Completeness of Fact Extractors and a New Approach to Extraction with Emphasis on the Refers-to Relation

by

Yuan Lin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2008

©Yuan Lin 2008

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis deals with *fact extraction*, which analyzes source code (and sometimes related artifacts) to produce extracted *facts* about the code. These facts may, for example, record where in the code variables are declared and where they are used, as well as related information. These extracted facts are typically used in software reverse engineering to reconstruct the design of the program.

This thesis has two main parts, each of which deals with a formal approach to fact extraction. Part 1 of the thesis deals with the question: How can we demonstrate that a fact extractor actually does its job? That is, does the extractor produce the facts that it is supposed to produce? This thesis builds on the concept of *semantic completeness* of a fact extractor, as defined by Tom Dean et al, and further defines *source, syntax* and *compiler completeness.* One of the contributions of this thesis is to show that in particular important cases (when the extractor is deterministic and its front end is idempotent), there is an efficient algorithm to determine if the extractor is *compiler complete.* This result is surprising, considering that in general it is undecidable if two programs are semantically equivalent, and it would seem that source code and its corresponding extracted facts are each essentially programs that are to be proved to be equivalent or at least sufficiently similar.

The larger part of the thesis, Part 2, presents Algebraic Refers-to Analysis (ARA), a new approach to fact extraction with emphasis on the Refers-to relation. ARA provides a framework for specifying fact extraction, based on a three-step pipeline: (1) basic (lexical and syntactic) extraction, (2) a normalization step and (3) a binding step.

For practical programming languages, these three steps are repeated, in stages and phases, until the Refers-to relation is computed. During the writing of this thesis, ARA pipelines for C, Java, C++, Fortran, Pascal and Ada have been designed. A prototype fact extractor for the C language has been created.

Validating ARA means to demonstrate that ARA pipelines satisfy the programming language standards such as ISO C++ standard. In other words, we show that ARA phases (stages and formulas) are correctly transcribed from the rules in the language standard.

Comparing with the existing approaches such as Attribute Grammar, ARA has the following advantages. First, ARA formulas are concise, elegant and more importantly, insightful. As a result, we have some interesting discovery about the programming languages. Second, ARA is validated based on set theory and relational algebra, which is more reliable than exhaustive testing. Finally, ARA formulas are supported by existing software tools such as database management systems and relational calculators.

Overall, the contributions of this thesis include 1) the invention of the concept of hierarchy of completeness and the automatic testing of completeness, 2) the use of the relational data model in fact extraction, 3) the invention of Algebraic Refers-to Relation Analysis (ARA) and 4) the discovery of some interesting facts of programming languages.

Acknowledgements

I would like to thank my supervisors, Richard Holt and Andrew Malton, for many years of guidance and inspiration. They have constantly taken care of their graduate students by pointing out relevant research, generating ideas, and finding work for them.

I thank my dissertation committee members, Professor Charlie Clarke, Professor Grant Weddell, Professor Kostas Kontogiannis, and Professor Hausi Müller, for their invaluable time and effort put into reading my thesis.

I also thank my current and previous members of SWAG, in particular, Jingwei Wu, Lijie Zou, Xinyin Dong, Cory Kapser and Ahmed Hassan. I appreciate their great friendship as well as their insightful comments on many ideas in this dissertation.

I am grateful to my parents and my sister for their support from China. I wish to especially thank my wife, Chunhui Meng for her support during the past six years of study and taking care of our baby, Erica Lin.

Table of Contents

List of Figures x
List of Tables xi
Chapter 1 Introduction 1
1.1 How to verify a fact extractor 1
1.1.1 Drawbacks in current approaches 1
1.1.2 Motivation 2
1.1.3 Completeness of a fact extractor 2
1.2 Algebraic Refers-to Analysis: A new approach to fact extraction 3
1.2.1 Current approach for extracting the Refers-to relation 4
1.2.2 Motivation 4
1.2.3 Algebraic Refers-to Analysis (ARA) 5
1.2.4 Validation of ARA 7
1.3 Contributions 8
1.4 Thesis Organization 8
Chapter 2 Related work 10
2.1 Extraction of the Refers-to relation 10
2.1.1 Ad hoc method 10
2.1.2 Attribute Grammars 13
2.2 Data models for facts in the source code 18
2.2.1 Conceptual models for extracted facts 18
2.2.2 Storage models for extracted facts 18
2.2.3 Schemas 19
2.3 Application of relational algebra in software engineering 20
2.3.1 Program analysis 20
2.3.2 Software repository exploration 21
2.3.3 Software architecture recovery and repair 21
2.4 Relational algebra software tools 22
Chapter 3 Completeness of fact extraction 24
3.1 Introduction 24
3.2 Compiler phases and ASGs 25
3.3 Schemas and interchange formats 25
3.4 Completeness of fact extractors 27

