An improved XML syntax for the Java programming language

Marko Topolnik
University of Zagreb, FER
Unska 3, Zagreb, Croatia (HR-10000)
marko.topolnik@fer.hr

***Abstract*— The benefits of using XML to encode programming language's source code have already been identified and XML syntaxes developed for this purpose. However, there is an absence of a systematic identification of the requirements that need to be met in order to fully leverage the potentials of XML and the key concept of the abstract syntax is being misinterpreted. Due to this the syntaxes have various deficiencies, a number of which are common to many of them. This paper clarifies the concepts, identifies these deficiencies, and proposes a new syntax for the Java programming language which avoids them. The superiority of the new syntax is demonstrated by comparing it with two of the most relevant publicly available syntaxes with the same purpose.**

I. Introduction

The benefits of using XML to encode programming languages' source code have already been identified [1], [2] and XML syntaxes developed for this purpose [3], [4], [5]. However, there is a lack of a systematic approach to the identification of more precise goals the syntax should achieve in order to fully leverage the potentials of this approach. Particularly, it appears that the central concept in the application of XML to source code, that of the abstract syntax, is not being interpreted properly.

In order to explore this subject, a research project named Jezix was started. The main goal of the project was the development of an XML syntax for the Java programming language that is based on the correct treatment of the concept of the abstract syntax, which allows it to optimally leverage the identified potentials of XML. The results of the research were validated by demonstrating the proposed syntax's superiority over existing syntaxes. This was achieved by solving several code analysis problems for the proposed and two of the most relevant existing syntaxes and then comparing the amount of effort needed and the properties of the resulting query code.

The next section gives an overview of source code and XML. It repeats some well-known results of the formal language theory, but also clarifies important concepts and defines some terms used later. Section III presents the two key benefits of using XML for source code. Section IV reflects on the effects of using XML on the quality of the machine-to-human interface. Section V presents the most sensitive requirements the XML syntax has to meet. Section VI gives a critical overview of other existing syntaxes and identifies the common root of many of their deficiencies. Section VII briefly introduces the proposed syntax and proceeds to the presentation of the results of its comparison with the two other syntaxes. Finally, the closing section summarizes the achievements and conclusions of the Jezix project.

II. Overview of source code and XML

A programming language has two important components:

1) A set of basic concepts and a syntax (called the *abstract syntax*) for combining them into a description of the desired behaviour of the program. The description has the form of a tree, called the *abstract syntax tree* (AST).
2) An alphabet of characters and a syntax (called the *concrete syntax*) for combining them into a textual encoding of the AST—the source code.

The text of the source code, then, is structured according to the concrete syntax, whose rules are usually separated into two layers for easier formalization, called the *lexical* and *syntactic* grammars. The compiling process begins with the *lexing* of the source code text into a sequence of *tokens* (character strings) according to the lexical grammar. The token sequence is then *parsed* according to the syntactic grammar and the result is a tree, called the parse tree, or the *concrete syntax tree* (CST). The syntactic grammar is formally a *context-free grammar* (CFG [6]), usually defined in the *Extended Backus-Naur Form* (EBNF). The rules of the grammar have to be such that each valid token sequence can be parsed into exactly one CST. Otherwise the grammar would be *ambiguous*: it would allow different iterpretations of the same source code text.

The AST is produced by processing the information found in the CST and possibly additional information from the broader environment, such as other code modules. The program description it represents can be regarded as the *meaning* of its source code document. The abstract syntax contains additional, higher-level constraints so that only a subset of valid CSTs produce a valid AST. In a typical programming language specification these constraints are only described in plain English, without precise formalization. The abstract syntax also has an important formal property that sets it apart from the concrete syntax: since it is never used for parsing, the idea of ambiguity as defined above—and thus the associated requirement—becomes irrelevant.

Although it can be argued what the essence of XML is, for the purpose of this paper it will be regarded as

a formal language for describing trees in textual form. That is, an XML document can be viewed as plain-text source code of a tree-describing language. The tree described by it is called the document's *infoset*. The concrete syntax of XML is formalized through EBNF in the specification document [7] and the abstract syntax is the generic syntax of the infoset, the one that has to be respected by any *well-formed* document. The generic infoset syntax is very lax, only defining several types of nodes and specifying general rules about their allowed combinations. It can be specialized through the use of a *schema*. Typically, XML schema languages (e.g., DTD and W3C XML Schema) allow the definition of what is essentially a CFG adapted to the generic syntax of the infoset. The document following the rules of a schema is said to be *valid* with respect to it.

In particular, the schema can be defined so that it corresponds with the abstract syntax of a programming language. In this paper, the term *source code infoset* will be used to refer to an infoset following such a schema. The paper considers the idea of using the XML encoding of the source code infoset in place of the regular source code.

