Source code modularization using lattice of concept slices

Conference Paper *in* Proceedings of the Euromicro Conference on Software Maintenance and Reengineering, CSMR · April 2004

DOI: 10.1109/CSMR.2004.1281420 · Source: IEEE Xplore

2 authors, including:

Kostas Kontogiannis
National Technical University of Athens
185 PUBLICATIONS **3,519** CITATIONS

SEE PROFILE

Source Code Modularization using Lattice of Concept Slices

by

Ahmed Raihan Al-Ekram

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of

Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2004

Author's Declaration

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.

Signature

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.

Signature

Borrower's Page

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

Acknowledgements

Acknowledgements.

Related Publication

Raihan Al-Ekram and Kostas Kontogiannis, “Source Code Modularization using Lattice of Concept Slices”, 8th European Conference on Software Maintenance and Re-engineering, Tampere, Finland, March 24-26, 2004.

Abstract

Most legacy systems have been altered due to prolonged maintenance to the point that they deviate significantly from their original and intended design and consequently, they lack modularity. Static source code analysis techniques like concept assignment, formal concept analysis and program slicing, have been successfully used by researchers for program understanding and for restoring system design properties. In our approach we combine these three techniques, aiming to gain on their individual strengths and overcoming their weaknesses.

This thesis presents a program representation formalism that we call the Lattice of Concept Slices and a program modularization technique that aims to separate statements in a code fragment according to the concept they implement or they may belong to. The lattice shows the relationship between the statements of a program and the domain concepts that might be implemented by the statements. Using the lattice as a primary data structure we present two algorithms for decomposing the program into appropriate modules. The goal is to achieve a modularization such that the modules are self-contained, side effect free and the code duplication among nodes is minimal.

We also present a language neutral program representation and analysis framework and a set of tools in order to perform the source code modularization and in general to facilitate language independent generic program analysis. XML sublanguages are defined to represent source code at different levels of abstractions. Generic source code analysis tools like concept identifier, program slicer, formal concept analyzer etc. are built on top of the framework to perform the source code modularization.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Description

Most legacy software systems have been altered due to prolonged maintenance activities so that they deviate from their original design and consequently they lack among other qualities modularity. Such systems may have been degenerated to a point that they consist of monolithic low cohesive subroutines each of which implementing numerous distinct domain concepts inside it. A domain concept is an idea or a task in the problem domain that is being implemented in the program, e.g. calculate interest, book a ticket etc. This makes these systems difficult to understand and maintenance tasks hard to perform. Automatic or semi-automatic decomposition of such systems into a more modular structure with each module preferably implementing a single domain concept facilitates better understanding and maintenance of the application. The result of the decomposition can also be used for program parallelization, object-oriented migration or other re-engineering activities.

Static source code analysis techniques like concept assignment, formal concept analysis, and program slicing, have long been used by researchers for program modularization. Concept assignment aims to associate specific meaning to specific parts

of a program. This technique can be used to extract program fragments that may be associated with a particular domain concept in a program. But the problem of concept assignment is that the code fragment that has been identified as a domain concept is not self-contained and not executable independently as a separate module. Program slicing is another useful technique that extracts an executable subset of the original program that preserves the behavior of the program with respect to a variable at a program point. The problem of slicing is that the decomposition is done based on very fine-grained program variables instead of domain concepts. Moreover different decompositions overlap with each other and may have a significant amount of duplicated code among the extracted modules. Even though each of the decompositions is executable by itself, the code duplication may cause side effects when implemented as separate modules. Formal concept analysis is used from a different perspective in modularization. Instead of decomposition, it has been used to identify groupings of subroutines and global data structures into modules. As a result it is not directly applicable in our context of modularization.

1.2 Objectives and Goals

The objectives of this thesis are twofold

- First we aim to develop a program representation framework and a toolset in order to perform the source code modularization and in general to allow language independent generic program analysis. Program representation is an important issue for program analysis and software re-engineering tasks. Parse trees and abstract syntax trees are source code representations at the finest level of granularity. Flow graphs and dependency graphs are the next level of abstractions of a program based on the control and the data flow dependencies among the

program components. Call graphs are abstractions of inter-procedural dependencies of the program. Development of a framework for program representations based on language domain models can make the representations programming language neutral over a number of languages of similar domain models, e.g. the group of procedural languages or the group of object-oriented languages. In order to enable the exchange of representations among different tools they should be portable, extensible and open.

- Second we aim to design a technique for automatic or semi-automatic decomposition of monolithic programs into modular structure. The goal is to achieve a modularization such that each module implements preferably a single domain concept, each module is self-contained, there is minimal duplication in code and there are limited side effects among the modules. The concept assignment technique can successfully identify code fragments associated with the domain concepts. Combining program slicing with concept assignment the code fragments can be made self contained and executable. Finally computing a concept lattice of the domain concepts and program statements in the code fragments will give the dependencies among the statements and domain concepts and as well as dependencies among the domain concepts. The lattice can help in identifying side effects and code duplications in the program fragments and thus help identifying modules that conform to the modularization goal.

1.3 Thesis Contributions

In order to facilitate the program analysis we have developed a language neutral program representation framework that represents program facts at different levels of abstraction in different XML-sublanguages. The XML-sublanguages are representation specific and not

programming language specific. This makes the framework programming language neutral, portable, extensible and suitable for the development of generic program analysis tools. The contributions can be summarized as:

- Development of a framework for source code neutral XML-based program representations. The framework consists of a multi-layered program representation formalisms and representation transformer tools working in a pipe and filter architectural style. At the lowest level of abstraction is the source code layer, the primary source for all other representations. The second level is the syntax tree layer; XML-sublanguages are defined for each of the programming languages to be analyzed. The third level is the language neutral representation layer; XML-sublanguages are defined for program facts, flow graphs, dependence graphs, call graphs and other representation formalisms that are not language dependent and can be extracted from the syntax tree. In between each pair of representation layers are the transformer tools that generates the representations at the higher level of abstraction from the level below it.
- Development of generic program analysis toolset on top of the representation framework. The toolset consists of a concept identifier, a program slicer and a concept analyzer.

To achieve our modularization goal we introduce a new program representation formalism that we call the Lattice of Concept Slices and propose modularization algorithms based on it. This approach combines concept assignment, slicing, and concept analysis aiming to capitalize on the individual strengths of these three analysis techniques while overcoming their shortcomings. The contribution can be summarized as

- Introduction of a new program representation formalism that we call the Lattice of Concept Slices. Building the lattice is a three stage process. First concept assignment is performed to identify the domain concepts implemented in the program and identify the code fragments associated with each of the domain concepts. Next slicing is performed based on the domain concepts to compute the Concept Slices from the code fragments to make them executable. And finally a formal concept analysis is done between the domain concepts and the program statements in the code fragments. The resulting lattice is the Lattice of Concept Slices. The lattice shows the contribution of the program statements towards the computation of the domain concepts implemented in the program.
- Development of program modularization algorithms based on the Lattice of Concept Slices. The lattice also depicts the side effects among the code fragments associated with the domain concepts; based on the side effects we propose a lattice clustering algorithm to group statements together in order to form a non-interfering set of modules. The lattice also demonstrates the amount of code duplication among the modules; we propose a lattice a restructuring algorithm to rearrange the statements in the lattice nodes so that it corresponds to a module sub-module hierarchy with minimal code duplication.

1.4 Thesis Organization

The remaining chapters of the thesis are organized as follows. Chapter 2 provides a brief overview of the related work in the area of program, representation, external representation formats, program comprehension and reverse engineering frameworks. Chapter 3 describes the program representation framework for language neutral analysis. Chapter 4 introduces the Lattice of Concept Slices representation formalism and presents

the algorithms for program modularization based on the lattice. Chapter 5 presents a prototype implementation of the framework and toolset. Finally Chapter 6 concludes the thesis with directions for future work.

Chapter 2

Related Work

In this chapter we present related work in the area of program representation, program comprehension and reverse engineering frameworks. We examine different source code representation formalisms and some higher-level abstractions of source code that focus on different aspects of a program. We examine some external formats for storing and exchanging these program representations. We then discuss some analysis techniques used for program comprehension and understanding. Finally we investigate some reverse engineering and re-engineering tools.

2.1 Program Representation Formalisms

While the source code is the primary source for performing any kind analysis on a program, for understanding and analyzing it at different levels of abstraction more structured and abstract source code and other intermediate representation formalisms are needed. These representations enable algorithmic analysis and manipulation of the programs at different levels of granularity. Following are some program representation formalisms widely used for program understanding and analysis.

2.1.1 Source Code Representations

Parse Tree and Abstract Syntax Tree [1] are source code representation techniques at the finest level of granularity. These data structures are the primary sources for constructing intermediate representations and performing analysis.

A **Parse Tree** is a hierarchical graphical representation of the derivations of the source code from its grammar. The interior modes of the tree represent the non-terminals and the leaves terminal symbols of the grammar. The source sentence can be reconstructed from a parse tree by reading the leaves from left to right. An example of a source sentence and its corresponding Parse Tree is shown in Figure 2.1

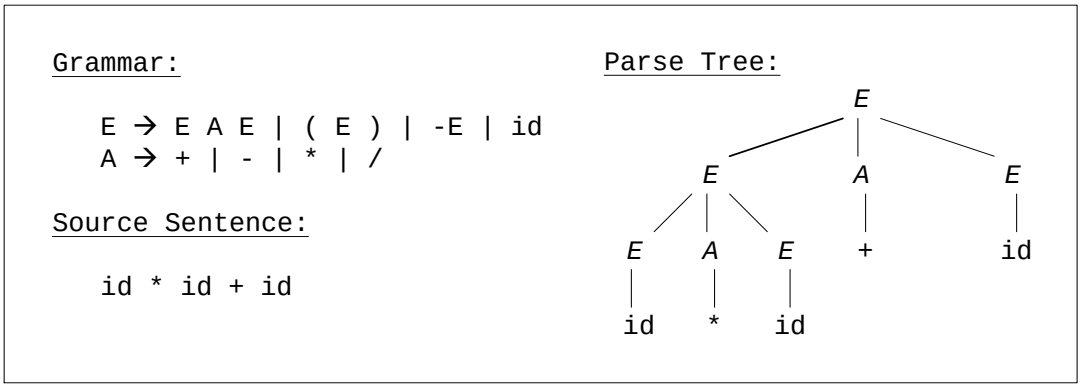

Figure 2.1: Example of Parse Tree

An **Abstract Syntax Tree (AST)** is a more economical representation of the source code abstracting out the redundant information from the parse tree. It represents the syntactic information contained in the source code without the grammar productions. The leaves of the tree represent operands and the interior nodes represent operators. Depth-first inorder traversal of the tree nodes reconstructs the source sentence. AST is used by compilers to analyze and transform source codes entities. Figure 2.2 shows the AST equivalent to the parse tree of Figure 2.1

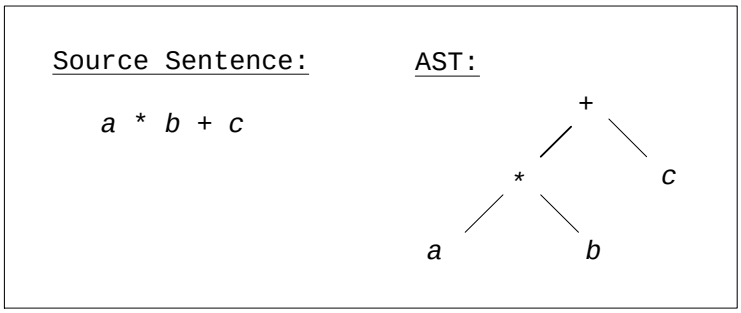

Figure 2.2: Example of Abstract Syntax Tree

Abstract Semantic **Graph (ASG)** [11] is similar to AST except that the nodes and arcs of the graph are typed and they have additional attributes associated with them. These attributes are used to embed semantic information e.g. scope of identifiers, type of variables etc. on top of the syntax.

2.1.2 Flow and Dependency Graphs

While parse tree and abstract syntax tree contain low level source code level details, they cannot be used directly for sophisticated and higher level analysis. Intermediate level representations are needed for analyzing flow of control and data in the program. These representations are programming language independent, making it possible to develop generic analysis algorithms.

A **Control Flow Graph (CFG)** [3] provides a normalized view of all possible flow of execution paths of a program. A CFG is a rooted directed graph where the nodes represent basic blocks and arcs represent possible immediate transfer of control from one basic block to another. A basic block is a sequence of consecutive instructions that are executed from start to finish without the possibility of branching except at the end. The CFG also includes a special node corresponding to the exit of the procedure. CFG is extensively used for code optimization and testing. It is also used to perform data flow

analysis, e.g. reaching definitions or use-def chaining, in the program [4] [5]. Figure 2.3 shows a function written in C and its corresponding CFG.

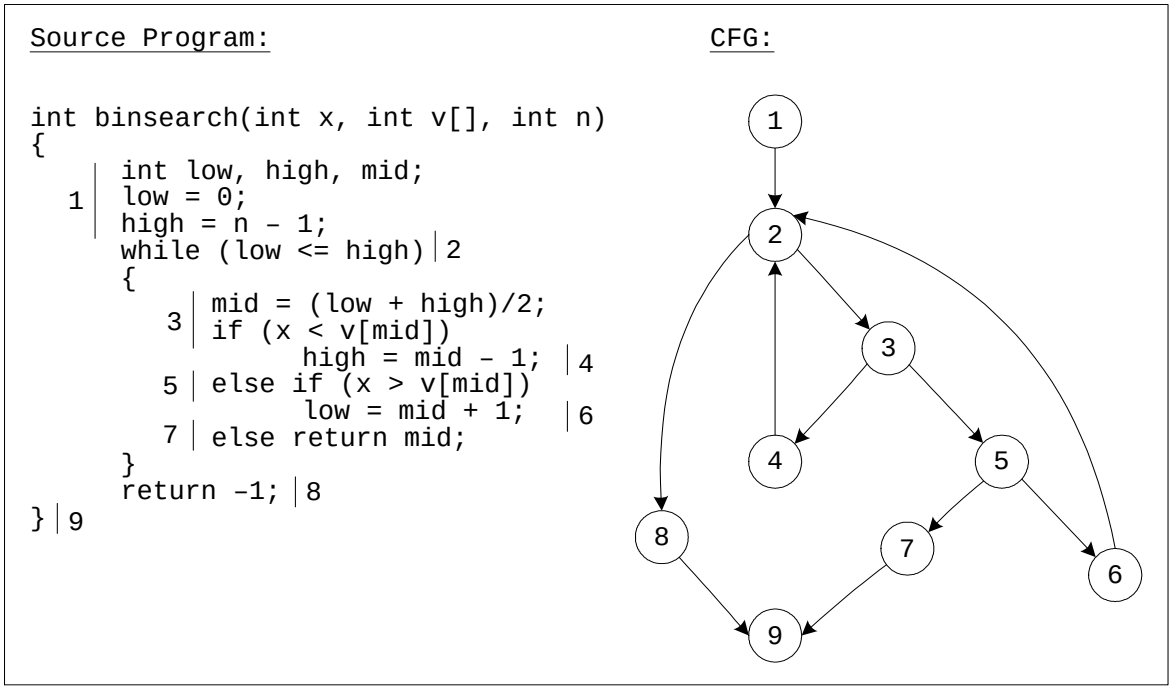

Figure 2.3: Example of Control Flow Graph

A Program **Dependence Graph (PDG)** [6] [7] is a combined explicit representation of both control and data dependences in a program. A PDG consists of a control dependence sub-graph and data dependence sub-graph. The control dependence sub-graph is similar to CFG but it does not include the fixed sequencing of operations of CFG, just the control flow relationships. The data dependence sub-graph represents possible flow of data values from one statement to another. The PDG is a directed graph whose nodes are connected by several kinds of arcs. The nodes represent statements and predicate expressions and the arcs represent control and data dependence. There is a special entry node, some initial definition of variable nodes and some final use of variable nodes. The control dependence arcs must start from the entry node or from the predicate nodes and are labeled either True or the truth-value of the predicate. The data dependence arcs

indicate possible flow of data values between nodes where the source node defines a variable and the destination node may use that particular data value. These arcs are labeled by the variable name and there can me more than one data dependence arc between any two nodes. PDG is mostly use for code optimization, e.g. parallelism detection, loop fusion, clone detection etc. It is also used for performing slicing for maintenance and re-engineering purpose. Figure 2.3 shows a function written in C and its corresponding PDG.

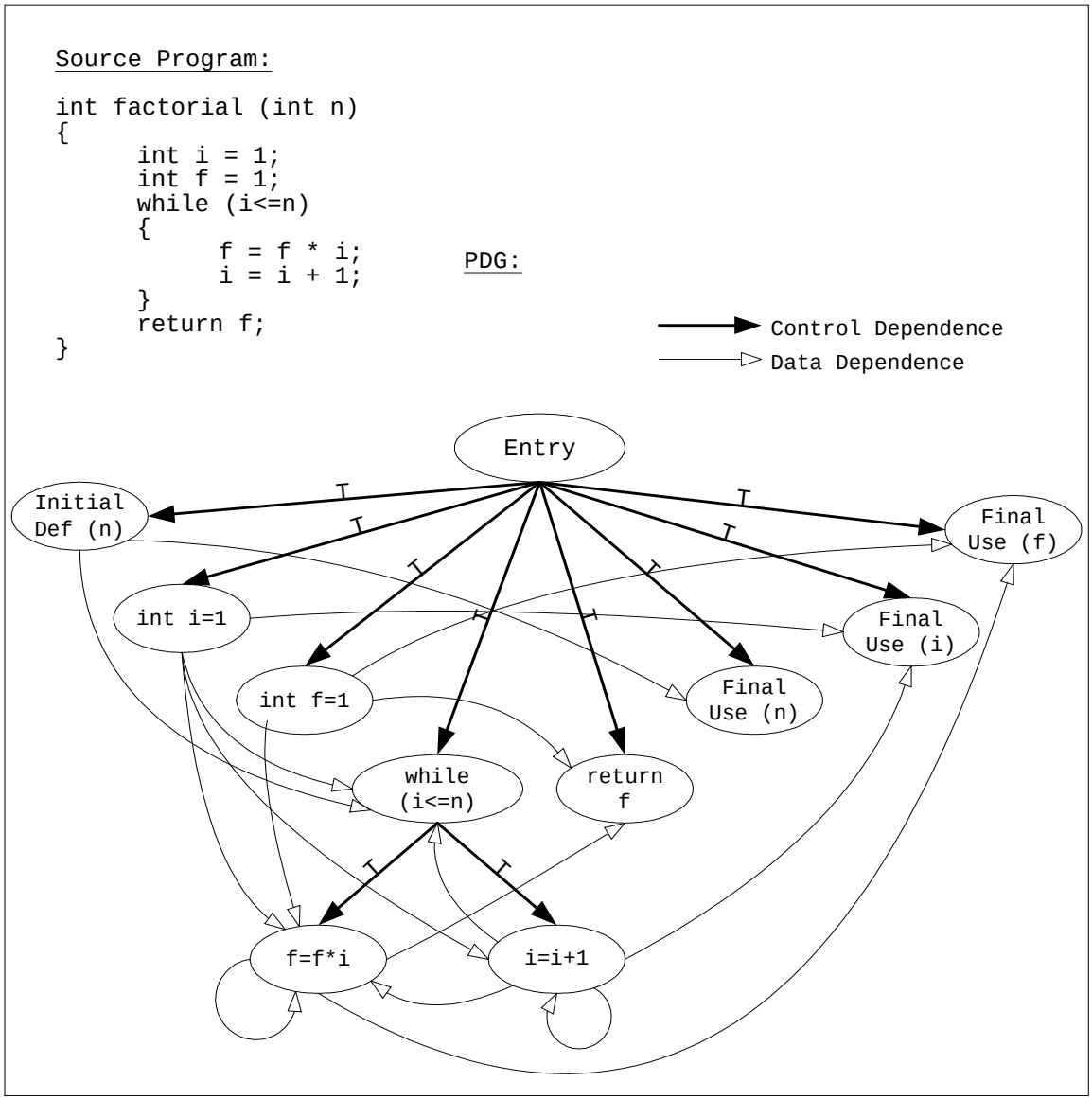

Figure 2.4: Example of Program Dependence Graph

The **System Dependence Graph (SDG)** [8] is an extension to PDG for programs with multiple procedures. The SDG is constructed by connecting the individual PDG of each procedure with some additional arc types. These arcs correspond to procedure calls, parameters passed and return values.