3.4.1 Four levels of completeness 27
3.4.2 Hierarchy of completeness 28
3.4.3 Semantic completeness 29
3.4.4 Relative completeness 29
3.5 Validating CPPX's semantic completeness 32
3.5.1 Why are assembly codes a_0 and a_1 identical? 32
3.5.2 Suite of test programs 34
3.5.3 The resolution problem 34
3.6 RCPPX: Recovering source from facts 35
3.6.1 Suffixing script 37
3.6.2 Nested syntax 38
3.6.3 TXL scripts 39
3.7 Errors found in CPPX 41
3.8 Conclusions and future work 42
Chapter 4 Algebraic Refers-to Analysis (ARA) as a new approach to fact extraction 43
4.1 Traditional approaches to fact extraction 43
4.1.1 Facts and their classification 44
4.1.2 Importance of Refers-to relation 44
4.1.3 Traditional approaches to extracting Refers-to relation 45
4.2 Motivation toward a relational approach to fact extension 46
4.3 Mathematical notation: Relational algebra 47
4.3.1 Binary relational algebra 48
4.3.2 N-ary relational algebra 49
4.4 Problem definition 49
4.5 ARA pipelines 50
4.5.1 Three-step pipeline 50
4.5.2 Tables and graphs used for summarizing the three-step pipeline 57
4.5.3 Multi-phase pipelines 58
4.6 Validation of ARA 59
4.7 Chapter summary 60
Chapter 5 ARA for C 61
5.1 C syntax related to Refers-to relation 61

5.1.1 Occurrences in C 61
5.1.2 Scopes in the C language 63
5.1.3 Visibility and name hiding 64
5.2 Overview of ARA for C 64
5.3 Stage one: Resolving unqualified occurrences 65
5.3.1 Basic Fact Extraction 65
5.3.2 Normalization 65
5.3.3 Binding Formulas 67
5.4 Stage two: Resolving qualified occurrences 67
5.4.1 Basic Fact Extraction 68
5.4.2 Normalization 70
5.4.3 Binding Formulas 72
5.5 Chapter summary 72
Chapter 6 ARA for Java 73
6.1 Java syntax related to Refers-to relation 73
6.1.1 Occurrences in Java 73
6.1.2 Scopes in Java 74
6.1.3 Visibility and name hiding 74
6.2 Overview of ARA for Java 75
6.3 Stage One: Resolving unqualified occurrences 76
6.3.1 Basic Fact Extraction 77
6.3.2 Normalization 78
6.3.3 Binding 82
6.4 Stage Two: Resolving qualified occurrences 82
6.4.1 Basic Fact Extraction 83
6.4.2 Normalization 83
6.4.3 Binding 85
6.5 Chapter summary 85
Chapter 7 ARA for C++ 86
7.1 C++ syntax related to the Refers-to relation 86
7.1.1 Occurrences in C++ 86
7.1.2 Scopes in C++ 87

7.1.3 Sub-object lattices 91
7.1.4 Other issues related to the Refers-to relation 93
7.2 Overview Of ARA For C++ 93
7.2.1 Phases in ARA for C++ 93
7.2.2 Inside each phase of ARA for C++: Trinary hiding relation 93
7.3 Resolving unqualified occurrences in ordinary statements 95
7.3.1 Basic Fact Extraction 95
7.3.2 Normalization 97
7.3.3 Binding Formulas 103
7.4 Chapter Summary 104
Chapter 8 Advantages of ARA, applications of ARA and miscellaneous 105
8.1 ARA is elegant, concise and insightful 105
8.1.1 Phases and stages makes ARA concise 105
8.1.2 ARA only uses a small number of Basic Fact Extraction 107
8.1.3 Normalization formulas are elegant and insightful 107
8.1.4 Binding formulas are the same for most languages 110
8.2 Validating ARA 111
8.2.1 Validating claim 1: Phases and stages must satisfy the language standard 111
8.2.2 Validating claim 2: Basic Facts must represent the source code according to the language standard 112
8.2.3 Validating claim 3: Formulas for Normalization and Binding must satisfy the language standard 113
8.3 All major components of ARA are supported by existing software 114
8.4 ARA has a wide range of applications in reverse engineering 115
8.5 Miscellaneous 116
8.5.1 Resolution of syntactic ambiguities 117
8.5.2 Macros, templates and generics 117
8.5.3 Features not supported by current ARA 117
8.6 Chapter summary 118
Chapter 9 Conclusions 119
9.1 Contributions to fact extraction 119
9.2 Future work 121
9.2.1 Applying ARA to dynamically scoped languages 121