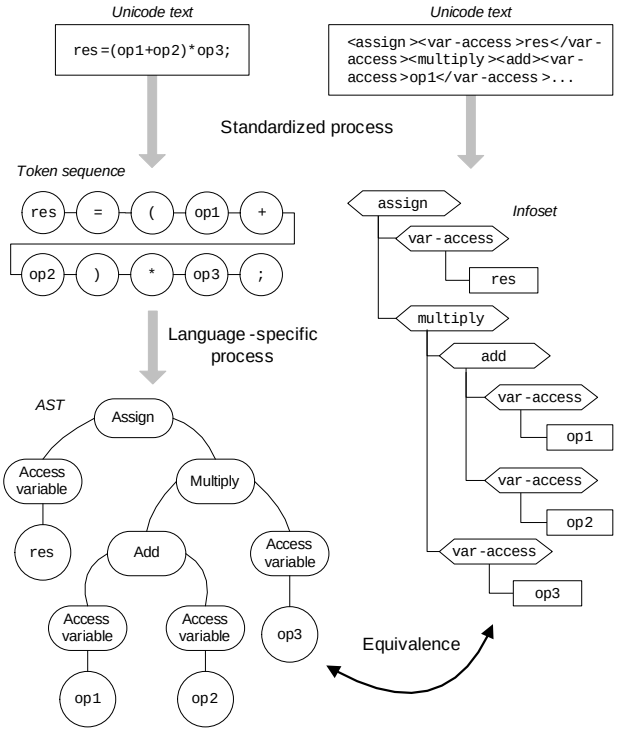

Fig. 1. Comparison: traditional source code and XML

III. The key benefits of XML

A. Task automation

Programming languages routinely use as the alphabet of their concrete syntax the character set of a global text-encoding standard, such as ASCII or the more recent Unicode. The application of a global encoding standard is a prerequisite for code portability because it has to be supported on every computing platform. It also makes it possible to develop generic text editors and text-processing languages that can be used in a very broad range of specific situations. These tools have been used extensively in software development to manage big software projects, automate analytical tasks, ease the coordination between remote developers, etc.

XML relies on the existing text-encoding standards and additionally standardizes an infoset-describing CFG. If traditional source code (with language-specific concrete syntax) is used, a generic process (shared by virtually all languages with minor variations) can be defined to lex the text into a series of tokens, but a language-specific process (involving the language's concrete syntax EBNF) is needed to produce the CST. In addition, the abstract syntax needed for transformation into the AST is not even formalized by the language specification. With XML the concrete syntax is always the same and a generic XML parser can produce the infoset of any document. In the scenario under consideration, the infoset will be equivalent to the AST. This difference is illustrated in Figure 1.

Owing to this, generic tools can be developed that manipulate the infoset directly instead of its plain-text encoding. Today there are several standardized infoset-processing languages, the most important being XSLT (W3C Recommendation since 1999) and XQuery (still in the Working Draft phase, but many tools already implement it). The first key benefit of XML is that the generic and widely supported infoset-processing languages can be employed to drastically increase (in comparison with text-processing languages) the complexity of automation problems that are feasible to solve.

B. Orthogonal extensions

There are many kinds of metadata, not needed to describe the AST, but whose presence inside the source code document is still desirable from a wider perspective. They can be placed inside *orthogonal extensions* of the source code document—"extensions" because they extend the document's primary content, and "orthogonal" because they do not influence the way it is processed by the compiler. The most well-known example of an orthogonal extension is embedded API documentation, such as Javadoc. There are issues with adding such data items to the traditional source code, stemming from the use of a language-specific concrete syntax. The concrete syntax provides only for the inclusion of the primary content in the source code. If metadata are inserted as additional tokens, they will break the syntactic rules. There are solutions, but they have various drawbacks. For example, Javadoc places its data inside comments. These data are ignored by the language parser and hence they are not a part of the resulting AST (except possibly as the unparsed text of the comments). Another approach, introduced in Java version 5, is to add support for orthogonal extensions explicitly in the concrete syntax. It allows extensions to be added inside "annotations" of the source code. The metadata can appear in the AST, but they can be placed only at predetermined locations. Even though it is

theoretically possible to add comments anywhere, Javadoc still restricts the points of extension to reduce the clutter. For example, documentation about a method is placed above its definition. This calls for redundancy—method parameters have to be referred to via their identifiers, for example. The link can be broken if the parameter name is changed, but not in both places at once. This is illustrated in Figure 2.

