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

Raihan Al-Ekram and Kostas Kontogiannis
Dept. of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada
Email: {rekram | kostas}@swen.uwaterloo.ca

Abstract

XML applications are becoming increasingly popular to define structured or semi-structured constrained data in XML for 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 code 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. Most of the XML representations proposed so far abstract the source code at the AST level. These AST representations are tightly coupled with the language grammar of the source code and hence require development of different tools for different programming languages to perform the same type of analysis. Moreover AST abstracts the program at a very fine level of granularity and hence not suitable to be used directly for higher-level sophisticated program analysis. As such, we propose XML applications for language neutral representation of programs at different levels of abstractions and by combining them we present a program representation framework in order to facilitate the development of generic program analysis tools.

1. Introduction

The Extensible Markup Language (XML) [20], a World Wide Web Consortium (W3C) [21] standard, has been widely accepted for storing and exchanging structured and semi-structured documents. Many XML sublanguages have been developed to define constrained data in XML format for special application areas, often by means of a Document Type Declaration (DTD) or XML Schema [22] definition. For example Mathematical Markup Language (MathML) [23] is defined for electronic interchange of mathematical symbols, equations and formulae or Voice Extensible Markup Language (VoiceXML) [24] is developed for voice markup and telephony call control to enable access to the Web using spoken interaction. Such markup languages are becoming increasingly popular because XML 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. Various XML applications namely JavaML [8] [12], CppML [12], srcML [9], PLIXML [10] and PascalML [10] have been proposed to represent the source code written in different programming languages. Some of them represent the complete AST of the source code while the others produce a partial AST representation by partially marking up the source at the focal point of analysis. Some of them are just syntax preserving, whereas the others preserve non-syntactic lexical information as well. But all of the AST representations are tightly coupled with the language grammar of the source code and hence require development of different tools for different programming languages to perform the same type of analysis. Moreover AST abstracts the program at a very fine level of granularity and hence not suitable to be used directly for higher-level sophisticated program analysis. As such, in this paper we propose XML applications for language neutral representation of programs at different levels of abstractions and by combining them we present a program representation framework in order to facilitate the development of generic program analysis tools.

The rest of the paper is organized as follows: Section 2 provides background information on different program representation formalisms at different levels of granularity and related work in representing them using XML. Section 3 presents language neutral AST representations based on generic language models. Section 4 discusses the proposed XML applications to represent the higher the artifacts. Section 5 presents the representation framework that will facilitate the development of generic program analysis tools. Section 6 describes a prototype implementation of the framework. Finally Section 7 concludes the paper.

2. Background and Related Work

In this section we discuss different source code representation formalisms and some higher-level abstractions of source code that focus on different aspects of a program. We also investigate the existing XML based external formats for storing and exchanging these program representations.

2.1 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.

2.1.1 Syntax Trees

A Parse Tree [1] 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) [1] is a more economical representation of the source code while 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 the 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 input for source code analysis and for constructing other representations for higher-level program analysis. 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.

2.1.2 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) [2] provides a normalized view of all possible flow of execution paths of a program. A CFG is a rooted directed graph showing the basic blocks in a program and the possible immediate transfer of control from one basic block to another. The CFG representation is extensively used for data flow analysis, code optimization and testing.

A Program Dependence Graph (PDG) [3] is a combined explicit representation of both control and data dependences in a program. The PDG is also a rooted directed graph that consists of nodes representing the statements and predicate expressions in the program and edges connecting them representing the control and data dependences between them. The control dependence edges are labeled either True or the truth-value of the predicate and the data dependence edges are labeled by the variable name that causes possible flow of data values between the nodes 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.

2.1.3 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) [4] 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 edge types to correspond to procedure calls, parameters passed and return values.

Call Graphs [5] [6] 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 also contain attribute labels (e.g. line number of the call or file name of the procedure) to enhance the graph with additional inormation. There can optionally be new entities in the graph (e.g. abstract data types and their usage relationships) in addition to the procedure calls. 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.

From the basic graph higher level call graph can be constructed to show relationships among files, modules or architectural entities instead of procedures. Other than inter-procedural data flow analysis for optimization, call graphs are also used for design recovery, architecture extraction or other reverse engineering analysis.

2.2 Program Representations using XML

Simic and Tolnik [7] explore the prospects of representing source code using XML in place of classical palin text format. They demonstrate that an XML grammar can improve the code structure, formatting, querying possibilities and will allow making orthogonal extensions to code for annotations, revision control, access control and documentation.