2.1.3 Call Graphs

Understanding the flow of control within a single subprogram is not sufficient for optimization or analysis of the complete system, which is comprised of many procedures and files. **Call Graphs** [9] [10] are program abstractions used in traditional inter-procedural analysis. It's a graphical representation of the caller or callee relationships among the procedures of a program. Call graphs can be extended by including attributes, e.g. line number or file name with each call and also by adding new entities, e.g. abstract data types and their usage relationships in addition to the procedure calls. From these attributes higher level call graphs can be constructed to show relationships among files, modules or architectural entities instead of procedures. Other than inter-procedural data flow analysis for optimization, call graphs are used for design recovery, architecture extraction or other reverse engineering analysis. Figure 2.5 shows a program writeen in C and the corresponding call graph for it.

Program Summary Graphs (PSG) are modification of Call Graphs to take into account the data flow and control flow in individual procedures. Information about actual reference parameters and global variables are contained at call sites and formal reference parameters and global variables at procedure entry and exit points. **Inter-procedural Flow Graphs (IFG)** are PSGs with additional information at each node for the locations of definitions and uses of reference parameters and global variables that can be reached across procedure boundaries.

Source Program:

```
main ()
{
      int x, y, p, q;
      scanf("%d %d", &x, &y);
      p = add(x, y);
      q = inc(p);
      printf("%d %d", p, q);
}

int add (int a, int b)
{
      return (a+b);
}

int inc (int a)
{
      return add(a, 1);
}
```

Call Graph:

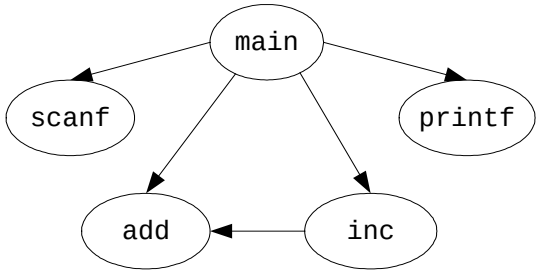

Figure 2.5: Example of Call Graph

2.2 External Representations

External representation formats are required for storing facts about a program for later use or for communicating the facts between different parts of a tool or exporting to a different tool. Following are some examples of external representaion formats.

2.2.1 ATerms and AsFix

The ATerms [12] or Annotated terms format is used in the ASF+SDF Meta-Environment [14] to exchange structural informations e.g. programs, specifications, parse trees, abstract syntax trees etc. among its components. ATerms are expressed in Syntax Definition Formalism (SDF) [13] which is comparable to BNF but more flexible since it can express both lexical and syntactic informations. Figure 2.6 shows some example ATerms given in [12].

```
constant    abc
numeral     123
literal     "abc", "123"
list        [], [1, "abc", 3], [1, 2, [3,2], 1]
functions   f("a"), g(1, []), h("1", f("2"), ["a", "b"])
annotation  f("a"){g(2, ["a", "b"])}, "1"{[1, 2, 3], "abc"}
```

Figure 2.6: ATerms example

```
Function Symbols:

  prod(L)     the production rule L
  appl(T1,T2) the application of rule T1 to the args T2
  l(L)        literal L
  w(L)        whitespace L


Syntax Rule for Boolean Expression:

  sort Bool
  context-free syntax
    true -> Bool
    false -> Bool
    Bool and Bool -> Bool

Source Sentence:

  true

AsFix Parse Tree:

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

Source Sentence:

  true and true

AsFix Parse Tree:

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

Figure 2.7: Example of AsFix Parse Tree

The **AsFix** [14] is an instance of ATerms used to represent parse trees. All the key words, whitespace and comments are preserved in the tree. The representations are self

descriptive and each application of a syntax rule contains the rule. As given in the example in [14], Figure 2.7 gives an example of parse tree in AsFix notation.

2.2.2 JavaML, CppML and OOML

```
Java Program:

public class Person
{
      private String name;
      public String getName()
      {
            return name;
      }
}

JavaML Representation:

<ClassDeclaration isPublic="True" Identifier="Person">

  <FieldDeclaration isPrivate="True">
    <Type Identifier="String"/>
    <VariableDeclarator Identifier="name"/>
  </FieldDeclaration>

  <MethodDeclaration isPublic="True" Identifier="getName">
    <ResultType>
      <Type Identifier="String"/>
    </ResultType>
    <Block>
      <ReturnStatement>
        <Expression>
          <PrimaryExpression>
            <Name Identifier="name"/>
          </PrimaryExpression>
        </Expression>
      </ReturnStatement>
    </Block>
  </MethodDeclaration>

 </ClassDeclaration>
```

Figure 2.8: Example of JavaML

JavaML and **CppML** [15] are XML based source code representation techniques for Java and C++ programs respectively. **OOML** [15] is an aggregate and more generic representation for all object oriented laguages. In this scheme the parse tree of the source program is represented as an XML DOM tree . A DTD is defined for each of the markup

languages to describe the language grammar. A well-defined API is available to make queries and extract facts from the DOM tree. Figure 2.8 shows a smple Java source program and it corresponding JavaML representation.

2.2.3 Rigi Standard Format

Rigi Standrad Format (RSF) [16] is a file format used by the tool Rigi to store facts about a program. It's a simple and intuitive format to make it easy to read and parse. An RSF file consists of a sequence of triples, one triple in each line. The format for a triple is three optionally quoted strings of the form: verb subject object

There can be a domain model file to specify the valid verbs for the RSF files. RSF can be used to represent a graph – the triplets can represent arcs between any two nodes, can bind values to attributes of nodes and arcs or can define the types of nodes and arcs. Figure 2.9 shows and example program written in C and some facts about the program in RSF.

```
Source Program: (main.c)          RSF File:

main ()                           type    main    function
{                                 type    inc     function
      int x, y;
      x = 5;                      call    main    inc
      y = inc(x);                 call    main    printf
      printf("%d", y);
}                                 file    main    "main.c"
                                  file    add     "main.c"
int inc (int a)
{                                 lineno  main    1
      return (a+1);               lineno  inc     9
}
```

Figure 2.9: Example of Rigi Standard Format

2.2.4 Tupple Attribute Language

The **Tupple Attribute (TA)** [17] language is used to represent graphs. Since the artifacts of a program are represented by different types of graphs, TA can be used store and exchange facts extracted from programs. TA also being a drawing laguage, it also focuses on the visual aspects of the graph. The TA language consists of two different sublanguages – the Tupple sublanguage to describe the nodes and arcs of the graph and the Attribute sublanguage to describe the attributes associated with the nodes and arcs. Each of the sublanguages have two levels – one to describe the facts, the other to describe the scheme for the facts. TA also supports inheritence among the entities of the language. Figure 2.10 shows the different parts of the TA language and Figure 2.11 shows an exmaple TA program from [18]

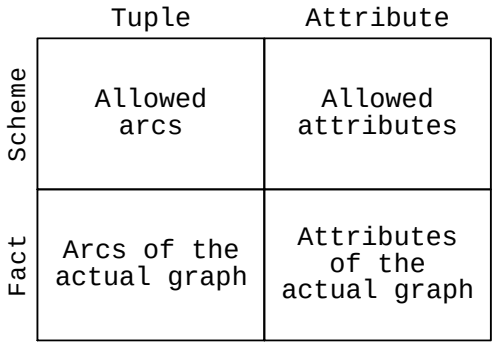

Figure 2.10: Components of TA language

```
SCHEME TUPLE :

Call Proc Proc // Procs call Procs
Ref  Proc Var  // Procs reference Vars

$INHERIT Proc Box // A Proc is a Box
$INHERIT Var  Box // A Var is a Box

SCHEME ATTRIBUTE :

Box { x y width = 70 height = 30 color }

Proc { color = ( 1.0 0.0 0.0 ) } // color red, using RGB encoding
Var  { color = ( 0.0 0.0 1.0 ) } // color blue

(Call) { color = ( 1.0 0.0 0.0 ) } // color red
(Ref)  { color = ( 0.3 1.0 0.3 ) } // color bright red

FACT TUPLE :

$INSTANCE P Proc // P is an instance of a Proc
$INSTANCE Q Proc // Q is an instance of a Proc
$INSTANCE V Var  // V is an instance of a Var

Call P Q // P Calls Q
Ref  Q V // Q References V

FACT ATTRIBUTE :

P { x = 50 y = 100 }
Q { x = 150 y = 100 }
V { x = 250 y = 100 }
```

Figure 2.11: Example of TA Program

2.2.5 Graph Exchange Language

Graph eXchange Language (GXL) [19] is an XML based language for describing graphs. It evolved from unification of graph description languages like GRAph eXchange format (GraX), Tgraphs, Tuple Attribute language (TA) and PROGRES. The conceptual data model of GXL is a typed, attrubuted and directed graph. Like TA, GXL describes both the instance data and the scheme of the data. Figure 2.12 shows a simple graph and its corrsponding GXL representation from [20].

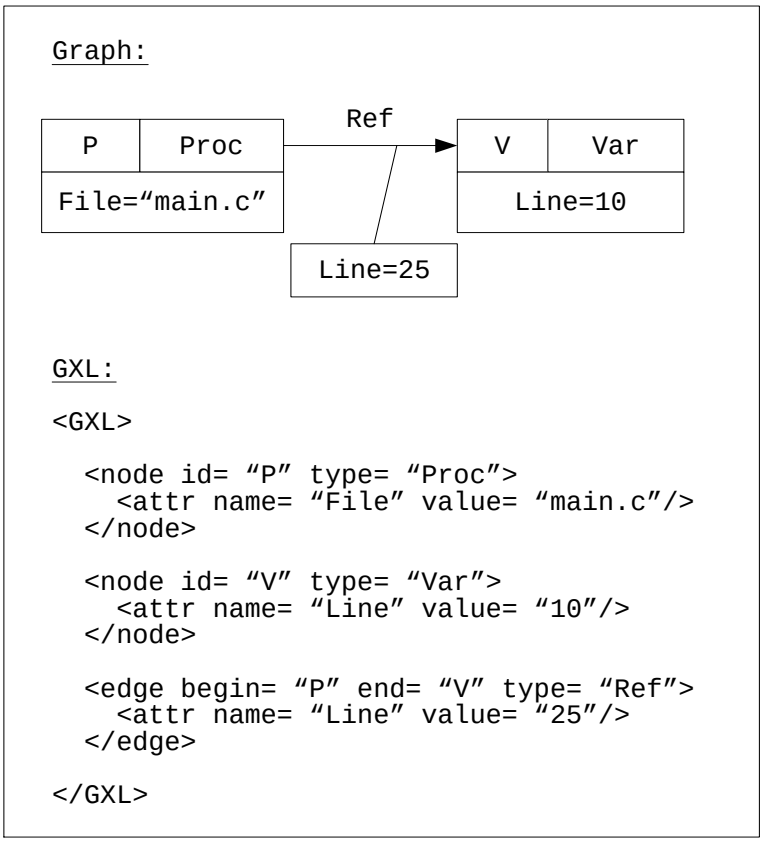

Figure 2.12: Example of GXL

2.3 Source Code Analysis and Program Comprehension

Source code analysis tecniques are tradiationally used for testing and optimization of programs. In software engineering these techniques are also used for tasks like program comprehension for design recovery, reverse engineering and re-engineering.

2.3.1 Data Flow Analysis

Data flow analysis techniques [2], extensively used by optimizing compliers, aims to analyze the behavior of programs in terms of their data flow properties. Data flow analysis are peformed by setting and solving systems of data flow equations at each basic block of

the CFG of the program. For the basic block S the set of equations is of the form given in Figure 2.13.

```
Out(S)        = Gen(S) ∪ ( In(S) – Kill(S) )

Gen(S)        Set of definitions in S that reaches end of S.
Kill(S)       Set of definitions that reaches beginning of S
              but are redefined in S.
In(S)         Information flowing in the block.
Out(S)        Information exiting the block
```

Figure 2.13: Data Flow Equations

The most common way to solve Data Flow Equations is via iteration. After building the control flow graph, *Gen* and *Kill* sets are computed compositionally from the statements for each block. The *In* and *Out* sets for each basic block is computed iteratively until their values converge; *Out* sets are computed in terms of *In* sets in forward iteration and *In* sets are computerd in terms of *Out* sets in backward iteration. Some typical data flow analysis techniques are presented below.

Reaching Definitions: Given a program point P, finding all the definitions of variables prior to P that reached the point P without being modified. In order to find the reaching definitions the initial *In* set for all the basic blocks are set to null set. In subsuquesnt iterations, for a basic the block B the *In* and *Out* sets are computed by the following equations until the *Out* set converges. The *Out* set gives the reaching definitions of the block

$$In(B) = \cup_P Out(P), \text{ P is a predecessor of B}$$
$$Out(B) = Gen(B) \cup (In(B) - Kill(B))$$

Live Variable Analysis: Given a variable x and program point P, finding out whether the the value of x at P could be used along some path in the flow graph starting at P. In

order to find live variables the initial *In* set for all the basic blocks are set to null set. In subsuquesnt iterations, for a basic the block B the *Out* and *In* sets are computed by the following equations until the *In* set converges. The *Out* set gives the live variables of the block

```
In(B)   = Use(B) ∪ ( Out(B) – Def(B) )
out(B)  = ∪s In(S), S is a successor of B
Def(B)  Set of variables assigned values in B prior to any use
Use(B)  Set of variables used in B prior to any definition
```

Available Expressions: An expression is available at program point P if every path from the start point to P evaluates the expression and after the last evaluation prior to P there is no subsequent definition to the variables of the expression.

Def-Use Chains: It is the list of all possible uses that can be reached by a given definition of a variable. **Use-Def Chains** is the list of all definitions that can reach a given use of a variable. The DU chaining problem can be formulated as data flow equation as follows

