Design and Implementation of an Integrated Software Maintenance Environment

by

Evangelos Mamas

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2000

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Software development methods and techniques have evolved greatly over the past few decades in terms of new programming paradigms, languages and support tools. This evolution enables the specification, design, implementation, and testing of large complex software systems. However, the existence of such large systems introduces the need for continuous software maintenance. This is an issue of particular importance in the area of legacy systems. Over the past years, highly specialized computer-aided software engineering (CASE) tools have been developed and used to perform maintenance and analysis of specific languages and even specific projects. After the development of numerous such tools for different maintenance tasks it has become apparent that there is a need to integrate all these tools into one collaborative environment. Tool integration allows for more complex maintenance tasks to be accomplished by combining the benefits of the individual tools. In our work, we propose an Integrated Software Maintenance Environment as a stepping-stone towards a solution to the software maintenance challenge. This thesis proposes a program representation scheme that is based on the XML markup language and a control integration architecture that allows the various CASE tools to be integrated in one collaborative environment. Moreover, this thesis discusses a method for inferring such program representations from the language grammar and how this method was used to create representations for the C++ and Java programming languages. Generic program representations for categories of languages can be built from specific language representations. A repository for persistent storage of software artifacts that uses the IBM universal database DB2 is presented. Finally, a collection of CASE tools for C++ and Java are integrated using the proposed representation schemes and an integration architecture that uses the Event-Condition-Action paradigm.

Acknowledgements

I cannot thank enough professor Kostas Kontogiannis for believing in me and for helping me with his valuable advice, encouragement and financial support throughout the course of my studies at University of Waterloo. THANKS Kostas!

I would like to thank my readers Ric Holt and Gordon Agnew for their valuable comments and suggestions. Special thanks also go to the IBM Center for Advanced Studies for providing funding, resources and a wonderful research environment which I was lucky enough to enjoy for the past few years. I would also like to thank all the members of the SWEN lab for their help, support and for putting up with me in general.

Very special thanks go to Angela, Patla, Deep, Jenny, Richard, Peter, Ladan and Kamran who made my stay in Waterloo a very fun and memorable experience. Last, but not least, I would like to thank my parents for their encouragement and support that made all this possible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Description

The importance of software maintenance has become increasingly apparent as large industrial software systems continue to evolve at a rapid pace. Numerous large software systems written in older programming languages using obsolete programming paradigms are currently in operation in many large organizations. These software systems are commonly referred to as "legacy systems" or in a more general way as "legacy software". Legacy systems can be identified by numerous characteristics. Some typical legacy systems have been developed at least a decade ago, they are mission critical, they have undergone continuous maintenance, they are large (>500KLOC), and their current architecture is not fully understood. Legacy systems undergo continuous evolution in order to implement new features, fix errors, and meet new functional and non-functional requirements. This task becomes very hard since there are no design documents for the whole system, and few if any architects of the original system still remain in the organization. Under these circumstances, introducing new features and maintaining existing ones is very likely to introduce more complexity into the legacy system. On the contrary, replacing these legacy

systems with completely new systems introduces more challenges. The new software will have most likely, to conform and re-implement accurately the existing business process model, and to provide similar if not enhanced functionality that the existing legacy system provides. Moreover, there are associated risks with complete re-development and replacement of a legacy system. These include, possible downtime when the new system will be introduced, incompatibility with existing databases or user interfaces, introduction of new faults, and learning time for the maintenance group. Computer-Aided Software Engineering (CASE) tools may assist in software maintenance tasks and will enable the transformation of legacy systems into systems that are based on modern programming techniques and languages.

In response to the problems of the legacy systems described above, numerous tools have been developed to assist developers in software maintenance tasks. These tools perform a variety of tasks such as computing software metrics, extracting control flow and data dependency graphs, visualizing the architecture, and recovering the design. Most of these tools address specific maintenance tasks such as identification of error prone code, data and control flow analysis, code restructuring, and redocumentation. The data model that each tool uses to process and store data and its analysis results, relates to the specific maintenance task that the tool addresses. It is not uncommon for organizations to have maintenance tasks implemented by different tools that possibly work on different operating environments. Furthermore, even when these tools perform different maintenance tasks, some of their basic components are providing similar functionality. For example, a source code parser for a particular programming language, is something that almost every CASE tool has to implement in order to perform an analysis task. Given the variety of tools that are available, and their respective strengths and weaknesses, one would like to be able to exchange data and analysis results that each tool generates in order to perform more complex maintenance tasks. In addition, making existing tools available in a

collaborative software maintenance environment would greatly facilitate many integrated maintenance tasks.

Some of the most important problems that the software maintenance community is currently facing are those of *tool integration* and *tool generalization*. Tool integration refers to the need for existing tools to be able to exchange data and analysis results so that software analysis techniques can be compared and combined to address more complex software maintenance tasks. Tool generalization refers to the requirements of being able to develop generic maintenance tools that can perform the most common maintenance tasks in a variety of similar programming language domains (i.e., imperative, object-oriented).

1.2 Objectives and Definitions

Building a tool that single-handedly addresses many and possibly diverse maintenance tasks is not possible. In order to address a wide variety of maintenance tasks an Integrated Software Maintenance Environment is required. Such an environment will allow for various CASE tools to collaborate so that maintenance activities can be facilitated by a group of, possibly distributed, software engineers. It has been established in the related literature [14] that tool integration can be divided into two major parts: data integration and control integration. Similarly, tool generalization can be divided into specific and generic program representations. Below we provide definitions for some of the key concepts discussed in this thesis.

- Data integration addresses the problem of representing and mapping data between various tools.

- Control integration addresses the area of distributed software tools that can collaborate in a network.

- Specific representations deal with representing information at the source code level.
- Generic representations deal with information at a higher level of abstraction that is language independent.

In our work, we attempt to solve these problems by proposing methodologies to achieve the following objectives:

1. Create source code representations for specific programming language domains that provide a common basis for developing new maintenance tools and facilitating the exchange of data between such tools.
2. Generalize the language-specific source code representations to higher-level program representations that allow the implementation of generic maintenance tools for various language domains.
3. Provide an architecture that facilitates control integration among existing tools. This enables the invocation and interaction between tools so that complex maintenance tasks can be addressed.
4. Propose an environment that supports software maintenance by CASE tool integration. By accomplishing all the previous objectives it will become possible to define an environment that uses the program representations as the basis for tool development, integration and generalization.

1.3 Contributions

This thesis reports on the design and implementation of an Integrated Software Maintenance Environment (ISME). An ISME is an extensible environment for developing software maintenance tools and integrating existing maintenance tools. What makes ISME

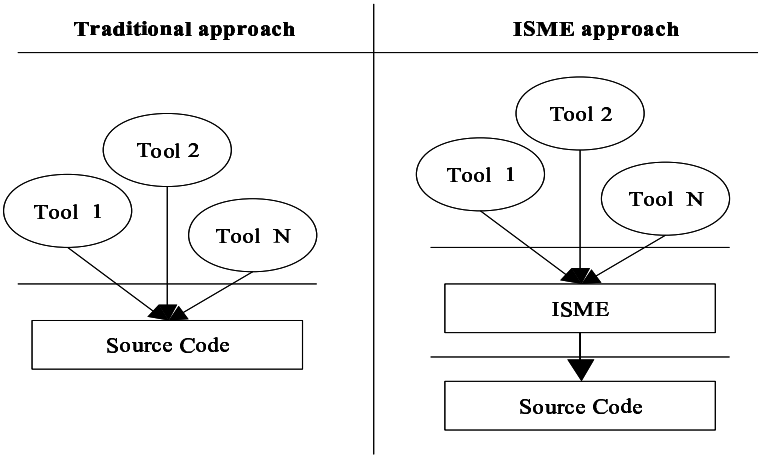

Figure 1.1: ISME Abstraction layer.

distinct from existing tools is that it does not target a specific programming language and it is not restricted to a set of predetermined maintenance tasks. Instead, it allows the users to define what language domain they want to analyze and what specific maintenance task they want to perform. This property of ISME can be visualized as an extra layer of abstraction between the source code and the maintenance tools as shown in Figure 1.1. To create such an abstraction layer it is necessary to create formalisms for representing programming languages and software analysis results. These formalisms are the core of the ISME environment and they are used as the basis for implementing software maintenance and analysis tools.

The main contributions of this thesis are:

1. The introduction of techniques to represent grammars for programming languages with Document Type Definition Documents (DTD). This enables us to represent source code information as XML documents.

2. The introduction of XML based representations for the Java and C++ program-

ming languages called JavaML and CppML respectively. In addition, a sample generic representation for the category of object-oriented languages called OOML is presented.

3. The implementation of tools that generate the JavaML and CppML representations from source code. The Java tool is built with a parser generator called JavaCC [40]. The C++ tool is based on the VisualAge C++ [35] IDE. Tools that generate the OOML representation from JavaML and CppML are also implemented.

4. The introduction of control and data integration techniques that will allow the development of a distributed environment for software maintenance tasks.

5. The implementation of a prototype ISME environment that allows software maintenance tasks to be developed for the Java and C++ programming languages using the JavaML, CppML and OOML representations. The prototype also demonstrates how the relational database DB2 [21] can be used as a repository for software artifacts.

1.4 Thesis Organization

Chapter 2 provides an overview of tools and technologies that are related to our work. Chapter 3 describes the details of modeling the different programming language domains. Chapter 4 introduces the proposed ISME environment, its architecture, data and control integration, and repository features. Chapter 5 describes the prototype ISME environment and the feasibility studies performed. Finally, Chapter 6 presents our conclusions and avenues for future work.

Chapter 2

Related Work

2.1 Introduction

In this chapter, we present related work in the areas of source code representation, and tool integration. The chapter is divided in four major sections. The Program Representations section, discusses approaches for representing information extracted from the source code level. The Reengineering Frameworks section, describes frameworks for generic analysis of software programs. The Modeling Environments section, presents generic technologies for modeling data and meta-data. The Tool Integration section, discusses frameworks for integrating software tools in distributed environments. Finally, the last section summarizes the related work presented in this chapter.

2.2 Program Representations

The area of program representation deals with techniques for representing source code information at a higher level of abstraction than source code text, and in a structured way, suitable for algorithmic manipulation with computers. Much work has already been done

in the area of program representation. In the following sections we present some program representations formalisms that have been used extensively, by numerous researchers, for source code analysis and software re-engineering.

2.2.1 Abstract Syntax Trees

An Abstract Syntax Tree, or AST in short, is a tree structure that represents the syntactic information contained in the source code [5]. Every node of the tree is an element of the language. The non-leaf nodes represent operators, while the leaf nodes represent operands. What is missing from the AST is syntactic information that is implicit from the structure of the AST. Punctuation tokens are a typical example of information not included in an AST. The AST notation is a commonly used structure in compilers for representing the source code internally in order to analyze it, optimize it, and generate binary code. To construct an AST it is necessary to first obtain the concrete syntax tree by using the grammar of the language, and then remove all the purely syntactic tokens such as parenthesis or semicolons. This implies that for every grammar there is one concrete syntax tree and many possible abstract syntax trees. In Figure 2.1, two examples of some source code and the corresponding ASTs are shown.

2.2.2 Program Dependence Graph

The Program Dependence Graph, or PDG in short, is a graph that combines control dependence and data dependence information. In a PDG, the nodes represent statements, expressions or regions of code, and the edges represent control or data values passed from one expression to another as well as control conditions that influence the order of execution. In contrast to an AST that represents the source code information, the PDG is used to represent information computed from the source code. In other words, the

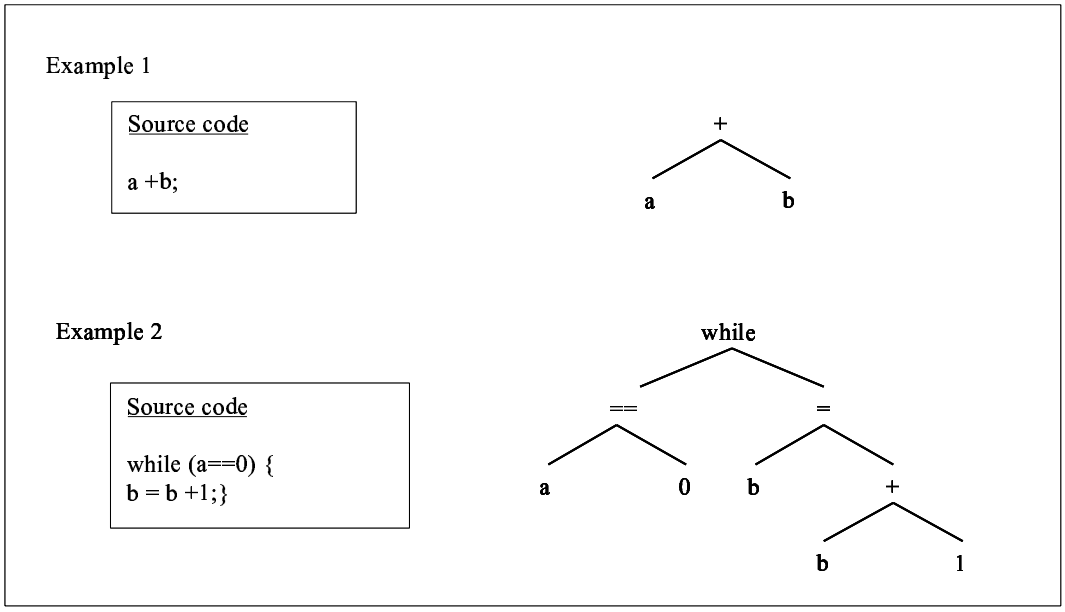

Figure 2.1: Abstract Syntax Tree examples.

dependencies expressed in a PDG are a result of processing the source code information directly or processing its corresponding AST. The control and data dependencies stored in a PDG are commonly used for software testing purposes as well as static and dynamic slicing techniques [33]. The PDG is also very useful in developing code optimization algorithms and in identifying parts of a program that can be made parallel [28]. The Control Flow Graph, or CFG in short, encodes control flow information [6]. In a CFG, the nodes represent statements and the edges represent transfer of control between statements. In order to construct a PDG it is necessary to use the control flow information, which is available from the CFG.

In Figure 2.2, we show the source code and the CFG for a small example found in [33], and in Figure 2.3 we show its corresponding PDG. For visual clarity we split the PDG into two subgraphs: control dependence subgraph (CDS) and data dependence subgraph (DDS). There are three types of nodes in a PDG graph: Statement nodes, which represent

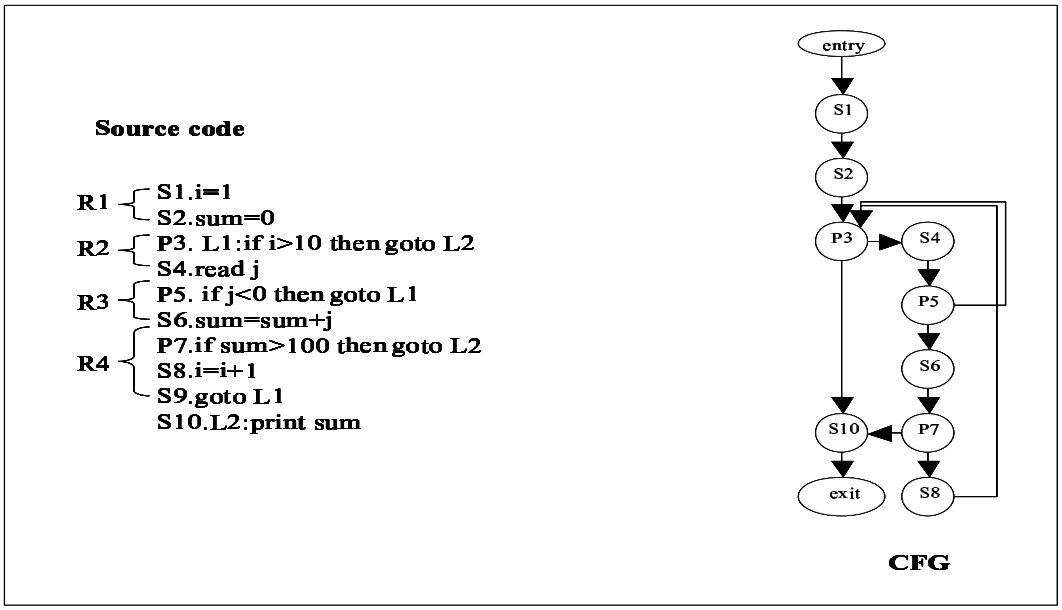

Figure 2.2: Source code and Control Flow Graph.

individual statements, are drawn as circles. Region nodes, which group statements with the same control dependencies, are drawn as circles. Predicate nodes, from which two edges can originate, are drawn as squares. The edges represent the control dependencies. In the CDS edges represent control dependencies while in the DDS edges represent data dependencies. For example, in Figure 2.3 in the CDS subgraph, both statement node S6 and predicate node P7 are control dependent on P5. Since S6 and P7 form region R3, the control dependency is illustrated as an edge from P5 to R3 and from R3 to nodes S6 and P7 in turn. In the DDS subgraph the edge from S2 to S6 indicates that S2 defines *sum* and S6 uses *sum*.

2.2.3 AsFix

AsFix is a parse tree representation for ASF+SDF terms and modules[47]. ASF+SDF is an algebraic specification formalism for describing the syntax and semantics of program-

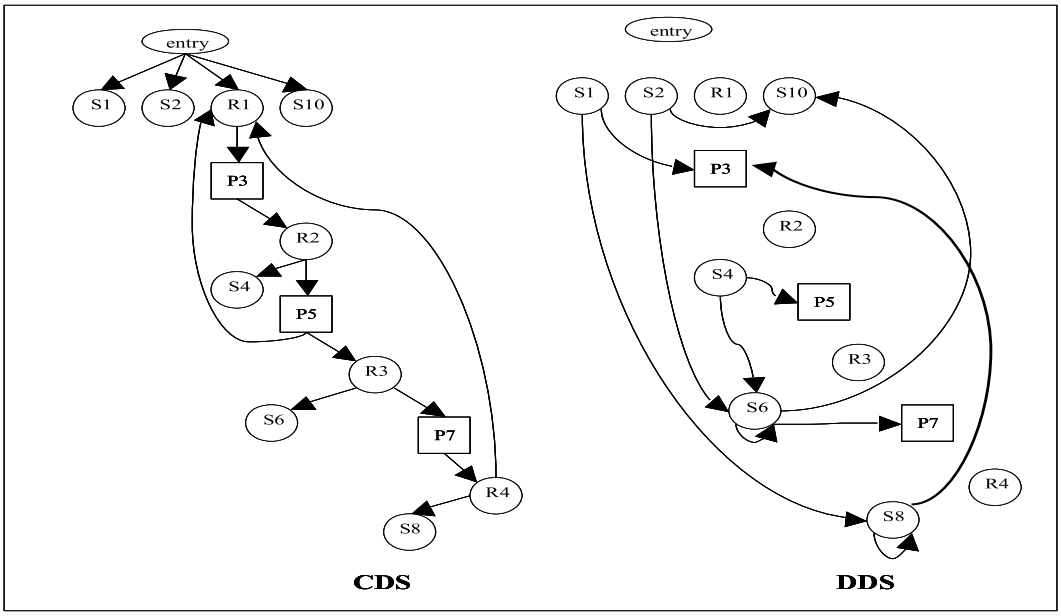

Figure 2.3: Control Dependence Subgraph and Data Dependence Subgraph.

ming languages. 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. According to its design goal, the ATerms format should be independent of any specific tool, but it should be capable of representing all data that is exchanged between tools. Here are some sample ATerms found in [47]:

constant abc

numeral 123

literal "abc" "123"

list [] [1, "abc", 3] [1, 2, [3,4], 5]

functions f("abc")

annotation f(123){color("red"), path([0,2,1])}

The AsFix representation contains all the information from the source code, including even white space and comments. One important concept of AsFix is that it is a self-descriptive representation. This means that every application of a context-free rule contains a copy of the actual rule. The following example, demonstrates this self-descriptive property of the AsFix representation. Consider the following set of function symbols:

`prod(L)` represents the production rule L

`appl(T1,T2)` represents the application of production rule T1, to the arguments T2

`l(L)` represents the literal L

`w(L)` represents whitespace L

Using the above symbols one can represent parse trees with ATerms. Consider the following production rules used to define Boolean expressions:

```
true -> Bool
false -> Bool
Bool and Bool -> Bool
```

The sentence "true" can be expressed by the following tree:

```
appl( prod("true -> Bool"), [l("true")] )
```

In a more complex example the parse tree for the sentence "true and true" is:

```
appl( prod("Bool and Bool -> Bool"),
   [appl(prod("true -> Bool"), [l("true")]),
   w(" "), l("and"), w(" "),
```

```
appl(prod("true -> Bool"), [l("true")])
])
```

2.2.4 Abstract Semantic Graph

Abstract Semantic Graph, or ASG in short, is similar to the AST we described previously but it contains many kinds of annotations. Just like a graph, an ASG has nodes and edges. Both the nodes and the edges are typed and edges also have their own properties. In [26] the ASG is defined as an AST with embedded semantic information. Specifically, in an AST, a reference to an entity is represented by an edge pointing to a simple leaf node that holds the name of the entity. In an ASG, a reference is represented by an edge pointing to the root of the (shared) subgraph in the ASG that represents the declaration of the entity [26].

The ASG is the representation used by the tool Datrix[23] which is currently used to model C++ and Java source code. In the Datrix tool, the ASG is annotated with a variety of information gathered from the source code at parse time (e.g. scoping information). In Figure 2.4, a small example of an AST, its ASG and its corresponding source code are illustrated.