There is a spectrum of levels of granularity at which source code is represented. Among them the AST representation provides the most detailed information from the source code. Hence most of the XML applications for source representation proposed so far are based on the AST notation of a program.

2.2.1 Java Markup Language (JavaML)

Badros [8] 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 of the source code information are stored as attributes on the element tags.

JavaML is a complete syntactic represention of the AST and hence the formatting and other lexical information in the source code are not preserved. In addition to representing the syntax of the source code, JavaML stores few 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.

2.2.2 Source Code Markup Language (srcML)

Collard et al. [9] 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 constructed from the C++ language grammar.

The srcML allows incomplete parsing of the source code to generate a partial AST by using a multi-pass multi-stage prasing technique with a partial grammar specification. This enables controlling the parsing upto the desired level of interest depending on the focus of the analysis. This approach of parsing, marking up only the selected constructs of interest while leaveing others as it is, is known as island parsing.

2.2.3 XMLizer

McArthur et al. [10] 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 uses a multi-weight parser that can generate ASTs of variable granularity by allowing designated syntactic construct to remain unparsed. This allows preserving certain lexical information, e.g. comments, by attaching them to unparsed constructs.

2.2.4 Agile Parsing

Cordy [11] 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 [19]. This approach selectively marks up only those AST nodes in the source that are relevant to a particular analysis 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.

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 grammar of the languages are used as the markup tags. As a result the markups are still strongly coupled with the respective language grammar.

2.2.5 Graph Exchange Language (GXL)

The Graph Exchange Language (GXL) [17] [18] is an XML based language for describing graphs. It evolved from unification of other existing graph description languages. Unlike the other representations discussed, GXL was not originally intended to represent the source code. Hence there is no schema defined in GXL to represent any software artifact. Instead, it provides features to specify the schema for the data as well as the data itself in the same format. The higher lever program representation formalisms being graphs in nature make GXL a good candidate for their representation.

3. Modeling Programming Languages

Even though AST is the fundamental source code representation formalism for building software analysis tools, AST represenatations are strongly tied with the corresponding language grammars. Which requires the development of different tools to perform same type of analysis for programs written in different programming languages. To enable portability of the representations and building generic tools the AST representations should be decoupled from the laguage grammars.

Over a family of programming languages the key concepts remain the same and they share many common features. For example object-oriented languages Java and C++ both have the notion of class, method/function, inheritance etc. Hence it is possible to develop a generic model of object-oriented languages by studying the grammars and a) identifying the commonalities and obtaining a generalization and b) identifying the variabilities and aggregating them at a higher level of abstraction. The AST representations based on the generic model will be able to handle constructs and represent source code from various object-oriented languages in a uniform format. Tools built on the generic format, e.g. a tool to extract object model, will be able to analyze programs written in any object-oriented language. The same argument holds for the family of procedural languages and so on.

3.1 Generic Procedural Model

Zou and Kontogiannis [14] [15] [16] proposed a generic model and an XML application, Procedural Markup Language (ProcML), for representing the procedural languages in XML. Their proposed model is derived from programming languages like C, Fortran, Pascal and COBOL.

In the first step the AST representation of individual languages are modeled using UML. The classes in the UML model encode the AST nodes, which are the basic language constructs and the attributes gathered in them. The class associations represent the attributes of non-primitive language syntactic types.

The second step is to identify the functionally equivalent constructs in different languages and generalize them at a higher level of abstraction. For example *subroutine* in Fortran and *function* in C denote similar concepts that can be generalized as a unique term *procedure*. Figure 1 presents a part of their proposed generic model for procedural languages.

