An XML-based Framework for Language Neutral Program Representation and Generic Analysis

For

Prof. Frank Tompa

CS 741: Non-Traditional Databases

By

Raihan Al-Ekram

Student Id: 20082523

rekram@swen.uwaterloo.ca

April 19, 2004

School of Computer Science, University of Waterloo

Table of Contents

Abstract

XML applications are becoming increasingly popular to define constrained XML data for some special application areas. In pursuit there is a growing momentum of activities related to XML representation of source code in the area of program comprehension and software re-engineering. The source and the artifacts extracted from a program are necessarily structured information that needs to be stored and exchanged among different tools. This makes XML to be a natural choice to be used as the external representation formats for program representations. But most of the proposed XML applications represent source code at a very fine level of granularity. As such, we propose XML applications for program representation at a higher level or granularity. By combining the proposed XML applications we present a framework for language neutral program representation at different levels of abstractions in order to facilitate the development of generic source code analysis tools.

1 Introduction

The Extensible Markup Language (XML) [1], a World Wide Web Consortium (W3C) [2] standard, has been widely accepted for storing and exchanging structured documents. Many XML sublanguages have been developed to define constrained XML data for some special application areas, often by means of a Document Type Declaration (DTD) or XML Schema definition [3]. For examples Mathematical Markup Language (MathML) [4] is defined for mathematical symbols, equations and formulae for electronic interchange or Voice Extensible Markup Language (VoiceXML) [5] developed for voice markup and telephony call control to enable access to the Web using spoken interaction etc. Such markup languages are becoming increasingly popular because of the features of XML – it is simple, easy to understand, extensible, searchable, open standard, interoperable and there is a wide range of tool support for creation, manipulation and transformation of XML documents automatically.

In pursuit there is a growing momentum of activities related to XML representation of source code in the area of program comprehension and software re-engineering. The source and the artifacts extracted from a program are necessarily structured information that needs to be stored and exchanged among different tools. Which makes XML to be a natural choice to be used as the external representation formats for program representations.

In this report we survey the existing tools and techniques for representing program artifacts in terms of XML. Our research shows that various XML applications namely JavaML, CppML, OOML, srcML, PLIXML, PascalML, FortranML etc. have been proposed to represent the source code of different languages in terms of their Abstract Syntax Tree. But very little have been done to XMLize the program representations at higher levels of abstractions. As such, we propose XML applications for higher-level program representations like Intra-Procedural and Inter-Procedural Flow and Dependence Graphs. We also present a framework for language neutral program representation at different levels of abstractions in order to facilitate the development of generic source code analysis tools.

The rest of the report is organized as follows. Section 2 provides a brief description of the program representation formalisms at different levels of granularity with their intended usage. Section 3 surveys the existing XML applications and tools for program representation. Section 4 presents the XML-based program representation framework. Section 5 discusses the proposed XML sublanguages for the intermediate representations. Finally Section 6 concludes the report with directions to future work.

2 Program Representation Formalisms

While the source code is the original artifact of a software system, it is written and stored in ASCII plain text format and is not suitable to be used directly for sophisticated program analysis. More structured and abstract representations are needed to enable algorithmic analysis and manipulation of programs. So the source code needs to be represented at different levels of granularity.

2.1 Syntax Trees

A Parse Tree [6] is a hierarchical graphical representation of the derivations of the source code from its grammar. The interior modes of the tree represent the non-terminals and the leaves terminal symbols of the grammar. An Abstract Syntax Tree (AST) [6] is a more economical representation of the source code abstracting out the redundant grammar productions from the parse tree. The source sentence can be reconstructed from a Depth-first inorder traversal of the tree nodes. The syntax trees are basic source code representations at the finest level of granularity. These data structures are used by compilers to analyze and transform source code entities. They also serve as the primary source for constructing other higher-level representations. The syntax trees are the abstraction of the source code in terms of the language grammar and hence are heavily dependent on the programming language.

2.2 Intra-Procedural Flow and Dependence Graphs

The next higher-level abstractions of source code are the flow and dependence graphs. These graph data structures are abstractions in terms of control flow and data flow of the program and can be represented in a programming language independent way. The intra-procedural graphs are for representing a single subroutine, procedure or function within a program.