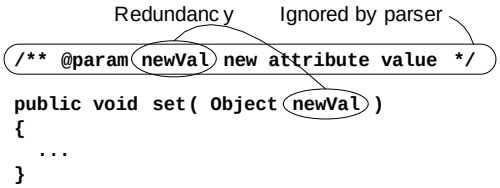

Fig. 2. Problems with Javadoc

XML's universal concrete syntax allows a new node to be added anywhere without changing any rules. Also, by employing the XML Namespaces standard, additional nodes can be marked as not being a part of the primary content—that is, as being metadata. There is no need for a programming language's schema to explicitly support this feature. Figure 3 illustrates this. Notice the prefix `doc:` on the description element. This places its name into a separate namespace and is used to identifiy it as the container of an orthogonal extension. Also notice how the description element is placed inside the parameter element and thus any redundancy is avoided.

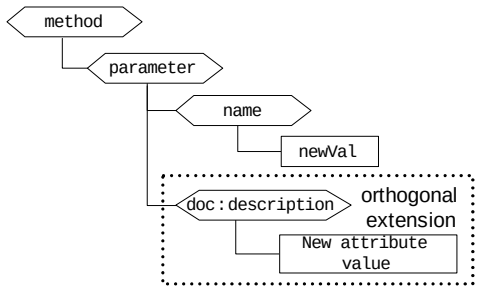

Fig. 3. Equivalent of Javadoc in XML

IV. Human interface issues with XML

The concept of a context-free grammar was developed in an attempt to formalize the syntax of natural languages. The reasons to employ a language-specific CFG for source code are twofold: using a CFG in general allows the code to inherit the structural characteristics of natural languages and using a language-specific CFG allows it to be closely related to the language's abstract syntax. This allows source code constructs to resemble natural language sentences and makes the code easy to read and write directly as text. On the other hand, with XML a simple, universal concrete syntax is used, generally unrelated to the abstract syntax of the programming language. The document looks much less like natural language text and this makes it less readable. The use of a universal syntax also introduces more redundancy so the documents are bigger, usually by a factor of three to five.

But, in a comparison between generic editors for text and XML as human interfaces, the XML editor can have advantages of its own. XML can be presented in the form of the infoset, not text. There will be no chance for the user to create an ill-formed (unparseable) document. If we accept that a global standard for a machine-processable schema definition will be an integral part of the XML framework, the editor would be able to monitor the user's input for mistakes and correct them at once. It would also be able to help the user with a list of nodes allowed at the insertion point. A text editor can achieve such functionality only by being language-specific.

A tree-based user interface with schema-based assisting seems like a very good interface for a beginner—node labels can make the source code self-explanatory. But to an experienced developer, working with the full tree structure could become a burden. He will prefer to author and read code using a more concise syntax. The use of XML-based source code does not preclude such a form of human interface, but it could be just one part of the whole development framework, which could employ a variety of human interface paradigms for code authoring and presentation.

V. Requirements on the syntax

The key to successfully applying XML for source code lies in a good design of the programming language's abstract syntax—more precisely, the schema of the source code infoset. It has to allow the potentials of XML to be fully leveraged. This section discusses the two most sensitive requirements, which are both very important and very involving to meet.

A. Query-friendliness

A query is generally a request to extract specified information from a data source. One of the two most important benefits identified above that XML can bring is the advantage in task automation and practically any code-processing task involves acquiring some information from the source code document. That means that the automatic processing of queries is crucial to task automation. If the infoset syntax is not carefully designed, a piece of information that could in principle be trivial to acquire might require a very complex query. So before every design choice is made, it has to be asked how it will influence the queries.

B. Strict WXS definition

As already mentioned, native XML editors can use a machine-processable schema definition in order to assist the code author by suggesting the correct syntax at every point and correcting mistakes. Also, any processing tool will be more reliable and easier to develop if it can rely on automatic validity checking of input documents. These and other reasons make it important for the schema definition to formalize as many rules of syntax as possible.

There is a choice of many XML schema languages today with varying degrees of expressiveness. The W3C

XML Schema (WXS) is overall the most expressive and, apart from the old DTD, the best-supported one. It is establishing itself as the global standard and can be expected to become an integral part of the XML framework. This means that support for it will be taken for granted on any computing platform.

In addition to automating validation, WXS introduces a rich set of built-in datatypes and provides for the definition of new ones, including complex datatypes. By using type information from the document's schema definition more sense can be made out of it in an automated way. For example, the new generation of core XML technologies (XPath 2.0 and XQuery 1.0 [8]) allow the formulation of more powerful queries based on type information.