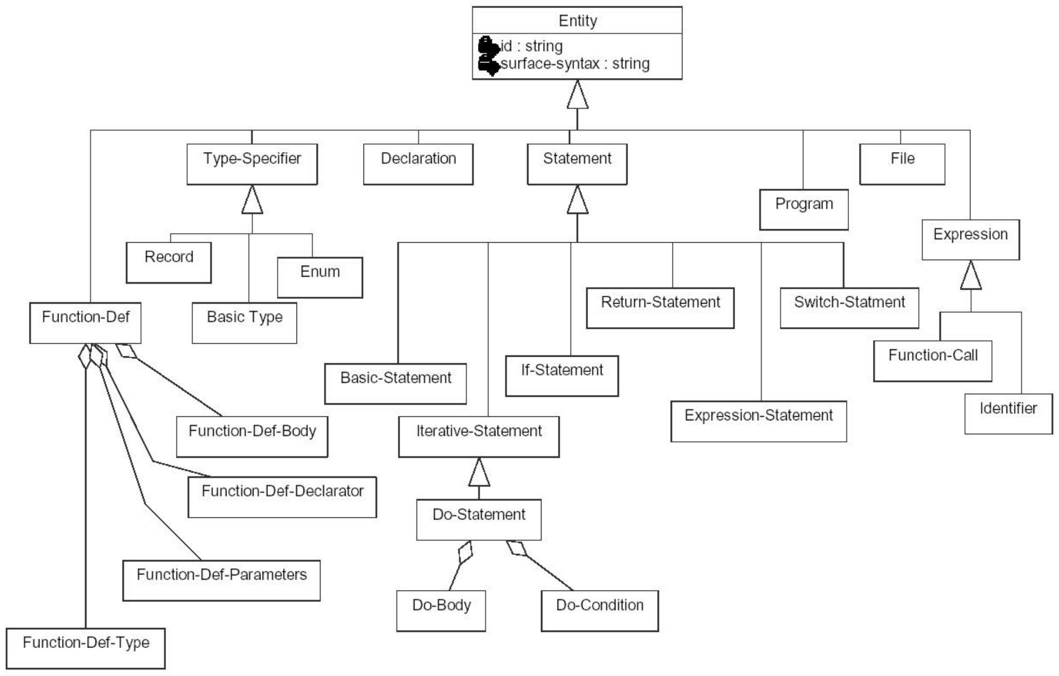

Figure 1: Generic Procedural Language Model

The UML diagram is a graphical representation of the model. In third step, for storage and model interchange, the UML models are encoded in XML DTD definitions. This results in one DTD for each of the languages – CML, FortranML and PascalML corresponding to C, Fortran and Pascal language and one for the generic model – ProcML. The produced DTDs are effectively the models and the XML representations of the ASTs are instances of them that will be validated against the models.

The generation of XML files representing the ASTs works as follows – a) XML ASTs for individual languages are generated in conformance to their own language model DTDs and b) XML ASTs for specific language models are transformed to the generic model using XSLT mapping programs.

3.2 Generic Object-Oriented Model

Mamas and Kontogiannis [12] [13] in their work proposed an XML application Object-Oriented Markup Language (OOML) as a generic model for representing object-oriented programming languages. OOML is derived by generalizing JavaML and CppML, language models for Java and C++ languages respectively. Table 1 lists some of the mappings from JavaML and CppML entities to OOML entities.

Table 1: JavaML, CppML to OOML Mapping

JavaML	**CppML**	**OOML**
CompilationUnit	Program	Program
ImportDeclaration	Include	Include
ClassDeclaration	Class	Class
MethodDeclaration	Function	Method
FieldDeclaration	Variable	Variable-Declaration
Block	LexicalBlock-Statement	Body
SwitchStatement IfStatement	SwitchStatement IfStatement	Conditional-Statement
DoStatement ForStatement WhileStatement	DoStatement ForStatement WhileStatement	Loop

4. Modeling Higher Level Artifacts

While the AST level representations are useful for some type of analysis, they are not usable for sophisticated higher-level analysis. For example in order to perform data flow analysis on a program the CFG representation of the program is required or in order to perform design recovery for a software system the call graph from the source programs is required. But the existing XML applications lack in defining program representations for abstractions at a level higher than the AST. The higher-level program abstractions are the intra and inter procedural flow and dependence graphs of a program. Among them the most commonly used representations in program analysis are CFG, PDG and Call Graph. As part of this research we propose XML applications CFGML, PDGML and CGML to represent these graph data structures respectively.

For each of the higher-level representations we first identified the basic elements that constitute the representation and the relationships among these elements. Based on it we developed a UML model for each of the representations. In doing so we realized that all these representations share some common elements. The common elements are the basic building block constructs of a program and the relationships among them. These constructs are statements, variables, data types, functions etc. and the relationships are the uses/definitions/declaration of the variables in the statements, declarations/calls to the functions etc. We call these constructs and relationships the *Facts*. The higher-level representations use the Facts and define new constructs and relationships, specific to the particular representation, on top of them.

