity involving XML grammars for source code. There is also much related activity, involving XML grammars for describing various analytical aspects of code. Such related technologies would benefit if the original source was already in XML. Below we mention several projects from both fields of research.

The Software Development Foundation (SDS) is an open architecture designed for developing tools for software development. The core of SDS is the XML-based Code Structure Format (CSF) [12]. The purpose of this format is to build a database of high-level analytical information about a software project's code base. The information is thus made available to various analytical tools. It does not contain full program details—for example, it does not describe method implementations.

Source Markup Language, srcML [15], is an XML format for source code markup. The markup forms a separate layer superimposed over source code. The source code remains untouched and retains all plain-text formatting (whitespace). The markup layer explicitly describes the inherent structure of the source code. This approach enables XML-oriented tools to gain access to semi-parsed code structure, while still leaving the source code intact. The code is only semi-parsed because, for example, data type specifications are left unparsed.

JavaML [10] is an XML format prototype proposed by Greg J. Badros. It is intended as a canonical representation of Java source code—replacing the Java source code as the baseline format for tools. It considers as content only those aspects of source code needed by the compiler to produce bytecode.

cppML [11] is an XML grammar for C++ code, taking an approach similar to JavaML, but adding more information about original plain-text source, such as line/column numbers. In type descriptions, it includes both parsed (XML attributes) and unparsed (simple string) description.

GXL [14] is an XML-based format for describing typed attributed graphs. It supports hierarchical graphs where nodes can be decomposed into subgraphs. It also supports hypergraphs where edges can connect more than two vertices. It is intended to serve as a standard exchange format for graph-based tools and has already taken roots in the software-engineering community. This format is not intended to represent the exact program code, but its higher-level analytical aspects. General technologies like XSLT and XQuery could be used to reverse-engineer program code (written in an XML grammar) into GXL and also to produce code based on GXL graphs.

The Software Concordance [13] is a hypermedia software development environment exploring how document technology and versioned hypermedia can improve software document management. A component of the IDE is a uniform document model for storing source code as well as all other project-related documents. The documents can contain hyperlinks and embedded multimedia content. XML is used to accomplish a flexible and multilayered document model which supports hypermedia and fine-grained versioning.

V. Conclusions and Future Work

In this paper, the benefits of encoding Java source code in XML were presented. The benefits include improved code structure and querying possibilities; code extensions, construction, and formatting; and referencing parts of code.

The important concept of grammar levels is introduced. We argue that the correct approach is to specify grammars at several levels of abstraction instead of only one and include them in a common development framework. Each level has its own uses because it possesses characteristics beneficial to some but not all aspects of development.

Future work should provide a blueprint for a complete architecture, with a detailed set of requirements for Java code grammars at various levels and their exact definitions in XML Schema. Java-specific features and standards need to be placed in an appropriate context of language-independent base standards and conventions. These base standards would include specifications for different types of extensions to programming languages' source code. Also, a wide range of tools need to be developed, such as Java-in-XML source code editors and rich format templates.

References

[1] Gosling, James, Joy, Bill, Steele, Guy, Bracha, Gilad, *The Java Language Specification, Second Edition*, Sun Microsystems, 2000.

[2] World Wide Web Consortium, *XML Schema Part 1: Structures*, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-1-20010502

[3] World Wide Web Consortium, *XML Schema Part 2: Datatypes, May 2001*, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502

[4] World Wide Web Consortium, *XML Path Language (XPath) Version 1.0*, November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116

[5] World Wide Web Consortium, *XQuery 1.0 and XPath 2.0 Data Model*, November 2002, http://www.w3.org/TR/2002/WD-query-datamodel-20021115

[6] World Wide Web Consortium, *XSL Transformations (XSLT) Version 1.0*. November 1999, http://www.w3.org/TR/xslt

[7] World Wide Web Consortium, *Extensible Stylesheet Language (XSL) Version 1.0*, October 2001, http://www.w3.org/TR/2001/REC-xsl-20011015

[8] World Wide Web Consortium, *XHTML™ 1.0, The Extensible HyperText Markup Language (Second Edition)*, August 2002, http://www.w3.org/TR/2002/REC-xhtml1-20020801

[9] Bloch, Joshua, *Effective Java Programming Language Guide*, Addison-Wesley, 2001.

[10] Badros, Greg J., *JavaML: A Markup Language for Java Source Code*. Proceedings of 9th International World Wide Web Conference (WWW9), Amsterdam, The Netherlands, May 13-15 2000.

[11] Mamas, Evan, Kontogiannis, Kostas, *Towards Portable Source Code Representations Using XML*, Proceedings of WCRE'00, Brisbane Australia, November 2000, pp.172-182.

[12] Sandø, Stig E., *CSF specification*, February 2000, http://sds.sourceforge.net/doc/csf.html

[13] Gupta, Satish C., Nguyen, Tien N., Munson, Ethan V., *The Software Concordance: A User Interface for Advanced Software Documents*, Proceedings of 6th IASTED International Conference on Software Engineering and Applications, November 2002.

[14] Winter, A., Kullbach, B., Riediger, V., *An Overview of the GXL Graph Exchange Language*, Software Visualization International Seminar (revised lectures), Germany, Springer Verlag, May 2001.

[15] Collard, Michael L., Maletic, Jonathan I., Marcus, Andrian, *Supporting Document and Data Views of Source Code*, DocEng'02, November 2002, McLean, Virginia USA.