Towards Portable Source Code Representations Using XML

E. Mamas
{evan@swen.uwaterloo.ca}
Dept. of Electrical &
Computer Engineering
University of Waterloo
Waterloo ON. N2L 3G1
Canada

K. Kontogiannis
{kostas@swen.uwaterloo.ca}
Dept. of Electrical &
Computer Engineering
University of Waterloo
Waterloo ON. N2L 3G1
Canada

Abstract

One of the most important issue in source code analysis and software re-engineering is the representation of source code text at an abstraction level and form suitable for algorithmic processing. Moreover, source code representation schemes must be compact, accessible by well defined application programming interfaces (APIs) and above all portable to different operating platforms and various CASE tools. This paper proposes a program representation technique that is based on language domain modes and the XML markup language. In this context, source code is represented as XML DOM trees that offer a higher level of openess and portability than custom-made tool specific Abstract Syntax Trees. The DOM trees can be exchanged between tools in textual or binary form. Similarly, the domain model allows for language entities to be associated with analysis services offered by various CASE tools, leading to an Integrated Software Maintenance Environment.

1 Introduction

There is a pressing demand for legacy software systems to be at all times current and operational. In order to achieve these objectives, software developers constantly maintain their legacy systems so that, they can be ported to new environments, and eliminate dependencies on obsolete programming languages, operating systems, or software architectures. A great problem in such maintenance tasks is accessing, organizing, and managing information related to the software system. This includes the source code itself, call graphs, data dependencies, links to external documentation, informal memos etc. It is estimated that up to 50% or more of a software engineer's time is spent on such information searching for related program understanding and maintenance tasks [Pressman97].

The project's objective is to investigate the requirements, design issues and implementation issues for systems which store, organize, and manage information related to large amounts of legacy code for the purpose of system maintenance and reengineering. The project is founded on the premise that software artifacts, such as source code, Abstract Syntax Trees (ASTs), call graphs, documentation, informal notes and memos from developers, can all be stored, organized, and managed by a generic system, which we shall call a Integrated Software Maintenance Environment (ISME). The ISME aims on providing support for software reengineering tasks. One can think of a ISME as a specialized DBMS, tailored to the representation of software-related information, also offering specialized interfaces for CASE plug-ins in support of software maintenance tasks. ISME is intended to make it easier to load and integrate all relevant information about a legacy system, as well as search, access and correlate information during a migration process by supporting the interface of the code base with existing CASE tools and transformation plug-ins.

2 Related Work

In this section we discuss related work performed by various research groups in the area of source code modeling and program representation. The area of program representation deals with the techniques and methodologies to represent information about a software system at various levels of abstraction that are suitable for algorithmic processing.

In this respect, program representation aims on facilitating source code analysis that can be applied at various levels

*This work was funded by the Natural Sciences and Engineering Research Council of Canada, the Consortium for Software Engineering Research, and IBM Canada Ltd., Centre for Advanced Studies. Inquiries for this paper can be sent to kostas@swen.uwaterloo.ca

of abstraction and detail namely at:

- the physical level where code artifacts are represented as tokens, syntax trees, and lexemes,
- the logical level where the software is represented as a collection of modules and interfaces, in the form of a program design language, annotated tuples, aggregate data and control flow relations,
- the conceptual level where software is represented in the form of abstract entities such as, components, abstract data types, and communicating processes.

These representations are achieved by parsing the source code of the system being analyzed at various levels of detail and granularity.

One such representation is the Abstract Syntax Tree. Abstract Syntax Trees or ASTs in short, are tree structures that represent all the syntactic information contained in the source code [Aho86]. Every node of the tree is an element of the language. The non-leaf nodes represent operators, while the leaf nodes represent operands. Abstract Syntax Trees suppress unnecessary syntactic details (whitespace, symbols, lexemes, punctuation tokens) and focus on the structure of the code being represented. The AST notation is the most commonly used structure in compilers to represent the source code internally in order to analyze it, optimize it and generate binary code for a specific platform.

Abstract Semantic Graph, or ASG in short, provides a rich abstract representation of source code text. ASGs are composed of nodes and edges. Nodes represent source code entities, while edges represent relations. Both the nodes and the edges are typed and have their own annotations that denote semantic properties [Devanbu96].

The ASG as a program representation scheme has been used by the Datrix [Datrix] and it is currently used to model C++ and Java source code. In Datrix the ASG is annotated with data and control flow information gathered from the source code at parse time (i.e. scoping, call graph information).

Program Dependence Graph, or PDG in short, is a graph that combines control flow and data flow information into a single structure [Ferrante87]. In a PDG, nodes represent statements, expressions or regions of code and edges represent data and control dependency information. The dependencies expressed in a PDG, are a result of processing the source code information or the corresponding AST. The PDG is useful for static and dynamic slicing techniques, program transformation and code optimizations.