4.1 FactML

The first step is to develop a UML model for the program Facts. The building block constructs of the Facts are represented as classes and the relationships among them are shown as associations or association classes. This results in classes named *Type*, *Variable*, *Statement*, and *Function* in the model. Each member of the *Variable* class is associated with a member of the *Type* class by its data type. A *Variable* and a *Statement* are related with a declaration relationship that is a simple association, whereas uses and definitions of a *Variable* in a *Statement* is more complex and requires an association class. There can be three different relationships between a *Statement* of and a *Function*. A *Function* is declared in a *Statement*, a *Function* consists of many *Statement* and a *Statement* can call one or more *Function*. Figure 2 presents the complete UML model of the Facts.

In the second step the UML model is transformed into an XML DTD declaration using following production rules

- Classes are mapped as elements
- Attributes of the classes are mapped as attributes in the elements
- Aggregations are mapped as sub-elements separated by or (|)

- Inheritances are mapped as sub-elements
- Simple associations are shown by IDREFs
- Association classes are mapped as elements showing the associations by IDREFs
- Elements with same tags originating from same node are grouped as sub-elements under one bigger element.

Figure 3 presents a part of the DTD derived from the UML model. For classes *Statement* and *Variable* in the UML model there is one element each in the DTD. Collections of them are grouped under bigger elements *Statements* and *Variables*. The optional IDREF attribute *function* in the *Statement* element refers to a *Function* element the statement is part of and the IDREF *declared* in *Variable* refers to a *Statement* the variable is declared in. The association class *UseDef* is mapped to its own element and grouped under a single *UseDefs* element.

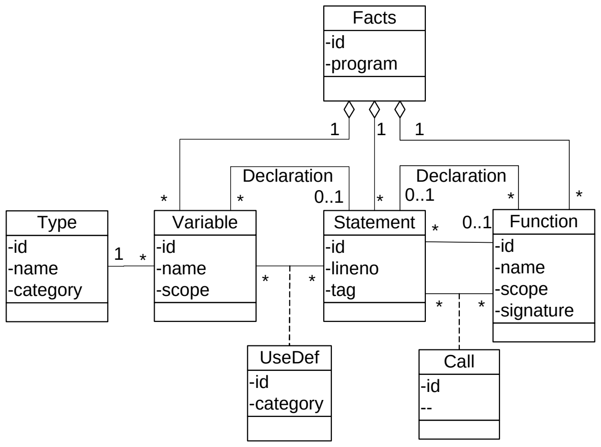

Figure 2: UML Model for Facts

```
…
<!ELEMENT Statements (Statement*)>
<!ELEMENT Statement EMPTY>
<!ATTLIST Statement
     id ID #REQUIRED
     lineno CDATA #REQUIRED
     tag CDATA
     function IDREF>
<!ELEMENT Variables (Variable*)>
<!ELEMENT Variable EMPTY>
<!ATTLIST Variable
     id ID #REQUIRED
     name CDATA #REQUIRED
     scope (Local|Global|Param|Ext) “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>
…
```

Figure 3: DTD for Facts, FactML

4.2 CFGML

A CFG is a directed graph indicating the basic blocks in a program and possible flows of control from one basic block to another. A basic block contains a sequence of consecutive program statements. The UML model and hence the XML DTD presented in Figure 4, describes these basic blocks and the flow of control among them. Description of any basic building block construct, e.g. Statement, is linked from the FactML using XLink [27].

Figure 5 shows an example C program and Figure 6 shows the corresponding CFG of the program as an instance of the CFGML. The FactML instance of the program is assumed to be stored as Facts.xml

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

Figure 4: DTD for CFG, CFGML

```
<1> main ()
<2> {
<3>    int a = 0;
<4>    if (a>3)
<5>       a = a+3;
<6>    a = 10;
    }
```

Figure 5: An Example C Program

```
<CFG>
 <Blocks>
  <Block id=1 label=1>
   <Statement id=2 xlink.href=”Facts.xml#3”/>
   <Statement id=3 xlink.href=”Facts.xml#4”/>
  </Block>
  <Block id=4 label=2>
   <Statement id=5 xlink.href=”Facts.xml#5”/>
  </Block>
  <Block id=6 label=3>
   <Statement id=7 xlink.href=”Facts.xml#6”/>
  </Block>
 </Blocks>
 <Flows>
  <Flow id=8  from=1 to=4 />
  <Flow id=9  from=1 to=6 />
  <Flow id=10 from=4 to=6 />
 </Flows>
</CFG>
```