2.2.5 Rigi Standard Format

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 meta-data. This allows RSF to use the same format to represent program information for a variety of programming languages without any changes in the format. The syntax of RSF is based on triplets. Sequences of these triplets are stored in self-contained files. Currently RSF is the base

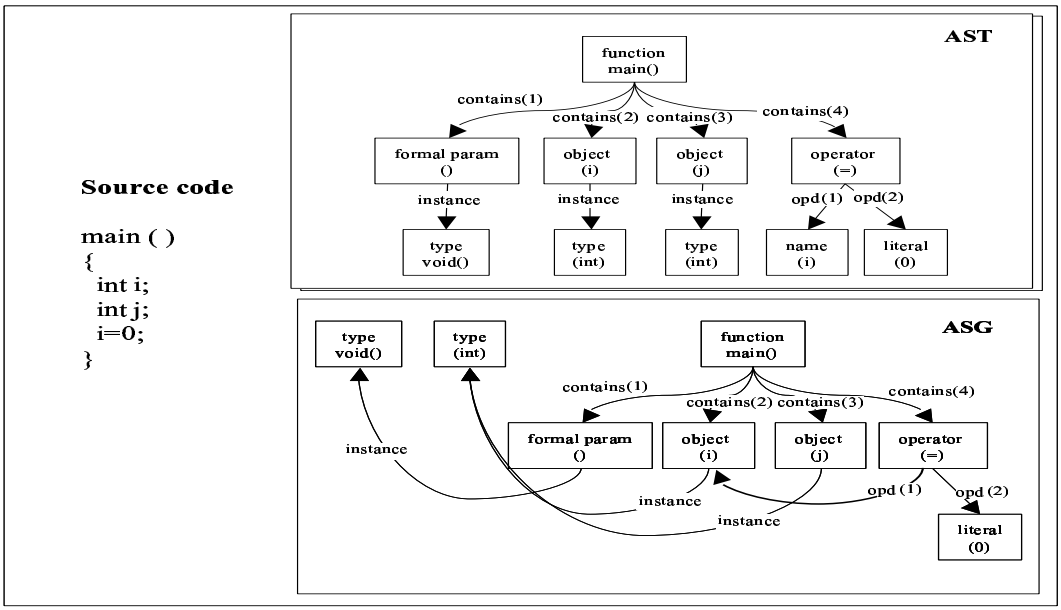

Figure 2.4: Abstract Semantic Graph example.

format for a reverse engineering tool called Rigi [46]. In Rigi, RSF is used to represent source code information for C programs. A small example of the RSF format is presented for the following source code:

```
main(){
 int x,y;
 x=1;
 printf("Done");
}
```

Corresponding RSF output:

```
type main Function
file main main.c
lineno main 1
call main printf
```

2.2.6 Tuple-Attribute Language

The Tuple-Attribute Language, or TA in short, is a language designed to represent graph information [34]. This information includes nodes, edges and any attributes the nodes and 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 as a "data interchange" format. The TA language is composed from two sub-languages, Tuple and Attribute. Furthermore, TA has two levels of abstraction, the Fact level and the Scheme level. Below we describe each sub-language as well as the Scheme level in more detail.

Tuple sub-language

The syntax of the Tuple sub-language allows for a sequence of triplets of information similar to the RSF format discussed previously. The semantic interpretation of every triplet is that of an edge in a typed graph. Multiple edges (with different types) can be defined between two nodes. Figure 2.5 shows a small TA program and its corresponding graph as found in [34].

Attribute sub-language

The Attribute sub-language allows to record attribute information for every node and

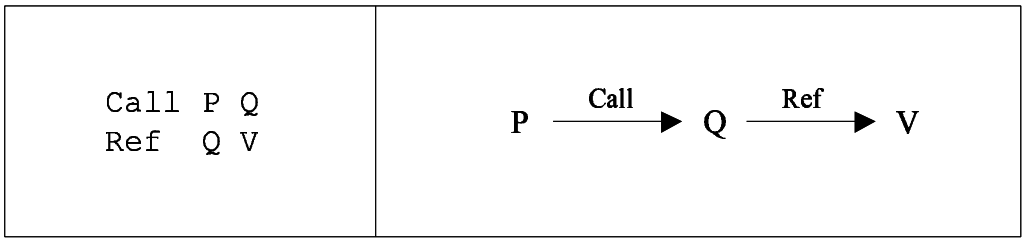

Figure 2.5: TA example.

edge in the graph. To specify an attribute for an entity, the entity of interest (node or edge) is followed by a left brace "{", then a list of items in the form "name=value" separated with white space, and finally followed by a right brace "}". For example, to describe the color and size of the node P we would write:

```
P { color=red size=100 }
```

Scheme Level of TA

This level of TA is used to describe the shape of the graph as opposed to the fact level that is used to describe the graph itself. The Scheme can be viewed as the meta level of the graph. It describes what edges and what attributes are allowed in the specific graph. For example the following triplet

```
Call Proc Proc
```

means that in the actual graph a Call edge can exist only between two Proc nodes. TA also allows for inheritance among entities with common characteristics. In Figure 2.6 we summarize the TA language and all its components.

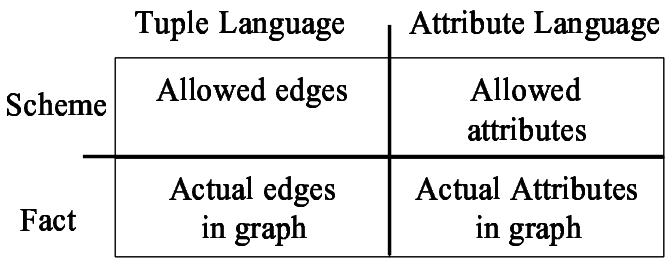

Figure 2.6: TA language overview.

2.2.7 Language Models for Object-Oriented Systems

The work on Language Models [11] deals with the problem of representing facts about Java and C++ programs. The representation of the facts is using the TA format previously described. Program facts can be represented at two different abstraction levels by using different schemas. These two levels are the Top-Level schemas and the Extracted Fact schemas.

The Top-Level schemas allow users to represent language-independent facts about programs and to perform generic analysis. At this abstraction level, there are three schemas defined: Dependency schema, UML Package Schema, and Subsystem schema. The Dependency schema is the most abstract schema and uses a single entity called *SourceElement* and a single relation called *dependsOn*. Some generic analysis can be performed using the Dependency schema. The UML Package schema, uses the UML package mechanism [10] to describe the architecture of a system using subsystems and modules. The Subsystem schema is used to divide large systems into groups by distinguishing between different types of *uses* relations.

The Extracted Fact schemas contain low level information extracted directly from the source code. The level of detail of the extracted facts is chosen to facilitate the reconstruction of software architectures. Only facts about source code elements that have an effect outside of a single source file are recorded. Besides source-code information, the

schemas also store compile time dependencies. A Java and a C++ model have been defined and prototype extractors have been developed for both models.

2.3 Reengineering Frameworks

This section describes frameworks for performing software analysis tasks on a variety of programming languages. The frameworks described here have evolved out of the need for a single environment that will enable software engineers to perform various reengineering tasks in a more systematic way. By systematic we mean that the framework makes it easy for the user to perform similar analysis tasks on a variety of different sources.

2.3.1 Refine

Refine is a Code-base Management System that supports analysis and transformation of software programs [36]. The code-base of Refine is built on top of Abstract Syntax Trees (ASTs). Types of nodes in the AST are represented as objects classes. Relations between nodes are represented as attributes of the object classes (nodes). A proprietary API is available for manipulating the objects that compose the AST. The architecture of Refine is illustrated in Figure 2.7.

A Language gateway is used to convert the source code information from different languages to the internal AST representation. Once the code has been converted to the Refine code-base, custom analysis and transformation programs can be built. Language-neutral data structures are also provided for building control and data flow graphs. Users can use the APIs to implement their own analysis tools, or use the language-specific analyzers that Refine provides.

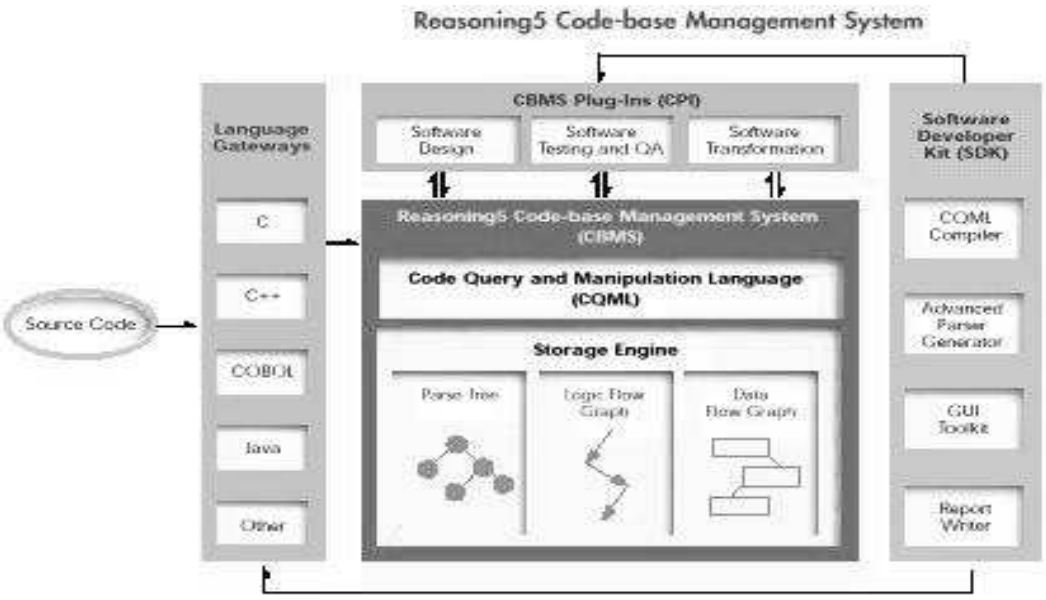

Figure 2.7: Refine Software Architecture.

2.3.2 Dali

Dali is a workbench that focuses on the integration of a variety of extraction, manipulation, analysis and presentation tools [37]. The first step when using Dali is to extract static and dynamic elements from the system under examination (View Extraction). The extraction process might involve the use of more than one tools. The extracted elements are stored in the repository which is implemented as an SQL database. The users are able to manipulate the extracted elements and create views to capture different properties of the system. The interaction between Dali and the users is accomplished with the use of a programmable command language called RCL that is part of the Rigi tool. Visualization of the views is accomplished by using the Rigi tool that also allows users to interact and manipulate the extracted facts through the views. The components and the activities of the Dali workbench are illustrated in Figure 2.8.

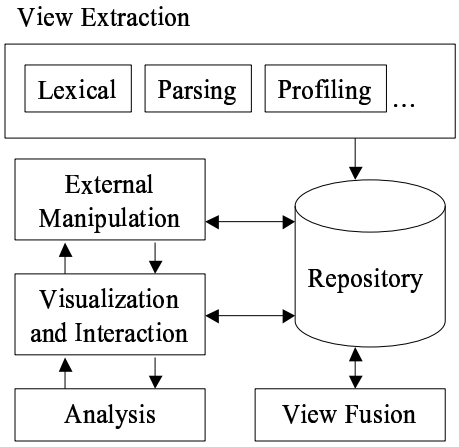

Figure 2.8: The Dali workbench

2.4 Modeling Environments

In this section, we look at some technologies that are available for modeling data and meta-data. The increasing amount of information made available to people through computers, has made it clear that there is a need for a way to model this data so that both humans and computers can easily process it. Furthermore, the increased amount of information gives rise to the need for a higher level of information (meta-information) that will be used to describe the actual information. An analogy to this problem is that of a library: We can think of the books inside the library as the information. As the library collects more and more books there is a need to archive and index these books in a fast and easy way. By creating an index for all the books, we actually generate more information based on the existing information. This is what we refer to as meta-data. Plain text files are great for storing content but fail to provide any context for the given content. One approach to dealing with this is to use structured languages. We will describe some commonly used structured languages and how they are used to store data and meta-data. We intentionally focus our overview on XML related technologies since

these are the technologies we use later on as the basis of our work.

2.4.1 Data

2.4.1.1 eXtensible Markup Language

XML is the acronym for eXtensible Markup Language, which has been developed by W3C (the World Wide Web Consortium). XML is a format for storing structured data intended for publishing or for exchange between different applications. XML derives from SGML (Standard Generalized Markup Language) and from HTML (HyperText Markup Language). SGML was initially proposed by Charles Goldfarb in the 1970's and was adopted by ISO in 1986 as an international standard for document markup. HTML was developed in 1992 as a language specific to Web pages. The disadvantages of SGML were that it was too hard to implement and one could argue that it was ahead of its time because it offered much more than what was needed at the time. With the evolution of the Web it became apparent that there was a need for a concise, compact and easy to use language to represent Web documents. HTML was designed for this very purpose. It is a compact language that cannot be easily extended but is practical for presenting information on the Web. The increasing amount of information available on the internet made it clear that the presentation and the actual contents of a document should be separated. A language that is more flexible than HTML was needed and W3C based the design of XML on their previous experience from SGML. What makes XML flexible is that it is a meta-language. This means that it is not a fixed language that is useful for describing predefined information. Instead, it allows the end user to specify custom tags and to create a logical structure based on those tags. Therefore, an XML document can be thought of as a self describing document in which the reader can interpret the tags and process them accordingly.

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 structure to be built. The physical structure allows components of the document to be stored separately inside or outside the document. Every element is defined by a start tag and an end tag. Between these tags, the contents of the element are listed. The contents could be other elements, raw text, or a combination of the two. Furthermore, every element can have many attributes that provide details about the content of the element. For example, to describe a book we can use 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. In Figure 2.9 we represent in XML the information about the book with the title "The story of my life", with ISBN 123456789, and with the two authors George Thomas and Tom Jones.

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

Figure 2.9: A simple XML example

The benefits of using a meta-language like XML are many. First of all, the document publishing applications can exploit it in order to develop better searching and indexing techniques. 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 to exchange data. A great deal of information on XML can be found in [12] and [16].

2.4.1.2 Document Type 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 defines all the elements that can be used, and describes how they can be used in relation to each other. This results in a hierarchical document. 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. An XML document is valid if its contents conform to the DTD specification. Therefore, we are able to enforce certain restrictions on the 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 used to define the structure of the XML document. The content of the elements is the basic block for building the structure. We can have empty elements, or elements that contain other elements, or text. A model group is used to describe what elements and text may occur within the current element. When only child elements are allowed then the element is defined to have element content. When text and elements are allowed then the element is said to have mixed content. Model groups are very flexible and they provide sequence and quantity control. The sequence control is accomplished by the combined use of sequences "," and choices "|". The use of parenthesis in conjunction with sequences and choices allows us to express very complex model groups. Quantity control is achieved with the use of quantity indicators. The following quantity indicators: "?", "+", "*" are used to express the fact that an element may appear: optionally, at

least once, or optionally and repeatedly. If there is no quantity indicator then every element must appear exactly once.

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 a 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.

In Figure 2.10 we present a sample DTD that can be used to enforce the logical structure of the example presented previously in the XML section in Figure 2.9.

2.4.1.3 XML Schema

XML Schema addresses the concept of validity of XML documents. As stated in [18], the XML schema language can be used to define, describe and catalogue XML vocabularies

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

Figure 2.10: A simple DTD example

for classes of XML documents. There is a need for a more expressive way of defining constraints specifications for XML documents than DTDs. It also important to be able to express these constraints using the XML language itself. This will allow the use of the same tools on both the XML documents and the specifications documents. XML Schema addresses the following issues [18]:

Structural Schemas

Structural Schemas provide a mechanism somewhat analogous to DTDs for constraining document structure (order, occurrence of elements, attributes), and address specific goals beyond DTD functionality including integration with namespaces, definition of incomplete constraints on the content of an element type, integration of structural schemas with primitive data types, and inheritance. Existing mechanisms use content models to specify part-of relations, while kind-of relations can only be denoted implicitly or informally. Making kind-of relations explicit would make both understanding and maintenance of these schemas easier.

Primitive and Complex Data Typing

Support for a wide range of primitive data types is available. These data types include byte, integer, float, double, date, time and strings. The primitive types can be used to create complex types for representing all possible information types. Lists and unions of

types are also supported.

Conformance

The relation of schemata to XML document instances, and obligations on schema-aware processors, is defined. The W3C working group has also defined a process for checking to see that the constraints expressed in a schema are obeyed in a document (schema-validation); the relationship between schema-validity and validity as defined in XML 1.0 is also defined.

It is expected that XML Schema will be used in a wide variety of applications such as publishing and syndication, electronic commerce transaction processing, supervisory control and data acquisition, traditional document authoring/editing governed by schema constraints, and meta-data interchange. The DTD example from Figure 2.10 in the previous section, can be expressed using XML Schema as illustrated in Figure 2.11.

2.4.2 Meta-data

2.4.2.1 Resource Description Framework

Resource Description Framework, or RDF in short, is a framework for expressing meta-data. It uses XML to describe Web resources, but the resources described can be of any type, XML or non-XML. RDF can be used in a number of different applications areas such as: resource discovery in search engines, describing the relationships of existing content on web pages, exchanging information among intelligent agents, content rating, and even describing the security policies on the web. It should be made clear that RDF does not specify vocabularies of descriptive elements. It specifies the mechanisms for defining elements as resources and for defining the relationships between resources.

RDF is used to make statements about resources. A statement says that a resource

```
<schema>
   <element name="book">
        <archetype order="sequence">
            <element ref="title" />
            <element ref="author" minOccurs='1' maxOccurs='*'/>
            <attribute name="ISBN" type="string" required="true"/>
        </archetype>
   </element>
   <element name="title">
        <archetype content="textOnly" type="string">
        </archetype>
   </element>
   <element name="author">
        <archetype content="textOnly" type="string">
        </archetype>
   </element>
</schema>
```

Figure 2.11: A simple XML-SCHEMA example

has one or more properties. Every property has a value and a type. The value can be a string, number or even another resource. A resource can be anything that has a URI (Unified Resource Identifier). Every statement can be seen as a triplet containing a resource, a property type and a property value. If, for example, we wanted to say that this document, which can be found at http://www.swen.uwaterloo.ca/ẽvan/thesis.html, has an author and that the author is Evan Mamas then we can visually present this information as shown in Figure 2.12 below.

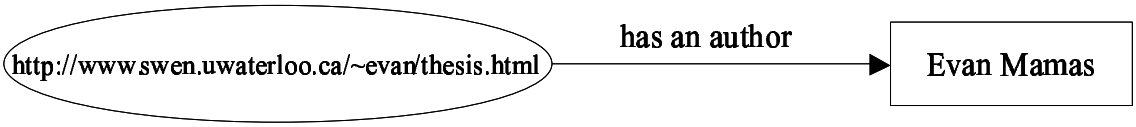

Figure 2.12: Visual representation of an RDF statement.

In RDF syntax the previous example would be expressed as follows:

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <rdf:Description about="http://www.swen.uwaterloo.ca/~evan/thesis.html">
      <Author>Evan Mamas</Author>
   </rdf:Description>
</rdf:RDF>
```

RDF allows far more expressive statements to be made when all its features are used. More information about RDF, its current status and its uses can be found in [17] and [32].

2.4.2.2 Meta Content Framework

Meta Content Framework, or MCF in short, is a way to describe files or collections of information. It is a meta-data model proposed by Netscape Communications [13]. A piece of information can be termed as an object. An object can have one or more properties. There exist property types and actual properties. The types can be thought of as property templates while the actual properties are the instantiations. So far, MCF sounds similar to XML. For example the author of a document could be represented in XML as:

```
<document author="James Brown">
```

In MCF all attributes are represented by elements so the previous XML example would look as follows:

```
<document>
    <author>James Brown</author>
</document>
```

However, MCF allows a property to be an object by itself. Instead of seeing the author property as a sub-object of the document object we have to see the author property as a separate object which can be linked directly with other objects. More complex relationships can be expressed by using the keywords: *unit*, *id*, *domain* and *range*. These relationships can be demonstrated using a Direct Linked Graph (DLG) in which nodes are properties and the links express the concept of properties pointing to other properties. The DLG for the above example is illustrated in Figure 2.13.

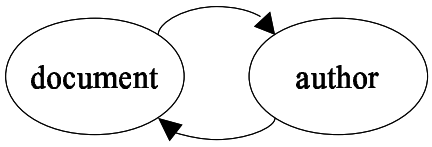

Figure 2.13: Direct Linked Graph example.

2.4.2.3 Extensible Information Brokers

Extensible Information Brokers, or XIB in short, is a framework that supports the integration and querying of diverse information sources. A number of information services exist on the Web today like Yahoo, Lycos and Altavista to name a few. Currently, searching for information requires the user to manually query each of these services and manually combine the query results. New search engines that tackle this problem, like Meta-Crawler, are making their appearance on the Internet. The reason for this tedious process is that every information service is built on a different proprietary format. The first solution that comes to mind is to have all the information services use a common for-

mat. This appears to be the trend for the next generation of information services. There still is a need though to automatically integrate the existing information services and that is exactly what XIB tries to achieve with the use of brokers. Brokers can be thought of as heterogeneous information services that perform a variety of tasks on behalf of their users. These brokers must be flexible and extensible. They must be easy to customize and to evolve, since the environment in which they will be used, will change quickly and drastically. XIB uses a service description language called XIBL as a communication language between various services. XIBL is an XML based language in which the input and the output of each service will be defined using DTDs and the actual data will be stored in XML format. In [38] there are three user groups identified for the XIB framework: wrapper engineers, broker engineers, and end users. Wrapper engineers are responsible for wrapping a particular information service and registering the service with a server. Wrapping involves the task of defining the service in terms of XIBL. Broker engineers, are responsible for generating brokers by selecting some of the registered services and defining how to integrate the services. Finally, end users utilize the brokers to automate the querying of the various wrapped information services. A set of tools for generating wrappers and brokers has already been developed. More information can be found in [38].

2.5 Tool Integration

In this section we examine methodologies and techniques that enable the integration of tools in distributed environments. Tools can be viewed as independent applications that provide some service under the premise that they accept input and generate output. To the user, the physical location (in terms of computer name or address) of these tools should not be important. What is important is that these tools can be invoked individually or

in any given combination to achieve a more complex task. In the following sections we provide an overview of some projects that deal with the integration of distributed services.

2.5.1 ToolBus

ToolBus is a framework that allows software tools to coordinate and to communicate with each other. A software tool could be a process running inside the ToolBus framework or a separate piece of software component running outside ToolBus. A scripting language based on process algebra [9] is used to describe the communication between the software tools. For every external tool, a language-dependent adapter is required to translate the data format used by ToolBus to that used by the external tool. This makes the development of tools more flexible since tool developers can choose the most appropriate format for their data. The overall architecture of the ToolBus system, as found in [3], is shown in Figure 2.14. Using either message passing or selective broadcasting, achieves the communication between the processes inside ToolBus. The communication between ToolBus and external tools is achieved by sending messages back and forth. Finally, more complex processes can be created by combining existing processes by using composition operators.

2.5.2 Control and Coordination of Complex Distributed Services

C3DS is a project that proposes a framework for distributed services. C3DS stands for Control and Coordination of Complex Distributed services. The objective of the C3DS project, as stated in [4], is to exploit distributed object technology to create a framework for complex service provisioning. By complex service provisioning it is primarily meant, the ability to compose a given service out of existing ones as well as the ability to exercise dynamic control over the execution of the service.

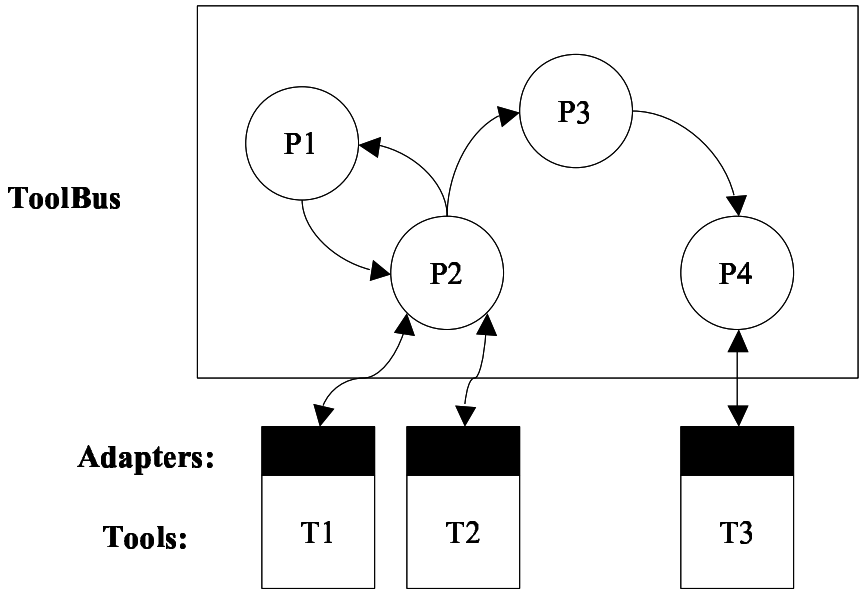

Figure 2.14: ToolBus architecture.

The C3DS framework defines two groups of requirements: service level and end user level. In the service level requirements, mechanisms are needed to dynamically add, extend, remove, or move component services in a dependable and predictable manner. In the end user requirements, the users must be able to specify, create, configure and manage services easily. In order to meet these two requirements, the C3DS framework is based on two emerging technologies: software agents and workflow management. In C3DS, agents are defined to be software entities that perform operations on behalf of a user, in order to achieve a goal. Typically, agents act as autonomous entities and can be either static or mobile. In many cases, when a software component has to be upgraded with more features in order to allow the user to customize the software functions, the incorporation of agents into the software is a very practical solution [4]. The use of agents is expected to meet the end user requirements. Workflow is a rule-based management software that allows one to direct, coordinate, and monitor the execution of complex tasks. These techniques were initially developed to automate office processes, but they can be applied

in a similar manner to service provisioning. Workflow management, can meet the service level requirements since it provides an environment for managing services in a dependable manner.

The C3DS framework is based on the integration of the software agents and the workflow management. The description of the software architecture, the services and their interconnections will be done with a language similar to ADL. ADL stands for Architecture Description Language and it describes the structure and components of large software systems. Once a user specifies the overall structure of the system using an ADL dialect, tools will automatically generate workflow scripts and agents to implement the specified system. These tools will be based on existing technologies like CORBA and Java that already provide functionality for distributed services and communication between software entities. In Figure 2.15, the overall architecture of the C3DS system is shown.

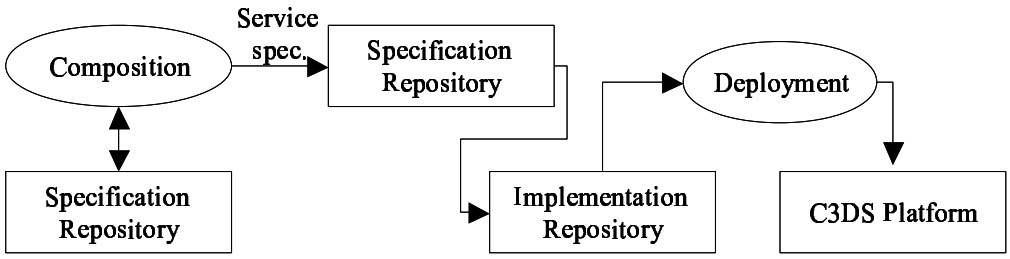

Figure 2.15: Overview of the C3DS framework.

2.5.3 BizTalk

BizTalk is a framework that provides a set of guidelines on how to publish schemas in XML and how to use XML messages to easily integrate software applications in order to build rich new solutions. It is an industry led initiative that has as its main goal not to provide a new standard, but to expedite the adoption of existing technologies

like XML and to use them in electronic commerce and application integration. BizTalk includes a design framework for implementing an XML schema and a set of XML tags used in messages sent between applications. The BizTalk website serves as a repository of schemas for a variety of industries. These schemas are available to everyone, and their wide acceptance and adoption may help companies integrate their applications with less effort. According to [1], the benefits of the BizTalk framework are the following:

Easier mapping across schemas.

By formalizing the process of defining business interchange schemas it would become an easier task for software developers to map the different schemas and enable businesses to communicate with each other.

Design target for software vendors.

By establishing a number of popular schemas the software vendors can rapidly incorporate these schemas into their next generation applications for electronic commerce and application integration products. Examples of such schemas include LandXML by Autodesk (used for representing land planning and survey information), and BBP-PurchaseOrder by SAP (used for representing business to business information for buying goods and services).

Framework for promoting standards.

The BizTalk Framework provides a platform for migrating an existing set of industry interchange standards to XML. This is especially useful for the Electronic Data Interchange (EDI) community.

Repository for BizTalk schemas.

The BizTalk framework provides a place where industry groups and developers will publish their schemas either publicly or privately among a group. Once a schema is published, the repository can support it with many features like versioning, dynamic detection, processes and visualization maps.

Showcases for best practices in developing XML interchanges.

Many organizations involved in the standardization of business interchange data formats are more skilled in business process modeling than in systems programming and XML. These groups can turn to the BizTalk Framework Web site to discover best practices for implementing their own schema or to discover pre-existing XML schemas they can use in their applications.

The principles for the BizTalk framework require that only the communication between applications be done through XML messages. It is recognized that software applications are written and deployed in a diversity of programming environments. BizTalk is primarily trying to address the need for these diverse applications to communicate. A variety of schemas for many industries are already available at the BizTalk website [2], along with a variety of tools and documents to support the use of these schemas.

2.5.4 Jini

The Jini system is a Java based technology that provides simple distributed mechanisms for programs to communicate and interoperate. In Jini, the notion of a service encapsulates a software or hardware component that offers its resources to others services in the system and uses other resources as necessary. According to [41], a Jini system consists of the following parts: infrastructure, programming model, and services. A set of

components provides an infrastructure for federating services in a distributed system. The programming model supports and promotes the production of reliable distributed services. Services can be made part of a federated Jini system and offer functionality to any other member of the federation.

In [41], a summary of the goals of the Jini system is discussed. Specifically, Jini enables users to share services and resources over a network, provide users easy access to resources anywhere on the network while allowing the network location of the user to change, and simplify the task of building, maintaining and altering a network of devices, software and users.

The Jini system is based on Java technology and therefore it inherits the benefit of portability throughout a wide range of platforms. Another benefit is the built-in security that allows the confidence to run code from other machines locally. One key assumption for such an environment to be realized, apart from the use of Java technology, is that every Jini enabled device has some memory and processing power. The dynamic nature of Jini allows services to be added, removed, and configured at runtime without the need of restarting the whole system. In order to support such a dynamic nature, the Jini system provides mechanisms for service construction, lookup, communication, leasing, and use in a distributed system.

	Infrastructure	Programming Model	Services
Base Java	Java VM RMI Java Security	Java APIs JavaBeans ...	JNDI Enterprise Beans JTS
Java + Jini	Discover/Join Distributed Security Lookup	Leasing Transactions Events	Printing Transaction Manager JavaSpaces Service

Figure 2.16: Jini Architecture Segmentation.

Overall, we can distinguish three categories of Jini components: infrastructure, programming model, and services. The infrastructure includes all components that enable building a Jini system. The programming model allows the creation of reliable services that are part of the system. Finally, the services are the entities within the Jini system. Even though Jini is an independent framework, it can be viewed as a network extension of Java technology. In Figure 2.16, we see how the Jini categories of components extend the Java technology.

2.6 Summary

This chapter summarizes the related work in the areas of source code representation, data modeling, and tool integration. In the Program Representations section we discussed numerous approaches for representing information found in the source code of programs written in various programming languages. This discussion provided an insight to the issues involved in developing and using such representations. In the Modeling Environments section we introduced some XML related technologies used to model data and meta-data. Since some of these technologies are used in the following chapters, the section served also as an introduction to XML. The Reengineering Frameworks section described frameworks for creating software maintenance and analysis tools. The objective of these frameworks is to facilitate the development of tools that can be reused in many programming language domains. The Tool Integration section focuses on technologies that allow tools to be used in distributed environments. The features available in the reengineering environments and those offered by distributed technologies, form the base requirements for the Integrated Software Maintenance Environment that we present in Chapter 4.

Chapter 3

Domain Models

Domain models are used to model information in specific domains. In our work, the domains that we model, are programming languages. Programming languages are defined by their syntax (what the programs look like), and by their semantics (what the programs mean). The syntax of a programming language can be specified using a context-free grammar[15] which is denoted in the Backus-Naur Form (BNF). The context-free grammar (grammar, for short) describes the hierarchical structure of the programming language by using terminals, nonterminals and production rules. Terminals, also known as tokens, are lexical elements of the language. Nonterminals represent sequences of terminals. Productions rules are composed of the left and the right side. The left side consists of a nonterminal and the right side consists of a sequence of terminals and/or nonterminals. The semantics of the language are much more difficult to describe than the syntax and are most commonly specified using informal descriptions. When programmers develop a program in a specific programming language, they have to understand both the grammar and the semantics of the language.

Typically, a program is stored in the file system as a plain text file. In order to translate the text file to executable code we use a compiler. A compiler has four major phases,

each of which transforms the program from one representation to another. The phases of the compiler as illustrated in Figure 3.1. In the first phase, the lexical analyzer (also known as scanner) reads the text file as a string of characters and recognizes sequences of tokens. In the second phase, the syntax analyzer (also known as parser) groups the tokens into grammatical phrases by using the grammar. The grammatical phrases are represented by a parse tree. Typically the parse tree is stripped from some syntactic tokens that are only used by the scanner and are not useful for translation. The resulting tree is an Abstract Syntax Tree (AST). In the third phase, the AST is read by the semantic analyzer which enforces the semantics of the language on the AST and adds annotations to it. Finally, in the fourth phase the code generator reads the annotated AST and generates the target machine code. In reality there are more phases involved in the compilation process (e.g. optimization) and there are additional data structures generated (e.g. symbol table). However, the simple model with the four phases that we described is complete enough for our purpose.

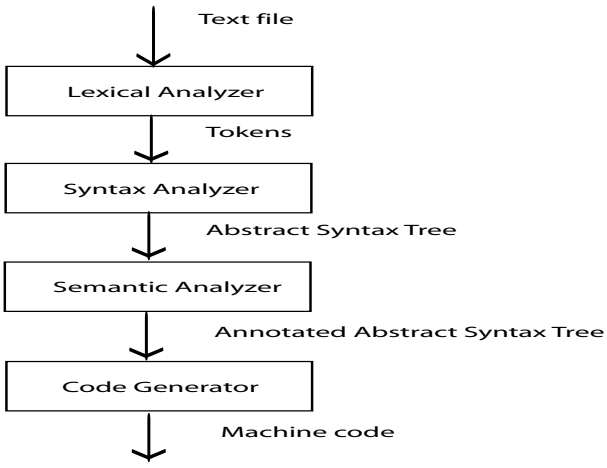

Figure 3.1: The structure of a compiler

It becomes apparent, after examining the compiler phases, that the AST is very useful in performing many software analysis and maintenance tasks. The compiler actually uses the AST to perform semantic checking, optimizations and to generate code. However, a compiler is typically used only to generate machine code from the source code, and most compilers available today discard at the end of the compilation all intermediate representations and data generated, including the AST. In addition, since ASTs are typically stored in memory, there is no standardized way to represent them as data structures or to store them in files. This has resulted in a number of software analysis tools that use different formats for storing intermediate information (like ASTs). This thesis presents an environment that allows for the use of a standardized modeling environment based on XML, to create a programming language representation that is more flexible and more portable than the AST generated by a typical parser. The generated representation of the source code is based on the domain model of the language. In this work we also propose generalized domain models for programming languages that share common features (e.g. object-oriented).

3.1 Requirements for domain models

Our goal is to develop program representations based on XML and language specific domain models that will store source code information at the same, or higher level of abstraction than an AST. Most data modeling approaches presented in the previous chapter, propose representations to accommodate specific types of maintenance tasks (e.g. construction of dependence graphs). The proposed source-code representation is simple but detailed enough to represent the syntax of a specific programming language in terms of a Document Type Definition specification document. On the other hand, the representation is extensible enough to enable the development of representations that

allow for specific analysis data to be represented in a uniform way. These analysis-specific representations not only are defined separately but also, they are linked to the domain model as annotations. When working with multiple domain models, it is possible to create a single model that accommodates the representation needs of all the subdomains. This single representation is typically an aggregation of all the constructs in all the different domains. Our approach is to try to develop more generic representations suitable for domains with similar characteristics. These representations are based on the identification of the common constructs of specific programming languages being modeled. The goal for these representations is to facilitate the design and the implementation of more generic analysis toolsets that can be applied to similar domains.

In proposing representations for programming languages, we need to use some guidelines to evaluate the quality of the representations. For this reason, we define the following requirements for developing a programming language domain model and the corresponding source code representation.

- **Easy to use**

  The representation must be easy to understand and to manipulate mechanically. No extra learning steps should be added to the tool developers. Keeping the representation of the domain model and formalism as close as possible to the grammar of the language is a strong requirement.

- **Extensible**

  The representation has to be extensible. The need to accommodate for evolving programming languages requires that the representation is easy to change. The representation should also be flexible enough to allow for other models to be linked to it as extensions or annotations. These annotations model and represent information related to specific analysis tasks (i.e. call graphs).

- **Widely supported**

  The representation must be widely supported in terms of published APIs, operating and hardware platform independence. The development of APIs that enable developers to easily read, store and change the information that uses these representations is very important. These APIs should be readily available for a variety of platforms.

- **Human readable**

  The representation has to be 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 tool developer to read the information directly makes the maintenance task easier.

3.2 Formalism required to represent domain models

The formalism and modeling environment developed in this thesis, is based on the requirements set out in the previous section. The use of XML as the representation language and formalism satisfies the majority of those requirements for its extensibility, portability, and wide adoption by the software community. As previously discussed, in the related work section in Chapter 2, XML has received wide support from many different communities overall. All these communities share the need for a common way to represent similar information. This common way will enable them to share data and integrate a variety of existing services over the network. The Biztalk framework discussed in the previous chapter explains how different industry segments are joining forces to develop and standardize a common representation for their area. Another reason for the wide support for XML is the numerous implementations of XML parsers that are readily available for various platforms [20, 29]. All these implementations are based on a single standard, so

the available APIs used to parse, represent and access the XML files are uniform across the various implementations. A variety of technologies that support and extend the XML language have been proposed and developed to accommodate the needs of specific communities. Therefore, we can safely claim that XML has been adopted as the modeling environment in many domains, including the software engineering domain.

3.2.1 Abstract Syntax Trees

Since our goal is to represent source-code at the syntactic level, it becomes clear that we need to determine how Abstract Syntax Trees (ASTs) [5] can be represented using the XML language. ASTs are composed of nodes and edges. Non-leaf nodes are typically used to represent operators and leaf nodes are used to represent operands. Edges have no type and are used to associate related nodes. When used between non-leaf nodes, they represent the parent-child node relationship. When used between non-leaf and leaf nodes, they associate the operators with the actual operands. To generate an AST we first need to generate a parse tree. The parse tree is the result of parsing the concrete syntax using the grammar of the language. The elimination of all the purely syntactic tokens such as parenthesis or semicolons from the parse tree yields the AST. The AST can be further annotated with extra information about the program to produce an annotated AST. In Figure 3.2, we illustrate all three types of trees for the simple expression $a + b$.

3.2.2 Extensible Markup Language

The basic constructs of the XML language are Elements and Attributes. Elements are allowed to contain other elements thus allowing hierarchical structures to be represented. Elements can also contain character data. Every element is identified by its name. In

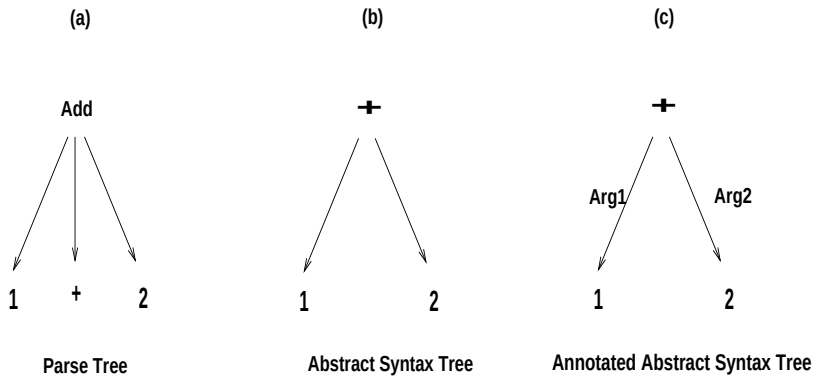

Figure 3.2: Parse Tree, AST, and Annotated AST for the $a + b$ expression and the grammar rule *Add: Integer '+' Integer*

addition, elements can hold information specific to the instance of the element by specifying attributes. When an XML document is seen as a hierarchical structure, Elements, Attributes, and Character Data become nodes of the tree. Character data can be thought of as a specialized element node. The edges between element nodes are used to represent the notion of containment. The edges between element and attribute nodes are typed. The type represents the attribute name and the attribute node holds the actual information that this attribute node has for the corresponding elements node. Two sample XML files and their corresponding tree views are shown in Figure 3.3 and Figure 3.4.

3.2.3 Mapping of ASTs to XML

Given a specific programming language, we need to define a representation to which every source code program can be mapped to. To accomplish this mapping from ASTs to XML trees, we need to define an algorithmic 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 valid for this grammar source code program), can be mapped to XML trees defined by the DTD.

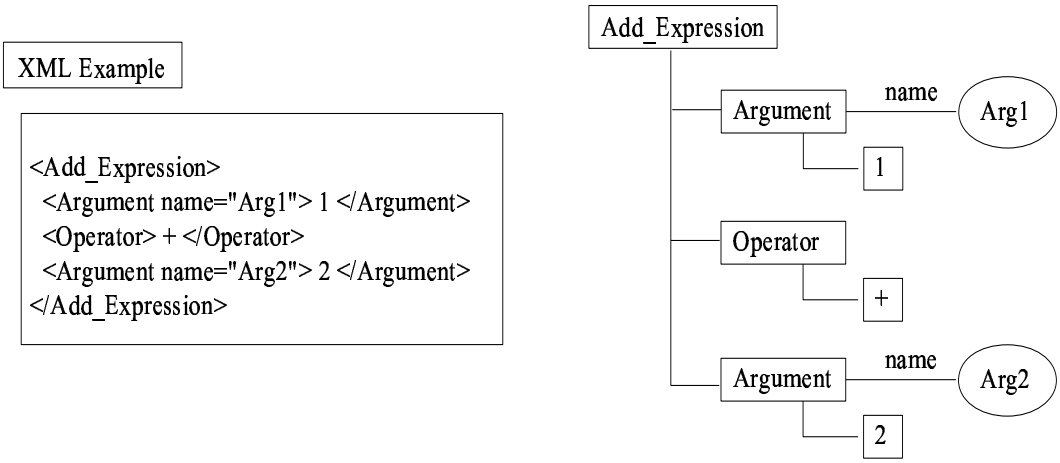

Figure 3.3: Sample XML file and its tree view for the AST in Figure 3.1

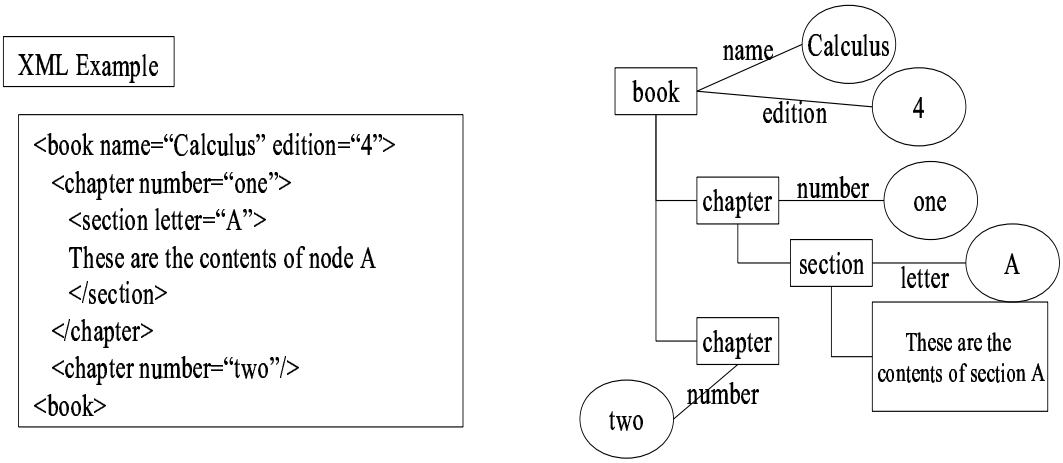

Figure 3.4: Sample XML file and its tree view.

In this section, we discuss a transformation methodology that assists in implementing a mapping from an EBNF grammar to a DTD. Below we provide a set of transformation rules for defining this mapping. Each of the following rules is accompanied by an example that demonstrates the production rule of the grammar and its corresponding DTD expression. In the examples we use a, b, c, to represent nonterminals and d, e, to represent terminals.

• Non-terminals are mapped to elements.

Grammar	**DTD**
a ::= b	<!ELEMENT a (b)>

• Choices of non-terminals are mapped to model groups using the "|" symbol.

Grammar	**DTD**
a ::= b |c	<!ELEMENT a (b|c)>

• Non-Terminals that contain sequences of characters are mapped to attributes.

Grammar	**DTD**
a ::= name	<!ELEMENT a EMPTY> <!ATTLIST a name CDATA>

•Optional non-terminals are mapped to model groups using the "?" symbol.

Grammar	**DTD**
a ::= b_{opt}	<!ELEMENT a (b?)>

• Lists of non-terminals are mapped to model groups using the "*" or "+" symbol.

Grammar	**DTD**
a ::= b b ::= c |b c	<!ELEMENT a (c+)>

• Terminals are mapped to attributes.

Grammar	**DTD**
a ::= d	<!ELEMENT a EMPTY> <!ATTLIST a d CDATA>

• Sequences of terminals are mapped to distinct attributes.

Grammar	**DTD**
a ::= d e	<!ELEMENT a EMPTY> <!ATTLIST a d CDATA !ATTLIST a e CDATA>

• Choices of terminals are mapped to named group attributes.

Grammar	**DTD**
a ::= d |e	<!ELEMENT a EMPTY> <!ATTLIST a value choice(d|e)>

The use and combination of the rules presented, allows us to generate elaborate mappings from grammars specified in EBNF 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. To demonstrate how the above rules are used let us consider a simple grammar that recognizes can recognize the following simple expressions like *a+b*, *a*b*, *a*b+c*. One possible grammar expressed in EBNF is illustrated below.