VI. Related work

Several publicly available XML source code syntaxes were analyzed with respect to the stated requirements. Some design choices which reduce their quality (make them satisfy the requirements to a lesser proportion) were repeatedly found in these syntaxes. For many of these deficiencies a common root was identified: the design of certain parts of these syntaxes is **influenced by programming language's concrete syntax**. This makes them describe certain meanings in a complicated, roundabout way that requires an unnecessarily sophisticated process to automate their understanding. This is reflected in the complexity of queries that have to access those meanings. The probable cause of this is the typical structure of what is called the AST inside compilers. The syntax of these trees is not the true abstract syntax; it is actually very close to the concrete syntax of the standard source code. This section presents these analyzed syntaxes.

A. Syntaxes used for comparison

JavaML by Greg Badros [3] shares a number of design goals with Jezix and is overall the most similar to Jezix's. The main difference is that it is designed to have a stronger connection with the standard source code. Its rules enable it to retain the exact choice of concrete syntax constructs and even some formatting. This is an additional source of weaknesses discussed below.

Uses special identifiers. The syntax does not explicitly support the expression types "*class variable name*", "*static method invocation*", "*qualified this*", "*qualified super*" and "*class literal*". These meanings are produced by using keywords as identifiers so that, after the code is transformed into the standard format, its meaning is appropriate. Thus there are special cases in which the meaning of the constructs changes according to rules analogous to the rules of Java's concrete syntax. This makes the understanding of the constructs difficult to automate. An example is the *class literal* expression, written `Object.class` in the standard syntax:

```
<field-access name="class">
  <var-ref name="Object"/>
</field-access>
```

Here a construct with the primary meaning "*field access*" is used to encode the class literal by employing the keyword `class` in place of the field name. Without reference to the standard Java syntax, such a rule makes no sense. It has a devastating effect on some queries, such as those used in the comparison with the Jezix syntax.

Incomplete. The syntax definitely does not support the expressions "*qualified class instance creation*" and "*qualified superclass constructor invocation*". They are not expressible even in a construct that would have the correct meaning after transformation into the standard source code.

Language-specific concrete syntax is retained for some constructs. This is in itself a deficiency, but it is made worse by the fact that the meaning of these constructs is context-dependent—it depends on the contents of other visible code modules. For example, the package name is not separated from type name (as in `java.util.List` which can, depending on the context, mean a nested class `List` inside the class `util` which is in the package `java`, or a class `List` in the package `java.util`). The situation is even more difficult in places where both variable identifiers and type names can occur. For example, in the construct

```
<field-access name="out">
  <var-ref name="System"/>
</field-access>
```

`System` might be the name of a local variable, parameter, attribute, or a type. Depending on the nature of the referenced entity, the meaning of the construct thoroughly changes. Another example of language-specific syntax is in array type specification, which uses brackets as part of the type name (as in `Object[]`). The meaning of this construct is not context-dependent, so the only problem is the need for language-specific string parsing.

Uses recursive forms to encode binary operations and array access. Even though they are formally correct, queries into such constructs are cumbersome and unreadable. The concrete syntax uses the infix notation for operations, which forces the binary operation model. The left operand can be an operation of the same type, thus forming a binary tree equivalent to a list. It is equally correct to define operations as accepting a list of two or more operands and the resulting constructs will be much easier to process. Multidimensional arrays are modelled in the concrete syntax as one-dimensional arrays of one-dimensional arrays. Access of an array element is then modelled as multiple nested accesses. It is equivalent and more practical to treat arrays directly as multidimensional.

Represents parentheses. Operator precedence, as well as parentheses made necessary by it, are strictly the consequence of the infix notatation used in the concrete syntax. In order to reduce the need for explicit operator binding, operator precedence is defined and parentheses are used when it needs to be overridden. The infoset syntax has no reason to use the infix notation and operators are always explicitly bound to their operands, so operator precedence and parentheses only add unneeded complexity.

Includes formatting hints for the standard format. The syntax supports the inclusion of line and column number of a statement in the standard source code. These

could have been treated as metadata and added through an orthogonal extension.

Schema definition in DTD lacks strictness. The DTD is in general less expressive than WXS, but the DTD for this syntax is also less strict than it could be. Some examples are:

- Any text allowed as the symbol of a binary operation.
- Any combination of operations allowed, breaking operator precedence and left associativity.
- Any combination of method modifiers (private, static, final, abstract, etc.) allowed in every context.

Requires constructor name. Constructors in Java have no name, but this syntax requires the name of the class to be redundantly repeated as constructor name.

Does not define its namespace. Every XML syntax should use a namespace. This is especially important for the implementation of orthogonal extensions. DTD has no direct support for namespaces, so its use is a problem in this context as well.