The Rigi Standard Format, or RSF in short, is a format for representing source code information. It is a generic, intuitive format that is easy to read and parse. The information that RSF is currently used for can be classified as metadata. This allows RSF to be generic and to use the same format to represent program information for a variety of programming languages without any changes. The syntax of RSF is based on entity-relation triplets of the form `<relation, entity, entity>`. Sequences of these triplets are stored in self-contained files. Currently RSF is the base format for the reverse engineering tool Rigi [Rigi].

The Tuple-Attribute Language, or TA in short, is a language designed to represent graph information [Holt98]. This information includes nodes, edges and any attributes the edges may contain. TA is easy to read, convenient for recording large amounts of data and easy to manipulate. The main use for TA is to represent facts extracted from source code through parsers and fact extractors. In this way TA can be considered to be a ”data interchange” format.

AsFix is a parse tree representation for terms and modules. The AsFix formalism is an instantiation of a more generic format called ATerms. ATerms are used to represent structured information that is to be exchanged between a collection of tools.

Graph Exchange Language, or GXL in short, is a proposed format for exchanging information among tools that analyze computer programs [Holt00]. The GXL format is suitable for representing typed graph information. A graph exchange format requires that both the schema and the data of the graph are represented in the format and GXL accomplishes this with the use of XML.

Finally, in the area of CASE tool integration the Refine Code-base Management System, developed by Reasoning, is a product that provides support for software analysis and transformation [Refine]. Refine uses ASTs for representing source code information and provides a proprietary Software Development Kit (SDK) for processing the information.

3 Portable Source Code Representations

In this section we provide a brief introduction to the XML language, the related Document Type Definition language and the existing interfaces for working with XML documents. Moreover, we explain how XML can be used to represent source code at the AST level and we discuss how programming language grammars can be mapped to DTDs.

3.1 eXtensible Markup Language

XML is the acronym for Extensible Markup Language and has been developed by W3C (the World Wide Web Consortium). XML is an ideal format for storing structured data intended for publishing or exchange between different applications. XML derives from SGML (Standard Generalized Markup Language) and in a way from HTML (HyperText Markup Language). SGML was developed in 1986

as an international standard for document markup. HTML was developed in 1992 as a language specific to Web pages. What makes XML flexible is that it allows the end user to specify custom tags and to associate semantic information to source code text.

An XML document has a logical and a physical structure. The logical structure allows the document to be divided into units called elements. These elements can contain other elements in turn thus allowing for a complex logical structure to be defined.

For example, to describe a book we need a book element, a title element and an author element. Also, this book will have a unique ISBN number that could be stored as an attribute to the book element. Here is how this would be expressed in XML:

```
<book ISBN="123456789">
  <title>The story of my life</title>
  <author>George Thomas</author>
  <author>Tom Jhones</author>
</book>
```

The benefits of using a meta-language like XML in the industry are many. First of all, the document publishing applications can exploit it to develop better searching and indexing techniques. The web applications would benefit by having the power to dynamically customize the way the same information is presented to the different users. The most important benefit of all will be that of data exchange. XML will allow the creation and use of common structures that will be used between applications not only to exchange data but also to communicate using messages. A detailed description on the structure and uses of XML can be found in [Bradley98] and [W3CXML].

3.2 DocumentType Definition

Document Type Definition, or DTD in short, is a technology directly related to XML documents. It provides a way of defining the logical structure of the XML document. The logical structure contains all the elements that can be used and describes how they can be used in relation to each other. This results in a document hierarchy. Without a DTD, an XML document can only be checked to determine if it is well formed. This means that every start tag is followed by a corresponding end tag. The use of a DTD allows us to check for validity of the XML document. Everything that is in the XML document must conform to the DTD specification. Therefore we are able to enforce certain restrictions on how the XML document can be composed and this makes it easy to create applications that process these XML documents. The DTD contains a number of declarations. Each declaration can be one of the following declaration types:

ELEMENT
Elements are the basic contents of an XML file and correspond to user defined tags (i.e. `book`, `author`) as illustrated in the example above.. We can have empty elements, or elements that contain other elements or text. A model group (tag) is used to describe enclosed elements and text.

ATTLIST
A list of attributes associated with a particular element can be declared using ATTLIST. Every attribute has a name, a type and a default value. The type determines the range of values the attribute may hold. The allowed types are: CDATA, NMTOKEN, NMTOKENS, ENTITY, ENTITIES, ID, IDREF, NOTATION or name group. The default value allows us to specify if the attribute is required, implied, default or fixed.

ENTITY
Entities are used to avoid repetition in XML documents. They are declared once and can be referred to many times. Both internal and external entities are allowed in DTDs. Internal entities are defined within the current DTD, while external entities reside in a separate DTD.

NOTATION
Notations are used to refer to data that is not in XML format. Notations can also be linked with entities by using the NDATA keyword.