```
In(B)       = UpExpUse(B) ∪ ( Out(B) – NewDef(B) )
out(B)      = ∪s In(S), S is a successor of B
NewDef(B)   Set of pairs (S,x) such that S is a statement that
            uses x, S is not in B, and B has a definition of x
UpExpUse(B) Set of pairs (S,x) such that S is a statement in B
            which uses x, no prior definition of x occurs in B
```

2.3.2 Slicing

Slicing as originally described in [21] is an abstraction of a program based on a particular behavior. A slice is defined to be an executable subset of the original program that preserves the original behavior of the program with respect to a slicing criteria <*P*, *V*>, which is a given variable *V* at a given program point *P*. The slice will consist of all the

statements of the program that may affect the value of *V* at point *P*. The original slicing algorithm was based on statement deletion using data flow analysis. A more widely used PDG based slicing algorithm is defined in [22] and [8]. [23] and [24] describes different slicing techniques and their applications.

Slicing a program consisting of a single monolithic procedure is called **Intra-procedural Slicing**, whereas slicing a program consisting of multiple procedures is called **Inter-procedural Slicing** and is based on the SDG of the program.

The usual slicing is done by backward traversal of the program starting from the critera and hence the slice obtained is referred to a **Backward Slice**. On the other hand a **Forward Slice** with respect to a criteria <*P, V*> will consist of all the statements of the program that might be affted by the value of *V* at *P*.

A **Decomposition Slice** with respect to a variable *V*, independent of a program point, is given by the union of all the slices with respect to the variable *V* and all possible program points *P*.

Slicing a program based on only the information statically available, e.g CFG PDG, is known as **Static Slicing**. In contrast **Dynamic Slicing** the dependences of only a specific program execution based on a given set of input is considered. So the critera for a dynamic slice also includes the inputs of the program.

Figure 2.14 shows the computed PDG and the program resulting from backward static slicing the PDG of the program given in Fugure 2.4 based on the critera of final use of the variable i.

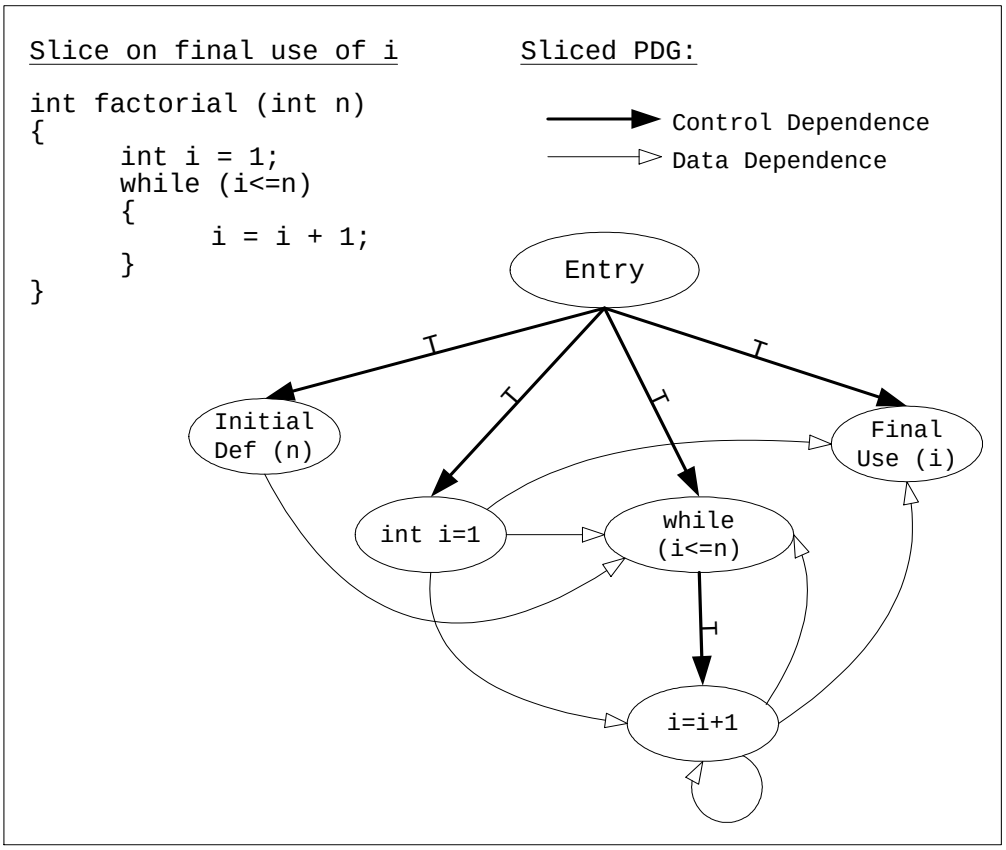

Figure 2.14: A Slice of the Program in Figure 2.4 on the Final Use of Variable i

The slicing algorithm starts working from the final use of i node of Figure 2.4 and then traces back to all nodes that has either a data flow arc or a control flow arc entering the current node. All the visited nodes are marked. The tracing back continues for all the marked nodes until there are no more nodes to mark. All the unmarked nodes and acrs entering or leaving the nodes are deleted from the PDG. The program corresponding to the resulting PDG is the computed slice.

Application of slicing includes program understanding to facilitate debugging and reverse engineering, program maintenance, program parallelization, integration of program variants and extracting reusable components.

2.3.3 Concept Assignment

The Concept Assignment [25] problem is the identification of human oriented domain concepts and assigning them to implementation oriented source code within a program. The simplest approach taken to automatically identify concepts in a program is parsing the source code for specific signatures or patterns. Having domain concepts assigned to a program facilitates the understanding of it for future maintenance tasks.

There are two major issues related to the concept assignment problem – segmenting the source code for locating extents of concepts in the program and binding appropriate concepts to these locations. A three-stage hypothesis-based concept assignment (HB-CA) approach is described in [26]. The process uses a knowledge base that contains a list of domain concepts implemented in the program and their indicators. The indicators can be identifiers, keywords, comments, regular expression etc.

In the hypothesis generation stage the source code is taken as input and scanned through to generate hypotheses of concepts and based on the knowledge base. The hypotheses are then sorted by the indicator position in the source code. In segmentation stage the sorted hypotheses are analyzed to group them into segments using an unsupervised competitive learning neural network. The output of the stage is a collection of segments each containing a number of hypotheses. In the concept binding stage the segments hypotheses are analyzed to identify the most evident concept. The segments are then labeled with their corresponding concepts.

Figure 2.15 shows part a COBOL program for mortgage payment calcualtion with two program segments corresponding to two concepts identified in it. The light shaded part is *Calculate Interest* concept and the dark shaded one is *Write Record* concept.

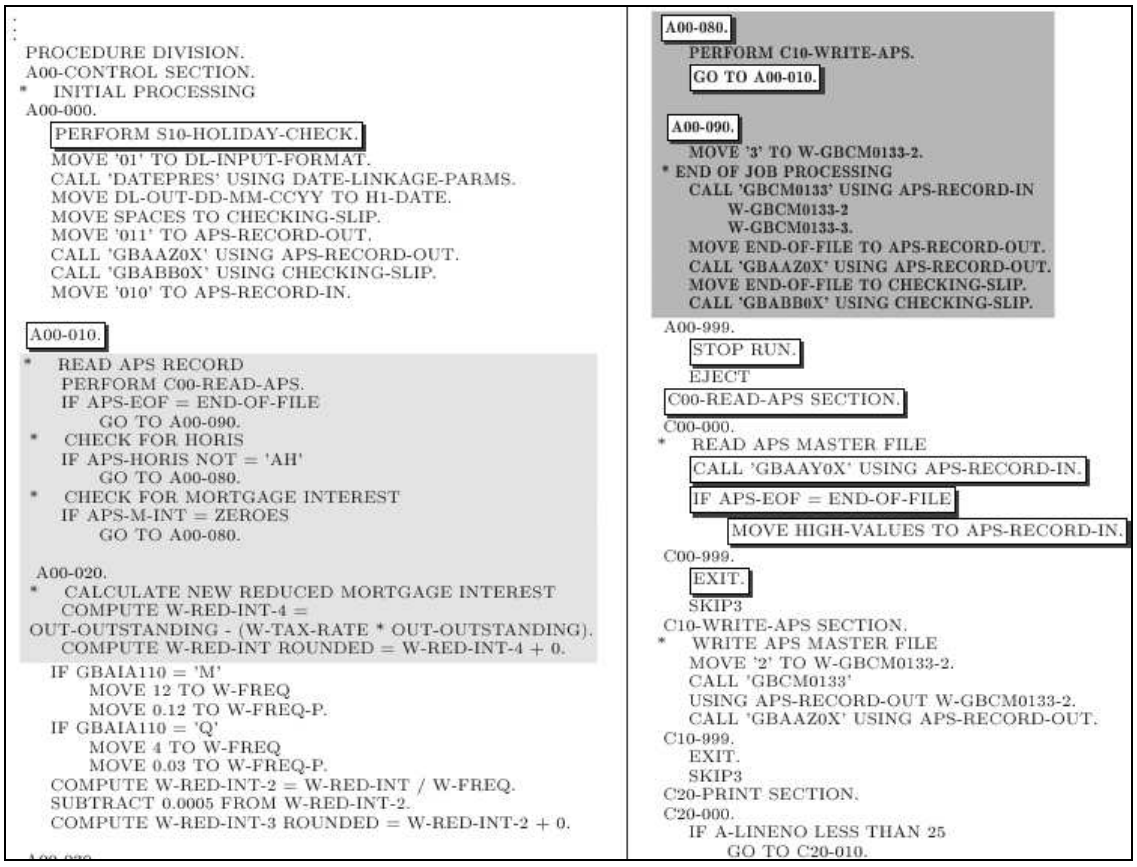

Figure 2.15: A COBOL program showing two concepts identified in it

2.3.4 Formal Concept Analysis

Formal Concept Analysis is a mathematical technique to identify grouping of objects that have common attributes and to represent them in a lattice structure to illustrate the generalization-specialization relationship among the groups.

Concept analysis starts with a context (*O*, *A*, *R*), a binary relation *R* between a set of objects *O* and their attributes *A*. A concept *C* (*E, I)* is a maximal collection of objects *E* (the extent) sharing common attributes *I* (the intent). A concept C_1 (E_1, I_1) is a sub-concept of another concept C_2 (E_2, I_2) if $E_1 \subseteq E_2$ or equivalently $I_2 \subseteq I_1$. The sub-concept relation is a partial order relationship that forms a lattice over the set of the concepts, each of the nodes of the lattice being a concept. For the infimum of the lattice the intent is empty and

the extent contains all the objects, whereas for the supremum the intent contains all the attributes and the extent is empty.

Concept analysis has been used as a data analysis method in other disciplines for a while. In software engineering its applications include program understanding, automatic modularization of legacy [27] [28], detection of configuration interference, class hierarchy transformation [29], restructuring source file [30] etc.

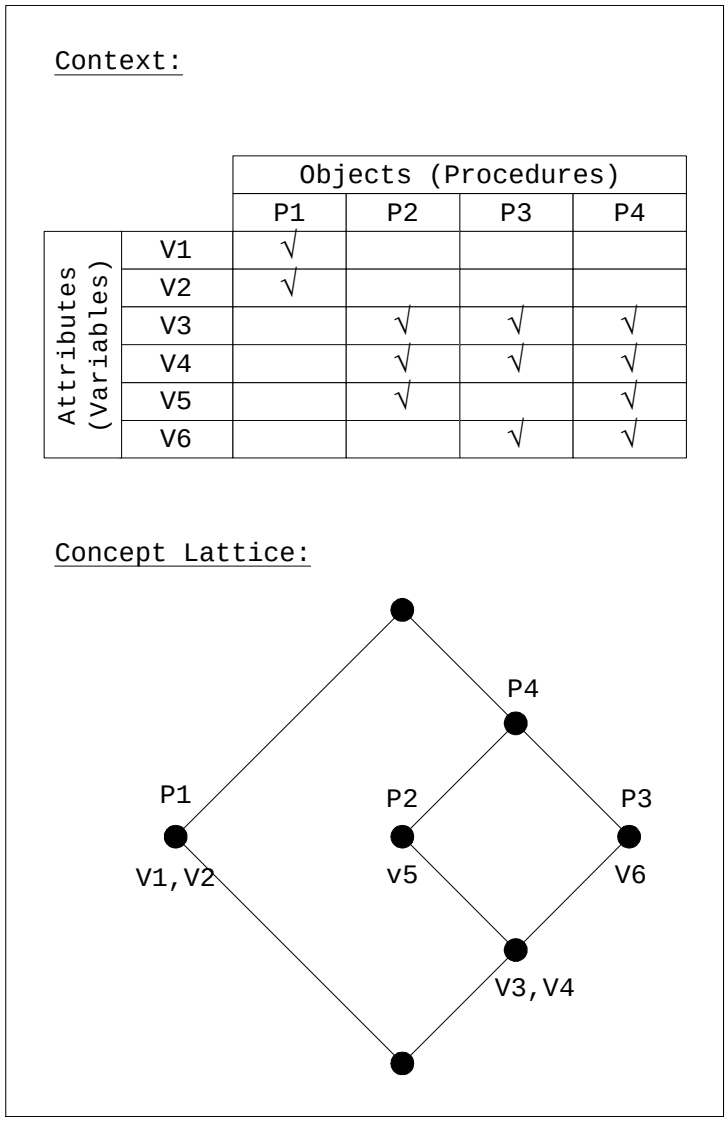

Figure 2.16: Example of Concept Lattice

For modularization purpose the procedures of a program are used as objects and the global variables as attributes. The usage of the variables by the procedures forms the relationship. Another approach is to use parameters and return types of the procedures as attributes instead of the global variables.

Figure 2.16 shows an example context and its corresponding concept lattice computed from a program consisting of 4 procedures and 6 global variables. The lattice consists of 7 concepts. The intent of the bottom most concept contains all the variables but the extent is empty. On the other hand the extent of the top most concept contains all the procedure but the intent is empty.

2.3.5 Composition

Source code analysis techniques like concept assignment, program slicing and formal concept analysis has been used in different combinations in the area of program comprehension and re-engineering.

A framework for unifying concept assignment with slicing is presented in [40]. The paper presents three algorithms for combining the two analysis techniques and demonstrates their application in reuse and reverse engineering. The algorithms are for computing an executable concept slice, key statement analysis and concept dependence analysis.

The idea of a lattice of a slice was introduced in [39] to analyze the relationship among the decomposition slices of a program. The lattice is basically a relationship graph of the decomposition slices of a program. The lattice is used it to identify the ripple affects of changes made to a program during the maintenance of the program. The paper also

gives a list of modifications that can be done in the program without any affect on the rest of the program.

The work of [39] was extended in [41] by performing formal concept analysis to build the concept lattice of decomposition slices as an extension of decomposition slice graphs. The paper demonstrates the use of the data structure in change impact analysis.

2.4 Reverse/Re-engineering Frameworks

In this section we describe some tools and frameworks for program analysis, comprehension, reverse engineering and re-engineering.

2.4.1 Software Refinery

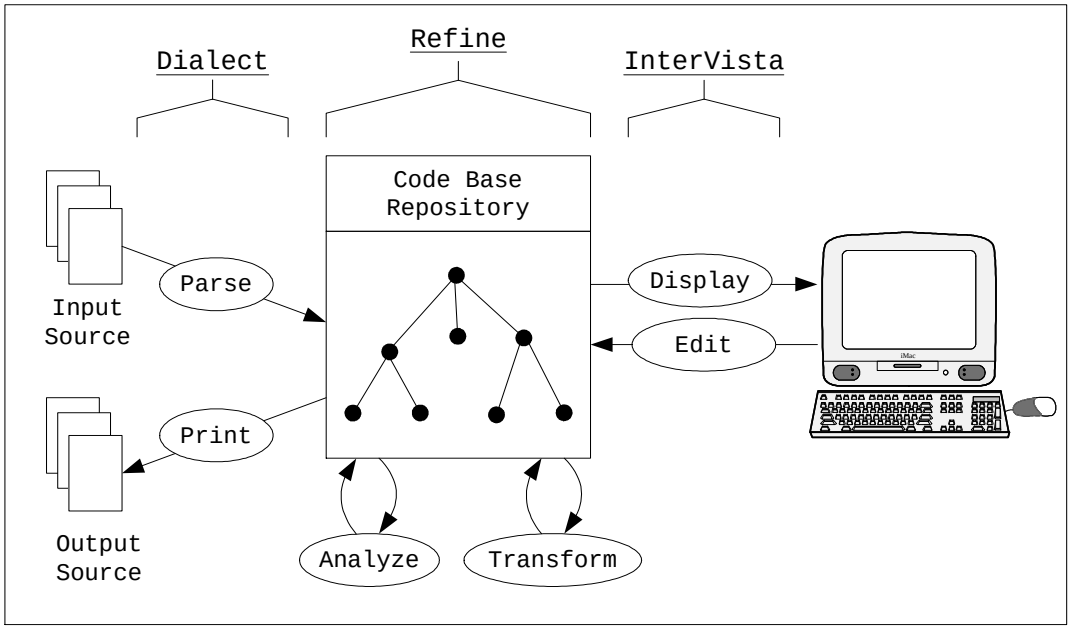

Figure 2.17: Architecture of Software Refinery

The Software Refinery [31] is a commercial round trip re-engineering framework developed by The Reasoning Systems Inc. It consists of three different toolset namely **Dialect**, **Refine** and **InterVista**. The core of the system is a Code Base Management

System (CBMS) called Refine to perform analysis and transformation of software programs. The source code is stored as a language independent annotated abstract syntax tree structure in an object-oriented repository in the CBMS. A high-level language is provided to query and update the repository to perform analysis and transformations on the source code respectively. Dialect is a language processing tools – a parser generator – that constructs a parser, pretty printer and pattern matcher from a specification of the syntax of any arbitrary language. The specification is a grammar written in a special high-level language. The generated parser then parses the source text of the language and produces objects in the repository. On the other hand the pretty printer reproduces the source text from the objects in the repository. The system comes with support for C, FORTRAN, ADA and COBOL. Any additional language can be added by providing a grammar for it and generating a parser for it. InterVista is a GUI built on top of X11 windowing system. Figure 2.17 sketches the architecture of the Software Refinery System.

2.4.2 Generic Understanding of PROgrams

The Generic Understanding of PROgrams (GUPRO) [32] is a framework at Institut für Softwaretechnik, Universität Koblenz-Landau for modeling, representing and analyzing software. The tool provides a way for multi-level understanding of heterogeneous software. The level of understanding required for analysis is described in a Concept Model using Extended Entity Relationship (EER) like graphical language.

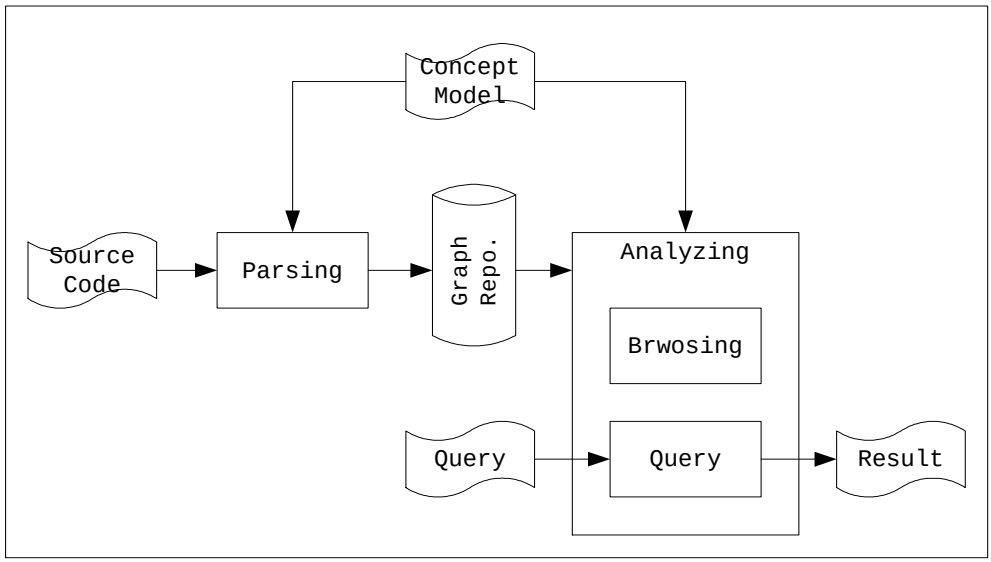

Figure 2.18: Architecture of GUPRO

The GUPRO toolset consist of three parts – a parser generator, a repository and an analyzing facility. The parser generator generates parser for any arbitrary language, which transforms source code into TGraph representation according to the concept mode. The repository stores the concept models and the TGraphs. Language independent query language GReQL for preparing arbitrary reports from the graphs. GUPRO also supports GXL as representation format. Figure 2.18 illustrates the architecture of the GUPRO system.

2.4.3 SWAG Software Architecture Toolkit

The SWAG Software Architecture Toolkit [33] is a software architecture extraction and presentation toolkit developed by the Software Architecture Group of University of Waterloo.

The toolkit follows three-stage pipeline architecture – extractor, manipulator and presenter. The extractor is a filter named **cppx** that extracts facts from C/C++ source files and stores them as a TA program. It can also generate the facts in GXL. The manipulator

consists of three filters – **prep** for cleaning up the TA programs, **linkplus** to link the cleaned files to create one large graph and **layoutplus** to cluster facts based on a given containment information of architectural entities and prepare the graph layout of the concrete architecture. Finally **lsedit** is the viewer for the generated software landscape. New filters can be easily added in the pipeline to perform other kind of analysis or design recover activities. Figure 2.19 shows the SWAGKit pipeline.

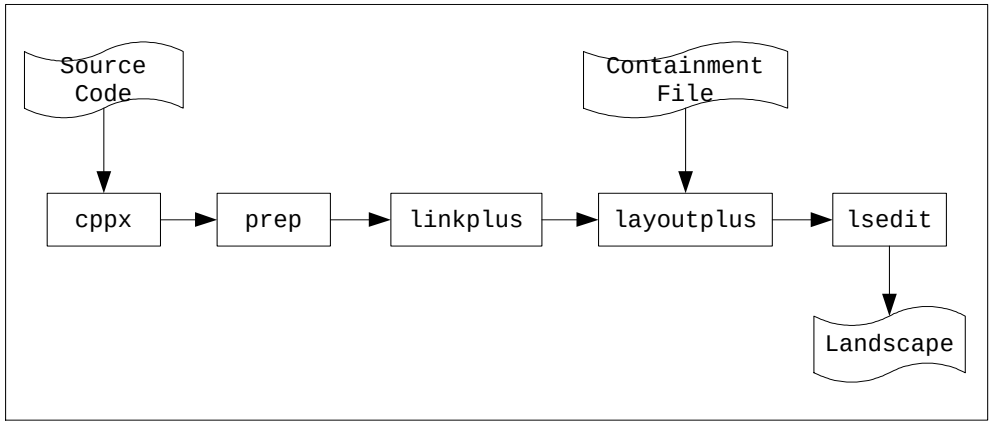

Figure 2.19: SWAGKit Pipeline

2.4.4 Integrated Software Maintenance Environment

The Integrated Software Maintenance Environment (ISME) [34] is a prototype toolset to facilitate software maintenance activities. It includes a source code transformer tool for parsing Java and C/C++ source text and generating language specific representations like JavaML and CppML or language independent representation like OOML. It also includes a simple analysis tool that computes different metrics on these representations. To facilitate tool integration ISME prototype contains a Rigi driver that processes OOML and generates RSF tuple from it, which then can be visualized in Rigi. ISME provides a framework to build high-level transformers on top of the source code representations to

map them into higher levels of abstraction. Figure 2.20 illustrates the ISME and its abstraction layers.

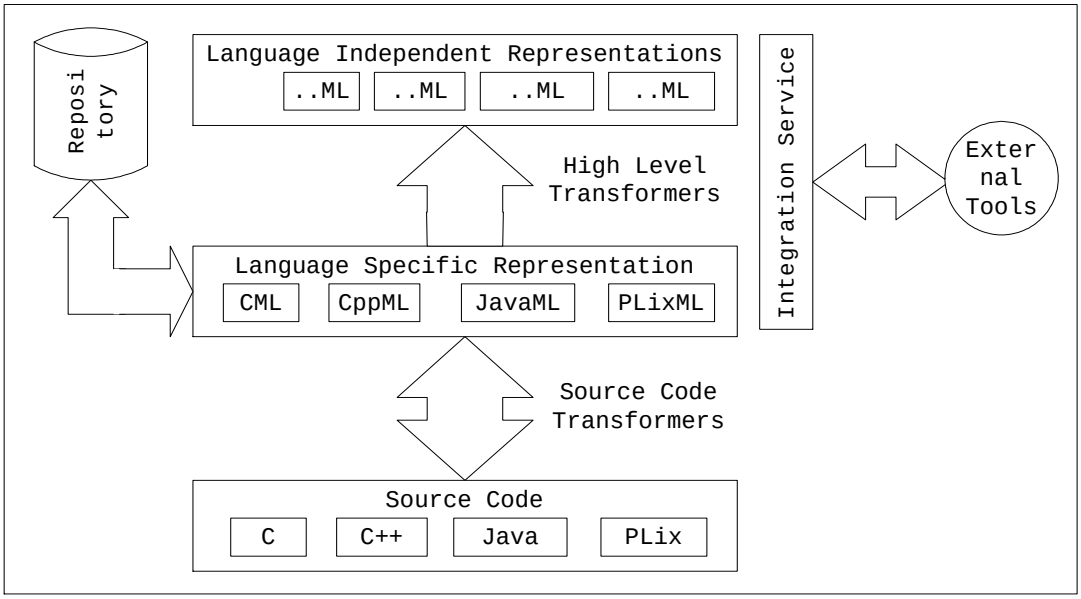

Figure 2.20: Architecture of ISME

Chapter 3

System Representation

In this chapter we present a framework for program representation and analysis. The framework enables programming language neutral representation of source code at different levels of abstractions. It also facilitates development of generic source code analysis tools using it.

3.1 XML-based Program Representation

There is a growing momentum of activities related to XML representation of source code facts and usage of XML as data exchange format among software tools. In this section we will discuss XML and its suitability as program representation format.

3.1.1 Extensible Markup Language

The Extensible Markup Language (XML) [35] is a World Wide Web Consortium (W3C) [36] standard for storing and exchanging structured documents. It was originally derived from Standard Generalized Markup Language (SGML) as a software and hardware independent flexible text format. Now the use of XML ranges form large-scale electronic publishing to exchanging multimedia data on the web.

Unlike HTML, which was designed to display data, XML was designed to described data. The data in an XML document are described using meaningful markup tags around them, which add semantics with the syntax to the data in a given domain. The tags are not predefined in XML. The Valid tags and the relationship among the tags can be described by defining a Document Type Definition (DTD) or XML Schema for a particular class of XML documents.

DTD:

```
<!-- employee.dtd 0.1 -->

<!ELEMENT employee (emp*)>
<!ELEMENT emp (name, address, phone*)>
<!ATTLIST emp eid ID>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ATTLIST phone type CDATA>
```

Document:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE employee SYSTEM "employee.dtd">

<employee>

  <emp eid="23452">
    <name> Jeremy Duncanson </name>
    <address> 68 Wabasha St, Winona, MN, 55987 </address>
    <phone type="Home"> 507-454-4498 </phone>
    <phone type="Mobile"> 507-452-1659 </phone>
  </emp>

  <emp eid="34529">
    <name> Faujiah Hajad </name>
    <address> 129 7th St W, Winona, MN, 55987 </address>
    <phone type="Home"> 507-454-1163 </phone>
  </emp>

</employee>
```

Figure 3.1: An Example XML Document and its DTD

An XML document is a hierarchical data structure consisting of a root element denoted by a single tag pair. All other elements are within the root element. Each element has a name and can have sub-elements nested inside it. The elements can have attributes

to provide additional information about them, the attributes being name and value pairs. The data itself reside inside the element tags. Figure 3.1 illustrates an example XML document that stores information about employees of an organization and the corresponding DTD that defines the data structure.