A Control Flow Graph (CFG) [7] provides a normalized view of all possible flow of execution paths of a program. A CFG is a rooted directed graph where the nodes represent basic blocks and arcs represent possible immediate transfer of control from one basic block to another. A basic block is a sequence of consecutive instructions that are executed from start to finish without the possibility of branching except at the end. The CFG representation is extensively used for data flow analysis, code optimization and testing.

A Program Dependence Graph (PDG) [8] is a combined explicit representation of both control and data dependences in a program. The PDG is a directed graph whose nodes are connected by several kinds of arcs. The nodes represent statements and predicate

expressions and the arcs represent control and data dependence. The control dependence arcs are labeled either True or the truth-value of the predicate. The data dependence arcs, labeled by the variable name, indicate possible flow of data values between nodes where the source node defines the variable and the destination node may use the data value of it. The PDG is used for code optimization, parallelism detection, loop fusion, clone detection etc. It is also used for performing slicing for maintenance and re-engineering purpose.

2.3 Inter-Procedural Flow and Dependence Graphs

Understanding the flow of information within a single subroutine is not sufficient for optimization or analysis of the complete system, which is comprised of many procedures and files.

The System Dependence Graph (SDG) [9] is an extension to PDG for programs with multiple procedures. The SDG is constructed by connecting the individual PDG of each procedure with some additional arc types. These arcs correspond to procedure calls, parameters passed and return values.

Call Graphs [10] [11] are program abstractions used in traditional inter-procedural analysis. It's a graphical representation of the caller or callee relationships among the procedures of a program, where the nodes indicate the procedures and the arcs indicate the calls. The nodes and arcs in a call graph may contain labels to include attributes, e.g. line number of the call or file name of the procedure. There can optionally be new entities in the graph, e.g. abstract data types and their usage relationships in addition to the procedure calls. From the basic graph higher level call graph can be constructed to show relationships among files, modules or architectural entities instead of procedures. An extention to call graph is the Program Summary Graph (PSG) that takes into account the reference parameters and global variables at the individual call points. Other than inter-procedural data flow analysis for optimization, call graphs are used for design recovery, architecture extraction or other reverse engineering analysis.

3 Program Representation using XML

Simic and Tolnik [12] in there work explores the prospects of representing source code using XML in place of classical palin text format. They demonstrate that an XML grammar will improve the code structure, formatting, querying possibilities and will allow making orthogonal extensions to code for annotations, revision control, access control and documentation.

3.1 Java Markup Language (JavaML)

Badros [13] proposes an XML application, namely the Java Markup Language (JavaML), to represent Java source code in terms of its AST in order to facilitate tools to peroform software engineering anlysis by leveraging the abundance of XML tools and technologies. The JavaML is defined by an XML DTD, where the elements represent the structure of the AST and most if the source code information are sotred as attributes on the element

tags. Figure 1(a) shows a sample Java code snippet and Figure 1(b) presents its corresponding JavaML representation.

```
public class FirstApplet extends Applet{
    public void paint (Graphics g){
    }
}
```

Figure 1(a): Sample Java code snippet

```
<java-source-program name="FirstApplet.java">
    <class name="FirstApplet" visibility="public">
        <superclass class="Applet"/>
        <method name="paint" visibility="public" id="meth-15">
            <type name="void" primitive="true"/>
            <formal-arguments>
                <formal-argument name="g" id="frmarg-13">
                    <type name="Graphics"/></formal-argument>
            </formal-arguments>
            <block>
            </block>
        </method>
    </class>
</java-source-program>
```

Figure 1(b): JavaML representation of the code snippet

In addition to representing the mere syntax of the source code, JavaML stores some semantic information as well. For example IDREF tags are used to refer to the declaration of a variable from the locations where it is used, which can be used for scope resolution or getting the type of a variable easily.

To demonstrate the concept, the author built a converter on top of the IBM Jikes Java complier framework to translate textual Java source code to JavaML and an XSLT stylesheet to transform JavaML back to textual form. Since JavaML represents the complete AST of the source code, preservation of syntactic details of every programming construcs may cause its size to explode. On the other hand since the AST abstracts out the comments and much of the formatting information, the original source code document cannot be regenerated from the JavaML.