JavaML by Evan Mamas [4] is another syntax with similar goals as Jezix, but different design principles. It is used in the IBM alphaWorks' Reengineering Toolkit for Java (ret4j). The syntax is explicitly based on the AST from Java compilers, which results in many of the following shortcomings.

Highly redundant construct for expressions. Even the simplest expressions, such as a literal number, have to be encoded with 19 elements nested inside each other. This follows from Java's concrete syntax EBNF that has to deal with the issue of grammar ambiguity. By virtue of this, the syntax enforces operator precedence and requires the explicit use of parentheses to override it.

Unnecessarily complex constructs. Some constructs, especially declarations (class, method, variable, etc.) and primary expressions (method invocation, member access, etc.) have unnecessarily complicated structure which results in processing overheads. This is an example of a class declaration:

```
<TypeDeclaration>
  <ClassDeclaration isPublic="True">
    <UnmodifiedClassDeclaration
      Identifier="DotTest" Extends="True">
      <Name Identifier="BaseClass"/>
      <Name Identifier="Interface1"/>
      <Name Identifier="Interface2"/>
      <ClassBody> ... </ClassBody>
    </UnmodifiedClassDeclaration>
  </ClassDeclaration>
</TypeDeclaration>
```

The same meaning is repeated by three nested elements: `TypeDeclaration`, `ClassDeclaration`, and `UnmodifiedClassDeclaration`. The `Extends` attribute is unexpectedly of boolean type and specifies whether the first in the list of `Name` elements that follow is the name of a superclass instead of an implemented interface.

Incomplete. The construct "*qualified super*" is not supported.

Schema definition in DTD lacks strictness. The most important example are the rules for defining accessibility levels. Each level has a separate attribute, for example `isPrivate` and `isPublic`. It is impossible in DTD (or in WXS) to enforce the setting of only one of them to true. This is an example where the very choice of a syntax rule prevents its formalization in the chosen schema language.

The following items are equivalent to the ones already explained for the Badros's syntax so they are just listed: **Language-specific syntax for type names; Type and variable names can occur in the same location; Requires constructor names; Does not define its namespace.**

B. Other syntaxes with similar goals

XQueryX [9] is an XML syntax, still under development, equivalent to the concrete syntax of the XQuery language (in [9] it is incorrectly called the abstract syntax). Apart from the fact that it models the concrete instead of the abstract syntax, two additional shortcomings were identified:

- XML attributes are completely avoided and simple elements are used instead. This enforces unnecessary syntactic constraints because subelements are required to appear in a specified order even though the order is semantically irrelevant.
- WXS's type hierarchy and the associated special attribute `xsi:type` attribute are used where the content model of an element depends on the value of an attribute. The special attribute always has the same name irrespective of the meaning it represents. It is also in a special namespace defined by WXS, and the `xsi` prefix will have to be bound to it whenever a query needs to access it.

XSLT [10] is the already mentioned infoset-processing language, which employs an XML syntax for its source code. The syntax is designed with the requirement on the source code to be readable and writable as pure text. This is achieved by employing a language-specific concrete syntax at the lower level of detail (e.g., for XPath expressions). This reduces the power of infoset-oriented source code manipulation and prevents the formalization of these parts of the syntax with a schema language.

o:XML [11] is an object-oriented programming language based on an XML syntax. It leverages XML in two ways: it introduces the "XML fragment" (a subtree of the infoset) as a basic data type and uses the infoset to hold the program description. Having XML fragments as data types makes it much easier than in conventional languages to manipulate complex data structures. o:XML allows XSLT-style (pattern-oriented) and DOM-style (object-oriented) infoset manipulation. The project employs the concept of orthogonal extensions to implement frameworks for embedded documentation and unit tests. What the language lacks from Jezix's point of view is a full representation of the AST in the source code infoset. For the same reasons as XSLT, it reverts to a language-specific syntax at the lower level of detail.

Superx++ [12] is also an object-oriented language utilizing an XML syntax for source code. Its interesting

property is that it models the working memory as an infoset, too, which should allow easy memory inspection (querying), object serialization etc. It employs a language-specific syntax for expressions. Instead of XPath, a proprietary language is used that is less expressive and has a different syntax. This is a shortcoming for a developer already familiar with XPath. The current syntax formalization in WXS is very lax, allowing arbitrary content for many elements.

XML-Encoded Source or XES [13] is an XML syntax for the encoding of the source code of multiple programming languages. Its generality is based on the similarity of programming languages' concrete syntaxes. Since different languages use the same syntactic constructs to denote different concepts, the meaning of constructs in XES is language-dependent. The syntax is formalized in WXS and, in spite of the claims to generality, it is tightly bound to the Java's concrete syntax.