W3C defines two standard interfaces for accessing XML documents – the Document Object Model (DOM) and the Simple API for XML (SAX). DOM is a tree based API to generate an internal tree structure for the XML document and allows an application to navigate and manipulate the tree. On the other hand SAX is an event-based API that generates events as it parses the XML document that an application can handle to perform various tasks.

A family of additional technologies is being developed to support and extend the capabilities of XML. Extensible Stylesheet Language (XSL) is a set of languages for XML transformation and presentation. It includes XSLT, XPath and XSL-FO. Extensible Stylesheet Language Transformations (XSLT) is a language to transform an XML document into other XML document. XML Path Language (XPath) is a language to locate information inside and XML document. XSL Formatting Objects (XSL-FO) specifies formatting semantics for XML. XML Query (XQuery) is an expression language to provide a flexible query facility to extract data from XML files as well as other databases. XML Linking Language (XLink) allows elements to be inserted into XML documents in order to create and describe links between resources.

3.1.2 XML Benefits

Following are some benefits of using XML as document format:

- **Simple:** XML documents are self-describing, simple and easy to understand by both humans and computer applications alike.
- **Open Standard:** XML is a hardware software independent non-proprietary format.
- **Tool Support**: Inexpensive off-the-shelf tools are available to process XML documents automatically. All the tools conform to APIs provided by W3C standards.
- **Internationalized:** XML is fully internationalized with all conforming processors required to support the Unicode character set in both its UTF-8 and UTF-16 encodings.
- **Interoperable:** XML enables information interchange by the use of platform independent data format that allows creation of common data structures to be used by different applications. Availability of translating technologies and tools like XSL and XSLT also facilitates interoperability.
- **Extensible**: Document format can be extended easily by creating new element and attribute tags to accommodate new information.
- **Searchable:** The document structure of XML and availability of technologies like XPath and XQuery make it possible to develop good searching and indexing techniques.

3.1.3 XML as Program Representation Format

The artifacts extracted from a program are necessarily structured information that needs to be stored and exchanged among different tools. The features of XML, discussed above, make it a good choice for external representation formats for program representations.

A description of the use of XML for source code representation at parse tree/AST level can be found at [37]. JavaML, CppML and OOML [15] are developed as part of ISME tool to represent source code for Java, C++ and any object-oriented language respectively. They take advantage of the tree structure of XML document and map the syntax tree to an XML DOM tree. The tree is validated against a DTD that described the language grammar.

On the other hand GXL [19] proposes are framework for representing any kind of typed, attributed and directed graphs as XML documents. Most of the program representation formalisms being graphs of different sort, GXL provides a mechanism to express them. The data structures of the graphs are also expressed as schema in GXL. Tools like SWAG Kit, GUPRO etc. uses GXL as a data exchange formats.

3.2 System Architecture

The Integrated Software Maintenance Environment (ISME) [34] provides an XML based source code representation framework based on the syntax trees of the source code. It provides source code transformer tools for parsing Java and C/C++ source text and generating language specific representations JavaML and CppML. It also provides a representation named OOML based on the generalized domain model for object-oriented languages. The uses of the representations are demonstrated with simple source code analysis tools, e.g. software metrics computation and source fact extraction. While the parse tree level representations are useful for some type of analysis, they are not usable for sophisticated higher level and language neutral generic analysis.

We extend the ISME framework to accommodate the additional layers of abstractions on top of the source code representations. The additional layers correspond to program

representations formalism like Control Flow Graph, Program Dependence Graph, Call Graph etc. Figure 3.2 presents the extended ISME framework for enabling generic program analysis. The framework follows a pipe and filter type architectural style. The pipe components are the different layers of abstractions of the program source and the filter components are the representation transformers and the analysis tools.

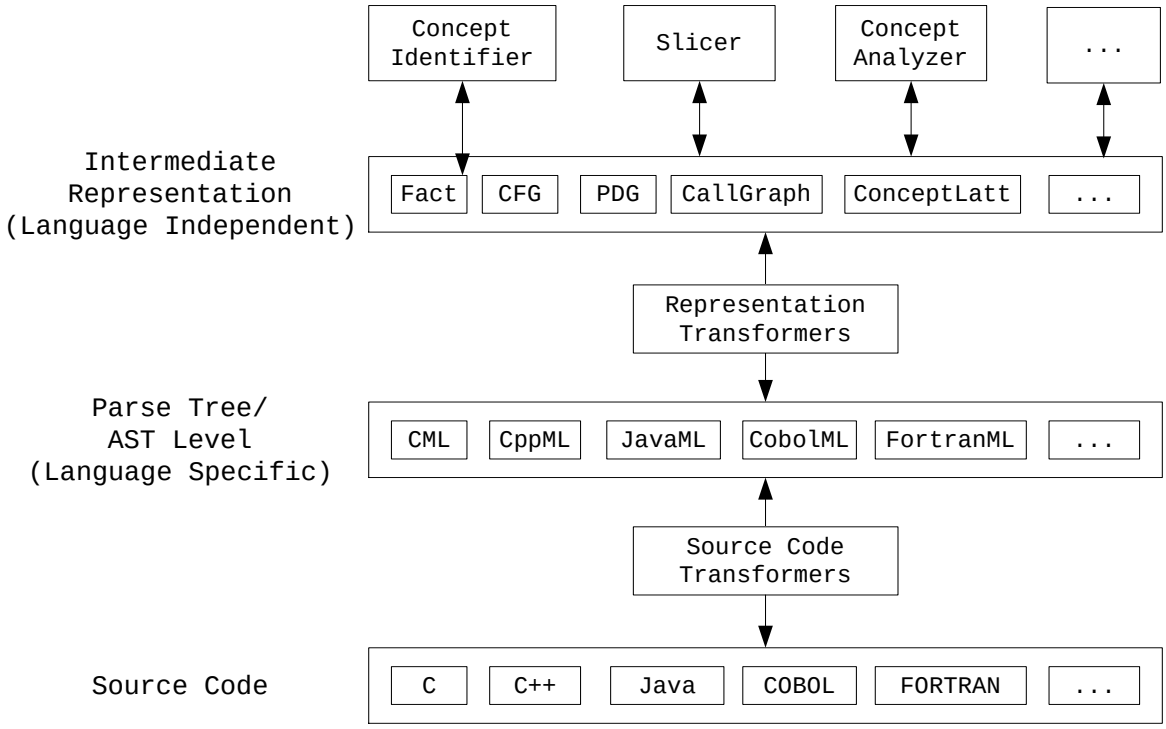

Figure 3.2: System Architecture of Extended ISME Framework

3.2.1 Layer of Abstractions

Source Code (Layer 0): This layer is the original source text of the program to be analyzed as it is.

Language Specific Representations (Layer 1): This is the first level abstraction of the source code in terms of the parse tree or the abstract syntax tree of the program. The

representations are programming language specific and hence one XML sub-language is defined for each programming language to represents its AST. CML, CppML and JavaML are defined to represent the AST of C, C++ and Java source code respectively. Similarly other XML sub-languages can be defined for other programming languages to be analyzed.

Language Independent Representations (Layer 2): This is the next level of abstraction of the program in terms of control flow graph, program dependency graph, call graph etc. These representations are programming language independent. One XML sub-language is defined for each of the representation formalism.

3.2.2 Tools

Source Code Transformers: The source code transformers are tools that parse the source text and generate the corresponding AST in an XML sub-language. There is one transformer for each of the languages to be analyzed.

Representation Transformers: These transformers convert the language specific representations into language independent representations of Layer 2. One transformer is required for each language and each intermediate representation.

Analysis Tools: Finally the program analysis tools that act on the language neutral representations to perform data flow analysis, slicing, concept analysis etc.

3.3 Language Independent Representations

The XML-sublanguages for language specific parse tree or AST level representations of Layer 1 are defined as part of the ISME tool. As part of our work we define XML-

sublanguages to represent important intermediate representations of Layer 2 that is necessary for generic program analysis.

3.3.1 The Facts (FactML)

The facts of a program are the building block constructs of the program and the basic relationships among them. These are used by the representation formalisms as basic units of composition. The constructs are statements, variables, data types and functions and the basic relationships are the uses or definitions of variables and calls to functions.

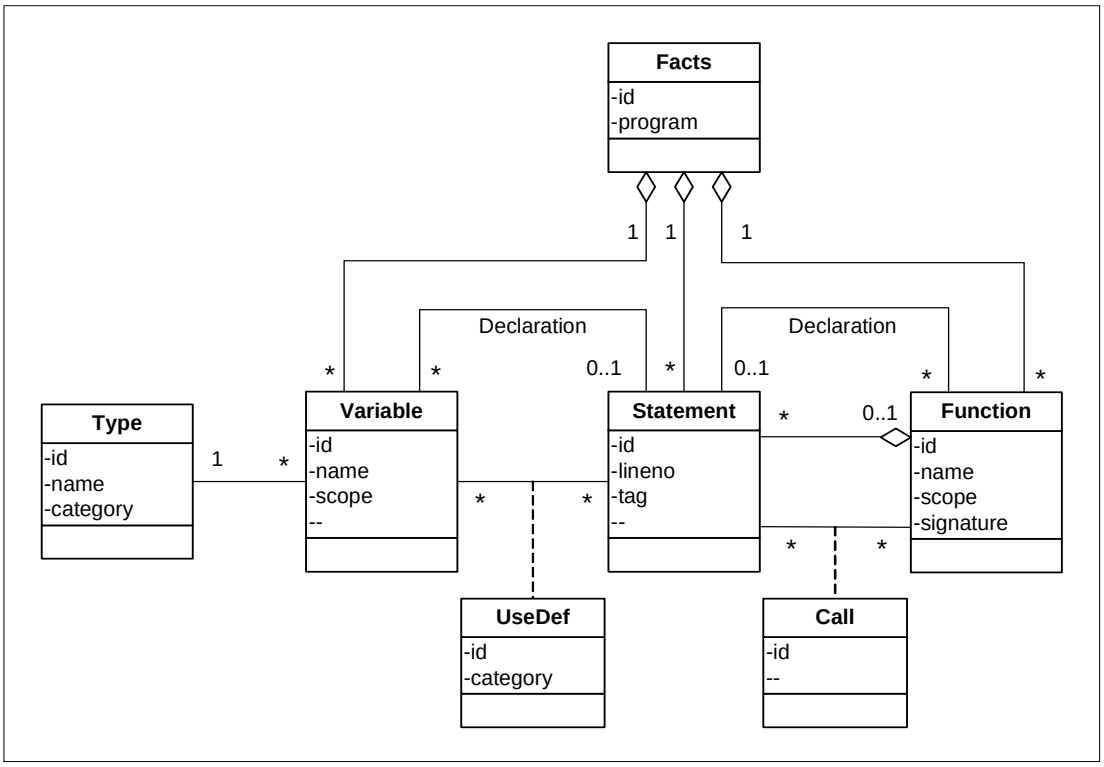

Figure 3.3: UML Model for Facts

Figure 3.3 presents the model of the facts in UML. The facts of a program consist of classes for the types, variables, statements, and functions in the program. The classes are associated with each other in various ways. Each member of the *Variable* class is associated with a member of the *Type* class by its data type. Declaration relationship of a

Variable a *Statement* is a simple association, whereas uses and definitions of a *Variable* in a *Statement* is more complex and require an association class. There can three different relationships between a *Statement* of a program and a Function. A *Function* is declared in a *Statement*, a *Function* consists of many *Statement* and a *Statement* can call one or more *Function*.

The UML diagram is a graphical representation of the model for facts. For storage and model interchange, the UML model is encoded in a Document Type Definition (DTD) in order to effectively validate the XML documents containing the facts. The production rules for transformation from UML to XML DTD are as follows

- Classes are mapped as elements
- Attributes of the classes are mapped as attributes in the elements
- Aggregations are mapped as sub-elements separated by or
- Inheritances are mapped as sub-elements
- Simple associations are shown by IDREFs
- Association classes are mapped as elements showing the associations by IDREFs
- Elements with same tags originating from same tree node are grouped as sub-elements under one big element.

Figure 3.4 presents the DTD corresponding to the UML model given in Figure 3.3 The root *Facts* element consists of *Statements, Types, Variables, Functions, UseDefs,* and *Calls* elements, which consists of the *Statement, Type, Variable, Function, UseDef,* and *Call* elements respectively grouped together under a bigger element. The *Facts* element has a *program* attribute to identify the program the facts belong to.

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

<!ELEMENT Facts (Statements?, Types?, Variables?, Functions?,
                 UseDefs?, Calls?)>
<!ATTLIST Facts
      program CDATA>

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

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

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

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

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

<!ELEMENT Calls (Call*)>
```

```
<!ELEMENT Call EMPTY>
<!ATTLIST Call
      id ID #REQUIRED
      statement IDREF #REQUIRED
      function IDREF #REQUIRED>
```

Figure 3.4: A DTD for Facts

The *Statement* elements represent each independent line of statements in the program. Each *Statement* has a *lineno* attribute to indicate line number of the statement. There is an optional *tag* attribute to indicate the statement type. This is the XML tag that is associated with the statement in its language dependent markup representation. There also is an optional *function* attribute to refer to the function the statement is part of.

The *Type* elements are the data types of all the variables referenced in the program. It has a *name* and a *category* attribute. The *category* attribute denotes if the type is a primitive data type, a composite type, an object or a pointer.

The *Variable* elements list all the variables referenced in the program. Each *Variable* has its *name* and a *declared* attribute associated with it. The *declared* attribute refers to a *Statement* element to indicate where the variable is declared in order to resolve the scope for it. This attribute is declared as optional in order to accommodate the variables that are defined externally. There is a *scope* attribute to indicate is the variable is local to a function or global or a parameter and a *type* attribute that refers to a *Type* element to indicate the data type of the variable. The *UseDef* elements indicate the actual uses and definitions of the variables in the program. Each element points to a *Variable* element and a *Statement* element where it is being used or defined.

The *Function* elements list all the functions called in the program. Each *Function* has its *name* and a *declared* attribute associated with it. The *declared* attribute refers to a *Statement* element to indicate where the function is declared in the program. This is an

optional attribute to accommodate the function defined outside this program. There is an optional *signature* attribute to uniquely identify overloaded functions. The *Call* elements are the actual function calls in the program. Each element points to a *Function* element and a *Statement* element where it is being called.

Figure 3.5 shows an example C program and Figure 3.6 shows its facts in XML

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

Figure 3.5: An Example C program