The following is a sample DTD that can be used to enforce the logical structure of the example presented previously in the XML section. By examining the DTD we see that a book must have one title and at least one author. Also a book has a unique ISBN number as an attribute. Both the title and author elements contain character data that stores the information.

```
<!ELEMENT book (title,author+)>
<!ATTLIST book  ISBN CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

3.3 Working with XML documents

The World Wide Web Consortium has defined a standard interface for accessing XML files called Document Object Model (DOM) [DOMspec]. Another interface called Simple API for XML (SAX) [SAXspec] has been developed by members of the XML-DEV mailing list.

Document Object Model (DOM)

DOM is a language neutral interface for allowing programs to access and update the structure and style of documents. DOM is a tree-based API that generates a internal tree structure of the document and allows an application to navigate and manipulate the tree. Every document is composed of Nodes and a variety of node types are defined in the DOM Core specification. The DOM tree for the XML example presented previously is shown in Figure 1. Methods for manipulating the tree or its components are provided by the specific parser implementation. The advantage of this interface is that the complete document exists in memory and it can be easily processed and manipulated. The disadvantage is that working with large documents imposes a large memory requirement.

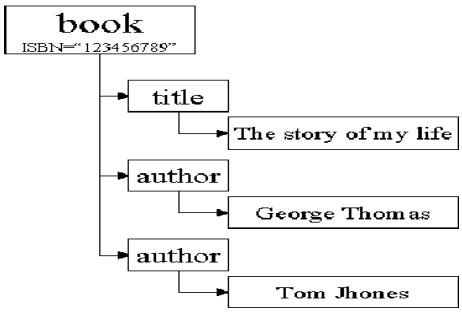

Figure 1. Sample DOM tree.

Simple API for XML (SAX)

SAX is an event-based API that reports event as it is parsing the XML document. An application that handles the events can be build to perform various tasks. The key difference when compared to DOM is that SAX does not built a tree representation of the XML file in memory. Therefore, SAX is more convenient for tasks that do not require the complete tree to be present in memory. A typical scenario in which SAX would be a better choice occurs when for example, we need to search a 10MB file to count the number of ”Author” elements. The system requirements by SAX are minimal when compared to DOM.

4 XML based representations for programming languages

XML-based markup languages offer great flexibility as well as the ability to represent documents as DOM annotated trees. Our objective is to develop XML-based program representations in which the corresponding DOM trees represent source code information at the same level as this is provided by an AST generated by a language parser. The proposed representation is simple but detailed enough to represent the complete syntax of a specific programming language in the form of a DTD schema and annotated source code in the form of a DOM tree. In addition, the DTD schema is extensible to allow for new entities and attributes to be added to model analysis results obtained by source code analysis tools. These analysis-specific schemata, can be defined separately and linked to the basic source code DTD schema. When working with multiple domain models, instead one can amalgamate and aggregate these models into more general ones.

For example, representations for programming languages that use XML as the modeling environment provide several advantages. Namely, these are:

- **Easy to understand and manipulate**
  By keeping the representation of the language as close as possible to the grammar of the language no extra learning burden should be added to the tool developers. Since the grammar is something that developers are already familiar with, our representations should approximate the grammar as much as possible.

- **Extensible**
  The need to accommodate for the evolving programming languages requires that the representation is equally easy to change. The representation should also be flexible enough to allow for other representations to be developed based on it as extensions.

- **Widely supported**
  The success of a program representation is based on how well it is supported. The development of APIs that enable developers to easily read, store and change the information based on the suggested representations is very important. These APIs should be readily available for a variety of platforms.

- **Human Readable**
  Given the nature of the software maintenance tasks, it is necessary for the format to be easily readable. Even though most processing will be done automatically by the tools, enabling a developer to read the information directly will make the development task easier.

4.1 Annotating Source Code using XML

4.1.1 Mapping ASTs to DTDs

Given a specific programming language we need to define a representation in which every valid source code document can be mapped to. To accomplish this mapping from ASTs to XML trees we need to define a method to map the grammar of the programming language to a Document

Type Definition (DTD). The mapping at this level will guarantee that all possible syntax trees defined by the grammar (and therefore any source code program), can be mapped to XML trees defined by the corresponding DTD. One of the requirements when defining new XML representations is to make them as easy to use as possible. In this context, we provide a set of general transformation rules that assist in implementing a good mapping from a grammar to a DTD. Below, a list of transformation rules for defining such mappings is presented.

- Non-terminal grammar symbols are mapped to elements.
- Sequences of non-terminals are mapped to model sequence groups (i.e. using the "," symbol).
- Choices of non-terminals are mapped to model choice groups (i.e. using the "|" symbol).
- Non-Terminals that contain sequences of characters are mapped to attributes.
- Optional non-terminals are mapped to model optional choice groups (i.e. using the "?" symbol).
- Lists of non-terminals are mapped to model set groups (i.e. using the "*" or "+" symbol).
- Terminals are mapped to attributes.
- Choices of terminals are mapped to named group attributes.
- Sequences of terminals are mapped to distinct attributes.

The use and combination of these transformation rules allow us to generate very elaborate mappings from grammars to DTDs. Once a grammar is mapped to a DTD, the ASTs for a specific program can be mapped to XML files. These XML files can then be used in place of the ASTs or the original source files for maintenance tasks.

4.1.2 Example Mappings

In order to demonstrate how the previous rules are used we present the following two simple examples. In the first one, we combine the rule for sequences of non-terminals and the rule for optional non-terminals. The resulting element declaration states that the element `a` contains an element `b` which is optionally followed by element `c`. In the second example we demonstrate the rule for mapping lists of non-terminals to model groups is used with in conjunction with the rule for mapping terminals to attributes. We see from the grammar rule that b is an artifact for expressing the fact that `c` can occur many times. The element declaration captures this concept by simply using the "+" symbol. The terminal is mapped to an attribute declaration by storing the literal in a string called `value`. In the table below the mappings for both examples are shown.

Grammar	**DTD**
a : b c_{opt}	<!ELEMENT a (b,c?)>
a : b Literal b : c b c	<!ELEMENT a (c+)> <!ATTLIST a value CDATA>

4.2 Java Markup Language (JavaML)

The generation of a program representation for Java is based on a parser generator tool called Java Compiler Compiler or JavaCC in short [JavaCC]. This tool was initially developed by Sun Microsystems and it is the most popular parser generator for Java. A parser generator is a tool that uses a BNF grammar as input and generates source code that can parse any instance of the BNF grammar. The popularity of JavaCC is most probably due to the grammar for Java that is shipped with the tool. The majority of Java source code parsers are built using JavaCC and the Java 1.1 grammar. The Java 1.1 grammar was developed by Sriram Sankar at Sun Microsystems and a copy of this grammar can be found in the distribution of JavaCC package.

The complete DTD that we generated based on the Java 1.1 grammar can be found at http://swen.uwaterloo.ca/ evan/javaml.html. Below we present a small example of a Java source file and its corresponding JavaML representation as it is automatically generated by adding semantic actions in the Java parser generated by JavaCC.

Java source code