Source Markup Language, srcML [5] uses XML tags to mark up the source code of the C++ programming language. Every detail of the standard source code is preserved and the tags are used to overlay a partial concrete syntax tree on top it. The tree is partial because, for example, it leaves expressions unparsed. This approach allows some benefits of XML to be leveraged without losing any aspects of the standard source code.

VII. Syntax comparison

A. The proposed syntax

The most involving part of the Jezix project was the development of an infoset syntax for the Java programming language and its formalization in WXS. Many of the features of the syntax are described implicitly in this paper as the lack of the shortcomings identified above in the other syntaxes. Some key features are also evident from the XQuery code presented in this section.

All the features of WXS were carefully studied in order to determine how they can be employed to maximize the schema definition's strictness. In particular, a feature of WXS that goes beyond the expressiveness of a CFG, uniqueness constraints, was employed to enforce the uniqueness of some identifiers, such as the names of attributes and method parameters. Another important addition to the strictness of the standard syntax's EBNF are the constraints on the combining of expressions according to their result type. For example, the schema allows addition, but not a reference type cast, to be an operand of multiplication. In the EBNF it is the other way round, due to operator precedence.

In order to demonstrate the reasons why Jezix's syntax is superior to the existing alternatives, several querying tasks were defined and their solutions developed in XQuery for the Jezix syntax and for two of the most similar and well-designed publicly available syntaxes. The tasks were defined to be both realistic (have a practical relevance) and focus on the identified deficiencies of the alternatives. A query that would have a simpler solution for an alternative syntax was not found. The only identified examples are queries where constructs are selected by how they look when written down in the standard Java source code format. This kind of query is irrelevant in the context where standard source code is replaced by XML. The solutions to two of the tasks used for comparison are presented here.

B. Query task 1

Task: find all occurrences of expressions satisfying the following pattern:

```
_ && _ && _ && ( _ || _ ) && ...
```

that is, a *conditional and* expression with four or more operands where the fourth operand is a *conditional or* expression with exactly two operands. For each matching expression, return this fourth operand. For example, for the equivalent of this source code:

```
boolean a,b,c,d,e,f,g,h,i,j,k,l,m,n,o;
...
a = b && c && d && ( e || f );
a = b && c && d && ( g || h ) && i;
a = b && c && d && ( j || k || l ) && m;
a = b && c && d && n | o;
```

this is what should be returned in case of the Jezix syntax:

```
<and-operations>
  <op4>
    <conditional-or>
      <variable-access name="e"/>
      <variable-access name="f"/>
    </conditional-or>
    <conditional-or>
      <variable-access name="g"/>
      <variable-access name="h"/>
    </conditional-or>
  </op4>
</and-operations>
```

Solution for Jezix

```
declare default element namespace
  "http://tel.fer.hr/jezix/java";
<and-operations> {
  for $op4 in document( "Task1.jxj" )
    //conditional-and/*[4]
    /self::conditional-or[ count(*)=2 ]
  return <op4>{$op4}</op4>
} </and-operations>
```

The text of the query corresponds closely with the description of the task. The main work is performed by the XPath expression that selects all *conditional and* expressions in the first step and then the fourth operand of each of them. In the third step it selects only those operands that are *conditional or* expressions and have exactly two operands. The development of this query required about ten minutes.

Solution for Badros's JavaML

```
<and-operations> {
  for $op4 in document( "Task1.jmlb" )
    //binary-expr[ @op="&amp;&amp;" ]
    [not(exists(binary-expr[@op="&amp;&amp;"]))]
    /parent::binary-expr[ @op="&amp;&amp;" ]
    /parent::binary-expr[ @op="&amp;&amp;" ]
    /*[2]/self::paren/binary-expr[ @op="||" ]
    [ not(exists(binary-expr[@op="||"])) ]
  return <op4>{$op4}</op4>
```

```
} </and-operations>
```

This query is quite a bit more complicated and much harder to understand. This is primarily due to the use of binary operations. The query has to descend through the nesting levels to reach the fourth operand and has to do additional checking at every level. It also has to deal with parentheses. The development of this query required about two hours.