9.2.2 Using ARA in program transformation 121
Appendix........ 123
Appendix A Visibility and name hiding rules in Java 123
Table 4. Hiding rules 124
Appendix B List of formulas in ARA for C++ 125
1. Phase for namespace declarations 126
2 Phase for using-directives 127
3 Phase for class declaration 128
4 Occurrences in class member definition 128
5 Occurrences in template definition 129
6 Occurrences in template instantiation 130
7 Ordinary statements 131
Bibliography 132

List of Figures

Figure 1.1 Three-step pipeline 6
Figure 1.2 Multi-phase pipeline 7
Figure 3.1: CPPX fact extractor. 24
Figure 3.2: Validating three kinds of completeness 28
Figure 3.3: Relative completeness of E and T. 31
Figure 3.4 Validating Completeness of CPPX. 33
Figure 3.5: TA from example C program as a diagram 38
Figure 4.1 Relational composition 48
Figure 4.2 The three-step pipeline 50
Figure 4.3 HasName for Example 4.1 52
Figure 4.4 HasKind for Example 4.1 52
Figure 4.5 Contains for Example 4.1 53
Figure 4.6 Relation B_3 (solid arrows) and an edge of H (dashed arrows) 56
Figure 4.7 The Refers-to Relation for Example 4.1 57
Figure 4.8 Data flow graph of the three-step pipeline 58
Figure 4.9 Multi-phase pipeline 59
Figure 5.1 ARA for C 63
Figure 5.2 Stages in ARA for C 65
Figure 5.3 Data flow graph for Stage two 68
Figure 5.4 Relations for Example 5.1 70
Figure 5.5 Relations for Example 5.2 71
Figure 5.6 Relations for Example 5.3 72
Figure 6.1 Data flow graph for resolving unqualified occurrences 76
Figure 6.2 Relation Contains (C) and Inherits (I) for Example 6.1 79
Figure 6.3 Data flow graph for qualified occurrences 83
Figure 7.1 Sub-object lattice for non-virtual base class 91
Figure 7.2 Sub-object lattice for virtual base class 92
Figure 7.3 Data Flow for resolving unqualified occurrences 96
Figure 7.4 Relation Parent (P) and Inherits (I) for Example 7.4 100
Figure 7.5 Relation Parent (P) and Uses (S) in Example 7.5 101
Figure 8.1 Multi-phase pipeline 107

List of Tables

Table 2.1 Language standards 11
Table 2.2 Features related to Refers-to relation 12
Table 2.3 Fact extractors surveyed 13
Table 2.4 Common extensions to Attribute Grammar 16
Table 2.5 The data models for reverse engineering tools 19
Table 2.6 Examples of application of relational algebra in software engineering 22
Table 2.7 Relational algebra software tools 23
Table 3.1 Four levels of completeness for a fact extractor 28
Table 4.1 Example entities in source code 44
Table 4.2 Example of relations in source code 45
Table 4.3 Facts in Example 4.1. 51
Table 4.4 Summary of three-step pipeline for ALGOL-60 like languages 57
Table 5.1 Basic Facts for unqualified occurrences 66
Table 5.2 Basic Facts for Stage two 68
Table 6.1 Occurrences in Java 74
Table 6.2 Basic Facts for unqualified occurrences 77
Table 6.3 Basic Facts for qualified occurrences. 84
Table 7.1 C++ occurrences 88
Table 7.2 Basic Facts needed in resolving unqualified occurrences in ordinary statements 95
Table 8.1 Seven languages in case studies. 106
Table 8.2 Formulas for Visibility (V) for different languages 109
Table 8.3 Formulas for Hiding (H) for different languages 110
Table 8.4 Binding formulas for different languages 111
Table 8.5 Test results on open source software 115

Chapter 1
Introduction

Fact extraction is the process of analyzing software artifacts and presenting the information about in a convenient form. It is a crucial step in the process of understanding the relationships among a system's elements and is usually automated by a software tool called *fact extractor*.

Over the past decade, many fact extractors have been developed to extract facts from different kinds of software artifacts, such as source code, object files, version control logs and so on. Various fact extraction approaches have been devised based on the theory of formal language and compilation. However, some fundamental questions remain open. In this dissertation, I answer two of them. First, how can we verify a fact extractor automatically? Second, compiler construction approaches such as Attribute Grammars are time consuming and error prone to use in fact extraction. Can we find a fact extraction approach that is reliable and easy to implement? This dissertation presents new solutions to these questions. The next two sections provide more detail about them.