```
public class Car{

 int color;

 public int getColor(){
  return color;}

}
```

JavaML representation

```
<ClassDeclaration Identifier="Car">

 <FieldDeclaration>
  <PrimitiveType Type="int"></PrimitiveType>
  <VariableDeclaratorId Identi-
fier="color"/>
 </FieldDeclaration>
```

```
 <MethodDeclaration Identifier="getColor">
  <ResultType>
   <PrimitiveType Type="int"/>
  </ResultType>
  <Block>
   <ReturnStatement>
    <PrimaryExpression>
     <Name Identifier="color"></Name>
    </PrimaryExpression>
   </ReturnStatement>
  </Block>
 </MethodDeclaration>

</ClassDeclaration>
```

4.3 C++ Markup Language (CppML)

The generation of a representation for the C++ programming language is based on a different approach than that for Java. Instead of using a parser generator we took advantage of a compiler product that maintains the intermediate representation of the code and provides access to it through an API. This product is the IBM VisualAge C++ [VACpp] compiler which is developed at the IBM Toronto lab. VisualAge features a customized source code repository which is called Codestore in which all the information generated during the compilation process is stored. Parsing, processing and code generation information can all be accessed using the provided APIs. The goal behind the architecture of VisualAge is to allow developers to maintain and analyze C++ source code. Our goal in using VisualAge is to demonstrate that a commercial product can be integrated and used as a tool in a more complex environment. Using a parser generator would have been another approach in generating a C++ representation. The complete representation for C++ that was generated using VisualAge can be found at http://swen.uwaterloo.ca/ evan/cppml.html. Sample source files and their representations are also available. The grammar that CppML was based on, was implicitly extracted from the Codestore APIs, which was obtained from the VisualAge development team and is expressive enough to represent ANSI C++ compliant source files.

4.4 Object Oriented Markup Language (OOML)

The object oriented programming paradigm has been implemented in a variety of programming languages. However the key concepts remain the same no matter what the implementation is. Here, we demonstrate how the DTD schemata for the Java and C++ programming languages, which are both object oriented, are aggregated to a more generic representation called OOML. It is noted that OOML does not have all the information that exist in a source file, because not all Object Oriented languages contain entities that can be aggregated.

In order to generate OOML representations it is possible to define mappings from the JavaML and CppML representations instead of using the source code directly. The implementation of such mappings can be done in two ways: The first involves the use of the XML APIs to construct a program that maps one representations to the other. The second approach involves the use of XSLT [XSLT] transformations which are designed to map one XML document to another. Both approaches will achieve the same goal and the developer will have to select one. It needs to be clear though that we are not interested in the implementation details of either approach. Our goal is to identify similar concepts in JavaML and CppML and describe how they can be mapped to OOML.

A small example of how common constructs in JavaML and CppML can be identified and mapped to the OOML representation below. The complete DTD for the OOML representation can be found at http://swen.uwaterloo.ca/ evan/ooml.html.

Both Java and C++, represent objects by using the concept of classes, class methods and class variables. An object is an abstract entity that contains some data and that is able to perform some kind of operations on its data. A class can be thought of as a template for creating an object. Every class has a name that uniquely identifies the class. The class variables define what data the object can store and the class methods define what kind of operations the object can perform. In OOML this information is expressed as follows:

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

The relevant parts of the JavaML representation are shown below:

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

In order to map JavaML classes to OOML classes we need the following mappings:

ClassDeclaration $\mapsto$ Class
UnmodifiedClassDeclaration.Name $\mapsto$ Class.Identifier
FeildDeclaration $\mapsto$ VariableDeclaration
MethodDeclaration $\mapsto$ Method

Representing objects in C++ can is done using the following sections from the CppML representation:

```
<!ELEMENT Class ((%Declaration;)*,BaseSpecifier*,
                 TemplateArgument*)>
<!ATTLIST Class name CDATA>
<!ENTITY % Declaration "(Function|Variable)">
```

The mappings from CppML to OOML are:

Class $\mapsto$ Class
Class.name $\mapsto$ Class.Identifier
Function $\mapsto$ Method
Variable $\mapsto$ Variable

5 An Integrated Software Maintenance Environment

The Integrated Software Maintenance Environment (ISME) is an environment that facilitates collaborative software maintenance activities. The environment allows source code to be represented in the form of DOM trees, and provides means for CASE tools to register their services with specific entities of the source code domain model. For example a tool that computes cyclomatic complexity metrics can be registered as a service associated with the `MethodDeclaration` language entity. Hence, source code entity nodes are linked with CASE tool services registered in the Integrated Software Maintenance Environment. In this environment a variety of new software maintenance tools can be developed for specific programming languages using XML representations as discussed in the previous section. Another feature of this environment is the ability to integrate existing tools by making use of these common XML representations. The tool integration is taken a step further by creating a distributed service manager module that allows inputs and outputs for every tool in the environment. These services are aware of all other services in the environment and are able to communicate with other services in order to perform more complex tasks via a control and data integration module. Services can be local or distributed, allowing tool developers to share the ISME environment and the corresponding registered CASE tools with other developers in a collaborative manner. An XML schema can be used to model and represent input and output data for the registered CASE tools.

The features of ISME can be grouped in the following three distinct categories:

- **Data Integration** deals with program representations and how they can be used to enable tools to communicate.
- **Control Integration** deals with making tools available as distributed services.
- **Repository Services** deals with the persistency and sharing of the processed data.

5.1 Data Integration

The term data integration refers to the features of the ISME environment that enable new maintenance tools to be built using common program representations and allow existing tools to communicate and exchange data. When talking about data integration it is important to make the distinction between two separate concepts: Source Domain Model integration and Analysis Domain Model integration. In Source Domain Model integration there is a need to use a common representation that all registered tools can use as input for their software maintenance tasks. Similarly, in Analysis Domain Model there is a need to create representations so that analysis results from registered tools can be modeled and automatically used as input to other tools.

By having such representations available it becomes possible to perform a variety of maintenance tasks. First of all, language-specific tools can be developed using the XML representation for that language. When tools use a common representation, it is easier to compare software maintenance and analysis algorithms without having to worry about the details of parsing the source files directly. Secondly, language-independent tools can be developed that perform generic analysis for a variety of sources. Thirdly, the language-specific and higher-level languages can be used to exchange data between existing tools. Finally, it is easier to create maintenance-specific representations that extend the language-specific representations to facilitate for exchange of maintenance results.

5.2 Control Integration

The term control integration encapsulates all the features that the ISME offers for creating and using distributed services out of software maintenance tools. In ISME, a service corresponds to a CASE tool that performs a specific task. In order for a service to become part of the ISME environment we need to describe its functionality, its input and output. In this section we explain how services can be created and localized, how they can register/deregister with the environment, how they can be configured dynamically and how they can be invoked as part of a more complex task. In our attempt to reuse existing technologies some of the concepts presented in this section are based on the Jini architecture[Jini] proposed by Sun Microsystems.

Before we proceed to the details of the services we present the key features of a service in the ISME environment namely, services that are distributed, dynamic, secure,

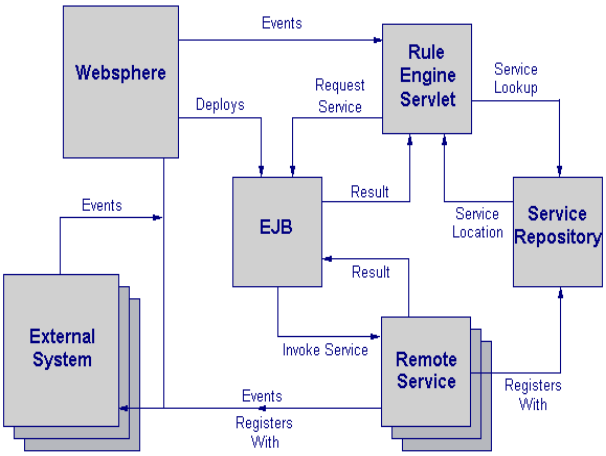

Figure 2. Architecture for service integration using the ECA paradigm.

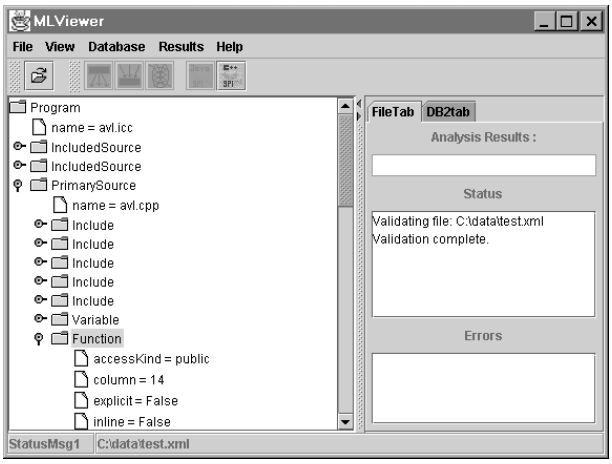

Figure 3. Screen shot of the prototype ISME.