3.2 Source Code Markup Language (srcML)

Collard et. al. [14] describes a technique to convert the C++ source code into an XML representation, namely the Source Code Markup Language (scrML), in order to use it for static extraction of facts. This is a markup technique where the tags are superimposed on the source code keeping the original code as it is. The markups explicitly describe the internal structure of the code preserving the comments and the formatting information. The srcML is defined by an XML DTD. Figure 2(a) shows a sample C++ code snippet and Figure 2(b) presents its corresponding srcML representation.

The srcML does not directly represent the AST, hence it does not require complete parsing of the source code to generate the complete AST. It uses a multi-pass multi-stage

prasing technique with partial grammar specification to parse and tag from higer level entities to their constituent lower level entities. This enables controlling the parsing upto the desired level of interest depending on the focus of the analysis to be performed on the source code. This appraoch of parsing and marking up only the selected constructs of interest, while leaveing others as it is, is know as island parsing.

```
// swap two numbers
if(a>b)
{
    t = a;
    a = b;
    b = t;
}
```

Figure 2(a): Sample C++ code snippet

```
<unit>
<comment type="line">// swap two numbers</comment>
<if>if<condition>(<expr><name>a</name>&gt;<name>b</name></expr>)</condit
ion><then>
<block>{
    <expr_stmt><expr><name>t</name> = <name>a</name></expr>;</expr_stmt>
    <expr_stmt><expr><name>a</name> = <name>b</name></expr>;</expr_stmt>
    <expr_stmt><expr><name>b</name> = <name>t</name></expr>;</expr_stmt>
}</block></then></if>
</unit>
```

Figure 2(b): srcML representation of the code snippet

srcML translator is constracted from ANLTR [26] pred-LL(k) grammar specification and a context stack. In the translator both pre and post actions are attached to the grammar specifications to markup the source code with appropriate start and end tags using the context stack.

3.3 XMLizer

McArthur et. al. [15] presents the XMLizer tool to transform source code of several programming languages into their respective XML representations in order to facilitate re-engineering and migration. The PL/IX Markup Language (PLIXML), the Pascal Markup Language (PascalML) and the Java Markup Language (JavaML) are defined with their own DTDs to represent PL/IX, Pascal and Java source code respectively. XMLizer representation of the source code is essentially the XML representation of the AST. In order to prevent the size of the representation from exploding, the tool uses a multi-weight parser that can generate ASTs of variable granularity by allowing designated syntactic construct to remain unparsed. This technique can also be used to preserve comments by attaching them to unparsed constructs. The XMLizer is also developed by modifying the ANLTR translator [26].

3.4 Agile Parsing

Cordy [16] in his paper describes a method for extending and generalizing the partial markup idea of island or multi-weight parsing using the agile parsing technique of the TXL [25]. This approach selectively marks up only those AST nodes in the source that are relevant to a particular task. Using grammar overrides and utilizing TXL's ordered ambiguity resolution a very precise form of constructs can be specified for markup, without any modification in the base grammar. Figure 3 shows a Java code snippet selectively marked for declaration and statement types.

```
<method_declaration>private boolean doKeyword(Segment line,int i,char c)
{
  <variable_declaration>int i1=i+1;</variable_declaration>
  <variable_declaration>int len=i-lastKeyword; </variable_declaration>
  <variable_declaration>byte id =
       keywords.lookup(line,lastKeyword,len);</variable_declaration>
  <if_statement>if(id!=Token.NULL)
  {
    <if_statement>if(lastKeyword!=lastOffset)
      <expression_statement>addToken(lastKeyword-lastOffset,Token.NULL);
        </expression_statement>
    </if_statement>
    <expression_statement>addToken(len,id);</expression_statement>
    <expression_statement>lastOffset=i;</expression_statement>
  }
  </if_statement>
  <expression_statement>lastKeyword=i1;</expression_statement>
  <return_statement>return false;</return_statement>
}
</method_declaration>
```

Figure 3: Selective AST markup in a Java code snippet

This parsing technique is programming language independent and has been used with grammars for Java, C++, COBOL, PL/I and RPG. There are no DTDs defined for the markups, the non-terminal symbols of the particular grammar are used as the markup tags.