1.1 How to verify a fact extractor

Fact extraction from source code is a fundamental activity for reverse engineering and program comprehension tools, because all subsequent activities depend on the data produced. Creating such a fact extractor is a challenging problem, especially for complex source languages such as C++ [DMH01] [FSH+01]. Consequently, it would be useful to have a convenient means to validate a fact extractor. Currently, test suites and benchmarks are the primary tools for this task [SHE02] [MNL96][AT98]. However, there are serious difficulties in justifying the test suite and in carrying out the tests on large input.

1.1.1 Drawbacks in current approaches

Conceptually, writing a test suite for a fact extractor is straightforward; it is similar to writing a test suite for a compiler. The difficulties appear when evaluating the output from a fact extractor. First, what is the standard for judging whether the answers correct. The extracted facts follow a variety of schemas, ranging from the abstract syntax tree level to the architectural level. They could be stored in a variety of formats, such as an in-memory repository or in a human-readable intermediate format, such as GXL [Win01]. In addition, the extracted facts that contain some variances might still be useful in practice. For example, we can use a fact extractor that translates (x >= 1) as (x > 0) when x is an integer, because the compiler GCC front end carries out the translation. .

Second, software tools for correlating the extracted facts with the original code are lacking and the task is often carried out by human. Writing a tool to check the accuracy of facts as specified by a

schema can be as difficult as writing an extractor itself. The feasibility of writing such a tool also seems in doubt because validating whether two arbitrary programs have the same meaning is known as un-decidable. As a consequence, benchmarking the fact extractor has been limited small hand-crafted inputs.

1.1.2 Motivation

The motivation for completeness of a fact extractor comes from CPPX, a fact extractor based on union grammars [DMH01]. It is composed of a series of transformations from GCC internal representations of compilation units to facts specified by the Datrix Schema [Bel01]. More importantly, throughout all the transformations, the compilation unit (as GCC internal representation or facts) is compliant to the union grammar of the GCC internal representation and Datrix Schema.

Besides being a fact extractor, CPPX proves that there exist transformations that map the source code into the extracted facts using the union grammar approach. Program transformation tools such as TXL also use this approach, which makes it possible to create recovery transformations that map the extracted facts back into the source code.

The CPPX approach motivates the research in validating fact extractors in two ways. First, the extracted facts and the original code are closely related and comparing these two is therefore easier than comparing two arbitrary programs. Second, we can choose the convenient form of representing facts when we compare the extracted fact and source code, either comparing two copies of the source code or comparing two copies of extracted facts. The concept of completeness of fact extractor is a result of this motivation.

1.1.3 Completeness of a fact extractor

This thesis builds on the concept of *semantic completeness* of a fact extractor, as defined by Tom Dean et al [DMH01], and further defines *source, syntax* and *compiler completeness.* One of the contributions of this thesis is to show that in particular important cases (when the extractor is deterministic and its front end is idempotent), there is an efficient algorithm to determine if the extractor is *compiler complete.* This result is surprising, considering that in general it is undecidable if two programs are semantically equivalent, and it would seem that source code and its corresponding extracted facts are each essentially programs that are to be proved to be equivalent or at least sufficiently similar.

The concept of completeness is elegant on paper, but is it useful in practice? To answer this question, we validated the completeness of CPPX. Perhaps the most obvious approach to validate CPPX is to run it against a large test suite of source programs, and to check that the extracted factbase for each test is correct. This approach would be very expensive, as it requires extensive manual work to do the checking or to create putatively correct factbases for comparison to CPPX generated

factbases. As we will explain in Chapter 3, our approach to validation of CPPX uses a test suite of programs, but avoids this manual step.

The work involved two major surprises. First, we were surprised to discover that the assembly code for an original source program and for the version of the source program recovered from its extracted factbase were identical. Second, we were surprised to find a mechanical test for a special case of a generally undecidable problem. Combining these two surprises, we developed a method for an automatic validation for semantic completeness of a fact extractor. These results (Part 1 of the thesis) are described in more detail in Chapter 3. We will now overview Part 2 of the thesis.

1.2 Algebraic Refers-to Analysis: A new approach to fact extraction

Facts in source code fall into three categories: lexical facts, syntactic facts and semantic facts. In compiler construction, the extraction of lexical and syntactic facts have a formal basis (parsing theory and formal languages) that is highly practical and have been widely used in fact extraction.

However, no formalism is widely used for the extraction of semantic facts, even though *semantic analysis* approaches such as *Attribute Grammars* have been formalized and shown to have promise in compiler construction. The fact is that even with the help of software tools, writing a fact extraction based on Attribute Grammars is time consuming and error prone.

This thesis assumes that traditional lexical analysis and syntactic analysis have been done, and then demonstrates how key aspects of fact extraction, the *Refers-to relation* in particular, can be elegantly specified using mathematical relations. Because many software tools support relational algebra, the new approach presented in this thesis requires minimal coding.