Solution for Mamas's JavaML

```
<and-operations> {
  for $op4 in document( "Task1.jmlm" )
    //ConditionalAndExpression/
    InclusiveOrExpression[4]
  let $parExp := $op4/*/*/*/*/*/*/*/*/*
  /UnaryExpressionNotPlusMinus/PostfixExpression
  /*/PrimaryPrefix/Expression
  where
    count( $op4/* ) = 1 and
    count( $op4/*/* ) = 1 and
    count( $op4/*/*/* ) = 1 and
    count( $op4/*/*/*/* ) = 1 and
    count( $op4/*/*/*/*/* ) = 1 and
    count( $op4/*/*/*/*/*/* ) = 1 and
    count( $op4/*/*/*/*/*/*/* ) = 1 and
    count( $op4/*/*/*/*/*/*/*/* ) = 1 and
    count( $op4/*/*/*/*/*/*/*/*/* ) = 1 and
    exists( $parExp ) and
    count( $parExp/* ) = 1 and
    count( $parExp/*/* ) = 1 and
    count( $parExp/*/*/* ) = 2
  return <op4>{$parExp/*/*}</op4>
} </and-operations>
```

In this case the query is very long and extremely redundant. This is due to the excessive nesting levels required for every expression in this syntax. The element names are also unnecessarily long, redundantly repeating the word *Expression*. This query took about three hours to develop.

C. Query task 2

Task: find all classes/interfaces mentioned in the source code document and return a sorted list of them with separated package name, containing types, and local type name. For example, for the equivalent of this Java code (note that `java.util.Arrays` does not refer to a type):

```
package jezix.srctest;
class TypeTest {
  TypeTest() {
    Class c = java.util.List.class;
    Nested java = new Nested();
    int j = java.util.Arrays.i;
    int k =
      jezix.srctest.TypeTest.Nested.n;
  }
  static class Nested {
    TypeTest util;
    static int n;
  }
  TypeTest Arrays;
  int i;
}
```

this is what should be returned:

```
<reference-types>
  <reference-type package="java.lang"
    name="Class"/>
  <reference-type package="java.util"
    name="List"/>
  <reference-type package="jezix.srctest"
    enclosing="TypeTest" name="Nested"/>
</reference-types>
```

Solution for Jezix

```
declare default element namespace
  "http://tel.fer.hr/jezix/java";
<reference-types> {
  for $type in document("Task2.jxj")
    //reference-type
  let
    $enclosing := string-join(
      $type/enclosing-type/@name,"."),
    $localName := $type/@name
  order by $type/@package,$enclosing,$localName
  return <reference-type
    package = "{$type/@package}"
    enclosing = "{$enclosing}"
    type = "{$localName}"/>
} </reference-types>
```

The main work (finding the required elements) is done by this trivially simple XPath expression:

```
document("Task2.jxj")//reference-type
```

The rest of the query deals with output formatting. The query required about half an hour, spent almost entirely in the search for a concise solution for output formatting.

Solutions for the JavaMLs

The solutions for both JavaMLs are quite similar so they are commented on together. The primary reason for the huge complexity of these solutions to a very simple task is that the JavaML syntaxes prescribe many different ways of referring to a type, depending on the context. Much of the complexity is also due to the use of keywords where identifiers are normally found and to the use of unparsed type names. In addition to being so complex, the queries do not fully satisfy the requirements because it is impossible to determine which parts of a composite type name refer to a package, and which to enclosing types. This is why instead of the two attributes, `package` and `enclosing`, there is only one: `package-and-enclosing`. The queries took about seven hours each to develop.

Function common to both solutions

```
declare function local:parseName(
$typeName as xs:string ) as xs:string* {
  let $names := tokenize( $typeName, "\." )
  return (string-join(
    $names[ not(position()=last()) ], "." ),
    $names[ last() ] )
}
```

Solution for Badros's JavaML

```
declare function local:isKeyword(
$name as xs:string ) as xs:boolean {
  $name="void" or $name="byte" or $name="char" or
  $name="short" or $name="int" or $name="long" or
  $name="float" or $name="double"
}
declare function local:isInScope(
$varRef as element() ) as xs:boolean {
  let $declsInScope := $varRef/ancestor::*/field |
    $varRef/ancestor::catch/formal-argument |
    $varRef/ancestor::*/formal-arguments/
      formal-argument |
    $varRef/ancestor::*/loop/init/local-variable |
```

```
    ( $varRef/preceding::* intersect
      ( $varRef/ancestor::*/local-variable ) )
  return exists(
    $declsInScope[ @name = $varRef/@name ] )
}
<reference-types> {
  let $root := document( "Task2.jmlb" )/*
  for $typeName in
    $root//superclass/@name |
    $root//implement/@interface |
    $root//throws/@exception |
    $root//import[ not(ends-with(@module,"*")) ]
      /@module |
    $root//type[ not(ends-with(@name,"[]")) ]
      [ not( @primitive = "true" ) ]/@name |
    $root//var-ref
      [ not( ends-with( @name, "[]" ) ) ]
      [ not( local:isKeyword( @name ) ) ]
      [ not( local:isInScope( . ) ) ]/@name
  let $parsedName := local:parseName( $typeName )
  order by $parsedName[1], $parsedName[2]
  return <reference-type
    package-and-enclosing="{ $parsedName[1] }"
    type="{ $parsedName[2] }" />
} </reference-types>
```