3.5 Generic Object-Oriented Domain Model

As part of the ISME framework Mamas and Kontogiannis [17] defines the Java Markup Language (JavaML) and the C++ Markup Language (CppML), XML sublanguages declared using DTDs, for representing Java and C++ source code in terms of their ASTs. A generator for JavaML from Java source code is developed using JavaCC parser generator [27]. On the contrary the CppML generator is developed using the CodeStore API of IBM VisualAge C++ complier.

The models of the different object-oriented languages share many common features, making it possible to develop a generalized superset domain model for the object-oriented language paradigm. Based on this idea Object-Oriented Markup Language (OOML) is developed as an aggregated and more generic representation for all object-oriented laguages. OOML representations can be generated by defining mappings from JavaML and CppML representations instead of directly manupilating the souce code.

Both Java and C++ represents objects by using the concept of classes, class variables and class methods. Figure 4(a) shows the declaration of a class in OOML derived from the correspoding JavaML declaration in Figure 4(b). Figure 4(c) gives the production rules used for the mapping

```
<!ELEMENT Class (VariableDeclaration*,Method*)>
<!ATTLIST Class Identifier CDATA>
```

Figure 4(a): OOML representation of a Class

```
<!ELEMENT ClassDeclaration (UnmodifiedClassDeclaration)>
<!ELEMENT UnmodifiedClassDeclaration (Name,ClassBody)>
<!ATTLIST UnmodifiedClassDeclaration Identifier CDATA>
<!ELEMENT ClassBody (FieldDeclaration|MethodDeclaration)*>
```

Figure 4(b): JavaML representation of a Class

```
ClassDeclaration                  →    Class
UnmodifiedClassDeclaration.Name   →    Class.Identifier
FeildDeclaration                  →    VariableDeclaration
MethodDeclaration                 →    Method
```

Figure 4(c): Mapping rules for translation from JavaML to OOML

3.6 Generic Procedural Domain Model

Zou and Kontogiannis [18] [19] in their work propose a generic domain model for representing the procedural languages in XML. In first step the domain models of the AST representations for the individual procedural languages are defined using XML DTDs. One DTD is defined for each of the C, Pascal and Fortran languages, namely the CML, the PascalML and the FortranML. In second step the domain models are enhanced with information such as unique identifier, linkage and analysis information. Finally the domain models are generalized by identifying common language structures found in the group of procedural languages such as files, functions, data types, variables, expressions, statements etc. and mapping them to their equivalents in the generic domain model, which can be called the ProcML. Table 1 shows an example mapping from the Fortran Domain Model to the Generic Domain Model.

3.7 Graph Exchange Language (GXL)

Graph Exchange Language (GXL) [20] [21] is an XML based language for describing graphs. It evolved from unification of graph description languages like GRAph eXchange format (GraX), Tgraphs, Tuple Attribute language (TA) and PROGRES. The conceptual data model of GXL is a typed, attrubuted and directed graph. GXL describes both the instance data and the scheme of the data in the same format.

Unlike the other representations discussed, GXL was not originally intended to represent the source code. But the higher lever program representation formalisms being graphs make GXL a natural choice for their representation. Figure 5(a) shows a simple Call Graph and Figure 5(b) presents its corresponding GXL representation.

Table 1: Generalization of the Fortran Domain Model

Fortran Domain Model	**Generic Domain Model**
Structure-Statement	Structure
Record-Statement	Struct Variable Declaration
Common-Statement	Global Variable Declaration
Programs	Program
Executable Program	File
Program Unit	Function-Def
Type-Statement	Declaration
Read-Statement	Function-Call
Call-Statement	Function-Call
Indexable-Name/function-Params	Function-call
Character-Statement	String
Equivalence-Statement	Union-Struct
Intrinsic-Statement	Function-Pointer

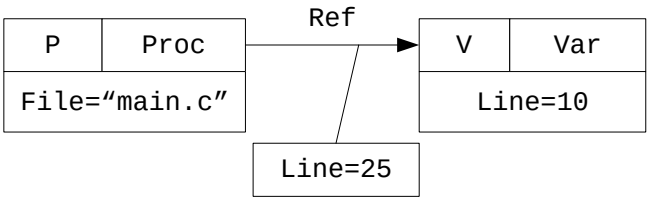

Figure 5(a): A Call Graph

```
<GXL>
    <node id= "P" type= "Proc">
        <attr name= "File" value= "main.c"/>
    </node>
    <node id= "V" type= "Var">
        <attr name= "Line" value= "10"/>
    </node>
    <edge begin= "P" end= "V" type= "Ref">
        <attr name= "Line" value= "25"/>
    </edge>
</GXL>
```