and easy to use and to integrate. Distributed means that services (i.e. CASE tools) can exist in any machine on the network. Users are able to select the right service by examining their descriptions and then provide the input and receive the output. Dynamic means that services can be added and removed to and from the ISME environment at any time without having to stop and restart the whole environment. In addition, the services can be configured dynamically to accommodate for updated features and new security requirements. Secure, means that features are implemented to make services available to a selected group of users based on access control lists. Finally, the ease of use and ease of integration can be accomplished by using the Event Condition Action (ECA) paradigm. Using ECA it will be possible to define transactions that will involve the combined use of many services [Mylo96]. Service integration enables us to combine existing services in order to define new and more complex ones. To provide support for this, ISME uses transactions and events as specified in the Jini architecture, as well as the Event-Condition-Action (ECA) paradigm as presented in [Gregory00]. Jini specifies a transaction protocol in which a series of operations within one or multiple services can be wrapped in a single transaction. The proposed service integration architecture is depicted in Fig.2.

5.3 Repository Services

The need to perform maintenance tasks in large and complex software systems imposes more requirements on the environment in which these maintenance tasks are to be performed. The first and most important one, is that of efficient storage. Efficient in this context encapsulates the need for large amounts of storage, as well as techniques to access and update the data in an easy and fast manner. The second requirement is that of version control. It is necessary to keep track of different versions of software that has been analyzed in the same way it is done for software that is developed. The third and final requirement is that of shared access. The nature of the ISME environment will allow multiple users to access and work on the same data. A safe way of sharing the same data is of great importance.

In the ISME environment most of the data exists either in plain text format or in XML format. A variety of services for storing the data are provided within the ISME. These services can exist locally or they can be accessed through the network as previously described. The first option is to use plain text files with some versioning software like RCS [RCS]. Another approach is to take advantage of the efficiency and scalability that database management systems offer. Using a database like DB2 with the XML Extender [DB2XML] software we are able to store and index entire XML documents in a relational database which offers persistent storage. This approach is very efficient for large documents that need to be searched and updated frequently.

6 Prototype

A prototype environment was built using the Java programming language. The reasons for selecting Java are numerous. First of all, the prototype is able to run on a variety of platforms for which a Java virtual machine has been developed. Secondly, integration with distributed technologies such as Jini will be easier. Finally, the majority of the supporting technologies are readily available for Java. By supporting technologies we refer to technologies such as XML APIs, DTD viewers and XSLT tools to name a few.

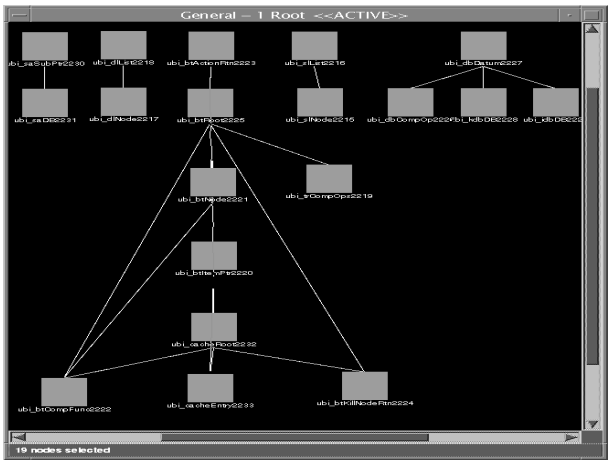

Figure 4. Invocation of the Rigi tool from ISME.

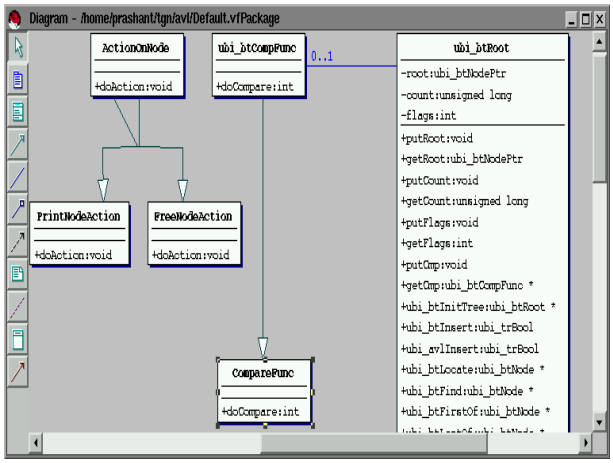

Figure 5. Invocation of the Together C++ tool from ISME.

The core part of the prototype is the XML tree viewer as shown in Fig. 3. The prototype is able to read JavaML, CppML and OOML representations from source files or from a DB2 Universal database for which there exists an XML to DB2 schema translator. Once a representation is loaded, the user is able to invoke tools that are registered to operate on the current representation. All tools are registered in a central repository and for each tool there exists a description and the input and output format. In addition, each tool can be registered to handle specific events depending on what part of the tree these events originated from. All the tools that are registered in the ISME prototype are linked to buttons on the toolbar at the top of the screen. In Fig. 3, a CppML representation of a source program is loaded, and one of the five buttons on the toolbar (left top) that represent registered tools is enabled. Each button corresponds to a different CASE tool registered in the ISME environment and can be active for the current source code entity selection. This means that only one tool is registered to handle this type of trees. In this case the tool maps the CppML representation to the higher level OOML. Once a tool is invoked the results are displayed in the right panel inside a textbox. In the current prototype, there is support for storing the analysis results as an XML file. This storing is achieved by using a very simple DTD as shown below. This DTD enables the user to save the analysis type and results associated with any element of the tree.