```
symbol ::= a | b | c
term ::= term * symbol
         | symbol
```

```
expr ::= expr + term
         | term
```

The corresponding DTD for the above grammar can be generated by using the transformation rules as follows. First we use the rule for choices of terminal and we map the production rule for *symbol* to a named group of attributes. Second, we combine the rules for choices of non-terminals with sequences of non-terminal to map the production rules for *term* and *expr* to elements with model groups. The resulting DTD for the example is illustrated below.

```
<!ELEMENT symbol EMPTY>
<!ATTLIST symbol value choice(a|b|c)>
<!ELEMENT term ((term,symbol) | symbol )>
<!ELEMENT expr ((expr,term) | term )>
```

Using this DTD we can now represent the expression $a*b+c$ in XML in the following way:

```
<expr>
   <expr>
      <term>
         <symbol value="a"/>
      </term>
      <symbol value="b"/>
   </expr>
   <term>
      <symbol value="c"/>
   </term>
</expr>
```

3.3 Domain specific models

The first use of the transformation rules is to generate representations for specific programming language domains. The domain-specific representations must meet all the objectives listed in the beginning of this chapter. Furthermore, it is very important to keep in mind that the representations generated must be complete. Complete means that all the information found in the source code should be available in the program representation. However, comments and white space are not part of the abstract syntax and therefor are not included. For this thesis, the entire Java and C++ grammars are mapped to their corresponding DTDs to provide XML representations for Java and C++ programs, called JavaML and CppML respectively. In the following two sections we describe how the transformation rules from the previous section were used to generate representations for the Java and the C++ languages.

3.3.1 Java Markup Language (JavaML)

The generation of a source code representation for Java is based on a parser generator tool called Java Compiler Compiler, or JavaCC [40] in short. 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 EBNF grammar as input and generates source code that can parse any instance of the EBNF grammar. The popularity of JavaCC is most probably due to the grammars for Java that are 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.

The complete DTD that we generated based on the Java 1.1 grammar can be found in Appendix A. The number of production rules in the Java 1.1 grammar is 130. In our

implementation we used the JavaCC grammar for Java with slightly fewer production rules. In the prototype developed there is no information loss. We should point out that comments are lost but this is not part of the AST of the source. However, maintaining the comments would only require some minor modifications to the parser and the lexer. Below we present some examples of how the guidelines were used to map the grammar to the DTD.

Every Java source file is considered to be a CompilationUnit that may contain an optional PackageDeclaration followed by zero or more ImportDeclrations followed by zero or more TypeDeclarations. The EBNF and JavaCC productions for this are shown below.

In EBNF form we have:

```
CompilationUnit ::=  PackageDeclaration{opt}
                     | ImportDeclarations{opt}
                     | TypeDeclarations{opt}
```

In JavaCC we have:

```
void CompilationUnit() :
{
  [ PackageDeclaration() ] ( ImportDeclaration() )* ( TypeDeclaration() )*
}
```

When this production is mapped to a DTD we make combined use of the guidelines that refer to sequences of non-terminal and optional non-terminals as well as quantified non-terminals. The corresponding DTD would be:

```
<!ELEMENT CompilationUnit
 (PackageDeclaration?,ImportDeclaration*,TypeDeclaration*)>
```

Another interesting thing to look at is the mapping of non-terminals that include terminals in the right hand side of the production rule. These are referred to as Identifiers or Literals in JavaCC. The use of the *Name* non-terminal is a good example. In EBFN this is expressed as:

```
Name ::= Identifier
         | Name . Identifier
```

In JavaCC *Name* is expressed as:

```
void Name() :
{
  <IDENTIFIER>
  ("." <IDENTIFIER>)*
}
```

This is mapped to an element called Name and an attribute is used to store the information that is contained in the Identifier. The corresponding DTD would look as follows:

```
<!ELEMENT Name (Name?)>
<!ATTLIST Name Identifier CDATA>
```

3.3.2 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 take 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++[35]

compiler which is developed at the IBM Toronto Lab. VisualAge contains a source code repository that is called *Codestore* in which all the information generated during the compilation process is stored in an IBM proprietary format. 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 even after the compiler has emitted the binary code. Our goal in using VisualAge is to demonstrate that a commercial product can be integrated and used in a more complex environment like the one proposed in this thesis. Using a parser generator would have been another approach in generating a C++ representation (i.e. the approach taken on generating the Java representation). Information loss due to preprocessing is something that we have to face, in our current implementation since some of that information is not made available by VisualAge. However, if we were to use a parser generator, we would capture all the information in the source code (excluding comments and white space). The resulting XML representation, would require more complicated processing, similar to the pre-processing that the compiler does in order for certain analysis tasks to be performed. The representation we generated using VisualAge can be found in Appendix B. The grammar that this was based on, was implicitly extracted from the *Codestore* APIs. Below we present an example of how the mapping was done using the mappings presented earlier in section 3.2.3.

In *Codestore* every component of the language is represented by a class. A Macro definition is used in the C++ language as a preprocessing directive to substitute some parameters with some corresponding replacements. The APIs define a Macro class that contains MacroParameter and MacroReplacement elements. The class is represented as follows:

```
class Macro{


public:
   char* name
   MacroParameters
   MacroReplacements


   ...
}
```

The corresponding DTD section that expresses the above structure is shown below.