```
<Facts>
   <Statements>
      <Statement id=1 lineno=1 function=”main” />
      <Statement id=2 lineno=2 function=”main” />
      <Statement id=3 lineno=3 function=”main” />
      <Statement id=4 lineno=4 function=”main” />
      <Statement id=5 lineno=5 function=”main” />
      <Statement id=6 lineno=6 function=”main” />
   </Statements>
   <Types>
      <Type id=7 name=”int” category=”Primitive” />
   </Types>
   <Variables>
      <Variable id=8 name=”a” scope=”Local” declared=3 type=7 />
   </Variables>
   <UseDefs>
      <UseDef id=9  category=”Def” statement=3 variable=8 />
      <UseDef id=10 category=”Def” statement=5 variable=8 />
      <UseDef id=11 category=”Def” statement=6 variable=8 />
      <UseDef id=12 category=”Use” statement=4 variable=8 />
      <UseDef id=13 category=”Use” statement=5 variable=8 />
   </UseDefs>
   <Functions>
      <Function id=14 name=”main” scope=”Internal” declared=1 />
   </Functions>
```

Figure 3.6: The Facts – facts.xml

3.3.2 Control Flow Graph (CFGML)

A CFG is a directed graph. The nodes represent basic blocks and arcs represent possible flow of control from one basic block to another. A basic block contains a sequence of consecutive instructions.

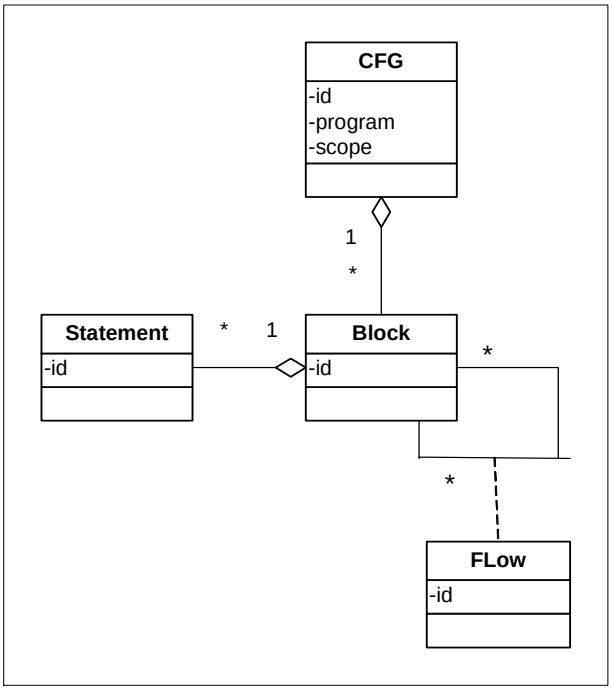

Figure 3.7: UML Model for CFG

Figure 3.7 presents the UML model for a CFG. As shown in the UML model, A CFG consists of classes for basic blocks and statements with part-whole relationships between them. Each *Block* is associated with another *Block* due to flow of control between them and can be described as an association class *Flow*.

Figure 3.8 shows the XML DTD for CFG corresponding to the UML model of Figure 3.7. While transforming to XML, the basic programming constructs are referred from the DTD for Facts using XLink instead of redefining them.

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

<!ELEMENT CFG (Blocks?, Flows?)>
<!ATTLIST CFG
       xmlns:xlink CDATA #FIXED “http://www.w3.org/1999/xlink”
       program CDATA
       scope CDATA>

<!ELEMENT Blocks (Block*)>

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

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

<!ELEMENT Flows (Flow*)>

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

Figure 3.8: A DTD for CFG

The XML sub-language for CFG consists of a root *CFG* element that can have a *program* and a *scope* attribute to identify the program and the function in the program for which the CFG is built for. The *CFG* element consists of two sub-elements – *Blocks* and *Flows*.

Blocks element consists of many *Block* elements, which are the basic blocks. Each *Block* element has a *label* attribute associated with it. There will be one *Block* that is the *start* of the graph and another that is *exit*; the rest are *regular*. Each *Block* in turn consists of many *Statement* elements. Each *Statement* refers to a *Statement* element in a *Facts* document with an XLink.

Flows element consists of many *Flow* elements whereas each *Flow* element joins two *Block* elements as given by the *from* and *to* attributes.

Figure 3.9 shows the XML representation of the CFG corresponding to the program in Figure 3.5

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

Figure 3.9: The CFG – cfg.xml

3.3.3 Program Dependence Graph (PDGML)

A PDG is a directed graph whose nodes are connected by several kinds of arcs. The nodes represent statements and predicate expressions. There is a special entry node, some initial definition of variable nodes and some final use of variable nodes. The arcs represent either control dependence or data dependence. The data dependence arcs are labeled with the variable that causes the dependence.

Figure 3.10 presents the UML model for PDG. A PDG consists of many *Node*, each of which that can be either a *Statement* or a *Special* node. Each *Special* node may be related to *Variable* to identify the variable in initial definition or final use. Each *Node* is

associated with another *Node* by either control and data flow between them and the relationships can be expressed as association classes *ControlFlow* and *DataFlow* respectively. Each *DataFLow* is also related with a *Variable* causing the flow.

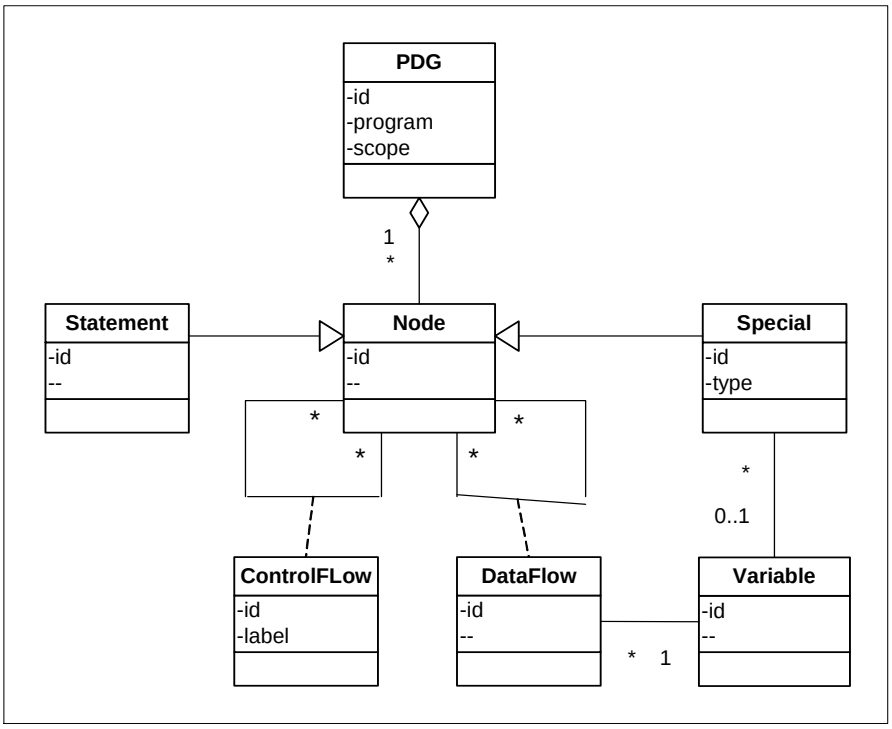

Figure 3.10: UML Model for PDG

Figure 3.11 shows the XML DTD for PDG corresponding to the UML model of Figure 3.10. While transforming to XML, the basic programming constructs are referred from the DTD for Facts using XLink instead of redefining them. The control and data flow elements are group together.

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

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

<!ELEMENT Items (Variables?, Nodes?)>

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

<!ELEMENT Nodes (Node*)>
<!ELEMENT Node (Special|Statement) >

<!ELEMENT Special EMPTY>
<!ATTLIST Special
      id ID #REQUIRED
      type (Entry|InitialDef|FinalUse) #REQUIRED
      variable IDREF>
<!ELEMENT Statement EMPTY>
<!ATTLIST Statement
      id ID #REQUIRED
      xlink:type (simple) #FIXED “simple”
      xlink:href CDATA #REQUIRED>

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

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

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

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

Figure 3.11: A DTD for PDG

The XML sub-language for PDG consists of a root *PDG* element that can have a *program* and a *scope* attribute to identify the program and the function in the program for which the PDG is for. The *PDG* element consists of two types of elements – *Items* and *Dependencies*.

The *Items* element consists of the *Nodes*, and *Variables* elements that in turn consist of many *Node* and *Variable* elements. Each *Node* element is either a *Special* or a *Statement* element. Each of the *Special* elements represents the special nodes of the PDG for the entry, initial definition of variables and final use of variables. Each *Statement* refers to a *Statement* element and each *Variable* refers to a *Variable* element in a *Facts* document with XLinks.

The *Dependencies* element consists of *ControlFlows* and *DataFlows* elements. *ControlFlows* consists of many *ControlFlow* elements whereas each *ControlFlow* joins two *Statement* or *DummyNode* elements as given by the *from* and *to* attributes. *DataFlows* consists of many *DatalFlow* elements whereas each *DataFlow* joins two *Statement* or *DummyNode* elements as given by the *from* and *to* attributes and the *over* attribute gives the *Variable* that causes the dependency.

Figure 3.12 shows the XML representation of the PDG corresponding to the program in Figure 3.5

```
<PDG>
   <Items>
      <Variables>
         <Variable id=1 xlink.href=”Facts.xml#8” />
      </Variables>
      <Nodes>
         <Node>
             <Special id=2 type=”Entry” />
         </Node>
         <Node>
             <Special id=3 type=”FinalUse” variable=1/>
         </Node>
         <Node>
             <Statement id=4 xlink.href=”Facts.xml#3” />
         </Node>
         <Node>
             <Statement id=5 xlink.href=”Facts.xml#4” />
         </Node>
         <Node>
             <Statement id=6 xlink.href=”Facts.xml#5” />
         </Node>
         <Node>
             <Statement id=7 xlink.href=”Facts.xml#6” />
         </Node>
      </Nodes>
   </Items>
   <Dependencies>
      <ControlFlows>
         <ControlFlow id=8 from=2 to=4 label=”True”>
         <ControlFlow id=9 from=2 to=5 label=”True”>
         <ControlFlow id=10 from=2 to=7 label=”True”>
         <ControlFlow id=11 from=2 to=3 label=”True”>
         <ControlFlow id=12 from=5 to=6 label=”True”>
      </ControlFlows>
      <DataFlows>
         <DataFlow id=13 from=4 to=5 over=1>
         <DataFlow id=14 from=4 to=6 over=1>
         <DataFlow id=15 from=7 to=3 over=1>
      </DataFlows>
   </ Dependencies >
</PDG>
```

Figure 3.12: The PDG – pdg.xml

3.3.4 Concept Lattice (LattML)

The ConExp [38], a tool to perform formal concept analysis, as part of it defines XML sub-language to describe the context and the lattice for formal concept analysis. Figure

3.13 shows the DTD from ConExp, Figure 3.14 shows screenshots of the tool with a context and its corresponding lattice and Figure 3.15 show the XML file that represents the context.

```
<!ELEMENT ConceptualSystem (Version, Contexts, Lattices)>

<!ELEMENT Version EMPTY>
<!ATTLIST Version MajorNumber CDATA MinorNumber CDATA>

<!ELEMENT Contexts (Context*)>
<!ELEMENT Context (Attributes, Objects)>
<!ATTLIST Context Identifier CDATA Type CDATA>

<!ELEMENT Atributes (Attribute*)>
<!ELEMENT Atribute (Name)>
<!ATTLIST Attribute Identifier CDATA>
<!ELEMENT Name (#PCDATA)>

<!ELEMENT Objects (Object*)>
<!ELEMENT Object (Name, Intent)>

<!ELEMENT Intent (HasAttribute*)>
<!ELEMENT HasAttribute EMPTY>
<!ATTLIST HasAttribute AttributeIdentifier CDATA>

<!ELEMENT Lattices (Lattice*)>
<!ELEMENT Lattice (AttributeMask, LineDiagram)>

<!ELEMENT AttributeMask (HasAttribute*)>
<!ELEMENT LineDiagram (ConceptFigures, LineDiagramSettings)>

<!ELEMENT ConceptFigures (LineDiagramFigure*)>
<!ELEMENT LineDiagramFigure (FigureCooords, Intent)>
<!ATTLIST LineDiagramFigure Type CDATA>

<!ELEMENT FigureCooords (Point2D)>
<!ELEMENT Point2D EMPTY>
<!ATTLIST Point2D x CDATA y CDATA>

<!ELEMENT LineDiagramSetting (AttributeLabelDisplayMode,
                              ObjectLabelDisplayMode)>
<!ELEMENT AttributeLabelDisplayMode EMPTY>
<!ATTLIST AttributeLabelDisplayMode Value CDATA>

<!ELEMENT ObjectLabelDisplayMode EMPTY>
<!ATTLIST ObjectLabelDisplayMode Value CDATA>
```

Figure 3.13: The ConExp DTD

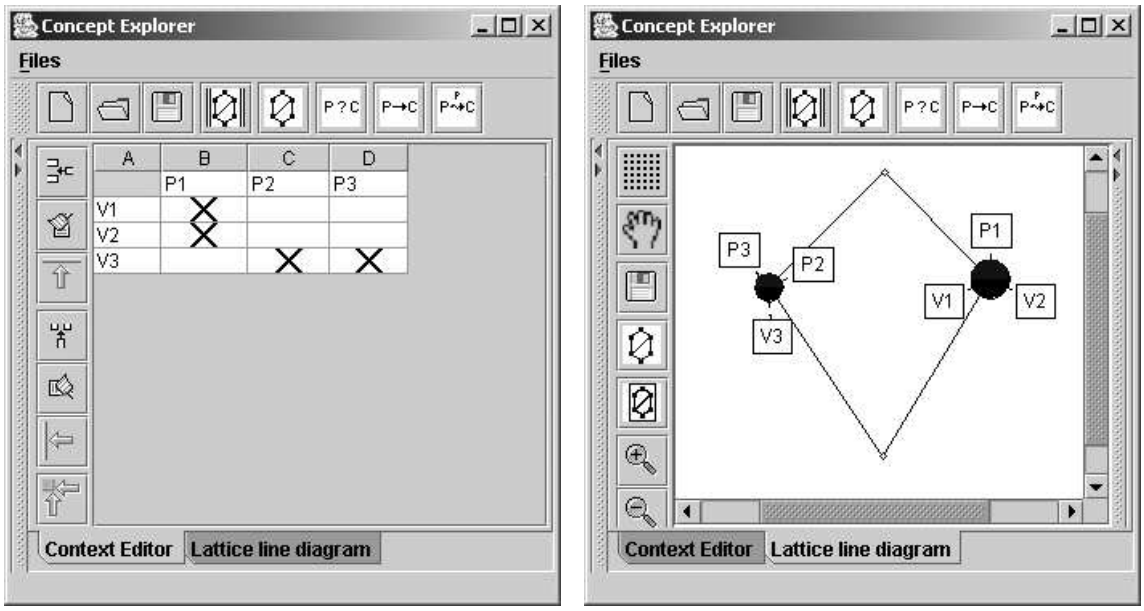

Figure 3.14: The ConExp Tool Showing a Context and Its Lattice

```
<ConceptualSystem>
   <Version MajorNumber="1" MinorNumber="0" />
   <Contexts>
      <Context Identifier="0" Type="Binary">
         <Attributes>
            <Attribute Identifier="0"> <Name>P1</Name> </Attribute>
            <Attribute Identifier="1"> <Name>P2</Name> </Attribute>
            <Attribute Identifier="2"> <Name>P3</Name> </Attribute>
         </Attributes>
         <Objects>
            <Object> <Name>V1</Name>
               <Intent> <HasAttribute AttributeIdentifier="0" /> </Intent>
            </Object>
            <Object> <Name>V2</Name>
               <Intent> <HasAttribute AttributeIdentifier="0" /> </Intent>
            </Object>
            <Object> <Name>V3</Name>
               <Intent>
                  <HasAttribute AttributeIdentifier="1" />
                  <HasAttribute AttributeIdentifier="2" />
              </Intent>
            </Object>
         </Objects>
      </Context>
   </Contexts>
</ConceptualSystem>
```

Figure 3.15: The XML File Describing the Context

Chapter 4

Source Code Modularization

In this chapter we present the source code modularization technique, which is based on the program representation formalism we propose, named Lattice of Concept Slices.

4.1 Existing Techniques

The source code analysis techniques discussed in Chapter 2 have been used by researchers for program comprehension and modularization. In this section we briefly discuss the existing techniques and their individual strengths and weaknesses

4.1.1 Concept Assignment

Concept assignment technique fragments the source code and assigns meaning to each of the fragments. Code segments corresponding to each fragment can be considered as candidate for modules. The advantage of this technique is that the fragmentation is done at right level of granularity, the domain concepts. But the drawback is that the code segments corresponding the domain concepts are not self-contained and not executable independently as separate modules.

Domain Model – The Adaptive Observer (DM-TAO) as part of the DESIRE tool [43] and Hypothesis Based Concept Assignment (HB-CA) [26] are the two most popular concept assignment techniques.

4.1.2 Program Slicing

Slicing gives an executable subset of the original program that preserves the behavior of the program with respect to a given variable at a given program point. Slicing has the advantage that the slices are self-contained and executable by themselves. But the problem of slicing is that the decomposition is done based on very fine-grained program variables instead of domain concepts. Modularization based on slicing may result into modules that contain a significant amount of duplicated code because of overlapping control flows. Moreover, even though each of the decompositions is self-contained, if the duplicated code modifies global program resources it may cause significant and undesirable side effects when deployed in separate modules.

The Unravel Project [44] and The Wisconsin Program-Slicing Tool [45] are among many tools available to perform program slicing. Both the tools work on ANSI C programs.

4.1.3 Formal Concept Analysis

Formal Concept Analysis is a mathematical technique used for identifying groupings of objects that have common attributes and representing them in a lattice structure to show the generalization-specialization relationship among the groups. In modularization, instead of decomposition concept analysis is used to identify grouping of program elements into modules. For example it is used to group together subroutines and global

data structures into ADTs for object-oriented migration. As a result this technique is not directly applicable to the type modularization we are interested in.

ConExp [38] and ToscanaJ [46] are two tools used for Formal Concept Analysis.

4.2 Lattice of Concept Slices

We now propose a program representation formalism for system modularization that we call the Lattice of Concept Slices. The lattice shows the contribution of the program statements towards the computation of the different domain concepts implemented of the program. Based on this representation we are going to propose our modularization algorithms. The goal is to achieve a modularization such that each module implements preferably a single domain concept, each module is self-contained, there is minimal duplication in code and there is no side effect among modules

The formation of the lattice is a three-stage process – domain concept identification, computation of concept slices and finally, building and analyzing the lattice.

4.2.1 Identification of Domain Concepts

The first step is the identification of domain concepts in the program. This can be done using exhaustive concept assignment techniques like HB-CA or DM-TAO. Instead we take the simpler approach of structural and informal analysis of the source code.

In our approach the software engineer provides a list of domain concepts that are taken from the functional specifications of the system and are implemented in the given program. Furthermore, she associates such domain concepts with one or more program elements such as variables and structural idioms in the source code. The associations may be based on the use or def of a particular data type or variable, call to a procedure or

method, a particular variable passed as parameter in a call, expressions matching on identifier naming or comments etc. These associations cannot be by no means exhaustive and only serve as a starting point of the analysis. In addition, the software engineer identifies some statements as key statements [40] that are believed to contribute the most in the computation of that domain concept

Some domain concepts can be identified automatically based on a set of general criteria. The rationale behind the criteria is that any information being sent outside from the program or any change in the internal state that is externally visible are information that will be used by other parts of the program and hence are candidates for being part of a domain concept. Such criteria can be the identifiers such as – return parameters of a function or method, modified formal parameters that have been called by reference, variables in output/print statements, global variables or class attributes been modified. These identifiers are candidates for domain concepts and the statements that modify these identifiers will be considered as part of the corresponding domain concept. The software engineer may accept or reject the suggestions made automatically.

The outcome of this step is a set of domain concepts and associated with each of them is a set of program statements that implement the concept, where some of the statements are marked as key statements. Each of the concepts is a candidate to form a possible module and the associated statements will comprise the statements for the module.

As an illustration of the technique consider Figure 4.1 that illustrates a simple line count program taken from [39] that counts the number of lines, words and characters in a text file. We are attempting to modularize main function. The function outputs the calculation results of three variables – *nl*, *nw* and *nc* statement 24, 25 and 26 respectively. Hence the automatic identification technique suggests the possible presence of three

domain concepts corresponding to these three variables. The software engineer confirms the suggestion and names the domain concepts as *Lines*, *Words* and *Chars* respectively. The *nl* variable is being computed in statement number 6 and 16. Statement 6 is the declaration and initialization and does not directly contribute to the computation of *nl*, whereas Statement 16 is the place where the main computation is being done. In this respect the Lines domain concept consists of statement 6, 16 and 24, where statement 16 is the key statement. Table 4.1 shows the list of domain concepts identified in this step and the statements associated with each of them.

```
1: #include <stdio.h>
2: #define YES 1
3: #define NO  2
4: void main()
5: {
6:     int nl = 0;
7:     int nw = 0;
8:     int nc = 0;
9:     int inword = NO;
10:    int c = getchar();
11:    while (c!=EOF)
12:    {
13:        char ch = (char) c;
14:        nc = nc + 1;
15:        if (ch=='\n')
16:             nl = nl + 1;
17:        if (ch==' ' || ch=='\n' || ch=='\t')
18:             inword = NO;
19:        else if (inword == NO)
20:        {
21:             inword = YES;
22:            nw = nw + 1;
           }
23:        c = getchar();
       }
24:    printf("%d \n", nl);
25:    printf("%d \n", nw);
26:    printf("%d \n", nc);
   }
```

Figure 4.1: The Line Count Program

Table 4.1: The Domain Concepts

Domain Concepts	**Statements**
Lines	6, **16**, 24
Words	7, **22**, 25
Chars	8, **14**, 26

In addition to the domain knowledge used by the software engineer to collect the significant variable that are believed to be associated with a specific domain concept, other semi-automated techniques can be also used. These include data mining, cohesion metrics, and data usage analysis [42].

4.2.2 Computation of Concept Slices

In this step we apply concept slicing as introduced in [40]. A concept slice is a slice of the program with respect to the domain concept. It is computed by taking slices of the program with respect to each of the statements belonging to a domain concept and then taking union of the slices. The outcome of this step adds more statements to the domain concepts and makes the domain concept executable. The concept slices corresponding to the domain concepts are candidates for possible modules.