```
<!ELEMENT Results (Analysis*)>
<!ATTLIST Results SourceFile CDATA #IM-
PLIED>
<!ELEMENT Analysis EMPTY>
<!ATTLIST Analysis
Type CDATA #REQUIRED
AppliesTo CDATA #REQUIRED
Result CDATA #REQUIRED>
```

In the following subsections we illustrate five simple services currently registered with the prototype ISME environment. The first three services correspond to tools that compute the *fan-in*, *fan-out*, and *McCabe* metric. The fourth and fifth services correspond to the Rigi and Together visualization tools.

6.1 Registered Tools

Simple tools that calculate software metrics were developed to demonstrate how the ISME environment facilitates software maintenance. These tools were developed to operate on the OOML representations and therefore they can be used to compute metrics for both Java and C++ source files. Once a source file has been parsed and the corresponding representation (JavaML or CppML) has been generated then the mapping tool is used to generate the OOML representation. All the information the tools need is available in the OOML representation and the developer saves time by implementing the tools only once. It must be noted that not all tools will be able to operate at the OOML level. The ones described here were intentionally selected to demonstrate this point.

Fan-in

For any given function, this tool computes the total number of functions in the program that call the currently selected function. This can be simply computed by counting the number of MethodCall elements (with the correct identifier) in the OOML representation. In addition we define the fan-in of a Class to be the sum of all the method fan-ins contained within the Class. Similarly we can define fanin for

source elements. All this means is that the fanin tool is able to perform it analysis on `Method`, `Class` and `Source` elements. Inside the prototype ISME once a OOML representation has been loaded and the user selects an appropriate node then the fanin tool on the toolbar becomes enabled.

Fan-out

Fanout is very similar to how fan-in works. However in this case fanout represents the number of calls to other functions that originate from a given function. As before, we can define the fanout metric for Classes and Sources. The fanout tool is also registered to handle events that originate from `Method`, `Class` and `Source` elements.

McCabe Cyclomatic Complexity

The Cyclomatic Complexity or McCabe function as it is more commonly known computes a number that describes how complex the given function is. To calculate this metric the possible execution paths for the function are determined by examining all the conditional statements and their interleavings. The result is defined to be the number of possible paths - 1. It should be noted that the Cyclomatic Complexity tool in ISME is registered and active only with `Method-Declaration` elements and not with `File` or `Class-Declaration` source code domain model entities.

Visualization Tools

Another category of tools we have integrated with the environment deals with visualization tools. In particular we have integrated in the ISME the *Rigi* and the *Together C++* tools. The Rigi tool is active for the `Program` entities, while the Together C++ tool is activated for the `File` and `Program` entities. In Fig.4 a Rigi invocation session for use relations between `ClassDeclaration` objects is depicted. Similarly, in Fig.5 an invocation of the Together tool depicting the object model of a `Program` is illustrated. New tools can be added as DTD schema extensions by specifying their name and their invocation signature.

7 Conclusion