Figure 5(b): GXL representation of the Call Graph

4 A Framework for Language Neutral Program Representation and Generic Analysis

While the AST level representations are useful for some type of analysis, they are not usable for sophisticated higher-level analysis. Moreover, in order to perform program analysis in a language independent way and to build generic analysis tools, language neutral representations are required at different levels of granularity of the source program. But the existing work lacks in defining program representations at a level higher

than the AST in terms of XML sub-languages. In this section we propose an XML-based multi-layered framework to represent program artifacts at different levels of abstractions in a language neutral way. We also demonstrate the usage of the framework for building generic program analysis tools. The different layers of the framework correspond to program representations formalism like AST, CFG, PDG, SDG, and Call Graphs etc. in XML format. Figure 6 presents the system architecture of the proposed framework.

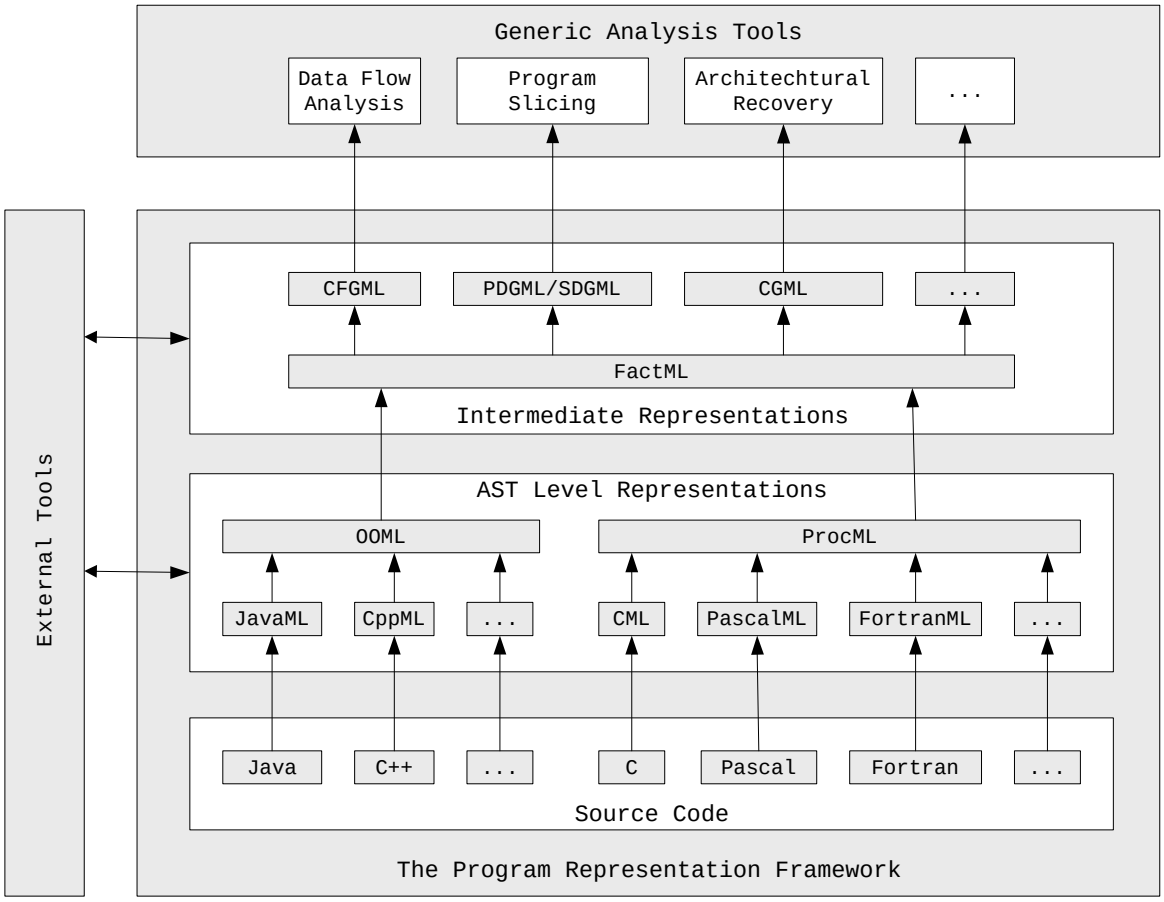

Figure 6: System architecture of the program representation framework

4.1 Source Code

Layer 0 of the framework is the original source text of the program to be analyzed.