The Refers-to relation is the relation between referential occurrences of an identifier and its definitional occurrence. Consider the following C program.

int x_1;

void f_2( ) {$_3$

int x_4;

x_5++;

$_3$}

For convenience, occurrences and blocks in examples in this thesis are assigned unique integers. There are three occurrences of identifier x, i.e. occurrences 1, 4 and 5. Among them, occurrences 1 and 4 are definitional occurrences and occurrence 5 is a referential occurrence, and occurrence 5 refers to occurrence 4. So, the Refers-to relation consists of a single pair <5, 4>.

Among semantic facts, the Refers-to relation is critical, for the following reasons. First, the Refers-to relation is a fundamental concept in programming languages. Programming language specifications, textbooks and critics of programming languages all spend considerable effort writing

about it. Extracting the Refers-to relation is also a critical step in compilation and program comprehension [App98]. Second, we can derive other static semantic facts based on the Refers-to relation and lexical and syntactical facts straightforwardly. When lexical and syntactical facts are represented as relations, the task of extracting other static semantic facts can be implemented as simple SQL statements. Third, the Refers-to relation is a key to understand the architecture of software systems [BHB99] [FHC01]. These reasons explain why we have concentrated on the Refers-to relation in this thesis.

1.2.1 Current approach for extracting the Refers-to relation

Currently, two approaches, ad hoc method and Attribute Grammars, are widely used in extracting the Refers-to relation. Based on language standards such as ISO C++ standard [C++03], ad hoc method is used the most often in fact extraction. Language standards communicate the rules for the language effectively and provide guidance for fact extraction and compiler construction. But they also have drawbacks. There are no software tools that can implement the language standards directly. Implementing these language standards manually using ad hoc method is time consuming and error prone.

Attribute Grammars were introduced in 1968 and some compiler construction tools have been developed based on Attribute Grammars. However, Attribute Grammars are not convenient for formalizing the Refers-to relation for two reasons. First, its formalism is not complete. It only formalizes the movement of information on the parser tree (or abstract syntax tree) and the rules on the Refers-to relation are hand coded instead of being formalized. It is also hard to link the Attribute Grammar back to the language standards and see whether the Attribute Grammars are correct. Second, the Attribute Grammars are inconvenient and hard to understand, especially compared with well-written language specifications such as ISO C++ standard and Java language specification. As a consequence, building a fact extractor based on Attribute Grammars is time consuming and error prone, even with the help of software tools.

1.2.2 Motivation

The thesis is partly motivated by the relational data model [Cod70][Cod82] and its applications in software engineering. As discussed in the related work (Chapter 2), many problems in reverse engineering and program comprehension can be formalized as relational algebra problems. The relational algebra model has the advantages of simple data structures, mathematically defined operations and a wide range of software tools that support it. Inspired by the relational data model, I invented *Algebraic Refers-to Relation Analysis* (ARA).

ARA introduces a new approach for specifying fact extraction. It differs from Attribute Grammars in three ways. First, in Attribute Grammar, fact extraction is a process of decorating the parse tree. In ARA, fact extraction is a query to find pairs of occurrences of identifiers that satisfy certain

conditions. For example, one of the conditions requires that two occurrences of identifier have the same name.

Second, Attribute Grammars specify the transportation of data on the parser tree, while ARA represents facts in the source code as relations and stores them in a relational database or an ASCII file.

Finally, Attribute Grammars require programmers to write code to implement the rules on the Refers-to relation. ARA translates these rules into relational algebra formulas and runs them on a relational database or other existing software tools. Because relational algebra is closely related to predicate logic [RG00], ARA formulas are similar to rules written in natural language and therefore are easy to be verified against language standards. In addition, many software tools including relational database management systems support ARA formulas. As a result, the major parts of ARA require no coding.

1.2.3 Algebraic Refers-to Analysis (ARA)

In its simplest form, ARA extracts the Refers-to relation in three steps, *Basic Fact Extraction*, Normalization and Binding. Collectively, these steps are called three-step pipeline. For a practical language such as C and Java, ARA is extended into a multi-phase pipeline. The following gives more detail about the pipelines.

1.2.3.1 Three-step pipeline

For simplest examples (see Chapter 4), the Refers-to relation can be extracted in three sequential steps, which we call the three-step pipeline (see Figure 1.1).

Step 1. Basic Fact Extraction. The information (facts) in the source code is represented as relations, which are called *Basic Facts*. This step is implemented using a scanner and parser. The computation of the Refers-to relation only needs a few simple facts from the source code.