```
4: void main()
5: {
6:     int nl = 0;
10:    int c = getchar();
11:    while (c!=EOF)
12:    {
13:        char ch = (char) c;
15:        if (ch=='\n')
16:            nl = nl + 1;
23:        c = getchar();
       }
24:    printf("%d \n", nl);
   }
```

Figure 4.2: Slice on the *Lines* Concept

```
4: void main()
5: {
7:     int nw = 0;
9:     int inword = NO;
10:    int c = getchar();
11:    while (c!=EOF)
12:    {
13:        char ch = (char) c;
17:        if (ch==' ' || ch=='\n' || ch=='\t')
18:            inword = NO;
19:        else if (inword == NO)
20:         {
21:            inword = YES;
22:            nw = nw + 1;
            }
23:        c = getchar();
       }
25:    printf("%d \n", nw);
  }
```

Figure 4.3: Slice on the *Words* Concept

```
4: void main()
5: {
8:     int nc = 0;
10:    int c = getchar();
11:    while (c!=EOF)
12:    {
14:        nc = nc + 1;
23:        c = getchar();
       }
26:    printf("%d \n", nc);
   }
```

Figure 4.4: Slice on the *Chars* Concept

Table 4.2: The Concept Slices

Domain Concepts	**Statements**
Lines	4, 5, 6, 10, 11, 12, 13, 15, **16**, 23, 24
Words	4, 5, 7, 9, 10, 11, 12, 13, 17, 18, 19, 20, 21, **22**, 23, 25
Chars	4, 5, 8, 10, 11, 12, **14**, 23, 26

Figures 4.2, 4.3 and 4.4 illustrate the concept slices corresponding to the domain concepts *Lines*, *Words* and *Chars*. Table 4.2 shows the refined statement list associated with the domain concepts after performing this step.

4.2.3 Building the Lattice

The third step is building the lattice of concept slices. This is done by performing a formal concept analysis on the concept slices. The context is formed by the relationship between the domain concepts and the program statements. The domain concepts make up the objects (extents) and statements make up the attributes (intents) of the context. A domain concept is considered to have a relationship with a statement if the statement is member of the concept slice for that domain concept. Figure 4.5 shows the context for the analysis. Performing a formal concept analysis on this context we obtain the lattice in Figure 4.6

	Lines	Words	Chars
4	X	X	X
5	X	X	X
6	X		
7		X	
8			X
9		X	
10	X	X	X
11	X	X	X
12	X	X	X
13	X	X	
14			X
15	X		
16	X		
17		X	
18		X	
19		X	
20		X	
21		X	
22		X	
23	X	X	X
24	X		
25		X	
26			X

Figure 4.5: The Context for the Lattice

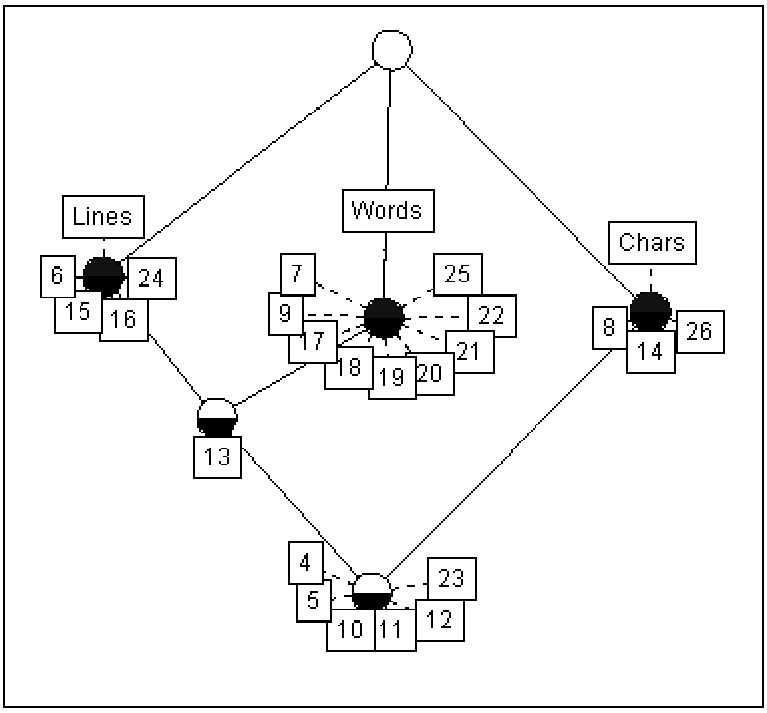

Figure 4.6: The Lattice of Concept Slices

The nodes of the lattice are called concepts, some of which correspond to the domain concepts. In the lattice each domain concept is related to all the statements in its node and all the nodes below it, whereas each statement is related to all the domain concepts in its node and all the nodes above it.

The lattice consists of 6 nodes, 3 of which are the domain concepts. The domain concept *Lines* consists of statements 6, 15, 16, 24, 13, 4, 5, 10, 11, 12 and 23. The statement 13 belongs to both *Lines* and *Words* domain concepts.

An observation from the lattice is that the flow in the lattice structure does not correspond to the control flow or data flow in the program. Rather the statements are shuffled up and down among the nodes. This lattice serves as an input to the modularization algorithms that is discussed in more detail in the following section.

4.3 Modularization Algorithms

Once the concept lattice has been built, the next step of the process is to perform modularization based on it. We present two different algorithms to do so. The first one is a lattice clustering algorithm that aims at keeping the modules as separate from each other as possible and only merge them together if there is any side effects identified. The second one targets the elimination of code duplication and organizes the program in a module/sub-module hierarchy.

4.3.1 Lattice Clustering Algorithm

A node of the lattice is called a critical node if any statement in the node contains a Principal Variable [40]. A Variable *V* in a set of statements *S* is a Principal Variable iff *V* is global or call-by-reference and is assigned in *S*. The existence of a critical node makes all other nodes above this node in the lattice interfering with each other. Furthermore, if the concept slices corresponding to the domain concepts above a critical node are decomposed as separate modules then they might have side-effect with each other since one module will change the internal state of the program and it might cause the other module to work possibly with incorrect data.

In this step we run the clustering algorithm presented below on the lattice to group domain concepts together to form a non-interfering set of modules. Figure 4.7 presents the algorithm and Figure 4.8 presents clustering examples as the algorithm is applied to the lattices.

If we apply the clustering algorithm on the lattice of Figure 4.8(b), in step 1.1 there will be three clusters identified for the three domain concept nodes – A{4}, B{5} and C{3}. Then in step 1.2 due to the presence of critical node 2, there will be a new cluster

containing nodes above his node – D{4, 5}. Then step 2 will cause clusters A, B and D to merge into a single cluster, so the clusters at this stage are – C{3} and D{4, 5}. Step 3 dictates that there will be two modules from this lattice fist containing the statements of the domain concept corresponding to node 3 and second containing the union of the statements of the domain concepts corresponding to nodes 4 and 5.

Lattice Clustering Algorithm

Input: Lattice of concepts
Output: A collection of modules

Step 1. In the bottom-up traversal of the lattice
  1.1. If this node is a domain concept create a new cluster containing this domain concept
  1.2. If this node is a critical node create a new cluster containing all the domain concepts in the sub-lattice starting from and above this node.
Step 2. Merge clusters with non-empty intersection
Step 3. Create a separate module for each of the clusters consisting of the union of the statements corresponding to the concept slices of the domain concepts inside each cluster.

Figure 4.7: Clustering Algorithm

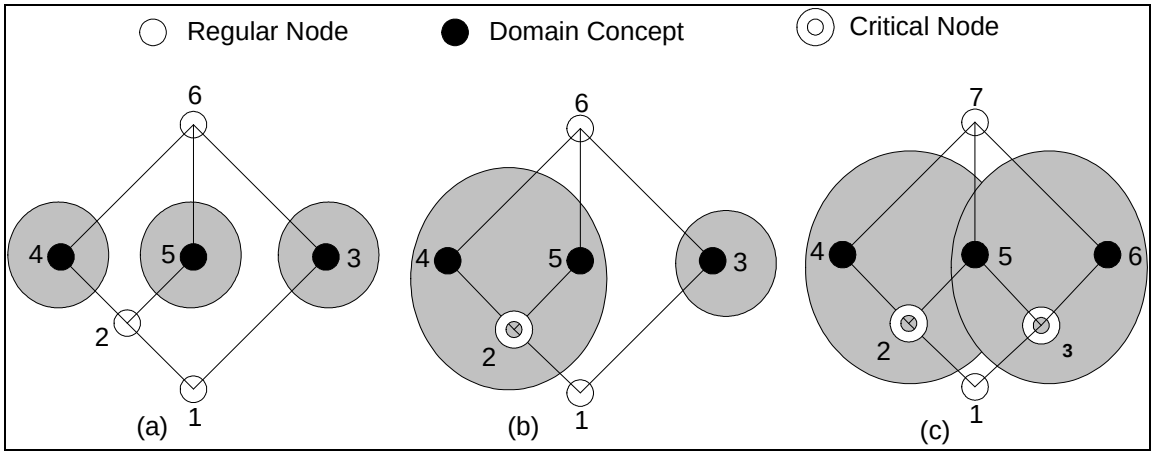

Figure 4.8: Lattice Clustering

In our line count example there is no critical node present in the lattice structure. Hence we have three clusters corresponding to three domain concepts. Resulting three modules, one for each *Lines, Words* and *Chars* concepts, consisting of the statements in their corresponding concept slice given in Figure 4.2, 4.3 and 4.4.

4.3.2 Lattice Restructuring Algorithm

The modularization we have achieved in the previous section contains some duplication of code among the modules. This is mostly because of the control flow they share while computing the domain concepts. Also if the computation of one domain concept depends on the result of another domain concept, all the code from the second domain concept will be duplicated in the module for the first one.

Since the statements of a node of the lattice belongs to all the domain concepts above it, the shared control flow and the shared computations among the modules will appear at the lower part of the lattice. Domain concept specific computations will reside in their corresponding nodes. But since statements in the nodes are not necessarily consecutive, rather shuffled up and down among the nodes of the lattice, the statements in a node may not be self-contained in a control flow.

We attempt to eliminate the code duplication by restructuring the lattice is a way such that the statements in the nodes will have a complete control flow and the lattice structure will correspond to a module sub-module hierarchy.

The restructuring is done by pushing down the statements from the nodes that disrupt the control flow. Since only the domain concepts of the nodes above the current node depend on the statements of the current node, pushing a statement down to the node below the current node still keeps it as part of those domain concepts. The pushed down

statement now becomes part of other domain concepts above the new location of the statement. This does not affect the computation of these other domain concepts since the other domain concepts were never dependent on the pushed down statement. It is to be noted that in worst case all the statements from all the nodes will be pushed down to the bottom node of the lattice and there will be no modularization at all. Figure 4.9 presents the restructuring algorithm.

Lattice Restructuring Algorithm

Input: Lattice of concepts
Output: A collection of modules

Step 1. In the top-down traversal of the lattice
- **1.1.** If there is a sub-lattice starting and below this node, push all the statements down from this node to the bottom node of the sub-lattice
- **1.2.** Identify groups of statements in this node such that the statements of the group and statements of the nodes above this node that have a self-contained and complete control flow
- **1.3.** If this node is a domain concept, and consecutive statements are not found, identify each key statement as a group
- **1.4.** Push down the other statements in this node to the nodes right below it
- **1.5.** If there are multiple groups in this node, split them into separate nodes
- **1.6.** Create a module from each statement group in this node. Declare all the variables used and defined in a module as parameters with variables being defined as call-by-reference and being used as call-by-value

Step 2. In the bottom-up traversal of the lattice
- **2.1.** From the module of this node call modules above this node from appropriate location.

Figure 4.9: Lattice Restructuring algorithm

If we apply the restructuring algorithm on the lattice of Figure 4.6, consecutive statements 15 and 16 at the node for *Lines* concept are complete in control flow. So they form a group and statements 6 and 24 are pushed down in the node below. Similarly consecutive statements 17, 18, 19, 20, 21 and 22 at the node for *Words* concept are complete in control flow. So they form a group and statements 7, 9 and 25 are pushed down. The node for *Chars* concept retains statement 14 since this is the key statement for this concept and statements 8 and 26 are pushed down. Figure 4.10 shows the final structure of the lattice.

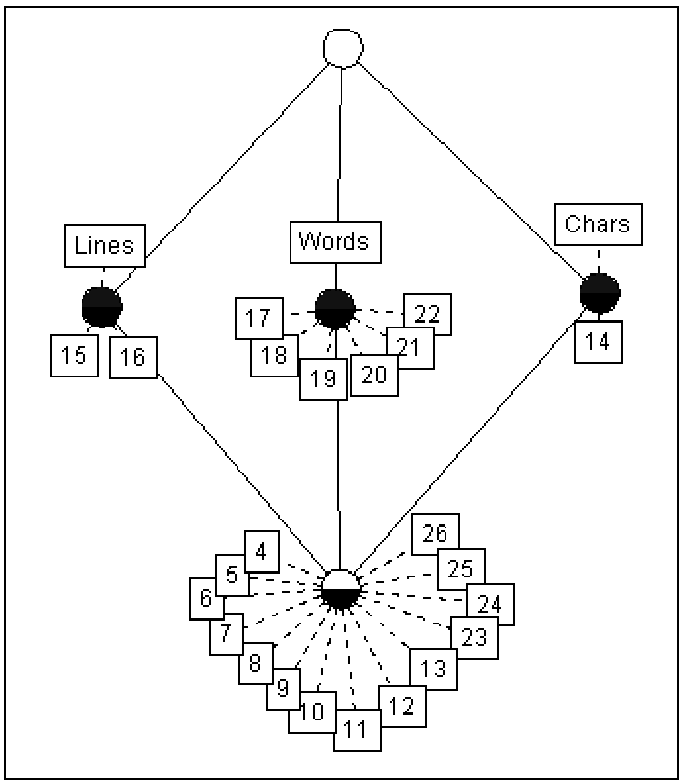

Figure 4.10: Restructured Lattice

One module will be created for each node in the lattice consisting of the statements in each node. All the variables defined in each of the modules will be declared as parameters called-by-reference and the variables used in the modules will be declared as parameters

called-by-value. Statements in the node at the bottom of the lattice will form the main controller module, which will call the other modules that actually computes the domain concepts. Figure 4.11 shows the line count program after the modularization is performed.

```
#include <stdio.h>
#define YES 1
#define NO  2

void chars(int &nc)
{
    nc = nc +1;
}
void lines(int &nl, char ch)
{
    if (ch=='\n')
        nl = nl + 1;
}
void words(int &nw, char ch, int &inword)
{
    if (ch==' ' || ch=='\n' || ch=='\t')
        inword = NO;
    else if (inword == NO)
    {
        inword = YES;
        nw = nw + 1;
    }
}
void main()
{
    int nl = 0;
    int nw = 0;
    int nc = 0;
    int inword = NO;
    int c = getchar();
    while (c != EOF)
    {
        char ch = (char) c;
        chars(nc);
        lines(nl, ch);
        words(nw, ch, inword);
        c = getchar();
    }
    printf("%d \n", nl);
    printf("%d \n", nw);
    printf("%d \n", nc);
}
```

Figure 4.11: The Modularized Line Count Program

Chapter 5

Prototype Tool

In this chapter we present a prototype implementation of the proposed extended ISME framework for program representation and analysis. The prototype is based on the architecture of the framework as shown in Figure 3.2 in order to facilitate language neutral program analysis and source code modularization tasks. The prototype was built using the Java programming language (JDK 1.3) to make it portable in a variety of operating platforms like Windows and different flavors of Unix. It also provided the benefit of readily available supporting technologies and API for XML parser, XPath expression, XSLT tools etc. In the following sections we present in details the various components of the tool.

5.1 Representation Transformers

These tools take the language specific representations of ISME and prepare language independent representations. One set of tools is required for each programming language. Our prototype tool works on JavaML, the AST representation for programming language Java, and generates intermediate representations for Java programs.

5.1.1 Positioning

The first step is to prepare the JavaML representation for further analysis. All the *switch* statements are transformed into equivalent *if-else* statements and all the *for* and *do-while* statements are transformed into equivalent *while* statements. So now the program will have only one type of branching and one type of looping statements making further analysis simpler.

The next step is to include an additional *lineNumber* tag in the JavaML representation that will indicate the starting of a program statement in the JavaML. This will enable us to identify which part of the JavaML belongs to which statement.

5.1.2 The Fact Extractor

The fact extractor works on the modified JavaML file. The file is parsed to extract information about statements, data types, variables, uses and defs of variables and function calls and the result is stored in a XML file that conforms the DTD given at Figure 4.3

All the variables declared, for example, in the JavaML can be found at the elements given by the XPath expression *//VariableDeclaratorId[@Identifier]*. The values of the attributes *Identifier* at these nodes give the names of the variables. The line number of the declaration can be found from the value of the *lineNumber* attribute found at the nearest node while traversing the XML tree upward. If the name of the parent node of the variable node is *FormalParameter* then it is a parameter declaration, if the name of the parent of the parent node is *LocalVariableDeclaration* then it is a local variable otherwise the variable is a field declaration.

The uses of variables are found from the identifiers in expressions that are not on the left hand side of an assignment statement. More specifically the XPath expression for finding the uses are *//PrimaryExpression/*/*[@Identifier]/following-sibling::*[1][name()!='AssignmentOperator']*

The defs of variables are found from different syntaxes. Identifiers on the left side of assignment statements are defs. A method call on an object type variable can modify the object and hence can be a def. Similarly an object passed as a parameter to a method call can also modify the object and can be a def for that object type variable. One approach to is to take the conservative approach and treat all the possible defs as actual defs. Instead we took a evidence based approach and ask the programmer interactively to confirm the defs of variables.

Function or method calls can be found at identifiers in expressions that can take arguments. The XPath expression is *//PrimaryExpression/*/*[@Identifier]/following-sibling::*[1][name()='Argument']*

5.1.3 The PDG Generator

The extracted facts along with the modified JavaML are the source for generating the PDG. This is an intra-procedural representation and hence works on a specific method of the program. This tool assumes *if-else* to be the only type of branching and *while* to be the only type of looping in the JavaML and there is no *goto* type transfer of control or *continue* or *break* statements.

First a control dependence sub-graph is created from the modified JavaML parse tree. Then the data dependence sub-graph is added from the uses and defines information from

the facts. The field members used in the methods comprised the *InitialDef* nodes and *FinalUse* nodes are created for all variables used or defined in the method.

5.2 Analysis Tools

Following tools are developed to facilitate generic analysis based on the intermediate representations. Given the intermediate representations generated from source code of any programming language as XML documents conforming the proposed DTDs, these tools will perform the required analysis uniformly.

5.2.1 Concept Identifier

Concept identifier works on the facts to identify concepts from hints such as

- Uses/defs of a particular variable
- Uses/defs of all the variables with their names matching a pattern
- Uses/defs of all the variables of a particular type
- Uses/defs of all the variables of all the types with their names matching a pattern
- Calls to a particular method
- Calls to the methods with their names matching a pattern

5.2.2 PDG Slicer

The PDG slicing tool works on the PDG based on the algorithm given in [8]. The algorithm assumes *if-else* to be the only type of branching and *while* to be the only type of looping statements and there is no *goto* statement for transfer of control or *continue* or *break* statements. It has the limitation of loosing the *catch* and *finally* blocks when slicing on a statement inside their corresponding *try* block for a java program. The output of the

slicer is a reduced PDG. The statements present in the resulting PDG will comprise the program Slice. The slicer can performs following kids of slicing

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

5.2.3 Formal Concept Analyzer

The ConExp tool [38] can be used on top of the framework to perform formal concept analysis on the contexts built from the facts or other intermediate representations.

5.3 Examples and Tool Operation Statistics

In this section we present some sample output generated by the prototype tool and evaluate proposed representations in terms of size and time. The output samples presented are generated from the simple Java program named *MyMath.Java* presented in Figure 5.1. The program consists of a single class *MyMath* and a single method *factorial*. The *factorial* method implements the algorithm for calculating the factorial of a number. Appendix A shows the JavaML file produced from this program after positioning. Figure 5.2 gives a pretty print of the positioned program.

Four other input files of different sizes were used to measure the size and time parameters. These files were chosen from a variety of sources ranging from student course projects to standard utility library. All the experiments were run in a Sun UltraSPARC III 440 MHz station with 512 MB RAM and running Solaris 8 operating System.