4.2 AST Representations

Layer 1 is the first level abstraction of the source code in terms of the AST of the program. We choose the AST representations proposed by Mamas and Zou to fit in this layer, since these representations include generic representations for object-oriented and procedural paradigm based on a generalized domain model. This layer consists of two sub-layers. Layer 1.1 consists of the ASTs representations in programming language specific markup languages – JavaML, CppML, CML, PascalML, FortranML etc. Layer

1.2 consists of the AST representations derived from the generic domain model of the language paradigms – OOML and ProcML.

4.3 Intermediate Representations

Layer 2 is the next level of abstraction of the program in terms of the different intra-procedural and inter-procedural graphs. This layer is also consisted of two sub-layers. Layer 2.1 represents the basic facts of a program in the FactML format. The facts of a program are the building block constructs of the program and the basic relationships among them. These are used by the representations above this layer as basic units of composition. These constructs are statements, variables, data types and functions etc. and the basic relationships are the uses or definitions of variables and calls to functions. Layer 2.2 are the representations for intra-procedural and the inter-procedural dependence and flow graphs of the program expressed as CFGML, PDGML, SDGML and CGML defined to represent CFG, PDG, SDG and Call graphs respectively.

4.4 Generic Analysis Tools

Various program analysis tools can be written on top of the representation framework. These tools work on language neutral program representations and enable the development of a single tool to perform a particular type of analysis on the source code written in any programming language. For example a generic data flow analysis tool can be written to work on the CFGML or a single slicing tool can be written to use the PDGML to perform program slicing on source code of any language.

4.5 External Tools

All the representations in the proposed framework are XML and hence can be easily transformed to any other formats using XSLT stylesheets [25] or XQuery [26] technology in order to enable exporting of data to an external tool. If the external tool also uses an XML representation for its data then it is straightforward to import the data using the same techniques. However if the external tool does not use XML representations, additional mapping tools needs to be developed to map the external formats to the internal XML representations.

5 Intermediate Representations

The XML-sublanguages for AST level representations of Layer 1 are already well studied. As part of our work we define XML-sublanguages to represent intermediate representations of Layer 2 that is necessary for generic program analysis.

5.1 FactML

The facts of a program consist of information about the basic constructs of th program and the relationships among them. The basic constructs can be data types, variables, statements and functions. The information relevant to a variable is its name and its scope, i.e., if it is local, global or external. A variable construct is associated with a type construct

to indicate its data type. A variable construct can also be associated with a statement construct in two possible ways – a variable is declared in a statement and a variable can be used or redefined in one or more statements. The FacML DTD given in Appendix A encodes such information about all the constructs in a program.

5.2 CFGML

A CFG is consisted of many basic blocks and the flow of controls among them. The basic blocks in turn consist of a sequence of statements. The CFGML in Appendix B describes its structure in an XML DTD. The basic constructs used in the CFGML are referred from the DTD defined for facts using XLink [27] instead of redefining them in it.

5.3 PDGML

The PDGML defined in Appendix C describes the encoding of the PDG in an XML DTD. There can be two kinds of elements in a PDG – the program constructs and the dependences among them. The dependences are of two kinds. The control flow dependences are between the statements and the dummy nodes and can be labeled either True or False. Whereas the data flow dependences are between the statements and the dummy nodes and are also associated with variables that cause the dependency. The basic constructs used in the PDGML are also referred from the DTD defined for facts using XLink.

6 Conclusion

In this report we presented a framework for language neutral program representation. The framework is based on a multi-layered abstraction of source code facts represented using several XML sublanguages. The framework adopts the existing XML applications for source code representation and defines new applications to represent higher-level program abstractions.