Step 2. Normalization. Using a set of relational algebra formulas, the Basic Facts are converted into *Normalized Facts* that are almost language independent. Each language has its own rules as to what kinds of occurrence are compatible, what regions of code text can be referred to by an occurrence and so on. These rules are translated into relational algebra formulas, which are used in this step.

Step 3. Binding. The Refers-to relation is computed from the Normalized Facts in the Binding step. In this step, all facts ARA needs have been extracted from the source code and stored as the Normalized Facts. Binding is a query to find the relation of occurrences of identifier that satisfy four *Binding Conditions*. The result of the query is the Refers-to relation.

The three-step pipeline is designed based on language standards. In particular, Basic Fact Extraction is based on the syntax of the language. Normalization and Binding are basically the translation of rules in the language standards.

Figure 1.1 Three-step pipeline

1.2.3.2 Multi-phase pipeline

The real world programming languages is complex. The Refers-to relation of some occurrences depends on the Refers-to relation of other occurrences and we have to compute the Refers-to relation using several iterations. ARA solves this problem by introducing the multi-phase pipeline (see Figure 1.2).

Each phase in the multi-phase pipeline resolves a subset of identifier occurrences such as all occurrences in the import declarations (in Java) or all occurrences in the ordinary statements. The union of results from phases produces the Refers-to relation for occurrences in the program. The language standards imply that some occurrences must be resolved before others. For example, the occurrences in the import declaration (in Java) affect the Refers-to relation of occurrences in the ordinary statements. Therefore, occurrences in the import declaration are resolved in an earlier phase than occurrences in the ordinary statements. As a result, ARA resolves occurrences in the import declaration in one phase and occurrences in the ordinary statements in a later phase.

Within each phase, there can be at most two stages. Stage 1 resolves the unqualified occurrences in the current phase such as occurrence of x in x++. Stage 2 resolves the qualified occurrences such as occurrence y in x.y. Each stage has three steps, Basic Fact Extraction, Normalization and Binding, as we have seen in the three-step pipeline. The results from the previous phases and stages are used by the current phase or stage.

The thesis shows that with the multi-phase pipeline, ARA can compute the Refers-to relation for C, Java, CPP, C++, Fortran, Pascal and Ada. The major concepts of the thesis have now been introduced. Now, let us consider the contributions of the thesis.