Program representation plays an important role on building tools that facilitate software analysis and software maintenance. One of the most popular representations is the Abstract Syntax Tree. However, building complete Abstract Syntax Trees (AST) for program analysis purposes requires not only a full parser but also customized semantic actions to be added to this programming language parser so that the AST can be easily analyzed and traversed by CASE tools. In this paper, we have presented an alternative to building Abstract Syntax Trees as a program representation scheme is to define a language model in terms of a DTD and automatically annotate source text with XML tags. An XML parser is used to parse the XML annotated source code and create a DOM tree that corresponds to an annotated AST. The benefit of using XML in this context is that the corresponding tree conforms with the language domain model as defined in the corresponding language DTD. Moreover, the DOM tree can be easily traversed and analyzed by using Java or C++ libraries that are publicly available.

CASE tools can be associated with specific language DTD entities and be activated only for these specific entities. In this context we have developed a prototype control integration architecture, where various CASE tools can be invoked by the unified interface. The XML annotated source code, its corresponding DOM tree, and the registered tools form an Integrated Software Maintenance Environment (ISME) by analogy to an Integrated Development Environment (IDE).

On-going and future work involves extending the current prototype environment to support representations for languages like Pascal, Cobol and PLIX. Finally, new tools can be added to the environment and linked using the ECA control integration paradigm. The prototype has been developed at IBM Canada, Center for Avdanced Studies, and is being linked for further evaluation with Visual Age for Java and Visual Age C++ compilers.

References

[Aho86] A. V. Aho, R. Sethi, and J.D. Ullman. *Compiler Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[Devanbu96] P.T. Devanbu, D. S. Rosenblum, A.L. Wolf. ”Generating Testing and Analysis Tools with Aria”, ACM Transactions on Software Engineering and Methodology, vol 5 no. 1, January 1996

[Datrix] BELL Canada, Datrix Group, ”Abstract Semantic Graph Reference Manual”, Version 1.3

[Rigi] University of Victoria, ”Rigi” at URL: http://rigi.uvic.ca, May 2000

[Holt98] R. Holt. ”An Introduction to TA: The Tupple-Attribute Language” at URL: http://plg.uwaterloo.ca/ holt/papers/ta.html, November 1998

[Bradley98] N. Bradley, ”The XML Companion”, Addison-Wesley, 1998

[W3CXML] World Wide Web Consortium, ”Extensible Markup Language (XML)” at URL: http://www.w3.org/XML, September 1999

[DOMspec] World Wide Web Consortium, ”Document Object Model (DOM) Level 1 Specification Version 1.0” at URL: http://www.w3.org/DOM, October 1998

[SAXspec] Megginson Technologie

[Alphawors] IBM Corporation, ”Alphaworks” at URL: http://www.alphaworks.ibm.com, May 2000

[JavaCC] Sun Microsystems, ”JavaCC The Parser Generator”, at URL: http://www.metamamta.com/JavaCC, May 2000

[VACpp] IMB Corporation, ”VisualAge for C++” at URL: http://www.ibm.com/software/ad/vacpp, September 1999

[XSLT] World Wide Web Consortium, ”XSL Transformations (XSLT) Version 1.0” at URL: http://www.w3.org/TR/xslt, November 1998

[DB2XML] IBM Corporation, ”DB2 XML Extender” at URL: http://www.ibm.com /software/data/db2/extenders/xmlext/

[Jini] Sun Microsystems, ”Jini Architecture Specification”, edition 1.0.1, November 1999

[Gregory00] R. Gregory, K. Kontogiannis, ”Customizable Service Integration in Web-enabled Environments”s, ”SAX 2.0: The Simple API for XML” at URL: http://www.meginson.com/SAX/, May 2000 , To be published, April 2000

[Mylo96] J. Mylopoulos, A. Gal, K. Kontogiannis, M. Stanley. A Generic Integration Architecture for Cooperative Information Systems. In *Proceedings of Co-operative Information Systems'96*. Brussels, Belgium, 1996

[Pressman97] Pressman, R., Software Engineering: A Practitioner's Approach

[RCS] GNUProject, ”Revision Control System (RCS)”, at URL: http://www.gnu.org/software/rcs/rcs.html

[Ferrante87] J. Ferrante, K. J. Ottenstein and J. D. Warren. ”The program dependence graph and its use in optimization”, ACM Transactions on Programming Languages and Systems, vol 9 no. 3, July 1987

[Holt00] R. C. Holt, A. Winter, A. Schurr. ”GXL: Toward a Standard Exchange Format”, To appear in WCRE 2000: Working Conference in Reverse Engineering, Brisbane, Australia, November 2000

[Refine] Reasoning Inc. ”CBMS WhitePaper” at URL: http://www.reasoning.com/tech/tech.html