```
<!ELEMENT Macro (MacroParameter*,MacroReplacement*)>
<!ATTLIST Macro name CDATA>
```

3.4 Generalized domain models

Given domain models that share common features (i.e. models that correspond to object-oriented languages), new generalized domain models can be developed. In this section, we discuss the rationale, the design, and the development of a generalized domain model that stems from the amalgamation of the Java and the C++ domain. Unlike the specific domain models that are used to represent information available at the source code level, the generic model captures the common characteristics of the domains they represent. This common information can be useful when developing generic maintenance tools. One such example would be to calculate object-oriented metrics, extract the object model, and even implement certain data dependence analysis algorithms.

It is possible to develop generalized domain models by studying grammars of object-oriented languages, identifying their most common entities, and then obtaining a gen-

eralization. Mapping specific domain models that are encoded as XML representations to other generalized domain models is easier than mapping different grammars. Implementing these mappings and transformations can be done by simply using XML APIs instead of developing complex mappings between EBNF grammars. The following section describes how a representation for object oriented languages can be generated based on JavaML and CppML representations.

3.4.1 Object Oriented Markup Language (OOML)

Many programming languages have been implemented using the object-oriented programming paradigm. In this section, we demonstrate how the Java and C++ programming languages, which are both object oriented, are mapped to a more generic representation called OOML. As previously stated, an easy way of defining this mapping is to use the JavaML and CppML representations as a starting point. The implementation of such a mapping can be done in two ways: The first involves the use of the XML APIs to construct a program that maps one representation to the other. The second approach involves the use of XSLT [19] transformations which are designed to map one XML document to another. Both approaches can achieve the same goal and the developer will have to select one. The goal of this section is to identify and discuss similar concepts that appear in JavaML and CppML and determine how they can be aggregated to a more general OOML representation. Table 3.1 lists all the domain model entities that were identified in the JavaML and CppML representations. In the first and second columns we list all the common JavaML and CppML domain model entities respectively. In the third column we list the mapping of the common entities that correspond to OOML. In some cases more than one entity of JavaML, or CppML maps to a single entity in OOML. An example of such a case is the Loop element. The introduction of the Loop element in our OOML representation allows us to perform more generic analysis tasks like the cy-

JavaML	**CppML**	**OOML**
CompilationUnit	Program	Program
CompilationUnit.source	PrimarySource, IncludedSource	Source
ImportDeclaration	Include	Include
ClassDeclaration	Class	Class
MethodDeclaration	Function	Method
FieldDeclaration	Variable	VariableDeclaration
Block	LexicalBlockStatement	Body
SwitchStatement, IfStatement	SwitchStatement, IfStatement	ConditionalStatement
DoStatement, ForStatement, WhileStatement	DoStatement, ForStatement, WhileStatement	Loop
Name	NameExpression	VariableUse
Name	FunctionCallExpression	MethodCall
Type	TypeDescriptor	Type
FormalParameter		Parameter

Table 3.1: Mapping of concepts from JavaML and CppML to OOML.

clomatic complexity at a language-independent level. Another point to observe in Table 3.1 is that in JavaML for example, the Name element could map to both a VariableUse or a MethodCall. Our implementation of the mapping tool determines what the Name maps to by examining its contents and its nesting in the XML document. The complete OOML representation can be found in Appendix C.

In a more formal way, the generalization of domain models specific to a programming language to a common domain model can be though of as a semantic mapping function. A semantic mapping function denoted here as S_d, generalizes a specific domain model denoted as $DomainModel_s$ and the current context denoted as *Context* to the generalized domain model denoted as $DomainModel_g$.

$$S_d : DomainModel_s \text{ x } Context_{opt} \mapsto DomainModel_g$$

The $DomainModel_g$ is a set of all the generic domain model entities. The $DomainModel_s$ is a set of the specific domain model elements that can be mapped to the some generalized elements in $DomainModel_g$. Finally, the *Context* represents information that is specific to $DomainModel_s$, and is necessary to perform the mapping to $DomainModel_g$. Such information may include attributes of some domain model elements, or the relationship among various domain model elements. In some cases the context is not necessary if the mapping of one element in $DomainModel_s$ to one in $DomainModel_g$ is direct.

Below, we present a small example of how common constructs in JavaML and CppML can be identified and mapped to the OOML representation below. Other mappings are possible but no matter what the mappings and the generalizations are, the premises and requirements for the Integrated Software Maintenance Environment that this thesis addresses, remain the same. Tools that use the OOML representation for maintenance tasks are discussed in more detail in Chapter 5. For example, both Java and C++ languages, represent objects by using the concept of classes, class methods and class variables. In this context, an object is an abstract entity that encapsulates some data and that is able to perform operations on this data. A class can be thought of as a template for creating an object. Every class has a name that uniquely identifies it. 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:

S_d(ClassDeclaration) $\mapsto$ Class
S_d(UnmodifiedClassDeclaration , Name attribute) $\mapsto$ Class.Identifier
S_d(FieldDeclaration) $\mapsto$ VariableDeclaration
S_d(MethodDeclaration) $\mapsto$ Method

Representing objects in C++ can be 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:

S_d(Class) $\mapsto$ Class
S_d(Class , name attribute) $\mapsto$ Class.Identifier
S_d(Function) $\mapsto$ Method
S_d(Variable) $\mapsto$ Variable

Another example of semantic mappings where the *Context* is used, is the case of the VariableUse and MethodCall domain model entities in OOML. In JavaML, the Name domain model entity is used to represent both a variable use or a method call. In order to distinguish which cases applies we need context information. In this case the context is the presence (or absence) of arguments after the Name. The semantic mappings for the Name domain model entity, from JavaML to OOML are:

S_d(Name , Absence of arguments) $\mapsto$ VariableUse
S_d(Name , Presence of arguments) $\mapsto$ MethodCall

The previous examples demonstrate how some simple domain model entities can be expressed in a more general domain model like OOML. All the mappings and their details are implemented as part of a Java program that is discussed in Chapter 5.

Chapter 4

Integrated Software Maintenance Environment

4.1 Introduction

The Integrated Software Maintenance Environment (ISME) is an environment that facilitates software maintenance. In this environment a variety of new software maintenance tools can be developed for specific programming languages using language-specific representations as discussed in Chapter 3. In addition, generic language-independent analysis tools can be developed using some higher-level representation languages. Such tools include components that compute a variety of software metrics (i.e. cyclomatic complexity, data complexity, fan-out, information flow), extract data and control flow graphs that allow the user to obtain an architectural description of the system being analyzed. Another feature of this environment is the ability to integrate existing tools by making use of common representations. The tool integration is taken a step further by creating a distributed service with specified inputs and outputs for every CASE tool in the environment. These services can be aware of all other services in the environment and are able to

exchange data and analysis results with other services in order to perform more complex software analysis tasks. Services can be local or distributed, thus allowing tool developers to use the ISME environment to share tools with other developers. Storing the input and output data for these tools should also be taken into consideration in ISME. It is not uncommon for software maintenance tasks, to require large amounts of storage space that needs to be accessed quickly and efficiently. In ISME, services that store data in a variety of systems and forms, and that use different persistent storage environments can coexist.The features of ISME can be grouped in the following three distinct categories:

1. **Data Integration**, which deals with program representations and how they can be used to enable tools to communicate.

2. **Control Integration**, which deals with mapping tools into distributed services and with the control and coordination aspects of these services.

3. **Repository Services**, which deals with the persistency and sharing of the processed data.

In the following sections, we discuss the objectives of the ISME environment, we present the overall architecture, and we discuss each of the three categories in more detail.

4.2 Objectives

The overall goal of the ISME environment is to facilitate complex software maintenance tasks by incorporating into a collaborative environment various specialized CASE tools. In order to achieve this goal, it is necessary not only to design new maintenance techniques, but also to improve and integrate existing ones. The objectives of the Integrated Software Maintenance Environment aim to address the following areas:

- Design of language-specific program representations using XML technologies. These representations must be expressive enough to capture all the information available at the source code level. They must also be extensible in order to accommodate for any future changes and additions.

- Enable the development of new generalized language-specific software maintenance tools that operate on the proposed XML representations. Working directly with these representations is easier and more efficient than working with the source code. An Integrated Software Maintenance Environment (ISME) facilitates this requirement.

- Enable the development of a set of language-independent software maintenance tools for specific maintenance tasks (i.e. to obtain an object model). These tools will allow specific maintenance tasks to be accomplished without the need of re-implementation for every language-specific representation.

- Enable tools to behave as distributed services that can be deployed and configured dynamically. These services would be aware of their environment and would be able to communicate and coordinate with other services to achieve more complex tasks.

- Reuse existing technologies and methodologies for data representation, service description and repository implementation. The focus for ISME is to bring together existing technologies rather than reinvent all components of such a complex environment.

4.3 System Architecture

So far the ISME was introduced in Chapter 1 as an abstraction layer between the source code and the software maintenance tools. The ISME and all of its components that create

this abstraction layer are illustrated in Figure 4.1. In this view of the system architecture, we focus on the functional components of the environment. The distributed nature of the system allows each component of the environment to exist as a service. These services will be discussed in more detail in the control integration section, later in the chapter.

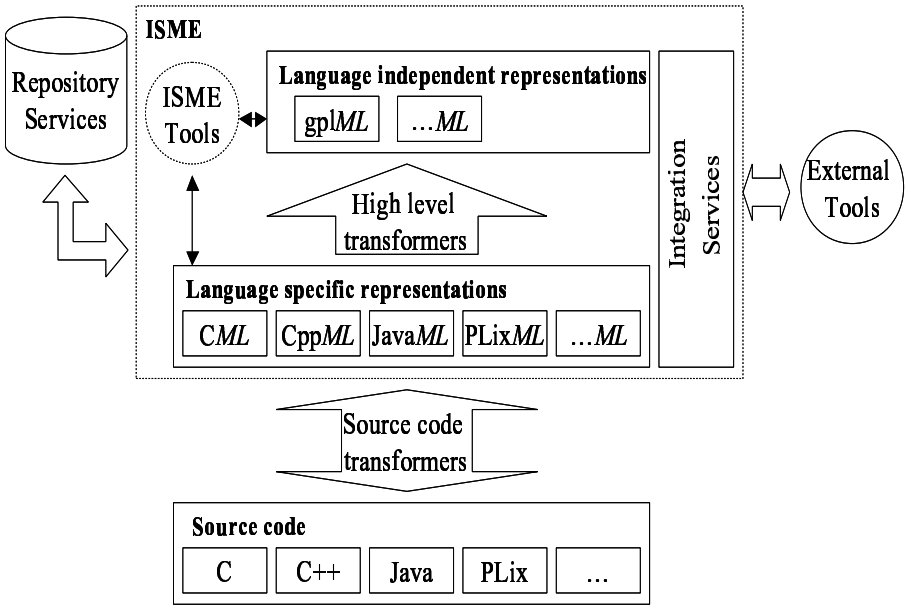

Figure 4.1: ISME architecture.

The major components of the ISME environment include: *Source code transformers*, *High-level transformers*, *ISME based tools*, *External tools*, *Repository Services*, and *Integration Services*. The source code transformers component, contains parsers that parse source code files and generate the corresponding XML based representation. Maintaining all the information found in the source code is a requirement for these parsers. For every programming language in which we wish to perform maintenance tasks for, a corresponding transformer should be developed. High-level transformers include all tools that allow to map the language-specific information to a more abstract level. Information loss may

be part of this transformation process. Nevertheless, generic maintenance tasks will be possible using the higher level information only. An integration layer allows for registered ISME tools to directly exchange information content encoded in XML. This processing is done using XML parsers that are widely available and very well supported [20]. In this context, knowledge of the generic XML APIs and understanding of the programming language domain model is required. External Tools, that use their own input and output formats, require knowledge of the programming language at hand, the internal representation of the tool and the API of the tool. Repository services are responsible for providing a scalable way of storing, accessing and sharing information used by the ISME environment. A variety of approaches can be taken to provide these services as will be discussed later in the chapter. Integration services provide the necessary mechanisms to allow external tools to use the information available in the ISME. Typically a mapping between the programming representation of the tool and of the ISME is used. Also these mappings will enable the ISME to act as translator between tools that want to communicate.

4.4 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. In order to perform data integration, it is necessary to identify the components of the ISME environment that will be utilized to achieve the integration. These components are: language-specific program representations, higher-level program representations and tool-specific representations. 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 representa-

tion 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, analysis-specific representations that extend the language-specific representations can be defined to facilitate the exchange of analysis results.

When talking about data integration it is important to make the distinction between two separate concepts: Domain Model integration and Software Analysis Results integration. In Domain Model integration, there is a need to use a common representation as input for their software maintenance tasks performed by the registered tools. Similarly, Software Analysis Results must be represented in such a way that the output of these tasks can be automatically used as input to other tools. These two separate concepts of data integration are discussed in more detail in the following two sections.

4.4.1 Domain Model Integration

This section discusses how existing maintenance tools can be integrated to use a common program representation as input to their analysis. The need for such integration has become apparent among various research communities that have developed several maintenance tools and need to compare them and integrate them to perform more complex tasks. Software reusability is another motivation behind this kind of integration. Using software components that have already been developed is a very attractive approach. This allows researchers to focus on developing high-level maintenance algorithms rather than dealing with low-level details. To perform such a task it is necessary to have language-specific representations that capture information available at the source code level. These representations are a core component of the ISME environment.

Using an Integrated Software Maintenance Environment, tool integration can be accomplished as shown in Figure 4.2. An external tool can be integrated with the ISME using two approaches: The first approach, is to develop a mapping tool that maps the format of the external tool to the XML-based representation used in the ISME environment. Such a mapping tool can be built using the standard XML APIs and it will provide the flexibility to the user to specify not only what the mappings are but how they are implemented. The second approach, requires that the format of the external tool is represented using XML. Then we can use the XSL Transformations (XSLT) [19] to define the mapping between the ISME program representation and the XMLlized format of the external tool. XSTL is a language for transforming one form of XML documents to another. Tools are readily available to perform this task. One of them is part of the IBM XML Tools product and it is freely available through IBM Alphaworks [20].

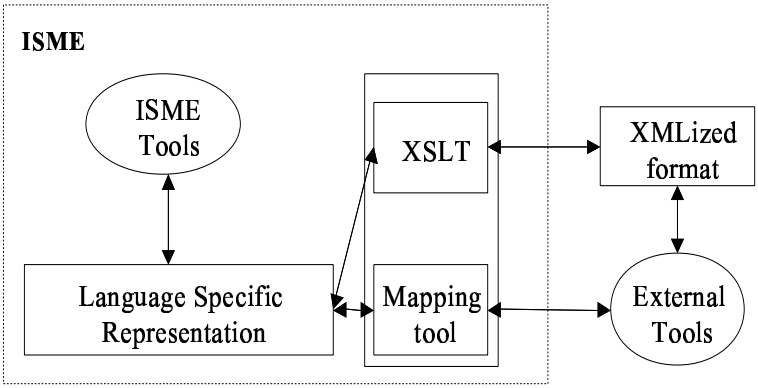

Figure 4.2: Data integration between ISME and external tools.

Integration between external tools can be accomplished in a similar way. Each tool can use one of the approaches discussed above in order to be integrated with the ISME environment. Once a tool has been integrated with ISME, it can use the ISME program representation as a data integration facility to exchange information.

The following example demonstrates how an external tool that constructs call graphs can be integrated with an ISME tool that parses source files and generates XML representations. Let us assume that the Call Graph tool is used to draw graphs that display all the calls that exist in the source code between functions. Let's also assume that the Call Graph tool is using the RSF format as an input, in order to compute the call graph information. For the Call Graph tool to work, the source code information must be represented using RSF. In this context, tool developers have two choices: The first is to develop their own source code parser that emits the RSF that their tool uses. This choice requires significant resources both in terms of time and effort. The second choice is to use an Integrated Software Maintenance Environment and take advantage of the parser that is already available and emits an XML representation. The developers will only have to create a mapping that processes the XML representation and emits RSF. Since the focus of the Call Graph tool developers is to create the call graph algorithm the developers would like to spend as less time as possible on other tasks such as creating a source code parser. All they need to do is use a readily available XML parser to obtain their information from the ISME environment. This example is shown in Figure 4.3 below.

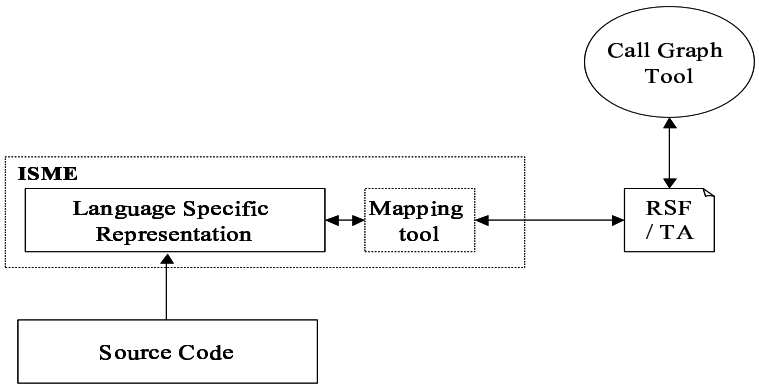

Figure 4.3: Call Graph Tool integration.

4.4.2 Software Analysis Results Integration

In this section, we describe how tools can be integrated and use the data generated by other tools. This integration can be performed based on the assumption that the tools use common program representations. If that is not the case, then the techniques previously described in the domain model integration section can be used to achieve this. When specific maintenance tasks have to be performed, it is a common approach to create models for storing intermediate information on which the actual analysis tasks can be performed. Given that such models are useful for a variety of maintenance tasks, it is important to represent them as an extension to the proposed program representation that the tools already use. We propose the use of XML to create domain models for specific analysis tasks that extend the common XML program representations.

As an example, consider that we have two software maintenance tools. One performs slicing and the other dead code identification. These two tools are both developed independently and they use the JavaML representation instead of working directly with the Java source code. The use of JavaML allows the tool developers to concentrate on the real algorithms rather than the details of parsing, storing and developing an intermediate representation for the source code. In such a scenario, both tools have to use a Control Flow Graph (CFG) as an intermediate step in order to either perform slicing or identify dead code. Therefore, it seems that if a common model for representing CFG could be developed, then both tool developers would benefit from it. Even more important, if the common model is based on the JavaML representation, that both tool developers already use, then the integration of the tools will become much easier.

Lets examine what the required information for creating a CFG is. In a CFG, the basic components are source code regions (usually called basic blocks) and control flow relationships between the regions. All the necessary information resides in the JavaML

representation. We only need to represent the basic blocks and their relationships. To do this, we need to create an element that represents the notion of a basic block. This can be easily accomplished by using XML Links and Pointers to identify parts inside XML documents. In the example illustrated in Figure 4.4 we present some source code and its corresponding XML representation to the right. The source code lines are grouped into four basic blocks S1, S2, S3 and S4. The call graph is constructed by using these four basic blocks. In order to store the call graph information we create a *source-region* and a *control-flow* element. The *source-region* element has a link to the corresponding source code element in the XML representation of the code. The *control-flow* element is used to express the flow of control from one source node to another.

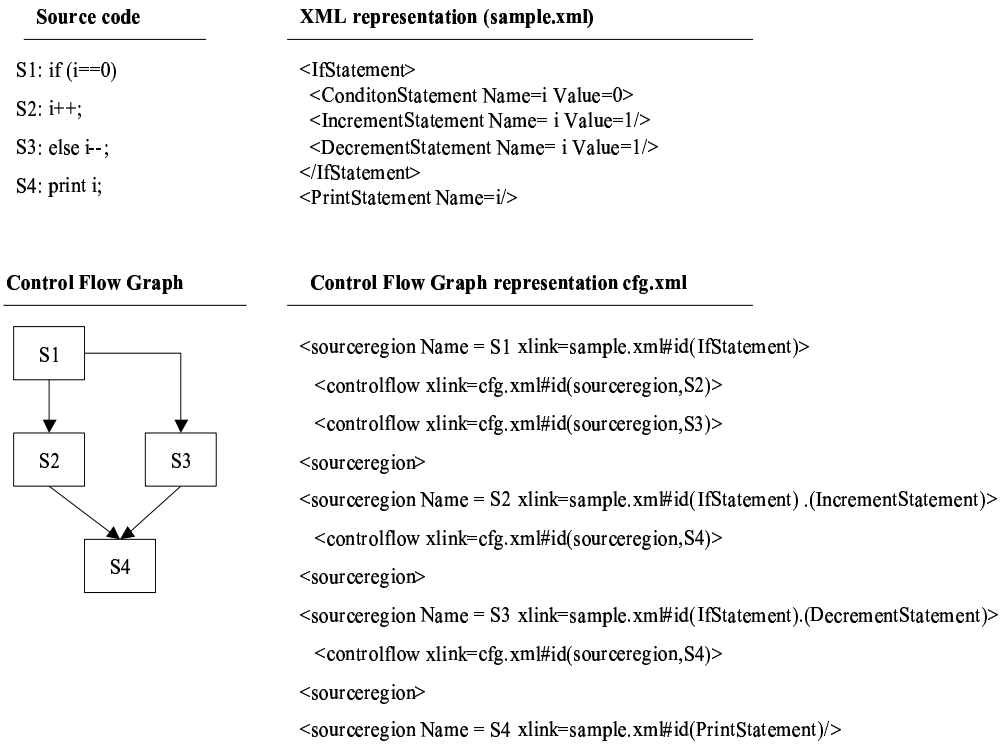

Figure 4.4: Control Flow Graph representation.

4.5 Control Integration

The term control integration in our context, includes all the features that an Integrated Software Maintenance Environment offers for creating and using distributed services out of software maintenance tools. In this context, a service is a software program 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 with the environment, how they can be configured dynamically, and how they can be invoked as part of a more complex task. Some of the concepts presented in this chapter are based on the Jini architecture. Before we proceed to the details of the services we present the key features of a service in the ISME environment: Distributed, dynamic, secure, and easy to use and to integrate. Distributed means that a service can exist in any machine on the network. Users are able to select the right service by examining its description and then supply the service with the input to 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 that was originally presented in the context of Active Databases [24]. Using ECA, it is be possible to define transactions that involve the combined use of many services.

4.5.1 Service localization

Localization of a service refers to the task of defining what a service is, where it is physically running, and what are the interfaces to access it. This information is also referred to as service attributes. When a new service is created it is necessary for the service provider to resolve all the localization issues. The localization information enables the service to become part of the ISME environment and makes the service available to other services and users. The localization issues involve describing the service, describing the interface of the service and configuring the service. Describing the service is the most simple and most important issue when creating a new service. A textual description of the service explains what the service does, how it does it, and what are the requirements for using this service. In an environment where many services exist, it becomes increasingly difficult to select the right service to use. Therefore, the description is the most important factor when selecting the right service. Describing the interface of the service requires that each service provide a list of the tasks it performs and the corresponding input and output formats for each task. It is not uncommon for a service to perform a single task. Whether one or more tasks are performed, specifying the correct input and output formats will allow the service to communicate with other services. The use of DTDs to specify the input and output format of a service is our approach. The existence of such DTDs allows the service to validate the input it receives for processing and it will prevent unnecessary processing of invalid input. When the output conforms to a DTD, it is possible to determine which tools can be combined to perform more complex services. Finally, configuration of the service involves issues such as runtime and security. The service provider has to decide where the service will physically run by selecting a host machine with a fixed IP address and allocating resources for the service as necessary. In addition, specifying the access control list enables the provider to establish a level of secu-

rity by blocking unauthorized users. The diagram below summarizes all the information required to localize a service.

Description:	A textual description of what the service does.
Interface:	A specification of the accessors and the message format of the service for both input and output.
Network address:	The ip address and port of the machine where the service is running.
Access:	A list of users who are authorized to access the service

If we were to create a service that parses Java source code and generates the JavaML representation for example, we would provide the following information to localize it:

Description:	This service parses Java source files and generates and xml based representation according that conform to the JavaML DTD. The generation representation contains all the information available in the source code and can be processed using any xml parser. The parser works with any version of Java up to 1.2
Interface:	**Input**: Java source files **Output**: JavaML files based on the following DTD: http://www.swen.uwaterloo.ca/ evan/JavaML/JavaML.dtd
Network address:	www.crete.uwaterloo.ca:265
Access:	emamas, kostas, rwgregor

4.5.2 Service registration

Once a service has been localized, it has to be deployed in the environment and inform the rest of the services of its existence. In this section, we describe how a service can register as part of the ISME environment and how it can be removed from it. These tasks can be accomplished by a set of Jini protocols known as discovery, join and lookup. The Jini architecture uses a lookup service that keeps track of all the registered services and their attributes. Discovery occurs when a service is looking for a lookup service. Join occurs when the lookup service has been discovered and the service wants to join it. Lookup occurs when a client wants to locate and invoke a service based on its interface. The following figures found in [41] describe the three protocols. In Figure 4.5 the discovery protocol is shown. A service provider is the originator of the actual service. First, the service provider needs to locate all lookup services. This is done by multicasting a request on the local network for any lookup services to identify themselves.

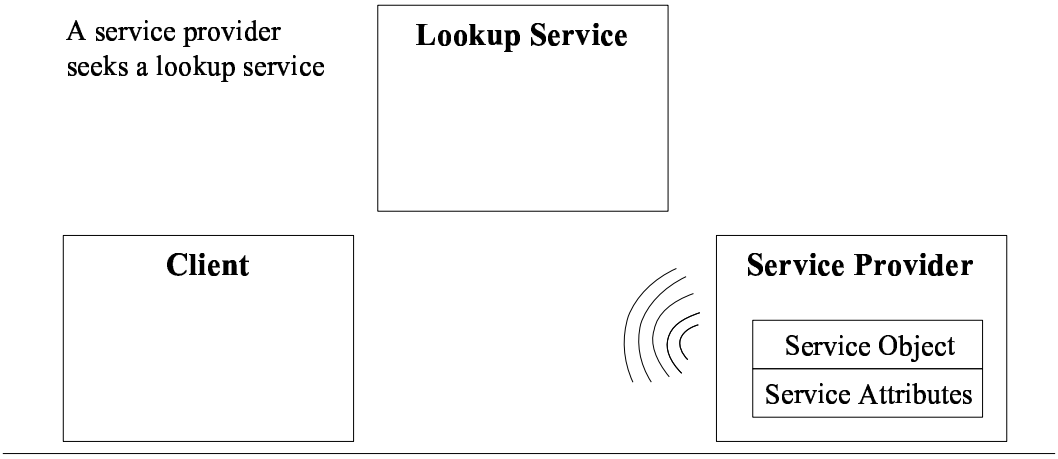

Figure 4.5: Jini Discovery.

Once the discovery phase is completed, the service provider loads an object to every lookup service found. The object contains the interface of the service and the all the required attributes to use it as discussed earlier. This activity is referred to as join and it is shown in Figure 4.6.

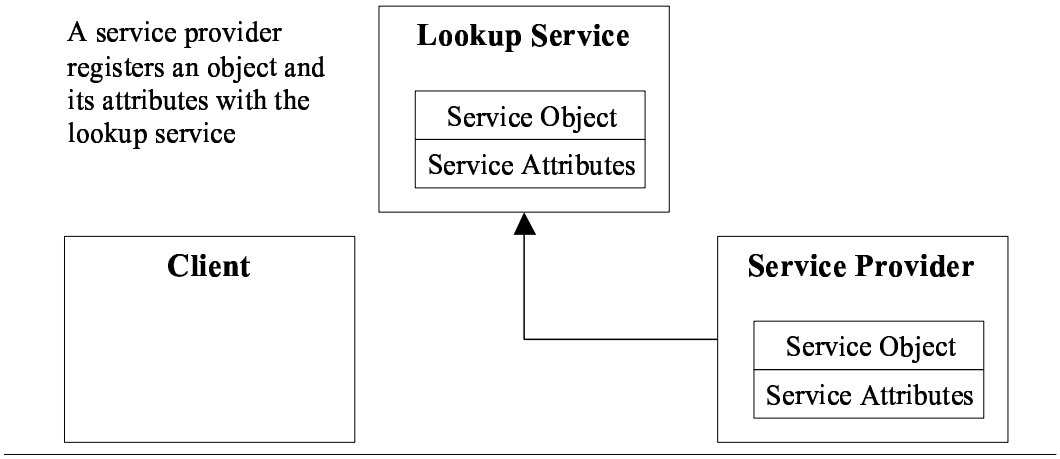

Figure 4.6: Jini Join.

In order for client services to find the newly registered services and use them, they have to use the lookup service. The client makes a request to the lookup service by specifying the service type and attributes that it wants to use. The lookup service provides an object to the client that satisfies the request as shown in Figure 4.7.

4.5.3 Service invocation

Service invocation deals with the aspects of invoking and using the services discovered in the environment. Once a service has been located using the lookup protocol, the client is able to invoke the service directly by using the service object (as illustrated in Figure

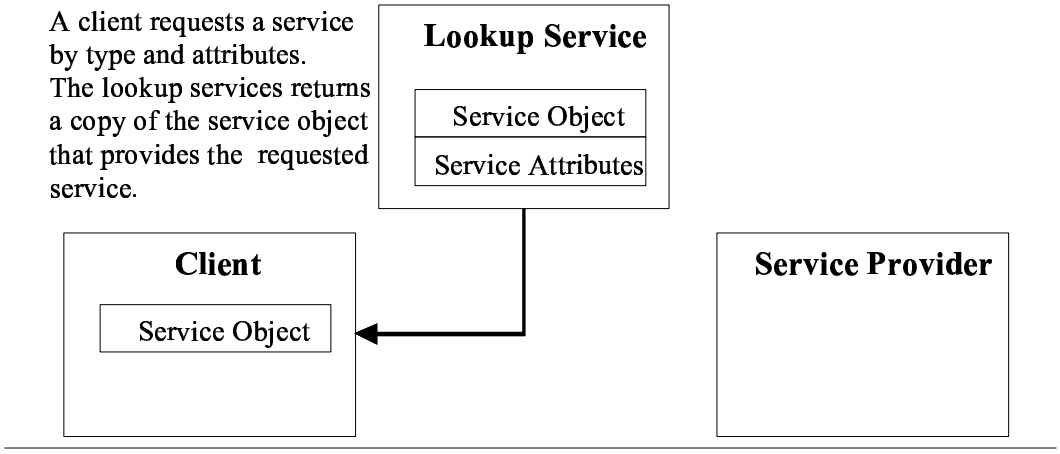

Figure 4.7: Jini Lookup.

4.8). The object is responsible for implementing the communication protocol between the two services. This approach provides great flexibility since service providers are free to implement the most appropriate protocol for their services. The only requirement is that the service object is written in Java. The actual service can be implemented in other languages as long as there exists a Java interface to invoke it.

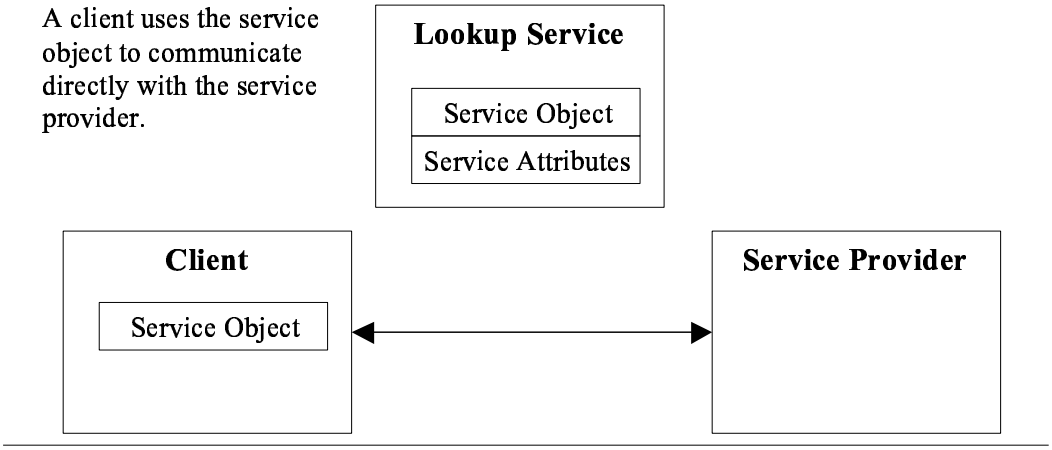

Figure 4.8: Client Uses Service.

Another feature of the environment is the notion of leasing [22]. When a service is requested, a lease is granted that allows access to the service for an amount of time. The service provider is responsible for negotiating the lease details with the client. Once a lease expires, the service provider can assume that the client does not require the resources anymore and the resources are freed. If the client still needs the resources, a lease renewal must be submitted before the lease expiration. Exclusive and Non-exclusive leases can exist. Exclusive leases guarantee that only one client is using the resources while non-exclusive allow sharing of the resources among many users. The leasing protocol is also used by the lookup service to make sure that registered services are still available.

4.5.4 Service integration

Service integration refers to the integrated use of existing services in order to define new services in the ISME environment. To provide support for this, ISME uses transactions and events as specified in the Jini architecture [41], as well as the Event-Condition-Action (ECA) paradigm as found in [44].

Jini specifies a transaction protocol in which a series of operations within one or multiple services can be wrapped in a single transaction. The protocol provided to handle this is based on a two-phase commit. The first phase of the transaction protocol is called the *voting phase* and the second is the *commit phase*. In the voting phase every service votes whether it has completed its part and it is ready to commit. In the second phase, the transaction coordinator issues a commit request to every service that has voted. The Jini Transaction protocol is very simple and its intention is to provide the basic interaction between services that is needed to implement the transaction semantics. The actual details on what the transaction does are implemented inside the service that coordinates the transaction.

Jini also supports distributed events. Objects allow other objects to register interest in

events and to receive a notification of when these events occur. In the ISME environment a service provider can decide whether to support specific events or not. Any events that are associated with a service will be known to any other service that wishes to invoke it using the discovery, join, lookup protocols. This event model is an extension of the event notification model used by JavaBeans [39] components.

In order to use the transaction and event protocols, we take advantage of the ECA paradigm. According to [44], ECA is a meta-language, implemented in XML, which allows the encoding of transaction logic by which processes interact. When new services are implemented, new requirements exist for how these services will interact with the existing ones. ECA allows the service provider to define scripts that check for events and perform actions under certain conditions. An ECA rule engine enacts ECA rules for all registered services. Whenever an event is received from a service of interest, the ECA engine finds the corresponding rules and after verifying that the specified conditions are met, it executes the specified actions. The actions are typically invocations of other services. Figure 4.9 illustrates the overall architecture of an integration environment that uses ECA to implement complex web-based services. The architecture has three major components: The web server that receives service notifications and delivers them to the rule engine. The rule engine that receives notifications and invokes services. The service repository that stores information about the services.

To demonstrate how services can be integrated using transactions, events and ECA scripts, let us consider the following example. We have two services in the ISME environment: one that parses Java source files and generates the JavaML representation and one that reads JavaML representation and reports the cyclomatic complexity of each method found in a JavaML representation. It becomes apparent that these two services

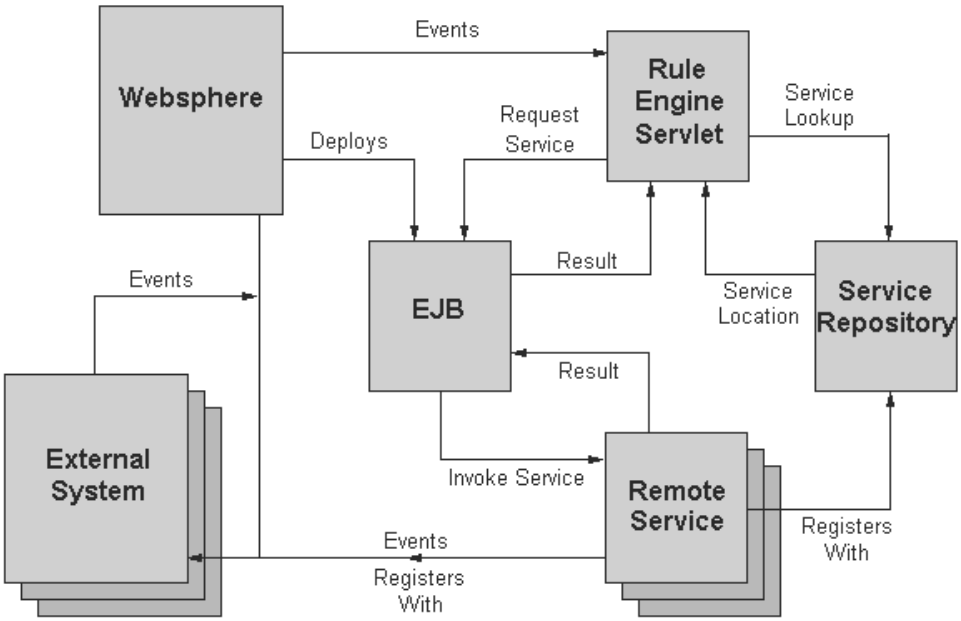

Figure 4.9: ECA architecture.

could be integrated into a higher-level service that reads Java source files and reports the cyclomatic complexity of each method found. This service automates the execution of the existing two separate services and hides all intermediate data from the client. The new service needs to implement the transaction logic and this can be done using ECA rules. For simplicity, we will refer to the parser service as A, to the cyclomatic complexity service as B, and to the new integrated service as C. The rule engine will contain the ECA rules defined in the Table 4.1.

This example does not need to use complex logic or the two-phase commit since it just uses two existing services in a pipeline. Nevertheless, using simple techniques such as the vote and commit and the ECA scripts it is possible to create very complex services by integrating existing ones.

Description	Event	Condition	Action
When service C is requested to perform its task it needs to invoke services A and B in order	Service C invoked	Input data is valid	Invoke service A
When service A is done it commits, and service B is invoked	Service A commits	Service A output is valid for service B input	Service B is invoked
When service B commits, service C returns the results	Service B commits	Service B output is valid according to the service C output	Service C returns the output received from service B

Table 4.1: Event Condition Action example.

4.6 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 one, is that of efficient storage. Efficiency 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 have been analyzed in the same way it is done for software that is developed. The third and final requirement is that of shared access. By its original specification, the ISME environment allows for multiple users to access and work on the same data. A safe way of sharing the same data is of great importance.

In such an integrated environment, most of the data will exist either in plain text format or in XML format. Therefore, it is important to take this into consideration when examining each of the following options: plain text files, control versioning systems and database management systems. Before we examine each of these options, it is important

to explain that no matter which option is adopted, it is still possible to implement it in a local computer or to make it available as another service inside ISME. This service is then accessed over the network by other services just as any other service in the environment. Multiple services that use different approaches could coexist. It is the responsibility of the user to select the most appropriate service for a maintenance task.

4.6.1 Text Files

This is the most common and easy to use approach. All the data is stored in plain text files regardless of what format the data uses. For each file, the following information is stored: name, size, last modification date, and access permissions. Most of the existing software maintenance tools use this approach when dealing with input and output data. Some of them also use proprietary binary formats to reduce file size and improve access speed.

The advantages of using text files include the ease of use and human readability. Easy of use comes as a result of working at the file level, which is a very simple concept. The majority of the applications in use today, allow users to store and retrieve data at the file level. In addition, working at the file level requires no specialized software and can be easily read and edited by a human. The majority of the programming languages that exist, provide routines that allow the user to work at the file level. These routines typically support the creation of new files and the modification of existing ones. There is no need for third party software, which makes it a cost-effective approach.

The disadvantages of this approach include the fact that it is inefficient for large amounts of data and has minimal concurrency control. When dealing with large files it is common to insert or modify small parts of the file that are scattered throughout the whole file. In this approach it is not possible to perform such operations since the whole file has to be processed again. Even when the data is well formatted (using XML

files) the file system does not understand any of these structures. Moreover, in most file and operating systems, when working at the file level there exists no standard way of concurrently accessing the same data. It is left up to the application developers to deal with how the files will be accessed within the application. File systems provide support for locking the files that are in use but this must be implemented in the application. The files system however provides access control over the files based on access permissions for groups of users.

4.6.2 Revision Control System and Concurrent Versions System

Revision Control System, or RCS in short, is a utility that keeps track of multiple file revisions and reduces the overall storage space. RCS allows for automatically storing and retrieving revisions of files, as well as merging and comparing them and keeping historical logs of the changes. The utility works at the file level and it allows the automation of many daily tasks. More information about RCS can be found in [31]. Concurrent Versions System, or CVS in short, is another utility that extends RCS to allow for concurrent access to a collection of files. Whereas RCS controls changes and revisions from one single directory, CVS allows the use of hierarchical collections of directories and subdirectories. With CVS, multiple users can concurrently access and modify the same files. This is accomplished by maintaining a single repository in which the master versions of all files are stored. When users need to modify some files, they can check them out, modify them, and then check them in the repository again. CVS has been a very popular tool in software development since it allows teams to works on the same files to produce a single project. In the same manner, it can be useful when performing software maintenance tasks.

The advantages of this approach include the ease of use, and that they offer inexpensive and excellent concurrency control. The RCS and CVS utilities are almost as easy

to use as working at the file level. However, users should be trained to understand the notions of checking files in and out of the repository and the concept of revisions. The majority of software development is currently done with these utilities or others that implement similar concepts. The maintenance of the utilities is minimal and as previously explained, they are very easy to use. The greatest advantage of these utilities is the concurrency support they provide. Latest releases of CVS even support concurrency with remote hosts using Internet protocols.

The main disadvantage of this approach is that it is inefficient for large amounts of data. Similarly to text files, RCS and CVS are not designed to work with files that are structured and therefore they cannot take advantage of those structures. However, the concept of revisions allows for more effective storage space use. For every revision, only the changes from the previous one are stored. When dealing with large amounts of data, if some of it remains the same over many revisions there will be significant savings in the storage space required.

4.6.3 Database Management System

The most common approach to store, access, and modify structured data is to use a Database Management System, or DBMS in short. Traditionally, relational databases store information in tables. Each table stores information about a single entity. The columns represent the attributes of the entity and the rows represent the actual values for an instance of this entity. If we consider a file to be an entity and the name, size and date its attributes, we can create a table that stores information about all files in our hard disk. Table 4.2 below is a good example.

Name	**Size**	**Date**
hello.c	1,256	01/01/2000
hello.h	345	01/01/2000
cppml.dtd	5,435	01/01/2000

Table 4.2: Sample table for storing files.

Using this tabular representation, we can represent very complex data representations. In reality, the tables cannot represent all the information inside the data structures. The tables simply store the information. The data representations can be mapped to an entity-relationship diagram (ERD) from which we derive the actual tables. It is then more appropriate to say that data can be represented by an ERD, which is in turn stored in tables. By looking at the tables, and by using the ERD, we can reconstruct the original data structures and all the information stored in them.

Using a DBMS to store information has numerous advantages over using simple files: Efficient storage,querying facilities, concurrent access support, and security. Databases are exceptional in allowing fast access to all the data stored within them. Searching for specific values and updating them is extremely fast due to the indexing abilities and fast algorithms they use. Moreover, databases are well known for their ability to store and manipulate large amounts of data. Updating data concurrently is a very well defined task in most of the DBMSs available today. Users are able to view and update the contents of tables at the same time. The DBMS employs transaction algorithms that coordinate how to read and write data to the databases without inconsistencies. Finally, within a DBMS, it is possible to provide different levels of security to meet the most demanding needs. Users and User groups can be granted access to specific tables and even columns inside the tables. In contrast to this, files only allow access control over the entire contents of each file.

However, there are also disadvantages when using a DBMS such as the complexity of

DBMS systems and the cost. Complexity deals with how easy it is to use DBMSs when compared to plain text files. While the concept of files is easy to understand and use daily, the concept of databases is a little more complex. Developing database applications requires understanding of all the underlying principles we discussed previously and knowledge of languages specific to databases like the Structured Query Language (SQL) used for querying databases. Mapping an existing data structure to an ERD is not a trivial task and significant consideration must be given to it. In addition, databases are very expensive systems to purchase and maintain. Typically the cost of purchasing such a system is minor when compared to the cost of the trained personnel required to program, operate and maintain the system.

Given the advantages that DBMSs offer us, it seems a good approach to use them as a repository of all the data used and generated by and from our software maintenance tools. All the representations used in the ISME environment are based on XML structures. In addition, most of the existing maintenance tools use representations that can be mapped to XML structures. Therefore, it seems crucial to be able to automate the mapping of XML structures to ERDs so that they can be stored in databases. DB2 XML Extender [21] from IBM does exactly that. Using the XML extender it is possible to store XML documents in the DB2 database as flat text files or map them to tables. The user is able to specify how the XML document will be stored in the database by creating a Document Access Definition (DAD) file. The DAD is defined in XML format. Using the DAD it is possible to associate an XML file with a DB2 database through two access and storage methods: XML columns and XML collections.

An XML column can be used to store the entire XML file as one piece inside the database as shown in Figure 4.10. For this purpose, the XML Extender provides a custom data type referred to as XMLCLOB. The XMLCLOB can be indexed using a set of functions defined by the XML Extender. This enables us to perform powerful

structural searches on the intact XML file. This approach is particularly useful when we want to use the database for management and search operations or as an archive of our XML files. The DAD file in this approach defines what tables are going to be created for indexing the selected parts of the XML file.

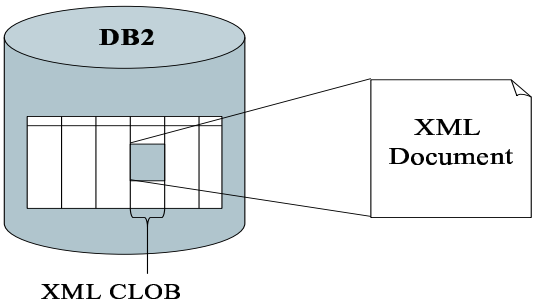

Figure 4.10: XML column inside DB2.

An XML collection can be used to map an XML file to a set of tables inside the database as shown in Figure 4.11. The XML Extender provides functions that map the hierarchical structure of the XML file to the relational structure of the database. This mapping is done using the DAD file in which every element defined in the DTD is mapped to a column in a table. When an XML file is stored in the database, the XML Extender processes and breaks down the files using the DAD and stores the values in the corresponding tables. When the XML document has to be retrieved from the database, the XML Extender automatically fetches the values stored in the tables and constructs the corresponding XML file. This approach is useful when entire XML files need to be stored but only small portions of them need to be retrieved. Also when working with large files, collections will provide a more efficient way of accessing the data.

The use of DAD files to implement both XML columns and collections is shown in Figure 4.12. Detailed examples of actual DAD files for both XML columns and collections

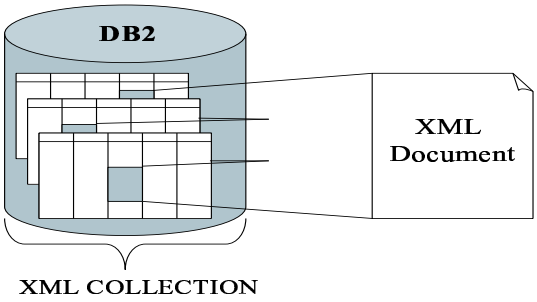

Figure 4.11: XML collection inside DB2.

are included in the Chapter 5.

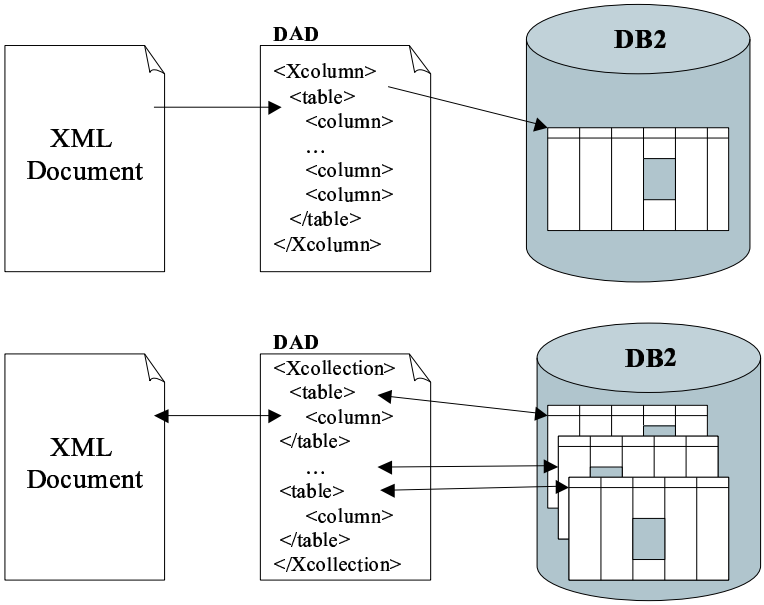

Figure 4.12: DAD files for XML columns and collections.

Chapter 5

Prototype tool and feasibility study

5.1 Introduction

In this chapter, we present the prototype ISME that is built as part of this work. All the components of the ISME are described in detail and examples are shown. The source code parsing tools that generate JavaML, CppML and OOML are evaluated for their performance in terms of size and processing time. Simple tools that compute metrics and operate directly on the XML source code representation are also presented. Finally, we explain how we used DB2 to store XML files using the aggregated OOML representation.

5.2 Prototype ISME

A prototype environment was built using the Java programming language. The reasons for choosing Java as the programming language are multiple. First of all, Java allows the final application to be portable. The prototype is able to run on a variety of platforms

for which a Java virtual machine has been developed. Secondly, ease of integration with the distributed technologies such as Jini. The Jini framework will become available from software vendors and we will only need to extend it to meet our requirements. Part of the extensions for the invocation and interactions of services, has already been developed in the ECA environment[44]. Finally, the majority of the supporting technologies are readily available for Java. By supporting technologies, we refer to technologies such as the XML APIs for processing XML files, DTD editors and viewers and XSLT tools for transformations.

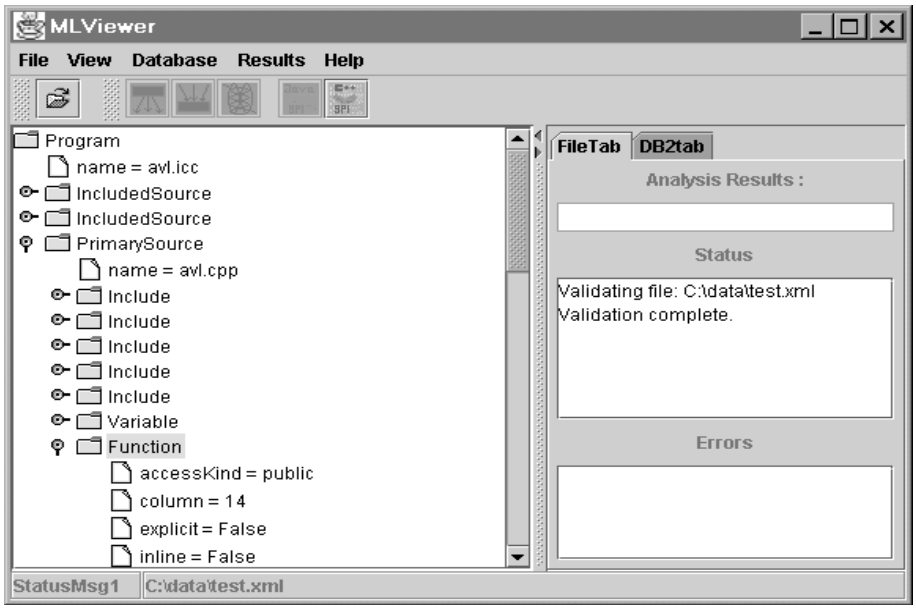

Figure 5.1: Screen shot of the User Interface of the prototype ISME.

The core part of the user interface of the prototype is the XML tree viewer as shown in Figure 5.1. The prototype is able to read JavaML, CppML and OOML representations directly from source files or from a DB2 database. When the XML file is loaded, it is validated using the corresponding DTD and then it is displayed on screen as a tree. The user is able to invoke tools that are registered to operate on the loaded representation,

by using the toolbar. All ISME tools are registered in a central repository and for each tool there exists a description and the input and output format definitions. In addition, each tool can be registered to handle specific events depending on what part of the tree these events originated from. For example, a tool that performs fan-in calculations for Method and Source elements found in the JavaML representation, will be registered in the central repository using the following XML document.