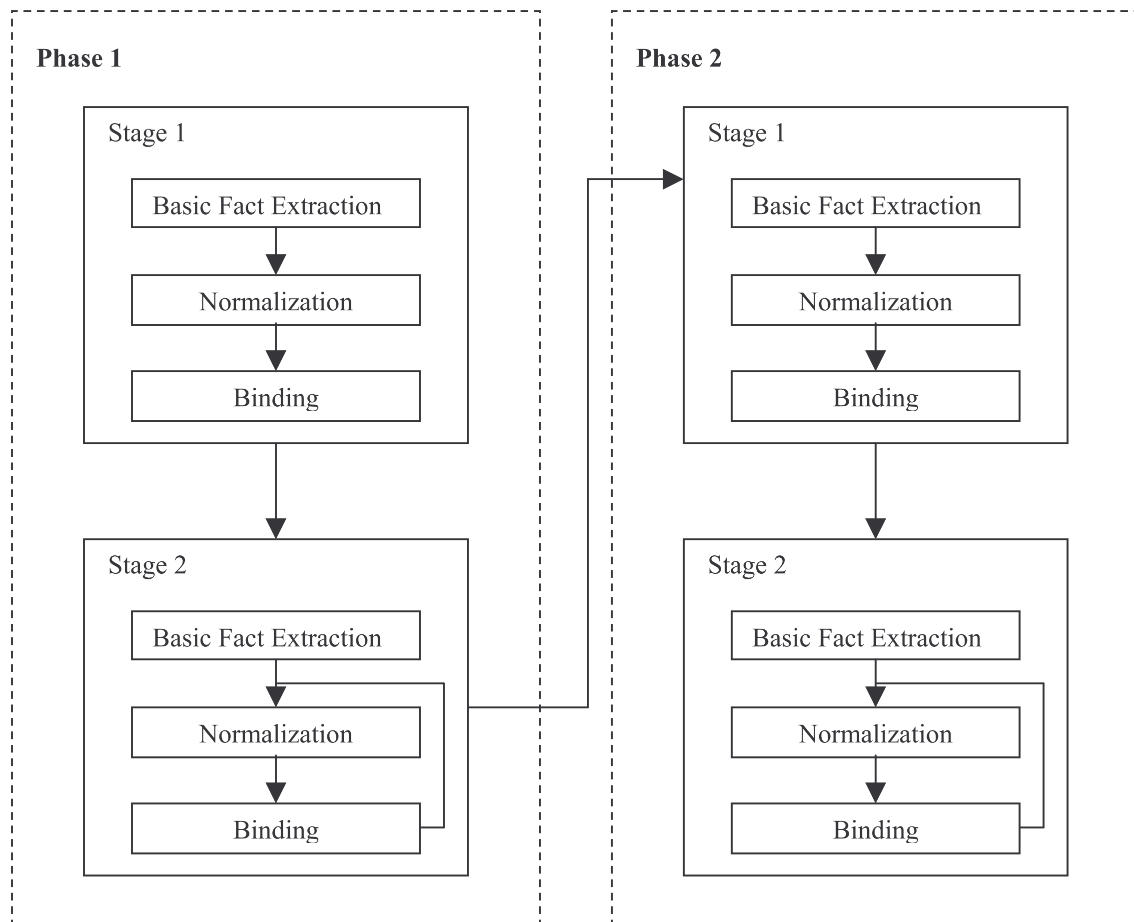

Figure 1.2 Multi-phase pipeline

1.2.4 Validation of ARA

Validation of ARA means to demonstrate that ARA satisfies programming language standards such as ISO standards. We validate ARA by showing that it is a transcription of the programming language standards, which produces formulas in set theory and relational algebra. For example, formulas for ARA for the Java language comprise a multi-phase pipeline. Validation of ARA for Java consists of proving the following three claims.

1. Phases and stages must satisfy the language standard;
2. Basic Facts must represent the source code according to the language standard;
3. Formulas for Normalization and Binding must satisfy the language standard.

Section 8.2 will give the detail of how this validation is done. It will show that rule x in the language is transcribed into a phase (or stage) y in ARA. Then, it will show the transcription is correct using predicate logic, set theory and relational algebra.

1.3 Contributions

The thesis makes a number of contributions to fact extraction and reverse engineering, as summarized here.

1. Inventing the concept of hierarchy of completeness and the automatic testing of completeness

The completeness of fact extractor is a measure for determining whether a fact extractor is accurate enough to be useful in reverse engineering. We can classify fact extractors into different levels based on their completeness. We can also validate how far a fact extractor is from achieving certain level of completeness.

2. Use of the relational data model in fact extraction

We formulate the extraction of the Refers-to relation as a query to find occurrences that satisfy the Binding Conditions. To extract the Refers-to relation, we work on a few facts in the source code that are represented as relations.

3. Inventing Algebraic Refers-to Relation Analysis (ARA)

The three-step pipeline and multi-phase pipeline are designed and applied to seven statically scoped languages, including C, Java, C++, CPP, Fortran, Pascal and Ada, based on the language standards.

4. Discovery of some interesting facts of programming languages

During the design of ARA, some interesting aspects of programming languages are discovered. For example, ARA only needs binary relations for C, CPP, Java, Fortran, Pascal and Ada. But we need both binary relation and trinary relations in ARA for C++. This implies, in a mathematical sense, that C++ is a more complex language than these other languages.

1.4 Thesis Organization

The dissertation is organized as the following. Chapter 2 discusses research work related to ARA. Part 1 of the thesis (Chapter 3) details my work on the completeness of fact extractors. Part 2 of the thesis (Chapters 4, 5, 6, 7 and 8) focuses on ARA. Specifically, chapter 4 gives the overview of ARA

while chapters 5, 6 and 7 present how ARA computes the Refers-to relation in C, Java and C++ programs. Chapter 8 summarizes the advantages of ARA and clarifies certain issues. Finally, Chapter 9 concludes the dissertation.

Chapter 2
Related work

This chapter presents the research work related to ARA. In specific, Section 2.1 explains the specification of extraction of the Refers-to relation. Section 2.2 explains the data models for representing facts. Section 2.3 presents the application of relational algebra in software engineering. Finally, Section 2.4 presents some software tools that support relational algebra.

2.1 Extraction of the Refers-to relation

The Refers-to relation is a key component of every programming language. Besides ARA, there are mainly two ways of implementing the extraction of the Refers-to relation, ad hoc method and Attribute Grammars, which are explained in the following.

2.1.1 Ad hoc method

Ad hoc method is based on the informal specification of the Refers-to relation in the language standards and implements the extraction of the Refers-to relation manually. The language standards describe the rules for the Refers-to relation in natural language and also provide examples to help readers to understand the rules. They are flexible and more importantly, specify the subject at the right level of abstraction. As a result, they communicate the rules for the language effectively and provide guidance for fact extraction and compiler construction.

But they also have drawbacks. As for the Refers-to relation, there are no software tools that can implement rules in these standards directly. And implementing the rules manually is time consuming and error prone.