The framework presented in this report suggests the particular AST representations in the Layer 1 and the tools to be used to generate these representations from the source code. In layer 2 only the intermediate representations are defined, a set of representation transformers needs to be developed to translate from Layer 1 representations to that of the Layer 2. Since the representations of both the layers are XML, the transformers can be built using XSLT stylesheets, XQuery technology or by programmatically manipulating the DOM [28] tree of the source XML. Finally the usefulness and effectiveness of the framework is to be validated by developing the generic program analysis tools on top of the framework and using them for the intended program analysis and comprehension task.

Appendix A: FactML DTD

```
<!-- facts.dtd 0.1                                              -->
<!-- A DTD for representation of facts as an XML document -->

<!ELEMENT Facts (Statements?, Types?, Variables?, Functions?,
                  UseDefs?, Calls?)>
```

```
<!ATTLIST Facts program CDATA>

<!ELEMENT Statements (Statement*)>
<!ELEMENT Statement EMPTY>
<!ATTLIST Statement
         id ID #REQUIRED
         lineno CDATA #REQUIRED
         tag CDATA
         function IDREF>

<!ELEMENT Types (Type*)>
<!ELEMENT Type EMPTY>
<!ATTLIST Type
         id ID #REQUIRED
         name CDATA #REQUIRED
         category (Primitive|Class|Struct|Pointer) “Primitive”>

<!ELEMENT Variables (Variable*)>
<!ELEMENT Variable EMPTY>
<!ATTLIST Variable
         id ID #REQUIRED
         name CDATA #REQUIRED
         scope (Local|Global|Parameter|External) “Local”
         declared IDREF
         type IDREF>

<!ELEMENT UseDefs (UseDef*)>
<!ELEMENT UseDef EMPTY>
<!ATTLIST UseDef
      id ID #REQUIRED
      category (Use|Def) “Use”
      statement IDREF #REQUIRED
      variable IDREF #REQUIRED>

<!ELEMENT Functions (Function*)>
<!ELEMENT Function EMPTY>
<!ATTLIST Function
         id ID #REQUIRED
         name CDATA #REQUIRED
         scope (Internal|External) “Internal”
         signature CDATA
         declared IDREF>

<!ELEMENT Calls (Call*)>
<!ELEMENT Call EMPTY>
<!ATTLIST Call
         id ID #REQUIRED
         statement IDREF #REQUIRED
         function IDREF #REQUIRED>
```

Appendix B: CFGML DTD

```
<!-- cfg.dtd 0.1                                              -->
<!-- A DTD for representation of a CFG as an XML document  -->

<!ELEMENT CFG (Blocks, Flows)>
```

```
<!ATTLIST CFG
   xmlns:xlink CDATA #FIXED “http://www.w3.org/1999/xlink”
         program CDATA
   scope CDATA>

<!ELEMENT Blocks (Block*)>

<!ELEMENT Block (Statement*)>
<!ATTLIST Block
         id ID #REQUIRED
         label CDATA #REQUIRED

<!ELEMENT Statement EMPTY>
<!ATTLIST Statement
   id ID #REQUIRED
   xlink:type (simple) #FIXED “simple”
   xlink:href CDATA #REQUIRED>

<!ELEMENT Flows (Flow*)>

<!ELEMENT Flow EMPTY>
<!ATTLIST Flow
         id ID #REQUIRED
         from IDREF #REQUIRED
         to IDREF #REQUIRED>
```

Appendix C: PDGML DTD

```
<!-- pdg.dtd 0.1                                                -->
<!-- A DTD for representation of a PDG as an XML document -->

<!ELEMENT PDG (Items, Dependencies)>
<!ATTLIST PDG
         xmlns:xlink CDATA #FIXED “http://www.w3.org/1999/xlink”
         program CDATA
         scope CDATA>

<!ELEMENT Items (Variables?, DummyNodes?, Statements?)>

<!ELEMENT Variables (Variable*)>
<!ELEMENT Variable EMPTY>
<!ATTLIST Variable
         id ID #REQUIRED
         xlink:type (simple) #FIXED “simple”
         xlink:href CDATA #REQUIRED>

<!ELEMENT DummyNodes (DummyNode*)>
<!ELEMENT DummyNode EMPTY>
<!ATTLIST DummyNode
         id ID #REQUIRED
         type (Entry|InitialDef|FinalUse) #REQUIRED
         variable IDREF>

<!ELEMENT Statements (Statement*)>
<!ELEMENT Statement EMPTY>
<!ATTLIST Statement
```

```
          id ID #REQUIRED
          xlink:type (simple) #FIXED “simple”
          xlink:href CDATA #REQUIRED>

<!ELEMENT Dependencies (ControlFlows?, DataFlows?)>

<!ELEMENT ControlFlows (ControlFlow*)>
<!ELEMENT ControlFlow EMPTY>

<!ATTLIST ControlFlow
          id ID #REQUIRED
          from IDREF #REQUIRED
          to IDREF #REQUIRED
          label (True|False) “True”>

<!ELEMENT DataFlows (DataFlow*)>
<!ELEMENT DataFlow EMPTY>
<!ATTLIST DataFlow
          id ID #REQUIRED
          from IDREF #REQUIRED
          to IDREF #REQUIRED
          over IDREF #REQUIRED>
```