```
<Tool name="fan-in"
description = "Calculates fan-in for method and sources">
   <Input type="XML" dtd="JavaML">
      <Element>Method</Element>
      <Element>Source</Element>
   </Input>
   <Output type=Integer>
   </Output>
</Tool>
```

All the tools that are registered in the ISME prototype are linked to buttons on the toolbar at the top of the screen. In Figure 5.1, a CppML representation is loaded and we can see that one of the five buttons on the toolbar is enabled. This means that only one out of the five tools is registered to handle the CppML representations. In this case, the tool maps the CppML representation to the higher level OOML. The other tools that are displayed in Figure 5.1, correspond to tools that compute metrics and are registered for specific elements for the OOML representation. Theses metrics tools are described in more detail later in this chapter. Once a tool is invoked, the results are displayed in the right pane inside a textbox. Currently, there is support for storing the analysis results as an XML file. Storing information 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 #IMPLIED>
<!ELEMENT Analysis EMPTY>
<!ATTLIST Analysis
Type CDATA #REQUIRED
AppliesTo CDATA #REQUIRED
Result CDATA #REQUIRED>
```

The current ISME prototype supports the use of DB2 to store, index and retrieve XML files. In Figure 5.2, we see how a database can be populated once an XML tree is loaded in the environment. The database pane on the right allows the user to search and retrieve parts of the XML document by using SQL statements. The details on how the XML representation is stored in the database are discussed later in this chapter.

5.3 JavaML and CppML examples

In this section, we present the output that is generated from our prototype for some simple Java and C++ programs.

5.3.1 JavaML example

In the following example we present a segment of a Java program. Specifically, a single Java class that contains a single field and a single method, along with its corresponding XML representation are presented.

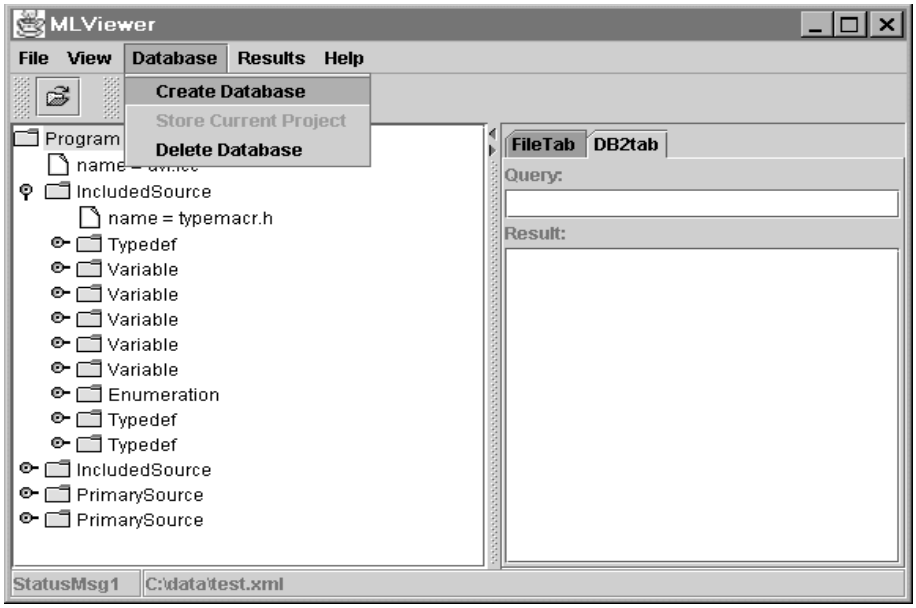

Figure 5.2: Screen shot of the prototype ISME with DB view.

Java source code

```
public class Car{
 int color;
 public int getColor(){
  return color;}
}
```

JavaML representation

```
<ClassDeclaration Identifier="Car">
 <FieldDeclaration>
  <PrimitiveType Type="int"></PrimitiveType>
  <VariableDeclaratorId Identifier="color"/>
 </FieldDeclaration>
 <MethodDeclaration Identifier="getColor">
  <ResultType>
```

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

5.3.2 CppML example

The C++ example is composed of two source files: putint.cpp and putint.h. Both, the C++ source code for these files, and the XML representation of the source files, which is stored in one XML document, can be found in Appendix D.

5.4 JavaML, CppML and OOML performance

In this section, we evaluate the proposed representations in terms of size and time. For each programming language, we used a number of input source files of different sizes to measure two parameters. The first is the size of the resulting XML based representation. It is expected that the size of the resulting XML based representations will be much greater than that of the actual source code. The reason for this is the extra characters required to represent the structure of the source file. The second factor is the time required by the source code parsers (or the mapping tool in the case of OOML) to generate these

representations. For each one of the two representations (JavaML and CppML), a table that summarizes the initial size of the source code, the size of the resulting XML based representations and the processing time required, is provided (Table 5.1, Table 5.2). All the experiments were performed using an Intel Pentium III processor running at 500MHz with 128MB of memory.

In Table 5.1, we summarize the results collected from processing Java source files. We can see that, on average, the output file which is based on the JavaML representation is 15 times larger than the input file which is plain Java source files. The ratio is higher than expected but a closer examination of the Java grammar used provides a good explanation for this. In the DTD for JavaML (Appendix A), we see that expressions are nested in such a way, so that every kind of expression is described in terms of other expressions. This means that to represent a Unary expression, we need to store many levels of other expressions in the XML file. In terms of time requirements, in Figure 5.3 we also graph the relationship between input file size and processing time. The result meets our expectations since the time required is directly related to the output file size (and input file size indirectly). Overall, the parsing of Java source files is very fast, even when very large files are generated as seen in the last row in Table 5.1

Source size	**JavaML size (bytes)**	**Time (sec)**
1359	4689	0.022
1382	4700	0.035
4257	83140	0.180
9519	201606	0.447
15440	98841	0.271
20733	509785	1.197
30343	655802	1.617
52028	573560	1.433
93720	2728477	6.432
207706	4081607	10.965

Table 5.1: Experimental results for JavaML.

Source size	**CppML size (bytes)**	**Time (sec)**
3927	17727	0.19
5030	19813	0.2
5365	31514	0.231
6305	34508	0.3
7171	65895	0.42
22376	75967	0.771
31983	85571	1.302
44375	190724	1.945
45576	284376	2.653
75065	455411	3.424
85922	267953	1.482

Table 5.2: Experimental results for CppML.

In Table 5.2, we summarize the results collected for the CppML representation. Since CppML is generated from parsing the VisualAge C++ Codestore the times shown correspond only to time required to generate the CppML representation from Codestore. The time required to generate the Codestore is not included since it represents the time for processing a complete project and not the single source file that we processed. In Figure 5.3 the relationship between the file size of the input source code and the file size of its corresponding XML representation, as well as the parsing and processing time required to generate the XML file are illustrated. In Figure 5.4, we present the same information but for C++ source code. As expected, the ratio between input and output is 5, much smaller than the ratio for JavaML. The reason for this as previously explained, is that JavaML representations are larger than expected due to the nesting of expressions. The processing also appears to be directly proportional to the output file size. Moreover, these results indicate that the proposed approach is scalable and tractable, in large systems.

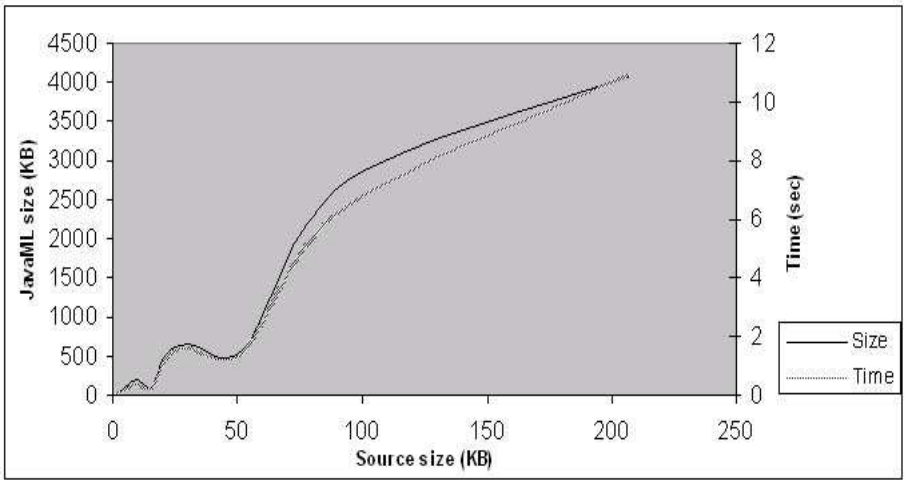

Figure 5.3: Graph of time and size results for JavaML.

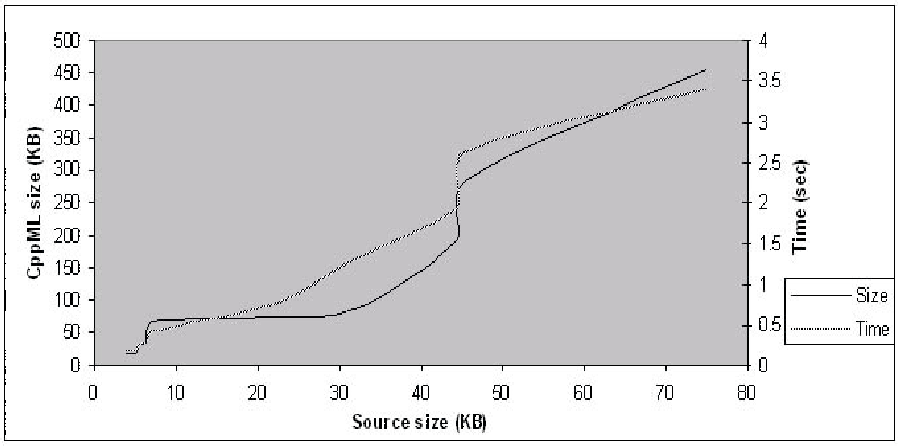

Figure 5.4: Graph of time and size results for CppML.

Finally, we summarize the results for the OOML representation. Since OOML retains only a fraction of the information found in either JavaML or CppML, it is expected that the resulting file size would be correspondingly small. In Tables 5.3 and 5.4, we present the results when using JavaML and CppML representations as the input files. The average ratio of the sizes in the first case is 0.07 while in the second is 0.25. We observe that the OOML representations that are generated from JavaML are approximately 3 times

smaller than those generated from CppML. This is exactly what we expected since the JavaML representation of the source code is 3 times larger when compared to that of the CppML and its corresponding source code. The processing times indicate that even large XML files can be processed fast enough to perform complex analysis tasks.