Figure 6: CFGML instance of the C Program

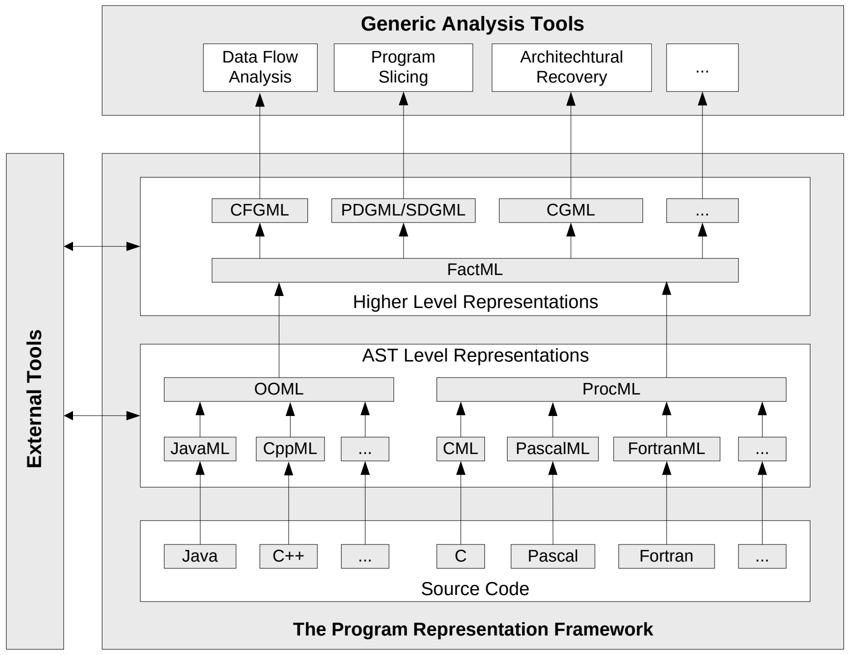

Figure 7: System Architecture for the Program Representation Framework

4.3 PDGML and CGML

Similarly PDG and Call Graphs can be modeled in UML and corresponding XML DTDs can be generated from them.

5. The Representation Framework

In figure 7 we present the multi-layered framework for language neutral representation of program artifacts. We also demonstrate the usage of the framework for building generic program analysis tools. The framework follows a pipe and filter type architectural style. The pipe components are the different layers of abstractions of the program source and the filter components are the representation transformers and the analysis tools.

5.1 Abstraction Layers

There are three distinct layers corresponding to three different levels of abstractions of source code in the framework. Layer 0 is the original source text of the program to be analyzed as it is.

Layer 1 is the first level of abstraction of the source code in terms of the AST of the program. We choose to adopt the AST representations proposed by Zou and Mamas to fit in this layer. Since these representations also include the generic representations for procedural and object-oriented language family, they will provide language neutral representations of the AST. This layer consists of two sub-layers. Layer 1.1 are the ASTs representations in programming language specific markup languages, i.e. JavaML, CppML, CML, PascalML and FortranML. Layer 1.2 are the AST representations derived from the generic model of the language family, i.e. ProcML and OOML.

Layer 2 is the next level of abstraction in terms of the different intra-procedural and inter-procedural graphs. This layer is also consists of two sub-layers. Layer 2.1 represents the basic facts of a program in the FactML format. Layer 2.2 is the representations for intra-procedural and the inter-procedural dependence and flow graphs of the program expressed as CFGML, PDGML, SDGML and CGML.

5.2 Transformers

A set of transformer tools is required to convert the representations from one level to the next higher level of abstractions. Some of them are source code transformers that are parsers of the source text in order to emit corresponding AST in the language specific XML format. There has to be one transformer for each of the languages to be analyzed.

The rest of transformers are XML to XML transformers. These transformers can be built using XSLT stylesheets [25], XPath/XQuery [26] or DOM [28] manipulation. There will be once transformer for each of the following conversions

- JavaML, CppML to OOML
- CML, PascalML, FortranML to ProcML
- OOML, ProcML to FactML
- FactML to CFGML, PDGML, CGML

5.3 Analysis Tools

Various program analysis tools can be written on top of the proposed framework. Since these tools will work on language neutral representations of the program, it is possible to develop of a single tool to perform a particular type of analysis on a source program 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.

All the representations in the proposed framework are XML and hence can be easily transformed to any other formats using XSLT or XQuery 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 are needed to map the external formats to the internal XML representations.