```
public class MyMath
{
   public int factorial (int n)
   {
       int i = 1;
       int f = 1;
       while (i<=n)
       {
              f = f * i;
              i = i + 1;
       }
       return f;
   }
}
```

Figure 5.1: A Simple Java Program MyMath.java

```
<1>
<2> public class MyMath
<3> {
<4> public int factorial(
<5> int n )
<6> {
<7> int i = 1;
<8> int f = 1;
<9> while (i <= n)
<10> {
<11> f = f * i;
<12> i = i + 1;
}
<13> return f;
}
}
```

Figure 5.2: Pretty Print of the Positioned MyMath.java Program

5.3.1 The Facts

Figures 5.3 shows the extracted facts *Facts.xml* from *MyMath,java*. Table 5.1 summarizes the results collected from fact extraction from all the input programs. It shows the relationships between the size of the source code and the size of the JavaML and FactML files and also the time to generate them. As illustrated in the graphs of Figure5.4, 5.5 and 5.6 the size of the ML files increases with the size of the source code as expected. Also the time required to generate the FactML file depends on both the size of the source code and the size of the FactML. One important observation from the graphs is that the time

required to generate the FactML grows at a faster rate than the time required to generate the JavaML, whereas the size of the FactML grows at a much slower rate than the size of the JavaML. This can be explained by the fact that for each statement in the source code the generation its corresponding JavaML requires constant time, making the JavaML generation a task of O(n) complexity. Whereas for the generation of the FactML each program construct needs to be examined with the rest of the constructs for a relationship between them, making it a task of $O(n^2)$ complexity.

```
<Facts program="MyMath.Java">
  <Statements>
    <Statement id="FS1" lineno=1 tag="CompilationUnit"/>
    <Statement id="FS2" lineno=2 tag="TypeDeclaration"/>
    <Statement id="FS3" lineno=3 tag="ClassBody"/>
    <Statement id="FS4" lineno=4 tag="MethodDeclaration" function="FF1"/>
    <Statement id="FS5" lineno=5 tag="FormalParameter" function="FF1"/>
    <Statement id="FS6" lineno=6 tag="Block" function="FF1"/>
    <Statement id="FS7" lineno=7 tag="LocalVariableDeclaration" function="FF1"/>
    <Statement id="FS8" lineno=8 tag="LocalVariableDeclaration" function="FF1"/>
    <Statement id="FS9" lineno=9 tag="WhileStatement" function="FF1"/>
    <Statement id="FS10" lineno=10 tag="Block" function="FF1"/>
    <Statement id="FS11" lineno=11 tag="StatementExpression" function="FF1"/>
    <Statement id="FS12" lineno=12 tag="StatementExpression" function="FF1"/>
    <Statement id="FS13" lineno=13 tag="ReturnStatement" function="FF1"/>
  </Statements>
  <Types>
    <Type id="FT1" name="int" category="Primitive"/>
  </Types>
  <Variables>
    <Variable id="FV1" name="n" scope="Parameter" declared="FS5" type="FT1"/>
    <Variable id="FV2" name="i" scope="Local" declared="FS7" type="FT1"/>
    <Variable id="FV3" name="f" scope="Local" declared="FS8" type="FT1"/>
  </Variables>
  <UseDefs>
    <UseDef id="FUD1" category="Use" statement="FS9" variable="FV2"/>
    <UseDef id="FUD2" category="Use" statement="FS9" variable="FV1"/>
    <UseDef id="FUD3" category="Use" statement="FS11" variable="FV3"/>
    <UseDef id="FUD4" category="Use" statement="FS11" variable="FV2"/>
    <UseDef id="FUD5" category="Use" statement="FS12" variable="FV2"/>
    <UseDef id="FUD6" category="Def" statement="FS5" variable="FV1"/>
    <UseDef id="FUD7" category="Def" statement="FS7" variable="FV2"/>
    <UseDef id="FUD8" category="Def" statement="FS8" variable="FV3"/>
    <UseDef id="FUD9" category="Def" statement="FS11" variable="FV3"/>
    <UseDef id="FUD10" category="Def" statement="FS12" variable="FV2"/>
```

```
  </UseDefs>
  <Functions>
    <Function id="FF1" name="factorial" scope="Internal"
              signature="MyMath: int factorial (int) " declared="FS4"/>
  </Functions>
</Facts>
```

Figure 5.3: Extracted Facts from MyMath.java Program

Table 5.1: Experimental Results for Fact Extraction

Program	Source Size (bytes)	JavaML Size (bytes)	FactML Size (bytes)	JavaML Time (ms)	FactML Time (ms)
MyMath.Java	187	19,438	2,030	2,016	224
Voter.java	3,822	322,694	17,502	3,259	1,487
GUI.java	4,994	358,641	20,697	3,270	1,498
UnboundedLife.java	10,831	688,671	33,800	3,783	3,849
PDG.java	22,200	1,134,171	81,072	4,575	11,403

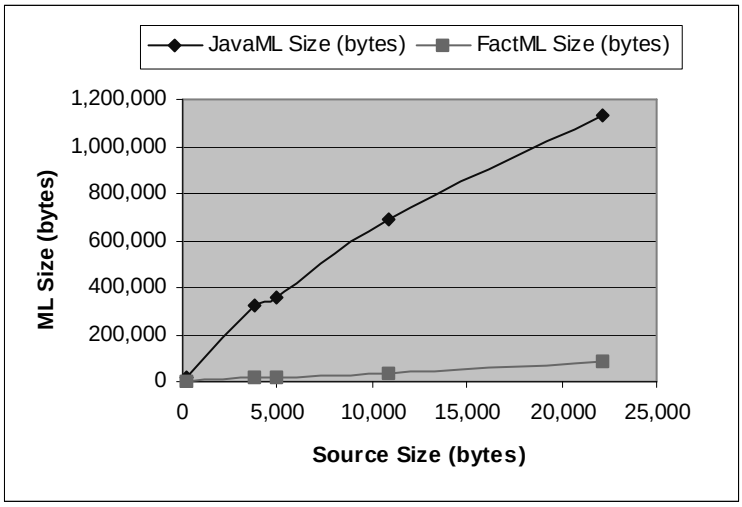

Figure 5.4: Plot of Source Code Size vs. the JavaML and FactML Sizes

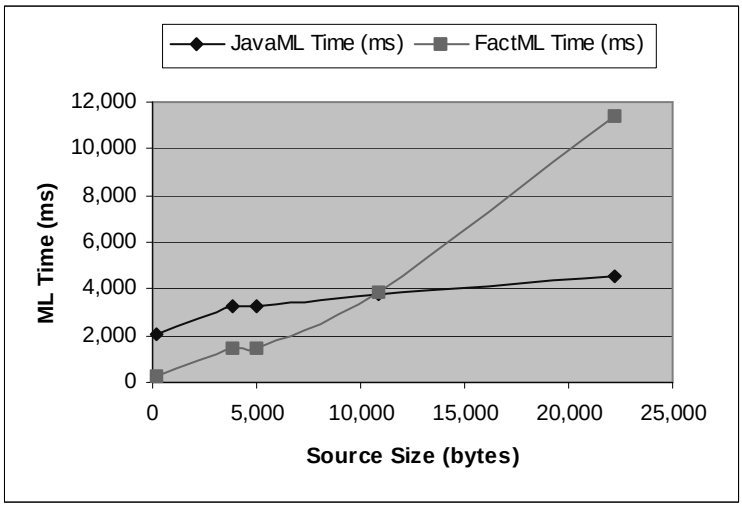

Figure 5.5: Plot of Source Code Size vs. the JavaML and FactML Creation Time

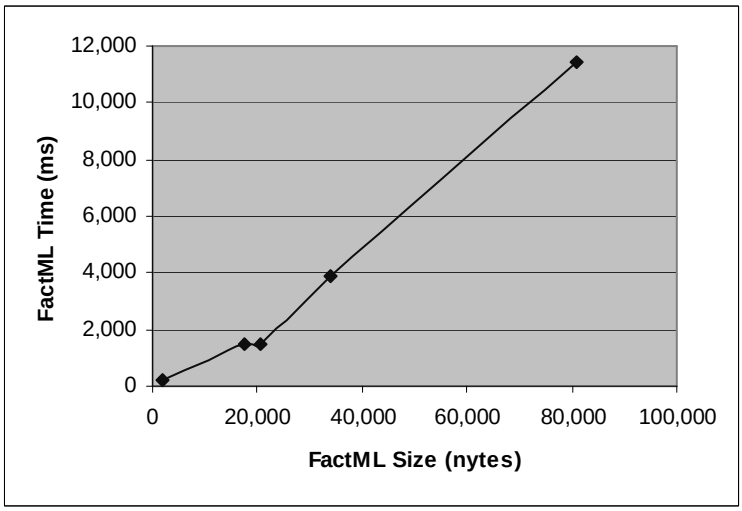

Figure 5.6: Plot of the FactML Size vs. the FactML Creation Time

5.3.2 The PDG

The program dependence graph generated for the *factorial* method of *MyMath* class is shown in Figure 5.7. Table 5.2 summarizes the relationship between the size of a method and the size of its corresponding PDGML and the time to produce it. The graph in Figure 5.8 indicates that even though the general tendency of the size of the PDGML is to

increase with the size of the method, it may not be the case always, specially if there is a low number of def-use chaining in the program.

```
<PDG xmlns:xlink="http://www.w3.org/1999/xlink"
 program="MyMath.Java" scope="int factorial (int)">

  <Items>
    <Variables>
      <Variable id="PV1" xlink:href="Facts.xml#FV1"/>
      <Variable id="PV2" xlink:href="Facts.xml#FV2"/>
      <Variable id="PV3" xlink:href="Facts.xml#FV3"/>
    </Variables>
    <Nodes>
      <Node>
        <Special id="PDN1" type="Entry"/>
      </Node>
      <Node>
        <Special id="PDN2" type="FinalUse" variable="PV1"/>
      </Node>
      <Node>
        <Special id="PDN3" type="FinalUse" variable="PV2"/>
      </Node>
      <Node>
        <Special id="PDN4" type="FinalUse" variable="PV3"/>
      </Node>
      <Node>
        <Statement id="PS1" xlink:href="Facts.xml#FS4"/>
      </Node>
      <Node>
        <Statement id="PS2" xlink:href="Facts.xml#FS5"/>
      </Node>
      <Node>
        <Statement id="PS3" xlink:href="Facts.xml#FS6"/>
      </Node>
      <Node>
        <Statement id="PS4" xlink:href="Facts.xml#FS7"/>
      </Node>
      <Node>
        <Statement id="PS5" xlink:href="Facts.xml#FS8"/>
      </Node>
      <Node>
        <Statement id="PS6" xlink:href="Facts.xml#FS9"/>
      </Node>
      <Node>
        <Statement id="PS7" xlink:href="Facts.xml#FS10"/>
      </Node>
      <Node>
        <Statement id="PS8" xlink:href="Facts.xml#FS11"/>
      </Node>
```

```
      <Node>
        <Statement id="PS9" xlink:href="Facts.xml#FS12"/>
      </Node>
      <Node>
        <Statement id="PS10" xlink:href="Facts.xml#FS13"/>
      </Node>
    </Nodes>
  </Items>
  <Dependencies>
    <ControlFLows>
      <ControlFlow id="PCF1" from="PDN1" to="PS1" label="True"/>
      <ControlFlow id="PCF2" from="PDN1" to="PDN2" label="True"/>
      <ControlFlow id="PCF3" from="PDN1" to="PDN3" label="True"/>
      <ControlFlow id="PCF4" from="PDN1" to="PDN4" label="True"/>
      <ControlFlow id="PCF5" from="PS1" to="PS2" label="True"/>
      <ControlFlow id="PCF6" from="PS1" to="PS3" label="True"/>
      <ControlFlow id="PCF7" from="PS3" to="PS4" label="True"/>
      <ControlFlow id="PCF8" from="PS3" to="PS5" label="True"/>
      <ControlFlow id="PCF9" from="PS3" to="PS6" label="True"/>
      <ControlFlow id="PCF10" from="PS3" to="PS10" label="True"/>
      <ControlFlow id="PCF11" from="PS6" to="PS7" label="True"/>
      <ControlFlow id="PCF12" from="PS7" to="PS8" label="True"/>
      <ControlFlow id="PCF13" from="PS7" to="PS9" label="True"/>
    </ControlFLows>
    <DataFLows>
      <DataFlow id="PDF1" from="PS2" to="PS6" over="PV1"/>
      <DataFlow id="PDF2" from="PS2" to="PDN2" over="PV1"/>
      <DataFlow id="PDF3" from="PS4" to="PS6" over="PV2"/>
      <DataFlow id="PDF4" from="PS4" to="PS8" over="PV2"/>
      <DataFlow id="PDF5" from="PS4" to="PS9" over="PV2"/>
      <DataFlow id="PDF6" from="PS4" to="PDN3" over="PV2"/>
      <DataFlow id="PDF7" from="PS5" to="PS8" over="PV3"/>
      <DataFlow id="PDF8" from="PS5" to="PS10" over="PV3"/>
      <DataFlow id="PDF9" from="PS5" to="PDN4" over="PV3"/>
      <DataFlow id="PDF10" from="PS8" to="PS8" over="PV3"/>
      <DataFlow id="PDF11" from="PS8" to="PS10" over="PV3"/>
      <DataFlow id="PDF12" from="PS8" to="PDN4" over="PV3"/>
      <DataFlow id="PDF13" from="PS9" to="PS6" over="PV2"/>
      <DataFlow id="PDF14" from="PS9" to="PS8" over="PV2"/>
      <DataFlow id="PDF15" from="PS9" to="PS9" over="PV2"/>
      <DataFlow id="PDF16" from="PS9" to="PDN3" over="PV2"/>
    </DataFLows>
  </Dependencies>
</PDG>
```

Figure 5.7: The PDG for the factorial Class of MyMath.java Program

Table 5.2: Experimental Results for PDG Creation

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

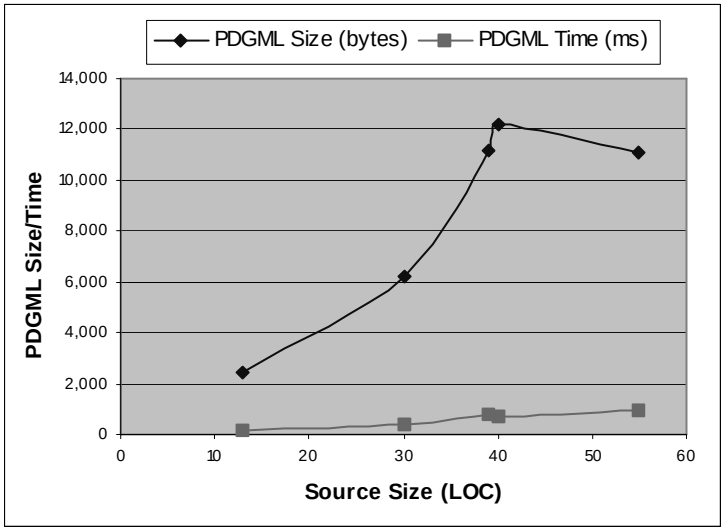

Figure 5.8: Plot of Source Code Size vs. the PDGML Size and creation Time

5.3.3 Slicing

Figure 5.9 shows the sliced PDG of the *factorial* method of *MyMath*, class based on the final use of the variable *i*. Table 5.3 and the graph in Figure 5.10 summarize the results of slicing based on the final uses of the variables given in the table.

```
<PDG id="PDG" xmlns:xlink="http://www.w3.org/1999/xlink"
 program="MyMath.Java" scope="int factorial (int)">
  <Items>
    <Variables>
      <Variable id="PV1" xlink:href="Facts.xml#FV1"/>
      <Variable id="PV2" xlink:href="Facts.xml#FV2"/>
      <Variable id="PV3" xlink:href="Facts.xml#FV3"/>
    </Variables>
    <Nodes>
      <Node>
        <Special id="PDN1" type="Entry"/>
      </Node>
      <Node>
        <Special id="PDN3" type="FinalUse" variable="PV2"/>
      </Node>
      <Node>
        <Statement id="PS1" xlink:href="Facts.xml#FS4"/>
      </Node>
      <Node>
        <Statement id="PS2" xlink:href="Facts.xml#FS5"/>
      </Node>
      <Node>
        <Statement id="PS3" xlink:href="Facts.xml#FS6"/>
      </Node>
      <Node>
        <Statement id="PS4" xlink:href="Facts.xml#FS7"/>
      </Node>
      <Node>
        <Statement id="PS6" xlink:href="Facts.xml#FS9"/>
      </Node>
      <Node>
        <Statement id="PS7" xlink:href="Facts.xml#FS10"/>
      </Node>
      <Node>
        <Statement id="PS9" xlink:href="Facts.xml#FS12"/>
      </Node>
    </Nodes>
  </Items>
  <Dependencies>
    <ControlFLows>
      <ControlFlow id="PCF1" from="PDN1" to="PS1" label="True"/>
      <ControlFlow id="PCF3" from="PDN1" to="PDN3" label="True"/>
      <ControlFlow id="PCF5" from="PS1" to="PS2" label="True"/>
      <ControlFlow id="PCF6" from="PS1" to="PS3" label="True"/>
      <ControlFlow id="PCF7" from="PS3" to="PS4" label="True"/>
      <ControlFlow id="PCF9" from="PS3" to="PS6" label="True"/>
      <ControlFlow id="PCF11" from="PS6" to="PS7" label="True"/>
      <ControlFlow id="PCF13" from="PS7" to="PS9" label="True"/>
    </ControlFLows>
    <DataFLows>
```