JavaML size (bytes)	**OOML size (bytes)**	**Time (sec)**
4689	647	1.212
4700	661	1.412
83140	3816	1.843
98841	6448	4.887
509785	29038	9.914
573560	34894	8.713
655802	43342	12.037
2728477	126596	35.451
4081607	192109	72.505

Table 5.3: Experimental results for JavaML to OOML.

CppML size (bytes)	**OOML size (bytes)**	**Time (sec)**
17727	5485	2.254
19813	6464	2.384
31514	8246	2.163
34508	13481	2.895
65895	19735	3.015
75967	12868	3.485
85571	17361	4.517
190724	43741	7.821
267949	62449	8.963
284376	66147	7.350
455645	117007	11.617

Table 5.4: Experimental results for CppML to OOML.

5.5 Tools for computing software metrics

Simple tools that calculate software metrics were developed to demonstrate how the proposed Integrated Software Maintenance Environment facilitates simple software maintenance tasks. These tools were developed as separate components to operate on the OOML representation 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 a transformation process is used to generate the OOML representation as discussed in Section 3.4. All the information the metric tools require is available in the OOML representation and the developer saves time by implementing the tools only once for both languages (C++ and Java). It must be noted that not all tools will be able to operate at the OOML level. The ones described here were specifically selected to demonstrate the advantages of working at the OOML level.

5.5.1 Fan-in

For any given method, this tool computes the total number of functions in the program that call the currently selected method. An approximation of this number 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 methods fan-ins contained within the Class. Similarly, we can define fan-in for source elements. All this means is that the fan-in tool is able to perform its 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 fan-in tool on the toolbar becomes enabled.

5.5.2 Fan-out

Fan-out is very similar to how fan-in works. However, in this case fan-out represents the number of calls to other methods that originate from a given method. As before, we can define the fan-out metric for Classes and Sources. The fan-out tools is also registered to handle events that originate from Method, Class and Source elements. It should be mentioned that constructing the call graph for object-oriented languages is a very complicated task as discussed in [45]. The approach we use is to assume that the declared and implementing types are the same. When the declared type resolves to a sub-type, or implementing type then more advanced call graph construction algorithms like Class Hierarchy Analysis [25, 27] and Rapid Type Analysis [8, 7] can be used.

5.5.3 Cyclomatic Complexity

The Cyclomatic Complexity or McCabe function, as it is more commonly known, computes a number that describes how complex the control flow of a 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. The implementation of this tool in ISME is registered only with Method elements.

5.6 Integration with Rigi

Rigi is a tool designed for program understanding and visualization. The basic component of Rigi is the graph model which is expressed using the RSF notation. The graph model allows the modeling of software artifacts such as subsystems, procedures, variables, and calls to name a few. These artifacts can be used for visualizing the structure of large software systems and for performing analysis tasks. As stated in [42], the goals of Rigi

are:

- to provide an infrastructure for research and practice in program understanding, and

- to discover abstractions in large software systems, and pass this information on to software engineers for maintenance and reengineering purposes.

In order to demonstrate how a tool like Rigi can be integrated with the ISME environment we created a mapping from OOML to a Rigi domain model. The domain model for Rigi contains a variety of high level artifacts such as the ones listed in Table 5.5. These artifacts capture the main structure of the program and also some relationships among artifacts like function and static data calls.

Nodes	**Arcs**
Program	call
Source	varuse
Class	level
Method	
Variable	

Table 5.5: Rigi domain for representing OOML.

In the prototype discussed in this thesis, a tool was implemented as a Rigi driver that processes OOML representations and generates the RSF tuples that Rigi could operate upon. The tool is available from the ISME toolbar as shown in Figure 5.5. Once an OOML representation is loaded in ISME the Rigi button becomes enabled. By clicking the button, the user is prompted to specify a filename in which the generated RSF will be stored. Once the RSF is generated, the Rigi tool is invoked and the user can start working with it immediately.

Figure 5.5: A view of the ISME toolbar.

As an example, we used the C++ AVL libraries source code. The AVL libraries are a set of APIs for creating and manipulating trees. The original version of the AVL libraries, which was implemented in C, can be obtained from GNU [30]. The C++ version we used was automatically generated from the C version as part of the work presented in [43]. Initially the C++ source code was parsed and represented using CppML. The resulting representation was then mapped to the more generic representation OOML using the mapping tools previously described. Finally, the OOML representation was used to generate the RSF information that is used by Rigi. In Figure 5.6, we see the structure of the AVL libraries as constructed by the Rigi tool. Existing analysis tools that are built on Rigi can be used on the generated RSF.

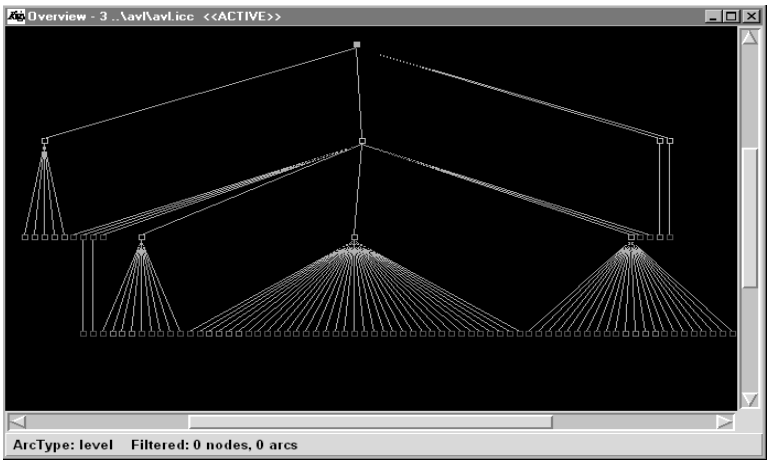

Figure 5.6: AVL program structure as seen in Rigi.

5.7 DAD Example

As described in Chapter 4, it is possible to use a database management system to store XML documents. In the ISME prototype, we created a mapping from OOML documents to the DB2 database by using the IBM XML Extender [21]. In the approach presented in this thesis, columns are used to index some key parts of the XML document to allow for fast searches. The XML document is stored as a large object while the indexed parts can still be retrieved individually. To illustrate how this mapping is performed we present the Document Access Definition file segment below.

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "c:\dxx\dtd\dad.dtd">
<DAD>
<dtdid>c:/data/cppml.dtd</dtdid>
<validation>NO</validation>
<Xcolumn>
   <table name="source_tab">
      <column  name="filename"
         type="varchar(50)"
         path="/Program/*[@name]"
         multi_occurrence="YES"/>
   </table>
</Xcolumn>
</DAD>
```

The DAD file is used to index every name of every source element that appears in the XML document. A table called source_tab is created and all the indexed source names are stored in the column called filename. Once the DAD file is constructed and

the corresponding database is created, we can use SQL commands to store or retrieve XML data. As shown in Figure 5.2 before, the prototype ISME allows the user to enter SQL queries and get the results from the database for a given XML file.

Chapter 6

Conclusion

6.1 Summary and Conclusions

The role of software maintenance has become an increasingly important component of software engineering. The task of building and maintaining large complex software systems imposes the need for software maintenance and analysis tools. Throughout the life cycle of software systems, maintenance ensures that systems remain functional and that they can evolve to meet new requirements.

The work presented in this thesis discusses an environment for supporting software maintenance tasks by integrating various CASE tools in one environment. The environment allows for maintenance tools to be developed independently and deployed in a distributed fashion. Language domain models are used to represent source code at the AST level and facilitate data interchange between the maintenance tools deployed in the environment. In addition, more complex maintenance tasks could be addressed by combining existing tools that perform simpler tasks. The environment presents a basis for a software development and maintenance environment where CASE tools can be integrated to support collaborative software engineering. By utilizing databases and versioning sys-

tems, the environment provides support for storing and working with large amounts of processed data.

The main contributions of this work can be summarized as follows:

- Design and implementation of a prototype for mapping grammars from programming languages to DTDs. The DTDs are used to describe source code as XML documents. These mappings assist developers in generating their own transformations to address specific source code representation requirements.

- Design and implementation of program representations based on XML that can be used to represent Java and C++ source code (JavaML and CppML respectively). These source code representations can be interchanged between tools. Composition of domain models to create aggregate source code representations has also been investigated.

- Design and implementation of tools that use Java and C++ as input and generate representations using JavaML and CppML respectively. In addition, tools that map the low level representations to the more generic one OOML have been implemented. Simple tools for calculating software metrics were also developed to demonstrate the use of the XML based representations.

- Design and implementation of data integration techniques that allow existing software tools to exchange input and output data. By employing intermediate representations, it was shown that software tools can be combined to reduce overhead and to compare results. Moreover, analysis results from tools could be used as input to other tools.

- Incorporation with repository services that allow for the storage of large amounts

of XML data. These services can be built on top of industrial strength database systems.

6.2 Future Work

The work presented here can be extended in various ways. Some extensions can focus on pure implementation work and some can address more theoretical issues. The following two paragraphs outline the possible future work in both areas.

Development of DTDs for a variety of programming languages can make the ISME environment an attractive solution to a wider audience. Languages like Pascal, PL/IX and Cobol in which most legacy software is written in are the first priority. The implementation of more complex maintenance tools than the ones presented here will help demonstrate the features of the ISME environment. Integration with third-party software engineering tools, will allow to present how data integration can be achieved in a more generic setting.

A more theoretical area of work that arises from the ISME work is the design of generic libraries that will allow easy manipulation of XML files. These libraries should include support for searching, transforming and unifying XML structures. Identifying the functionality of these libraries will require the study and analysis of the features that the most common maintenance task require. Finally, feasibility studies need to be performed to deal with the ease of use, the user interface and the distributed nature of the ISME environment. These studies can provide valuable feedback in improving ISME and providing better support for collaborative software development and maintenance.

Bibliography

[1] Biztalk framework. http://www.biztalk.org/Biztalk/framework.asp.

[2] Biztalk website. http://www.w3.org/TR/NOTE-MCF-XML/MCF-tutorial.html.

[3] *The Toolbus*. http://adam.wms.uva.nl/ olivierp/toolbus/toolbus.html.

[4] C3ds: Control and coordination of complex distributed services. http:// www.newcastle.research.ec.org/c3ds/programme.html, 1999.

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

[6] F.E. Allen. Control flow analysis. In *ACM SIGPLAN*, 1970.

[7] D.F. Bacon. Fast and effective optimization of statically typed object-oriented languages. PhD thesis,University of California at Berkeley, 1997.

[8] D.F. Bacon and P.F. Sweeney. Fast static analysis of c++ virtual function calls. In *In proceedings of OOPSLA,*, pages 324–341, 1996.

[9] J. C. M. Baeten and W. P. Weijland. Process algebra. In *Cambridge Tracts in Theoretical Computer Science 18*. Cambridge, University Press, 1990.

[10] Grady Booch, James Rumbaugh, and Ivan Jacobson. *The Unified Modelling Language User Guide*. Addison Wesley, 1999.

[11] Ivan Thomas Bowman. Architecture recovery for object-oriented systems, 1999.

[12] N. Bradley. *The XML Companion*. Addison-Wesley, 1998.

[13] T. Bray and R.V. Guha. An mcf tutorial. http://www.w3.org/TR/NOTE-MCF-XML/MCF-tutorial.html.

[14] A W. Brown, D. J. Carney, E.J. Morris, D. B. Smith, and P.F. Zarrella. *Principles of CASE Tool Integration*. OXFORD, 1994.

[15] N. Chomsky. Three models for the description of language. In *Transactions on the Information Theory 2:3*, pages 113–124, 1998.

[16] World Wide Web consortium. Extensible markup language (xml). http://www.w3.org/XML.

[17] World Wide Web consortium. Resource description framework (rdf) schema specification 1.0. http://www.w3.org/TR/2000/CR-rdf-schema-20000327.

[18] World Wide Web consortium. Xml schema requirements. http://www.w3.org/TR/NOTE-xml-schema-req.

[19] World Wide Web consortium. Xsl tranformations (xslt) version 1.0. http://www.w3.org/TR/xslt.

[20] IBM Corporation. Alphaworks. http://www.alphaworks.ibm.com.

[21] IBM Corporation. Db2 xml extender. http://www.ibm.com/software/data/db2 /extenders/xmlext/.

[22] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 1994.

[23] DATRIX. *Abstract Semantic Graph Reference Manual*. BELL Canada, version 1.3 edition.

[24] U. Dayal. Active database management systems. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, 1998.

[25] J. Dean, D.Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *In proceedings of ECOOP*, 1995.

[26] P.T. Devanbu, D. S. Rosenblum, and A.L. Wolf. Generating testing and analysis tools with aria. In *ACM Transactions on Software Engineering and Methodology*, volume 5, pages 42–62, 1996.

[27] A. Diwan, J.E.B Moss, and K.S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *In proceedings of OOPSLA*, pages 292–305, 1996.

[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *The program dependence graph and its use in optimization*, pages vol 9, no. 3, 319–349, 1987.

[29] Apache Software Foundation. The apache xml project. http://xml.apache.org.

[30] GNUProject. Gnu's not unix. http://www.gnu.org.

[31] GNUProject. Revision control system (rcs). http://www.gnu.org /software/rcs/rcs.html.

[32] E. R. Harold. *XML Bible*. IDG Books Worldwide, 1999.

[33] M.J Harrold, B. Malloy, and G. Bothermel. Constructing program dependence graphs using a parser. *ACM International Symposium on Software Testing and Analysis*, 1993.

[34] R. Holt. An introduction to ta: The tuple-attribute language. http:// plg.uwaterloo.ca/ holt/papers/ta.html, 1998.

[35] IBM. Visualage c++. http://www.ibm.com/software/ad/vacpp.

[36] Reasoning Inc. Cbms whitepaper. http://www.reasoning.com/tech/tech.html, 2000.

[37] Rick Kazman and S. Jeromey Carriere. View extraction and view fusion in architectural understanding. In *Fifth International Conference on Software Reuse*, 1998.

[38] J Lu, J. Mylopoulos, and J. Ho. Towards extensible information brokers based on xml. In *To appear in 12th Conference on Advanced Information Systems Engineering*, 2000.

[39] Sun Microsystems. Javabeans. http://java.sun.com/beans.

[40] Sun Microsystems. Javacc the parser generator. http://www. metamata.com/JavaCC.

[41] Sun Microsystems. *Jini Architecture Specification*, 1.0.1 edition, 1999.

[42] Hausi Muller. Rigi, a visual tool for understanding legacy systems. http:// www.rigi.csc.uvic.ca.

[43] Prashant Patil. Migration of procedural systems to object oriented, 1999.

[44] K. Kontogiannis R. Gregory. Customizable service integration in web-enabled environments. To be published.

[45] Derek Rayside, Steve Reuss, Erik Hedges, and Kostas Kontogiannis. The effect of call graph construction algorithms for object-oriented programs on automatic analysis. In *In proceedings of IWPC*, Limerik, Ireland, 2000.

[46] University of Victoria. *RIGI*. http://rigi.uvic.ca.

[47] M.G.J van den Brand et al. Implementation of a prototype for the new asf+sdf meta-environment. In *2nd International Workshop on the Theory and Practise of Algebraic Specifications*, Amsterdam, 1997.

Appendix A

DTD for JavaML representation

```
<!ENTITY % ClassBodyDeclaration "(FieldDeclaration|MethodDeclaration
|Initializer|ConstructorDeclaration|NestedClassDeclaration
|NestedInterfaceDeclaration)">
<!ENTITY % InterfaceMemberDeclaration "(FieldDeclaration|MethodDeclaration
|NestedClassDeclaration|NestedInterfaceDeclaration)">
<!ENTITY % Statement "(LabeledStatement|IfStatement|WhileStatement
|ForStatement|Block|EmptyStatement|StatementExpression|SwitchStatement
|DoStatement|BreakStatement|ContinueStatement|ReturnStatement
|SynchronizedStatement|ThrowStatement|TryStatement)">
<!ENTITY % BlockStatement "(LocalVariableDeclaration|(%Statement;)
|ClassDeclaration|InterfaceDeclaration)">

<!ELEMENT CompilationUnit (PackageDeclaration?,ImportDeclaration*,
TypeDeclaration*)>
<!ATTLIST CompilationUnit
  Name CDATA #IMPLIED>
<!ELEMENT PackageDeclaration (Name)>
```

```
<!ELEMENT ImportDeclaration (Name)>
<!ATTLIST ImportDeclaration Single (True|False) "False">
<!ELEMENT TypeDeclaration (ClassDeclaration|InterfaceDeclaration)?>
<!ELEMENT Literal EMPTY>
<!ATTLIST Literal
Type (Integer|FloatingPoint|Boolean|Character|String|Null) "Integer"
Value CDATA #REQUIRED>
<!ELEMENT Type (PrimitiveType|Name)>
<!ATTLIST Type ArraySize CDATA #IMPLIED>
<!ELEMENT PrimitiveType EMPTY>
<!ATTLIST PrimitiveType
Type (boolean|byte|short|int|long|char|float|double) "boolean">
<!ELEMENT Name EMPTY>
<!ATTLIST Name Identifier CDATA #IMPLIED>
<!ELEMENT ClassDeclaration (UnmodifiedClassDeclaration)>
<!ATTLIST ClassDeclaration
isAbstract (True|False) "False"
isFinal (True|False) "False"
isPublic (True|False) "False">
<!ELEMENT UnmodifiedClassDeclaration (Name?,Name*,ClassBody)>
<!ATTLIST UnmodifiedClassDeclaration
Identifier CDATA #IMPLIED
Extends (True|False) "False">
<!ELEMENT NestedClassDeclaration (UnmodifiedClassDeclaration)>
<!ATTLIST NestedClassDeclaration
isStatic (True|False) "False"
isAbstract (True|False) "False"
isFinal (True|False) "False"
isPublic (True|False) "False"
```

```
isProtected (True|False) "False"
isPrivate (True|False) "False">
<!ELEMENT ClassBody ((%ClassBodyDeclaration;)*)>
<!ELEMENT FieldDeclaration (Type,VariableDeclarator+)>
<!ATTLIST FieldDeclaration
isPublic (True|False) "False"
isProtected (True|False) "False"
isPrivate (True|False) "False"
isStatic (True|False) "False"
isFinal (True|False) "False"
isTransient (True|False) "False"
isVolatile (True|False) "False">
<!ELEMENT VariableDeclarator (VariableDeclaratorId,VariableInitializer?)>
<!ELEMENT VariableInitializer (ArrayInitializer|Expression)>
<!ELEMENT VariableDeclaratorId EMPTY>
<!ATTLIST VariableDeclaratorId
Identifier CDATA #IMPLIED
ArraySize CDATA #IMPLIED>
<!ELEMENT MethodDeclaration (ResultType,MethodDeclarator,Name*,Block?)>
<!ATTLIST MethodDeclaration
isPublic (True|False) "False"
isProtected (True|False) "False"
isPrivate (True|False) "False"
isStatic (True|False) "False"
isAbstract (True|False) "False"
isFinal (True|False) "False"
isNative (True|False) "False"
isSynchronized (True|False) "False">
<!ELEMENT MethodDeclarator (FormalParameter*)>
```

```
<!ATTLIST MethodDeclarator
Identifier CDATA #IMPLIED
ArraySize CDATA #IMPLIED>
<!ELEMENT FormalParameter (Type,VariableDeclaratorId)>
<!ATTLIST FormalParameter
isFinal (True|False) "False">
<!ELEMENT Initializer (Block)>
<!ELEMENT ConstructorDeclaration  (FormalParameter*,Name*
,ExplicitConstructorInvocation?,Block?)>
<!ATTLIST ConstructorDeclaration
isPublic (True|False) "False"
isProtected (True|False) "False"
isPrivate (True|False) "False"
Identifier CDATA #IMPLIED>
<!ELEMENT ExplicitConstructorInvocation (PrimaryExpression?,Arguments?)>
<!ATTLIST ExplicitConstructorInvocation
Type (this|super) "this">
<!ELEMENT InterfaceDeclaration (UnmodifiedInterfaceDeclaration)>
<!ATTLIST InterfaceDeclaration
isAbstract (True|False) "False"
isPublic (True|False) "False">
<!ELEMENT NestedInterfaceDeclaration (UnmodifiedInterfaceDeclaration)>
<!ATTLIST NestedInterfaceDeclaration
isStatic (True|False) "False"
isAbstract (True|False) "False"
isFinal (True|False) "False"
isPublic (True|False) "False"
isProtected (True|False) "False"
isPrivate (True|False) "False">
```

```
<!ELEMENT UnmodifiedInterfaceDeclaration (Name*,InterfaceBody?)>
<!ATTLIST UnmodifiedInterfaceDeclaration
Identifier CDATA #IMPLIED>
<!ELEMENT InterfaceBody ((%InterfaceMemberDeclaration;)*)>
<!ELEMENT ArrayInitializer (VariableInitializer*)>
<!ELEMENT Block ((%BlockStatement;)*)>
<!ELEMENT LocalVariableDeclaration (Type,VariableDeclarator+)>
<!ATTLIST LocalVariableDeclaration
isFinal (True|False) "False">
<!ELEMENT EmptyStatement EMPTY>
<!ELEMENT LabeledStatement ((%Statement;))>
<!ATTLIST LabeledStatement
Identifier CDATA #IMPLIED>
<!ELEMENT StatementExpression (PreIncrementExpression|PreDecrementExpression
|(PrimaryExpression,(AssignmentOperator,Expression)?))>
<!ATTLIST StatementExpression Type (Increment|Decrement|None) "None">
<!ELEMENT SwitchStatement (Expression,((SwitchLabel,(%BlockStatement;)*)*))>
<!ELEMENT SwitchLabel (Expression?)>
<!ATTLIST SwitchLabel Type (case|default) "case">
<!ELEMENT IfStatement (Expression,(%Statement;),(%Statement;)?)>
<!ELEMENT WhileStatement (Expression,(%Statement;))>
<!ELEMENT DoStatement ((%Statement;),Expression)>
<!ELEMENT ForStatement (ForInit?,Expression?,ForUpdate?,(%Statement;))>
<!ELEMENT ForInit (LocalVariableDeclaration|StatementExpression*)>
<!ELEMENT ForUpdate (StatementExpression*)>
<!ELEMENT BreakStatement EMPTY>
<!ATTLIST BreakStatement Identifier CDATA #IMPLIED>
<!ELEMENT ContinueStatement EMPTY>
<!ATTLIST ContinueStatement Identifier CDATA #IMPLIED>
```