References

[1] XML.ORG, www.xml.org

[2] World Wide Web Consortium, www.w3c.org

[3] XML Schema, www.w3.org/XML/Schema

[4] MathML, www.w3.org/Math/

[5] VoiceXML, www.w3.org/TR/voicexml20/

[6] Alfred V. Aho and Jeffrey D. Ullman. Principles of Compiler Design. Addison-Wesley Publishing Company. April 1979.

[7] Francis E. Allen. Control flow analysis, ACM SIGPLAN Notices, Volume 5 Issue 7. July 1970.

[8] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. The Program Dependence Graph and Its Use in Optimization. ACM Transactions on Programming Languages and Systems. July 1987.

[9] Susan Horwitz, Thomas Reps and David Binkley. Intreprocedural Slicing Using Dependence Graphs. ACM TOPLAS, Volume 12 No 1. January 1990.

[10] D. Callahan, A. Carle, M. W. Hall, K. Kennedy. Constructing the Procedure Call Multigraph. IEEE Transactions on Software Engineering, Volume 16 Issue 4. April 1990

[11] G. C. Murphy, D. Notkin and E. S. Lan. An empirical study of static call graph extractors. Proceedings of the 18th International Conference on Software Engineering. March 1996.

[12] Hrvoje Simic and Marko Topolnik. Prospects of Encoding Java Source Code in XML. Conference of Telecommunications, 2003.

[13] Greg J. Badros. JavaML: A Markup Language for Java Source Code. International World Wide Web Conference, 2000.

[14] Michael L. Collard, Huzefa H. Kagdi and Jonathan I. Maletic. An XML-based Lightweight C++ Fact Extractor. International Workshop on Program Comprehension, 2003.

[15] Gregory McArthur, John Mylopoulos and Siu Ng. An Extensible Tool for Source Code Representation Using XML. Working Conference on Reverse Engineering, 2002.

[16] James R. Cordy. Generalized Selective XML Markup of Source Code Using Agile Parsing. International Workshop on Program Comprehension. 2003

[17] Evan Mamas and Kostas Kontogiannis. Towards Portable Source Code Representations using XML. Working Conference on Reverse Engineering, 2000.

[18] Ying Zou and Kostas Kontogiannis. A Framework for Migrating Procedural Code to Object Oriented Platforms. Asia Pacific Software Engineering Conference, 2001.

[19] Ying Zou and Kostas Kontogiannis. Incremental Transformation of Procedural Systems to Object Oriented Platforms. Computer Software and Applications Conference, 2003.

[20] Ric Holt , Andy Schürr, Susan Elliott Sim and Andreas Winter. Graph Exchange Language. http://www.gupro.de/GXL/

[21] R. C. Holt, A. Winter and A. Schürr. GXL: Towards a Standard Exchange Format. Working Conference on Reverse Engineering, 2000.

[22] James R. Cordy, C. D. Halpern and E. Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. Computer Languages, January 1991

[23] ANother Tool for Language Recognition (ANTLR), www.antlr.org

[24] Java Compiler Compiler (JavaCC), javacc.dev.java.net

[25] The Extensible Stylesheet Language Family, www.w3.org/Style/XSL/

[26] XML Query, www.w3.org/XML/Query

[27] XML Linking, www.w3.org/XML/Linking

[28] Document Object Model, www.w3.org/DOM/