6. A Prototype Implementation

We have developed a prototype toolset based on the proposed framework. Our prototype works on the JavaML-OOML representation of Mamas and Ret4J [29] toolkit to generate JavaML-OOML instances of Java programs. Minor modification is done to Ret4J to include a *lineNumber* attribute in the generated XML.

6.1 Analysis Tools

As part of the toolset we have developed a fact extractor that takes an OOML file as input and generates a FactML file. The tool works on the DOM tree of the OOML instance and makes XPath query to extract information from it. We have developed a PDG generator that works on both the OOML and FactML files and generates a PDGML instance.

The toolset also consists a PDG slicer that slices a PDGML instance and emits a reduced PDG based on the algorithm given in [4]. The statements remaining in the sliced PDG will comprise the program slice. The slicer can perform the following kinds of slicing:

- Backward slicing for a given program point and a variable use and the final use of a given variable
- Backward decomposition slicing for the uses of a given variable
- Forward slicing for a given a program point and a variable use
- Forward decomposition slicing for the definition of a given variable

6.2 Operational Statistics

In this section we evaluate the proposed framework in terms of the sizes and the time required to generate the representations by the prototype toolset. Five input files of different sizes were used to measure the size and time parameters. These files were chosen from a variety of sources ranging from student course projects to standard utility library. The prototype was developed using the Java programming language (JDK 1.3) and all the experiments were run in a Sun UltraSPARC III 440 MHz station with 512 MB of RAM and running Solaris 8 Operating System.

Table 2 presents the size of the generated FactML files and the time required to generate them by the fact extractor tool. The size of the FactML is approximately 5 times the source code. Table 3 summarizes the relationship between the size of a method and the size of its corresponding PDGML and the time taken to produce it. Even though the general tendency of the size of the PDGML is to increase with the size of the method, it may not be the case always. When there is a low number of def-use chaining in the program, the number of edges in the graph is low and it will result in a smaller PDGML size. Finally Table 4 shows the results of slicing based on the final uses of a given variable. The size of the slice compared to the size of the method shows the same property as the size of the PDG. The time required to slice a PDG is quiet reasonable and depends on the size of the source.

Table 2: Experimental Results for Fact Extraction

Program	Source Size (bytes)	FactML Size (bytes)	FactML Time (ms)
MyMath.Java	187	2,030	224
Voter.java	3,822	17,502	1,487
GUI.java	4,994	20,697	1,498
UnboundedLife.java	10,831	33,800	3,849
PDG.java	22,200	81,072	11,403

Table 3: Experimental Results for PDG Creation

Class: Method	Size (LOC)	PDGML Size (bytes)	PDGML Time (ms)
MyMath:factorial	13	2,466	154
UnboundedLife:restore	30	6,194	402
GUI.java	39	11,144	748
PDG:backwardSlice	40	12,166	711
Voter:fix	55	11,086	945

Table 4: Experimental Results for Slicing

Class:Method:Variable	Source Size (LOC)	Slice Size (LOC)	Slicing Time (ms)
MyMath:factorial:i	13	10	2
UnboundedLife:restore:x	30	13	3
GUI.java:labels	39	24	10
PDG: backwardSlice:list	40	31	26
Voter:fix:game	55	23	10

7. Conclusion

In this paper we presented a framework for language neutral program representation. The framework is based on a multi-layered abstraction of source code artifacts represented using several XML applications. The framework adopts the existing XML applications for source code representation and defines new applications to represent higher-level program abstractions. The framework is extensible, new representations and tools can be added to it to facilitate different generic analysis tasks.

The obtained operational statistics from the prototype toolset show that the tools operate fairly accurately and with reasonable performance. The sizes of the different intermediate representations and the time required to generate them are reasonable. As a conclusion, this paper provides the fundamental mechanism to build generic tools that will perform program analysis independently of the programming language used to write the program.

8. References

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

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

[3] 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.

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

[5] 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

[6] 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.

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

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

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

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

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

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

[13] Evan Mamas. Design and Implementation of Integrated Software Maintenance Environment. MASc Thesis, Department of Electrical and Computer Engineering, University of Waterloo. 2000.

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

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

[16] Ying Zou. Techniques and Methodologies for the Migration of Legacy Systems to Object Oriented Platforms. PhD Thesis, Department of Electrical and Computer Engineering, University of Waterloo. 2003.

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

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

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

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

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

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

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

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

[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

[29] Reengineering Toolkit for Java, www.alphaworks.ibm.com/tech/ret4j