```
<!ELEMENT ReturnStatement (Expression?)>
<!ELEMENT ThrowStatement (Expression)>
<!ELEMENT SynchronizedStatement (Expression,Block)>
<!ELEMENT TryStatement (Block,(FormalParameter,Block)*,Block?)>
<!ATTLIST TryStatement HasFinally (True|False) "False">
<!ELEMENT Arguments (Expression*)>
<!ELEMENT PrimaryExpression (PrimaryPrefix,PrimarySuffix*)>
<!ELEMENT PrimaryPrefix (Literal|Expression|AllocationExpression
|ResultType|(Name,Arguments?))?>
<!ATTLIST PrimaryPrefix
isThis (True|False) "False"
isSuper (True|False) "False"
Identifier CDATA #IMPLIED>
<!ELEMENT PrimarySuffix (AllocationExpression|Expression|Arguments)?>
<!ATTLIST PrimarySuffix
isThis (True|False) "False"
Identifier CDATA #IMPLIED>
<!ELEMENT ResultType (Type?)>
<!ELEMENT AllocationExpression ((PrimitiveType,ArrayDimsAndInits)
|(Name,(ArrayDimsAndInits|(Arguments?,ClassBody?))?))>
<!ELEMENT ArrayDimsAndInits (Expression|ArrayInitializer)>
<!ATTLIST ArrayDimsAndInits ArraySize CDATA #IMPLIED>
<!ELEMENT Expression (ConditionalExpression,(AssignmentOperator,Expression)?)>
<!ELEMENT AssignmentOperator EMPTY>
<!ATTLIST AssignmentOperator Type (Simple|Multiply|Divide|Remainder|Plus|Minus
|LeftShift|RightShift|RightShiftUnsigned|AND|XOR|OR) "Simple">
<!ELEMENT ConditionalExpression  (ConditionalOrExpression
,(Expression,ConditionalExpression)?)>
<!ELEMENT ConditionalOrExpression (ConditionalAndExpression+)>
```

```
<!ELEMENT ConditionalAndExpression (InclusiveOrExpression+)>
<!ELEMENT InclusiveOrExpression (ExclusiveOrExpression+)>
<!ELEMENT ExclusiveOrExpression (AndExpression+)>
<!ELEMENT AndExpression (EqualityExpression+)>
<!ELEMENT EqualityExpression (InstanceOfExpression, (EqualityOperator
,InstanceOfExpression)*)>
<!ELEMENT EqualityOperator EMPTY>
<!ATTLIST EqualityOperator Type (Equal|NotEqual) "Equal">
<!ELEMENT InstanceOfExpression (RelationalExpression,Type?)>
<!ELEMENT RelationalExpression (ShiftExpression,(RelationalOperator,
ShiftExpression)*)>
<!ELEMENT RelationalOperator EMPTY>
<!ATTLIST RelationalOperator Type (Less|Greater|LessOrEqual|GreaterOrEqual)
"Less">
<!ELEMENT ShiftExpression (AdditiveExpression
,(ShiftOperator,AdditiveExpression)*)>
<!ELEMENT ShiftOperator EMPTY>
<!ATTLIST ShiftOperator Type (Left|Right|RightUnsigned) "Left">
<!ELEMENT AdditiveExpression (MultiplicativeExpression
,(AdditiveOperator,MultiplicativeExpression)*)>
<!ELEMENT AdditiveOperator EMPTY>
<!ATTLIST AdditiveOperator Type (Add|Subtract) "Add">
<!ELEMENT MultiplicativeExpression (UnaryExpression
,(MultiplicativeOperator,UnaryExpression)*)>
<!ELEMENT MultiplicativeOperator EMPTY>
<!ATTLIST MultiplicativeOperator Type (Multiply|Divide|Remainder) "Multiply">
<!ELEMENT UnaryExpression ((UnaryOperator,UnaryExpression)
|PreIncrementExpression|PreDecrementExpression|UnaryExpressionNotPlusMinus)>
<!ELEMENT UnaryOperator EMPTY>
```

```
<!ATTLIST UnaryOperator Type (Plus|Minus|BitwiseComplement|LogicalComplement)
 "Plus">
<!ELEMENT PreIncrementExpression (PrimaryExpression)>
<!ELEMENT PreDecrementExpression (PrimaryExpression)>
<!ELEMENT UnaryExpressionNotPlusMinus ((UnaryOperator,UnaryExpression)
|CastExpression|PostfixExpression)>
<!ELEMENT CastExpression ((Type,UnaryExpression)
|(Type,UnaryExpressionNotPlusMinus))>
<!ELEMENT AssignementOperator EMPTY>
<!ATTLIST AssignementOperator Type (Simple|Multiply|Divide|Remainder|Plus
|Minus|LeftShift|RightShift|RightShiftUnsigned|AND|XOR|OR) "Simple">
<!ELEMENT PostfixExpression (PrimaryExpression)>
<!ATTLIST PostfixExpression Type (Increment|Decrement|None) "None">
```

Appendix B

DTD for CppML representation

```
<!ENTITY % Declaration "(PredefinedType|Function|Variable|Class
|Enumeration|Typedef|Friend|Namespace|Using|UsingDirective)">
<!ENTITY % Expression "(NameExpression|ThisExpression|LiteralExpression
|FunctionCallExpression|UnaryExpression|BinaryExpression|ConditionalExpression
|CastExpression|FunctionCastExpression|SizeofTypeExpression
|TypeidofTypeExpression|OffsetofExpression|NewExpression|DeleteExpression
|ThrowExpression|MemberExpression|ParenthesesExpression)">
<!ENTITY % Statement "(BreakStatement|CaseLabelStatement|ContinueStatement
|DeclarationStatement|DefaultStatement|DoStatement|ExpressionStatement
|ForStatement|GotoStatement|IfStatement|LabelStatement|LexicalBlockStatement
|ReturnStatement|SwitchStatement|WhileStatement)">

<!ELEMENT Program (PrimarySource|IncludedSource)*>
<!ATTLIST Program
name CDATA #IMPLIED>
<!ELEMENT PrimarySource (Include*,Macro*,(%Declaration;)*)>
<!ATTLIST PrimarySource
```

```
name CDATA #IMPLIED>
<!ELEMENT IncludedSource (Include*,Macro*,(%Declaration;)*)>
<!ATTLIST IncludedSource
name CDATA #IMPLIED>
<!ELEMENT Include EMPTY>
<!ATTLIST Include name CDATA #IMPLIED
system (True|False) "False">
<!ELEMENT PredefinedType EMPTY>
<!ATTLIST PredefinedType
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public"
kind (Ellipsis|Wchar_t|Void|Bool|Char|SignedChar
|UnsignedChar|ShortInt|SignedShortInt|UnsignedShortInt
|Int|SignedInt|UnsingedInt|LongInt|SignedLongInt
|UnsignedLongInt|LongLongInt|SignedLongLongInt
|UnsignedLongLongInt|Float|Double|LongDouble) "Ellipsis">
<!ELEMENT Function (TypeDescriptor, FunctionBody?)>
<!ATTLIST Function
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public"
kind (ExternFileScope|StaticFileScope|Constructor|Destructor
|ConversionOperator|StaticMember|RegularMember)
"ExternFileScope"
operator (New|Delete|ArrayNew|ArrayDelete|Call|Arrow|PreIncrement
|PreDecrement|BitwiseNot|LogicalNot|Negate|UnaryPlus|Address
```

```
|Indirect|PostIncrement|PostDecrement|Assign|AddAssign
|SubtractAssign|MultiplyAssign|DivideAssign|RemainderAssign
|RightShiftAssign|LeftShiftAssign|AndAssign|XorAssign|OrAssign
|LogicalOr|LogicalAnd|BitwiseOr|BitWiseXor|BitwiseAnd|Equal
|NotEqual|GreaterOrEqual|LessOrEqual|Greater|Less|LeftShift
|RightShift|Add|Subtract|Multiply|Divide|Remainder|ArrowStar
|Subscript|Comma) "New"
virtual (True|False) "False"
purevirtual (True|False) "False"
inline (True|False) "False"
explicit (True|False) "False">
<!ELEMENT FunctionBody ((%Statement;)*)>
<!ELEMENT Variable (TypeDescriptor*, Initializer?)>
<!ATTLIST Variable
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public"
kind  (ExternFileScope|StaticFileScope|StaticMember
|RegularMember|BitFieldMember|Enumerator
|AutoLocal|StaticLocal|FunctionParameter
|TemplateAliasParameter) "ExternFileScope"
bitfieldWidth CDATA #IMPLIED
enumeratorValue CDATA #IMPLIED
UnnamedBitfield (True|False) "False"
UnboundedArray (True|False) "False"
mutable (True|False) "False">
<!ELEMENT Initializer (DirectInitializer|ExpressionInitializer
|BraceListInitializer|ImplicitInitializer)>
```

```
<!ELEMENT DirectInitializer ((%Expression;)*)>
<!ELEMENT ExpressionInitializer ((%Expression;))>
<!ELEMENT BraceListInitializer (Initializer*)>
<!ELEMENT ImplicitInitializer EMPTY>
<!ELEMENT NameExpression EMPTY>
<!ATTLIST NameExpression
name CDATA #IMPLIED>
<!ELEMENT ThisExpression EMPTY>
<!ELEMENT LiteralExpression (TypeDescriptor*)>
<!ATTLIST LiteralExpression
value CDATA #IMPLIED>
<!ELEMENT FunctionCallExpression ((%Expression;),(%Expression;)*)>
<!ELEMENT UnaryExpression ((%Expression;))>
<!ATTLIST UnaryExpression
kind (opPreIncrement|opPreDecrement|opBitwiseNot
|opLogicalNot|opNegate|opUnaryPlus|opAddress|opIndirect
|opPostIncrement|opPostDecrement|opSizeof|opTypeid
|opVaStart|opVaArg) "opPreIncrement">
<!ELEMENT BinaryExpression ((%Expression;),(%Expression;))>
<!ATTLIST BinaryExpression
kind (opAssign|opAddAssign|opSubtractAssign
|opMultiplyAssign|opDivideAssign|opRemainderAssign
|opRightShiftAssign|opLeftShiftAssign|opAndAssign
|opXorAssign|opOrAssign|opLogicalOr|opLogicalAnd
|opBitwiseOr|opBitwiseXor|opBitwiseAnd|opEqual
|opNotEqual|opGreaterOrEqual|opLessOrEqual|opGreater
|opLess|opLeftShift|opRightShift|opAdd|opSubtract
|opMultiply|opDivide|opRemainder|opArrowStar|opDotStar
|opSubscript|opComma) "opAssign">
```

```
<!ELEMENT ConditionalExpression ((%Expression;),(%Expression;),(%Expression;))>
<!ELEMENT CastExpression (TypeDescriptor*,(%Expression;))>
<!ATTLIST CastExpression
kind (Simple|Static|Dynamic|Reinterpret|Const) "Simple">
<!ELEMENT FunctionCastExpression (TypeDescriptor*,(%Expression;)*)>
<!ELEMENT SizeofTypeExpression (TypeDescriptor*)>
<!ELEMENT TypeidofTypeExpression (TypeDescriptor*)>
<!ELEMENT OffsetofExpression ((%Expression;))>
<!ELEMENT NewExpression (TypeDescriptor*,(%Expression;)*,DirectInitializer?
,(%Expression;)?)>
<!ATTLIST NewExpression
global (True|False) "True">
<!ELEMENT DeleteExpression ((%Expression;))>
<!ATTLIST DeleteExpression
global (True|False) "True"
vector (True|False) "True">
<!ELEMENT ThrowExpression ((%Expression;))>
<!ELEMENT MemberExpression ((%Expression;))>
<!ATTLIST MemberExpression
name CDATA #IMPLIED
kind (Dot|Arrow|Offsetof|MemberOffset|BraceListInitialized)
"Dot"
byteOffset CDATA #IMPLIED
bitOffset CDATA #IMPLIED
arrayIndexOffset CDATA #IMPLIED>
<!ELEMENT ParenthesesExpression ((%Expression;))>
<!ELEMENT BreakStatement EMPTY>
<!ATTLIST BreakStatement
line CDATA #IMPLIED
```

```
column CDATA #IMPLIED>
<!ELEMENT CaseLabelStatement EMPTY>
<!ATTLIST CaseLabelStatement
value CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT ContinueStatement EMPTY>
<!ATTLIST ContinueStatement
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT DeclarationStatement ((%Declaration;),Initializer)>
<!ATTLIST DeclarationStatement
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT DefaultStatement EMPTY>
<!ATTLIST DefaultStatement
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT DoStatement ((%Expression;),LexicalBlockStatement)>
<!ATTLIST DoStatement
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT ExpressionStatement ((%Expression;))>
<!ATTLIST ExpressionStatement
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT ForStatement ((%Statement;)*,(%Expression;)?,(%Statement;)?
,LexicalBlockStatement?)>
<!ATTLIST ForStatement
```

```
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT GotoStatement EMPTY>
<!ATTLIST GotoStatement
label CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT IfStatement ((%Statement;)?,LexicalBlockStatement?
,LexicalBlockStatement?)>
<!ATTLIST IfStatement
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT LabelStatement EMPTY>
<!ATTLIST LabelStatement
label CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT LexicalBlockStatement ((%Statement;)*,ConstructorInitializer*
,ExceptionHandler*)>
<!ATTLIST LexicalBlockStatement
tryBlock (True|False) "True"
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT ReturnStatement ((%Expression;)?)>
<!ATTLIST ReturnStatement
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT SwitchStatement ((%Statement;)?,LexicalBlockStatement?)>
<!ATTLIST SwitchStatement
```

```
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT WhileStatement ((%Statement;)?,LexicalBlockStatement?)>
<!ATTLIST WhileStatement
line CDATA #IMPLIED
column CDATA #IMPLIED>
<!ELEMENT Class ((%Declaration;)*,BaseSpecifier*,TemplateArgument*)>
<!ATTLIST Class
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public"
kind  (Class|Struct|Union) "Class"
abstract (True|False) "False"
anonymous (True|False) "False"
unnamed (True|False) "False"
TemplateName CDATA #IMPLIED>
<!ELEMENT Enumeration (TypeDescriptor*, Variable*)>
<!ATTLIST Enumeration
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public"
unnamed (True|False) "True">
<!ELEMENT Typedef (TypeDescriptor*)>
<!ATTLIST Typedef
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
```

```
accessKind (public|private|protected|unspecified) "public">
<!ELEMENT Friend ((%Declaration;))>
<!ATTLIST Friend
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public">
<!ELEMENT Namespace ((%Declaration;)*) >
<!ATTLIST Namespace
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public"
unnamed (True|False) "True"
alias (True|False) "True">
<!ELEMENT Using EMPTY>
<!ATTLIST Using
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public"
declname CDATA #IMPLIED>
<!ELEMENT UsingDirective EMPTY>
<!ATTLIST UsingDirective
name CDATA #IMPLIED
line CDATA #IMPLIED
column CDATA #IMPLIED
accessKind (public|private|protected|unspecified) "public"
declname CDATA #IMPLIED>
```

```
<!ELEMENT TypeDescriptor (FunctionParameter*,TypeDescriptor*,Exception?
,(%Expression;)?,(%Declaration;)?)>
<!ATTLIST TypeDescriptor
name CDATA #IMPLIED
arraysize CDATA #IMPLIED
kind (Pointer|Reference|Function|PointerToMember|Array
|PredefinedType|NamedType|UnnamedType) "Pointer"
constant (True|False) "False"
volatile (True|False) "False">
<!ELEMENT FunctionParameter (TypeDescriptor*,(%Expression;)?)>
<!ATTLIST FunctionParameter
identifier CDATA #IMPLIED
register (True|False) "False">
<!ELEMENT Exception (TypeDescriptor*)>
<!ELEMENT BaseSpecifier EMPTY>
<!ATTLIST BaseSpecifier
name CDATA #IMPLIED
accessKind (public|private|protected) "public"
virtual (True|False) "False">
<!ELEMENT TemplateArgument ((%Expression;)|TypeDescriptor*)>
<!ELEMENT Macro (MacroParameter*,MacroReplacement*)>
<!ATTLIST Macro
name CDATA #IMPLIED>
<!ELEMENT MacroParameter EMPTY>
<!ATTLIST MacroParameter
name CDATA #IMPLIED
kind (Terminator|Ignored|Concatenated|Expanded|Stringized) "Terminator">
<!ELEMENT MacroReplacement EMPTY>
<!ATTLIST MacroReplacement name CDATA #IMPLIED>
```

```
<!ELEMENT ConstructorInitializer ((%Expression;)*)>
<!ATTLIST ConstructorInitializer
kind (unknown|virtualBase|directNonVirtualBase|dataMember) "unknown">
<!ELEMENT ExceptionHandler (TypeDescriptor*,LexicalBlockStatement
,LexicalBlockStatement)>
<!ATTLIST ExceptionHandler
identifier CDATA #IMPLIED>
```

Appendix C

DTD for OOML representation

```
<!ELEMENT Program (Source)*>
<!ATTLIST Program
Name CDATA #REQUIRED
Language (Cpp|Java) "Java">
<!ELEMENT Source (Include*,VariableDeclaration*,Method*,Class*)>
<!ATTLIST Source
Name CDATA #REQUIRED
Type (Primary|Included) "Primary">
<!ELEMENT Include EMPTY>
<!ATTLIST Include
Name CDATA #IMPLIED
isSingle (True|False) "True">
<!ELEMENT Class (VariableDeclaration*,Method*)>
<!ATTLIST Class
Identifier CDATA #REQUIRED
isAbstract (True|False) "False">
<!ELEMENT Method (Type?, Parameter*, Body?)>
```

```
<!ATTLIST Method
Identifier CDATA #REQUIRED
Access (Public|Private|Protected|Default) "Default">
<!ELEMENT Body (ConditionalStatement|Loop|VariableDeclaration|VariableUse
|MethodCall)*>
<!ELEMENT ConditionalStatement (ConditionalStatement|Loop|VariableDeclaration
|VariableUse|MethodCall)*>
<!ELEMENT Loop (ConditionalStatement|Loop|VariableDeclaration
|VariableUse|MethodCall)*>
<!ATTLIST Loop
Type (for|do|while) "for">
<!ELEMENT VariableDeclaration (Type)>
<!ATTLIST VariableDeclaration
Identifier CDATA #REQUIRED
isInitializer (True|False) "False">
<!ELEMENT VariableUse EMPTY>
<!ATTLIST VariableUse
Identifier CDATA #REQUIRED
Type (read|write) "read">
<!ELEMENT MethodCall (Argument*)>
<!ATTLIST MethodCall
Identifier CDATA #REQUIRED>
<!ELEMENT Type EMPTY>
<!ATTLIST Type
Name CDATA #REQUIRED
isPrimitive (True|False) "False"
isArray (True|False) "False"
isPointer (True|False) "False">
<!ELEMENT Parameter (Type?)>
```

```
<!ELEMENT Argument (Type?)>
```

Appendix D

CppML example

C++ source code (putint.cpp)

```
#include "putint.h"
#include <stdio.h>

int global_var;
int putint(int param) {
   int local_var;
   local_var=global_var*param;
   printf("The number is:%d\n",local_var);
}
```

C++ source code (putint.h)

```
#ifndef _PUTINT_H_
#define _PUTINT_H_
 extern int putint(int param);
 extern int global_var;
#endif
```

CppML representation

```
<Program name="..\putint\putint.icc">
 <IncludedSource name="putint.h">
  <Function name="putint" line="5" column="18" accessKind="public"
   kind="ExternFileScope" virtual="False" purevirtual="False" inline="False"
   explicit="False">
    <TypeDescriptor name="int (int)" kind="Function" constant="False"
    volatile="False">
     <FunctionParameter identifier="param" register="False">
      <TypeDescriptor name="int" kind="PredefinedType" constant="False"
      volatile="False"></TypeDescriptor>
     </FunctionParameter>
     <TypeDescriptor name="int" kind="PredefinedType" constant="False"
     volatile="False"></TypeDescriptor>
    </TypeDescriptor>
  </Function>
  <Variable name="global_var" line="4" column="18" accessKind="public"
  kind="ExternalFileScope" mutable="False">
   <TypeDescriptor name="int" kind="PredefinedType" constant="False"
   volatile="False"></TypeDescriptor>
  </Variable>
 </IncludedSource>
 <PrimarySource name="putint.cpp">
  <Include name="putint.h" system="False"></Include>
  <Include name="stdio.h" system="True"></Include>
  <Variable name="global_var" line="4" column="11" accessKind="public"
  kind="ExternalFileScope" mutable="False">
   <TypeDescriptor name="int" kind="PredefinedType" constant="False"
   volatile="False"></TypeDescriptor>
   <Initializer>
```

```
  <ImplicitInitializer></ImplicitInitializer>
 </Initializer>
</Variable>
<Function name="putint" line="6" column="11" accessKind="public"
kind="ExternFileScope" virtual="False" purevirtual="False" inline="False"
explicit="False">
 <TypeDescriptor name="int (int)" kind="Function" constant="False"
 volatile="False">
  <FunctionParameter identifier="param" register="False">
   <TypeDescriptor name="int" kind="PredefinedType" constant="False"
   volatile="False"></TypeDescriptor>
  </FunctionParameter>
  <TypeDescriptor name="int" kind="PredefinedType" constant="False"
  volatile="False"></TypeDescriptor>
 </TypeDescriptor>
 <FunctionBody>
  <LexicalBlockStatement tryBlock="False" line="6" column="29">
   <DeclarationStatement line="7" column="11">
    <Variable name="local_var" line="7" column="15" accessKind="public"
    kind="AutoLocal" mutable="False">
     <TypeDescriptor name="int" kind="PredefinedType" constant="False"
     volatile="False"></TypeDescriptor>
    </Variable>
    <Initializer>
     <ImplicitInitializer></ImplicitInitializer>
    </Initializer>
   </DeclarationStatement>
   <ExpressionStatement line="8" column="11">
    <BinaryExpression>
```

```
       <NameExpression name="local_var"></NameExpression>
       <BinaryExpression kind="opMultiply">
        <NameExpression name="global_var"></NameExpression>
        <NameExpression name="param"></NameExpression>
       </BinaryExpression>
      </BinaryExpression>
     </ExpressionStatement>
     <ExpressionStatement line="10" column="11">
      <FunctionCallExpression>
       <NameExpression name="printf"></NameExpression>
       <LiteralExpression value="&quot;The number is:%d\n&quot;">
        <TypeDescriptor name="const char [18]" arraysize="18" kind="Array"
        constant="False" volatile="False"></TypeDescriptor>
        <TypeDescriptor name="const char" kind="PredefinedType"
         constant="True" volatile="False"></TypeDescriptor>
       </LiteralExpression>
       <NameExpression name="local_var"></NameExpression>
      </FunctionCallExpression>
     </ExpressionStatement>
    </LexicalBlockStatement>
   </FunctionBody>
  </Function>
 </PrimarySource>
</Program>
```