```
      <DataFlow id="PDF1" from="PS2" to="PS6" over="PV1"/>
      <DataFlow id="PDF3" from="PS4" to="PS6" over="PV2"/>
      <DataFlow id="PDF5" from="PS4" to="PS9" over="PV2"/>
      <DataFlow id="PDF6" from="PS4" to="PDN3" over="PV2"/>
      <DataFlow id="PDF13" from="PS9" to="PS6" over="PV2"/>
      <DataFlow id="PDF15" from="PS9" to="PS9" over="PV2"/>
      <DataFlow id="PDF16" from="PS9" to="PDN3" over="PV2"/>
    </DataFLows>
  </Dependencies>
</PDG>
```

Figure 5.9: Slice of the factorial method for Final Use of i from MyMath.java Program

Table 5.3: Experimental Results for Slicing

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

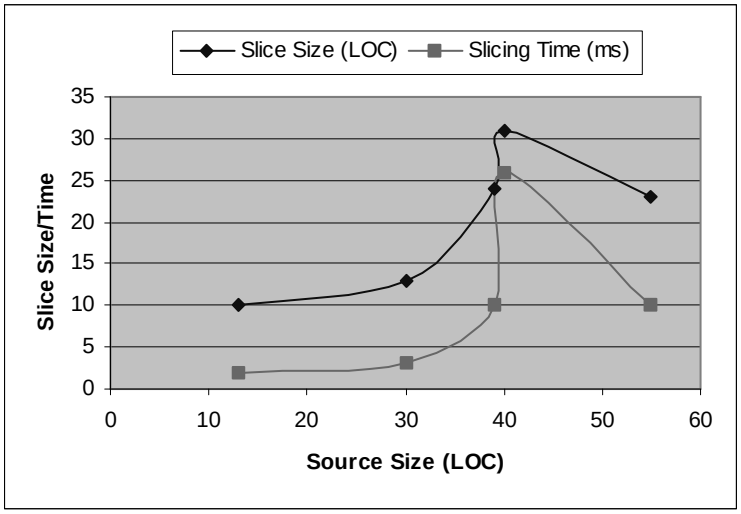

Figure 5.10: Plot of Source Code Size vs. the Slice Size and Creation Time

Chapter 6

Conclusion

6.1 Summary

The work presented in this thesis presents a program representation framework, a source code modularization technique and a toolset to facilitate such analysis.

The modularization aims to identify stand-alone code fragments that can be implemented as modules that deliver cohesive domain functionality. The algorithms are applied on a lattice based program representation formalism that is built by combining concept assignment, concept analysis, and slicing. The lattice representation models the contributions of the individual statements of a program towards the computation of different domain concepts implemented by it. The lattice structure serves as a primary data structure for program modularization. The algorithms to discover modules and sub-modules are applied on the concept lattice by clustering and restructuring it so that nodes in the lattice contain consecutive statements that form complete control flows.

In order to facilitate the program analysis such as source code modularization we also developed a language independent program representation framework to represent program facts at different levels of abstraction in XML-sublanguages. This makes the

representations portable and facilitates the development of generic analysis tools on top of the framework.

6.2 Thesis Contributions

The main contribution of the thesis can be summarized as:

- Development of a framework for source code neutral XML-based program representations. The framework consists of a multi-layered program representation formalisms and representation transformer tools. At the lowest level of abstraction is the source code itself. The second level is the XML-sublanguages for the syntax trees The third level is the XML-sublanguages for language neutral representations for program facts, flow graphs, dependence graphs, call graphs etc. that can be extracted from the syntax tree. In between each pair of representation layers are the transformer tools that generates the representations at the higher level of abstraction from the level below it.
- Implementation of generic program analysis toolset, a concept identifier and a program slicer, on top of the representation framework.
- Introduction to a new program representation formalism that we call the Lattice of Concept Slices. Specifically we combined concept assignment, program slicing and formal concept analysis in order to examine the relationships among the program statements and their contribution towards the computation the domain concepts implemented by them in the form of a lattice.
- Development of program modularization algorithms based on the Lattice of Concept Slices. Specifically we propose two modularization algorithms based on clustering and restructuring the lattice. The lattice clustering algorithm groups

nodes together to form a non-interfering set of modules. Whereas the lattice restructuring algorithm restructures the lattice by splitting nodes or pushing statements down the lattice in order to create a module sub-module hierarchy in the lattice.

6.3 Future Work

The proposed modularization process has been applied in small and medium sized programs. However, additional work is to be done in order to evaluate and apply the modularization process on larger programs with inter-procedural dependencies. One future direction is to use Inter-Procedural Dependency graphs, and incremental analysis so that larger program segments and groups of functions that relate to specific concepts can be individually analyzed. The proposed framework at the moment focuses on intra-procedural representations and analysis and needs to be extended in order to incorporate inter-procedural representations and analysis.

Another interesting future direction is to apply this modularization technique in migration of monolithically developed Servlet-based web applications into Model-View-Controller (MVC) pattern J2EE architecture. Early web-based multi-tier applications were developed using Java Servlet technology in a monolithic way by embedding all the business logic and presentation logic inside one Java program. The business logic in such programs computes the domain concepts implemented in the Servlet. Similarly, the presentation logic relates also special type of domain concepts that deal with the user interface. Decomposed modules from the domain concepts that correspond to the business logic can be migrated as JavaBeans/EJBs while modules that correspond to the presentation logic can be migrated into JavaServer Pages.

Appendix A

Positioned JavaML for the MyMath.java program

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CompilationUnit SYSTEM "javaml.dtd">
<CompilationUnit lineNumber="1" parentLine="0">
  <TypeDeclaration lineNumber="2" parentLine="1">
    <ClassDeclaration isAbstract="False" isFinal="False" isPublic="True">
      <UnmodifiedClassDeclaration Extends="False" Identifier="MyMath">
        <ClassBody lineNumber="3" parentLine="2">
          <MethodDeclaration isAbstract="False" isFinal="False" isNative="False" isPrivate="False"
           isProtected="False" isPublic="True" isStatic="False" isSynchronized="False" lineNumber="4"
           parentLine="3">
            <ResultType>
              <Type ArraySize="0">
                <PrimitiveType Type="int"/>
              </Type>
            </ResultType>
            <MethodDeclarator ArraySize="0" Identifier="factorial">
              <FormalParameter lineNumber="5" parentLine="4">
                <Type ArraySize="0">
                  <PrimitiveType Type="int"/>
                </Type>
                <VariableDeclaratorId ArraySize="0" Identifier="n"/>
              </FormalParameter>
            </MethodDeclarator>
            <Block lineNumber="6" parentLine="4">
              <LocalVariableDeclaration isFinal="False" lineNumber="7" parentLine="6">
                <Type ArraySize="0">
                  <PrimitiveType Type="int"/>
                </Type>
                <VariableDeclarator>
                  <VariableDeclaratorId ArraySize="0" Identifier="i"/>
                  <VariableInitializer>
                    <Expression>
                      <ConditionalExpression>
                        <ConditionalOrExpression>
                          <ConditionalAndExpression>
                            <InclusiveOrExpression>
                              <ExclusiveOrExpression>
                                <AndExpression>
                                  <EqualityExpression>
                                    <InstanceOfExpression>
                                      <RelationalExpression>
                                        <ShiftExpression>
                                          <AdditiveExpression>
                                            <MultiplicativeExpression>
                                              <UnaryExpression>
                                                <UnaryExpressionNotPlusMinus>
                                                  <PostfixExpression Type="None">
                                                    <PrimaryExpression>
                                                      <PrimaryPrefix isSuper="False" isThis="False">
                                                        <Literal Type="Integer" Value="1"/>
                                                      </PrimaryPrefix>
                                                    </PrimaryExpression>
                                                  </PostfixExpression>
                                                </UnaryExpressionNotPlusMinus>
                                              </UnaryExpression>
                                            </MultiplicativeExpression>
                                          </AdditiveExpression>
                                        </ShiftExpression>
                                      </RelationalExpression>
                                    </InstanceOfExpression>
                                  </EqualityExpression>
```

```
                  </AndExpression>
                </ExclusiveOrExpression>
              </InclusiveOrExpression>
            </ConditionalAndExpression>
          </ConditionalOrExpression>
        </ConditionalExpression>
      </Expression>
    </VariableInitializer>
  </VariableDeclarator>
</LocalVariableDeclaration>
<LocalVariableDeclaration isFinal="False" lineNumber="8" parentLine="6">
  <Type ArraySize="0">
    <PrimitiveType Type="int"/>
  </Type>
  <VariableDeclarator>
    <VariableDeclaratorId ArraySize="0" Identifier="f"/>
    <VariableInitializer>
      <Expression>
        <ConditionalExpression>
          <ConditionalOrExpression>
            <ConditionalAndExpression>
              <InclusiveOrExpression>
                <ExclusiveOrExpression>
                  <AndExpression>
                    <EqualityExpression>
                      <InstanceOfExpression>
                        <RelationalExpression>
                          <ShiftExpression>
                            <AdditiveExpression>
                              <MultiplicativeExpression>
                                <UnaryExpression>
                                  <UnaryExpressionNotPlusMinus>
                                    <PostfixExpression Type="None">
                                      <PrimaryExpression>
                                        <PrimaryPrefix isSuper="False" isThis="False">
                                          <Literal Type="Integer" Value="1"/>
                                        </PrimaryPrefix>
                                      </PrimaryExpression>
                                    </PostfixExpression>
                                  </UnaryExpressionNotPlusMinus>
                                </UnaryExpression>
                              </MultiplicativeExpression>
                            </AdditiveExpression>
                          </ShiftExpression>
                        </RelationalExpression>
                      </InstanceOfExpression>
                    </EqualityExpression>
                  </AndExpression>
                </ExclusiveOrExpression>
              </InclusiveOrExpression>
            </ConditionalAndExpression>
          </ConditionalOrExpression>
        </ConditionalExpression>
      </Expression>
    </VariableInitializer>
  </VariableDeclarator>
</LocalVariableDeclaration>
<WhileStatement lineNumber="9" parentLine="6">
  <Block lineNumber="10" parentLine="9">
    <StatementExpression Type="None" lineNumber="11" parentLine="10">
      <PrimaryExpression>
        <PrimaryPrefix isSuper="False" isThis="False">
          <Name Identifier="f"/>
        </PrimaryPrefix>
      </PrimaryExpression>
      <AssignmentOperator Type="Simple"/>
      <Expression>
        <ConditionalExpression>
```

```
        <ConditionalOrExpression>
          <ConditionalAndExpression>
            <InclusiveOrExpression>
              <ExclusiveOrExpression>
                <AndExpression>
                  <EqualityExpression>
                    <InstanceOfExpression>
                      <RelationalExpression>
                        <ShiftExpression>
                          <AdditiveExpression>
                            <MultiplicativeExpression>
                              <UnaryExpression>
                                <UnaryExpressionNotPlusMinus>
                                  <PostfixExpression Type="None">
                                    <PrimaryExpression>
                                      <PrimaryPrefix isSuper="False" isThis="False">
                                        <Name Identifier="f"/>
                                      </PrimaryPrefix>
                                    </PrimaryExpression>
                                  </PostfixExpression>
                                </UnaryExpressionNotPlusMinus>
                              </UnaryExpression>
                              <MultiplicativeOperator Type="Multiply"/>
                              <UnaryExpression>
                                <UnaryExpressionNotPlusMinus>
                                  <PostfixExpression Type="None">
                                    <PrimaryExpression>
                                      <PrimaryPrefix isSuper="False" isThis="False">
                                        <Name Identifier="i"/>
                                      </PrimaryPrefix>
                                    </PrimaryExpression>
                                  </PostfixExpression>
                                </UnaryExpressionNotPlusMinus>
                              </UnaryExpression>
                            </MultiplicativeExpression>
                          </AdditiveExpression>
                        </ShiftExpression>
                      </RelationalExpression>
                    </InstanceOfExpression>
                  </EqualityExpression>
                </AndExpression>
              </ExclusiveOrExpression>
            </InclusiveOrExpression>
          </ConditionalAndExpression>
        </ConditionalOrExpression>
      </ConditionalExpression>
    </Expression>
  </StatementExpression>
  <StatementExpression Type="None" lineNumber="12" parentLine="10">
    <PrimaryExpression>
      <PrimaryPrefix isSuper="False" isThis="False">
        <Name Identifier="i"/>
      </PrimaryPrefix>
    </PrimaryExpression>
    <AssignmentOperator Type="Simple"/>
    <Expression>
      <ConditionalExpression>
        <ConditionalOrExpression>
          <ConditionalAndExpression>
            <InclusiveOrExpression>
              <ExclusiveOrExpression>
                <AndExpression>
                  <EqualityExpression>
                    <InstanceOfExpression>
                      <RelationalExpression>
                        <ShiftExpression>
                          <AdditiveExpression>
                            <MultiplicativeExpression>
```

```
                                <UnaryExpression>
                                  <UnaryExpressionNotPlusMinus>
                                    <PostfixExpression Type="None">
                                      <PrimaryExpression>
                                        <PrimaryPrefix isSuper="False" isThis="False">
                                          <Name Identifier="i"/>
                                        </PrimaryPrefix>
                                      </PrimaryExpression>
                                    </PostfixExpression>
                                  </UnaryExpressionNotPlusMinus>
                                </UnaryExpression>
                              </MultiplicativeExpression>
                              <AdditiveOperator Type="Add"/>
                              <MultiplicativeExpression>
                                <UnaryExpression>
                                  <UnaryExpressionNotPlusMinus>
                                    <PostfixExpression Type="None">
                                      <PrimaryExpression>
                                        <PrimaryPrefix isSuper="False" isThis="False">
                                          <Literal Type="Integer" Value="1"/>
                                        </PrimaryPrefix>
                                      </PrimaryExpression>
                                    </PostfixExpression>
                                  </UnaryExpressionNotPlusMinus>
                                </UnaryExpression>
                              </MultiplicativeExpression>
                            </AdditiveExpression>
                          </ShiftExpression>
                        </RelationalExpression>
                      </InstanceOfExpression>
                    </EqualityExpression>
                  </AndExpression>
                </ExclusiveOrExpression>
              </InclusiveOrExpression>
            </ConditionalAndExpression>
          </ConditionalOrExpression>
        </ConditionalExpression>
      </Expression>
    </StatementExpression>
  </Block>
</WhileStatement>
<ReturnStatement lineNumber="13" parentLine="6">
  <Expression>
    <ConditionalExpression>
      <ConditionalOrExpression>
        <ConditionalAndExpression>
          <InclusiveOrExpression>
            <ExclusiveOrExpression>
              <AndExpression>
                <EqualityExpression>
                  <InstanceOfExpression>
                    <RelationalExpression>
                      <ShiftExpression>
                        <AdditiveExpression>
                          <MultiplicativeExpression>
                            <UnaryExpression>
                              <UnaryExpressionNotPlusMinus>
                                <PostfixExpression Type="None">
                                  <PrimaryExpression>
                                    <PrimaryPrefix isSuper="False" isThis="False">
                                      <Name Identifier="f"/>
                                    </PrimaryPrefix>
                                  </PrimaryExpression>
                                </PostfixExpression>
                              </UnaryExpressionNotPlusMinus>
                            </UnaryExpression>
                          </MultiplicativeExpression>
                        </AdditiveExpression>
```

```
                                    </ShiftExpression>
                                  </RelationalExpression>
                                </InstanceOfExpression>
                              </EqualityExpression>
                            </AndExpression>
                          </ExclusiveOrExpression>
                        </InclusiveOrExpression>
                      </ConditionalAndExpression>
                    </ConditionalOrExpression>
                  </ConditionalExpression>
                </Expression>
              </ReturnStatement>
            </Block>
          </MethodDeclaration>
        </ClassBody>
      </UnmodifiedClassDeclaration>
    </ClassDeclaration>
  </TypeDeclaration>
</CompilationUnit>
```

Bibliography

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

[2] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing Company. 1986.

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

[4] Francis. E. Allen and J. Cocke. A program data flow analysis procedure. Communications of the ACM, Volume 19 Issue 3. March 1976.

[5] J. R. Allen, Ken Kennedy, Carrie Porterfield and Joe Warren. Conversion of control dependence to data dependence. Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. January 1983

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

[7] Susan Horwitz and Thomas Reps. The Use of Program Dependence Graphs in Software Engineering. Proceedings of the 14th International Conference on Software Engineering. May 1992.

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

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

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

[11] Bell Canada. Datrix Abstract Semantic Graph Reference Manual. http://www.iro.umontreal.ca/labs/gelo/datrix/refmanuals/asgmodel-1.4.pdf

[12] M. Brand, P. Klint and P. Olivier. ATerms: Exchanging Data between Heterogeneous Tools for CASL, 1998. ftp://ftp.brics.dk/Projects/CoFI/Notes/T-3/doc.ps.Z

[13] J. Heering, P. R. H. Hendriks, P. Klint and J. Rekers The Syntax Definition Formalism SDF – Reference Manual. ACM SIGPLAN Notices, Volume 24 Issue 11. November 1989.

[14] Mark Brand, Tobias Kuipers, Leon Moonen and Pieter Olivier. Implementation of a Prototype for the New {ASF+SDF} Meta-Environment. Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications. November 1997.

[15] Evan Mamas and Kostas Kontogiannis. Towards Portable Source Code Representations using XML. Proceedings of Seventh Working Conference on Reverse Engineering. November 2000.

[16] Rigi Standard Format. http://www.rigi.csc.uvic.ca/rigi/manual/node52.html

[17] R. C. Holt. An Introduction to TA: The Tuple-Attribute Language. February 1997. http://swag.uwaterloo.ca/pbs/papers/ta.html

[18] R. C. Holt. TA as a Drawing Language. April 1997. http://swag.uwaterloo.ca/pbs/papers/tadraw.html

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

[20] R. C. Holt, A. Winter and A. Schürr. GXL: Towards a Standard Exchange Format. Proceedings of the 7th Working Conference on Reverse Engineering. November 2000.

[21] Mark Weiser. Program Slicing. IEEE Transactions on Software Engineering. July 1984.

[22] K.J Ottenstein and L.M. Ottenstein. The Program Dependence Graph in a Software Development Environment. Proceedings of the ACM Software Engineering Symposium on Practical Software Development Environments. April, 1984.

[23] Frank Tip. A Survey on Program Slicing Techniques. Journal of programming languages. 1995.

[24] Andera De Lucia. Program Slicing: Methods and Application. Proceedings of First IEEE International Workshop on Source Code Analysis and Manipulation. November 2001

[25] Ted J. Biggerstaff, Bharat G. Mitbander and Dallas Webstar. The Concept Assignment Problem in Program Understanding. Proceedings of the 15th International Conference on Software Engineering. May 1993.

[26] Nicolas Gold and Keith Bennett. Hypothesis-based Concept Assignment in Software Maintenance. IEE Proceedings on Software. August 2002.

[27] C. Lindig, and G. Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. Proceedings of the 19th International Conference on Software Engineering. May 1997.

[28] M. Siff and T. Reps. Identifying Modules via Concept Analysis. IEEE Transactions on Software Engineering, Volume 25 Issue 6. November 1999.

[29] G. Snelting. Software Reengineering based on Concept Lattices. Proceedings of the Fourth European Conference on Software Maintenance and Reengineering, March 2000.

[30] G. Antoniol, G. Casazza, M. di Penta and E. Merlo. A method to re-organize legacy systems via concept analysis. Proceedings. 9th International Workshop on Program Comprehension. May 2001.

[31] Refine User's Guide. Reasoning Systems Inc. 1992.

[32] J. Ebert, M. Kamp, A. Winter. A Generic System to Support Multi-Level Understanding of Heterogeneous Software. Fachbericht Informatik, Universität Koblenz-Landau, Institut für Informatik, Koblenz. 1997.

[33] The Software Architecture Toolkit. http://www.swag.uwaterloo.ca/swagkit/

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

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

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

[37] Ying Zou and Kostas Kontogiannis. Towards a Portable XML-based Source Code Representation. Workshops of XML Technologies and Software Engineering, International Conference on Software Engineering. May 2001.

[38] ConExp Homepage at Darmstadt University of Technology http://www.mathematik.tu-darmstadt.de/ags/ag1/Software/ConExp/

[39] Keith B. Gallagher and James R. Lyle. Using Program Slicing in Software Maintenance. IEEE Transactions on Software Engineering, Volume 17 Issue 8. August 1991.

[40] Mark Harman, Nicolas Gold, Rob Hierons and Dave Binkley. Code Extraction Algorithms which Unify Slicing and Concept Assignment. Proceedings of Ninth Working Conference on Reverse Engineering. October 2002.

[41] Paolo Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Transactions on Software Engineering, Volume 29 Issue 6. June 2003

[42] K. Sartipi, K. Kontogiannis. A User-assisted Approach to Component Clustering. In Journal of Software Maintenance: Research and Practice (to appear 2004).

[43] Ted J. Biggerstaff, Josiah Hoskins and Dallas Webstar. DESIRE: A System for Design Recovery. MCC Technical Memo. April 1989.

[44] The Unravel Project. http://hissa.nist.gov/unravel/

[45] The Wisconsin Program-Slicing Tool. http://www.cs.wisc.edu/wpis/slicing_tool/

[46] ToscanaJ. http://sourceforge.net/projects/toscanaj/