Solution for Mamas's JavaML

```
declare function local:isInScope(
$name as element() ) as xs:boolean {
  let $declsInScope :=
    $name/ancestor::Block[
      exists( parent::TryStatement ) ]
      /preceding-sibling::FormalParameter[1]
      /VariableDeclaratorId |
    ( $name/ancestor::*/FieldDeclaration |
      $name/ancestor::MethodDeclaration
        /MethodDeclarator |
      $name/ancestor::ConstructorDeclaration |
      $name/ancestor::ForStatement/ForInit
        /LocalVariableDeclaration |
      $name/preceding::* intersect
        $name/ancestor::*/LocalVariableDeclaration
    )/*/VariableDeclaratorId
  return exists( $declsInScope[
    @Identifier = $name/@Identifier ] )
}
let $root := document( "Task2.jmlm" )/*
return <reference-types> {
  for $nameEl in $root//Name[
    exists(
      parent::ConstructorDeclaration |
      parent::MethodDeclaration |
      parent::UnmodifiedClassDeclaration |
      parent::AllocationExpression ) or
    exists( parent::ImportDeclaration ) and
      not( ends-with( @Identifier, "*" ) ) or
    exists( parent::Type ) and
      not( @Identifier = "void" ) or
    exists( parent::PrimaryPrefix ) and
      not( local:isInScope( . ) ) ]
  let $parsedName :=
    local:parseName( $nameEl/@Identifier )
  order by $parsedName[1], $parsedName[2]
  return <reference-type
    package-and-enclosing="{ $parsedName[1] }"
    type="{ $parsedName[2] }" />
} </reference-types>
```

D. Comparison summary

Table I summarizes the results of comparison presented above:

- **Time** is the approximate time in minutes required to develop the solution.
- **Length** is the length of the XQuery code in characters (optional whitespace was not counted).

TABLE I
SUMMARIZED RESULTS OF COMPARISON

Solution	Time	Length
Jezix 1	10	215
Badros 1	120	323
Mamas 1	180	622
Jezix 2	30	377
Badros 2	420	1362
Mamas 2	420	1373

VIII. CONCLUSION

The key benefits that the application of XML to programming language source code can bring are better support for task automation and for orthogonal extensions. In order to fully leverage these potentials, the document's infoset should correspond with the AST of the source code. There are several XML syntaxes for the Java programming language that use this approach, but their design suffers from various deficiencies. Their main origin is in the fact that they are influenced by Java's concrete syntax, which makes them describe some meanings in a complicated, roundabout way. This can have very negative effects on the form of queries that have to access these meanings. As a part of the Jezix project, a new syntax was designed that avoids these deficiencies and its superiority was demonstrated by comparing it with two of the most relevant publicly available syntaxes. The comparison was made by specifying several querying tasks and developing a solution in XQuery for the three syntaxes. Queries for the Jezix syntax are shorter, more readable and required significantly less effort to develop.

REFERENCES

[1] H. Simic and M. Topolnik, "Prospects of encoding Java in XML," in *Proceedings of ConTEL 2003*, June 2003, pp. 635–641.

[2] E. Armstrong, "Encoding source code in XML, a strategic analysis," http://www.treelight.com/software/encodingSource.html, 2000.

[3] G. J. Badros, "JavaML: a markup language for Java source code," in *Proceedings of the 9th international World Wide Web conference on Computer networks*. North-Holland Publishing Co., 2000, pp. 159–177.

[4] Y. Zou and K. Kontogiannis, "Towards a portable XML-based source code representation," in *Proceedings of XML Technologies and Software Engineering (XSE2001)*, 2001.

[5] M. L. Collard, J. I. Maletic, and A. Marcus, "Source code files as structured documents," in *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, 2002.

[6] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, pp. 137–167, 1959.

[7] "Extensible Markup Language (XML) Version 1.1," http://www.w3.org/TR/2004/REC-xml11-20040204/, World Wide Web Consortium, 2004.

[8] "XQuery 1.0 and XPath 2.0 Data Model," http://www.w3.org/TR/xpath-datamodel/, World Wide Web Consortium.

[9] "XML Syntax for XQuery 1.0 (XQueryX)," http://www.w3.org/TR/2003/WD-xqueryx-20031219, World Wide Web Consortium, 2003.

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

[11] M. Klang, "XML and the art of code maintenance," in *Proceedings of Extreme Markup Languages 2003*, 2003.

[12] "Superx++, XML-based OO Programming," http://xplusplus.sourceforge.net.

[13] "XML-encoded Source," http://www.euclideanspace.com/software/language/xes/index.htm.