Because language standards are often referred to in this thesis, we give some detail about them here. The language standard usually first gives an overview of the Refers-to relation and then clarifies the specific rules as the rules are needed. During the writing of this thesis, the standards (informal specification) of seven programming languages are studied, including C, CPP, C++, Java, Fortran, Pascal and Ada. These programming languages are statically typed. The language standards are listed in Table 2.1. Also listed in the table is the clause where the standards address the concept of Refers-to relation. All these standards use the concept of scope of declarations to specify the Refers-to relation.

Language	Standards	Overview of Refers-to Relation
Ada	ISO/IEC 8652:1995(E)	8.2 Scope of declarations
C	ISO/IEC 9899:1999	6.2.1 Scopes of identifiers, 6.2.3 Name spaces of identifiers
CPP	ISO/IEC 9899:1999	6.10.3 macro replacement
C++	ISO/IEC 14882:2003	3.3 Declarative regions and scopes; 3.4 Name lookup
Fortran	ISO/IEC 1539:2004	16.1 scope of global identifiers; 16 scope of local identifiers
Java	The Java Language Specification, 3rd Edition	6.3 scope of a declaration; 6.4 Members and inheritance; 6.5 Determining the meaning of a name
Pascal	ISO 7185:1990 (standard Pascal) and ISO/IEC 10206:1990 (extended Pascal)	6.2 blocks, scopes, activations, and states

Table 2.1 Language standards

2.1.1.1 Language standards vs. ARA

Now, consider the specific rules for the Refers-to relation. Table 2.2 lists the features for Refers-to relation in the seven languages. As a reminder, the list includes features in the latest versions of the languages and some features might not be in a given language in the original design. For example, inheritance is added to Ada after Ada95. The consequence of such enhancements is that the languages are becoming alike, at least from fact extraction's point of view.

Because the standards are written in natural language, it is not surprising to find that the descriptions vary quite a bit from one standard to another, even for the same feature. For example, the following is the description of name hiding in ISO/IEC C++ standard (clause 3.3.7).

> A name can be hidden by an explicit declaration of that same name in a nested declarative region or derived class…
>
> In a member function definition, the declaration of a local name hides the declaration of a member of the class with the same name, The declaration of a member in a derived class (clause 10) hides the declaration of a member of a base class of the same name, …

Language	Standard	Refers-To Relation Features
C	ISO/IEC 9899:1999 [C99]	Qualification, containment, order of occurrences
CPP	ISO/IEC 9899:1999 [C99]	Order of occurrences
Java	Java Language Specification, 3rd Ed. [GJSB05]	Qualification, containment, single inheritance, import
C++	ISO/IEC 14882:2003 [C++03]	Qualification, containment, order of occurrences, multiple inheritance, namespace using-directives, namespace alias, argument-dependent name lookup
Fortran	ISO/IEC 1539-1:2004 [For04]	Containment, uses clause, contains clause, interface clause
Pascal	ISO/IEC 10206: 1990(E) [Pas90]	Qualification, containment, order of occurrences, uses clause, import/export clause
Ada	ISO/IEC 8652:1995(E) [Ada95]	Qualification, containment, order of occurrences, single inheritance (Ada95 only), redefine

Table 2.2 Features related to Refers-to relation

The above description is more concise than the similar description in the Java language specification, which is the following.

> A declaration d of a type named n shadows the declarations of any other types named n that are in scope at the point where d occurs throughout the scope of d.
>
> A declaration d of a field, local variable, method parameter, constructor parameter or exception handler parameter named n shadows the declarations of any other fields, local variables, method parameters, constructor parameters or exception handler parameters named n that are in scope at the point where d occurs throughout the scope of d.
>
> A declaration d of a method named n shadows the declarations of any other methods named n that are in an enclosing scope at the point where d occurs throughout the scope of d.

In general, the specification in ISO/IEC C/C++ [C++03] standard and Java language specification [GJSB05] is closer to the relational algebra formulas used in ARA.

2.1.1.2 Ad hoc method and fact extraction

Most compilers and fact extractors are written using ad hoc method. During the writing of this thesis, a survey of existing fact extractors was carried out. The fact extractors studied are listed in Table 2.3. The goal of the survey is not be complete but to confirm that writing a fact extractor using existing approaches is a time consuming and error prone task.

Table 2.3 classifies fact extractors into three categories based on the technique they employ. The first row includes fact extractors that use only scanners. Although these fact extractors are relatively easy to implement, they are in general not accurate due to the lack of parsers.

The second row includes the fact extractors that consist of scanner, parser and semantic analyzer. These fact extractors are similar to compiler front end and therefore, time consuming to develop. In addition, testing these fact extractors is also time consuming.

The last row includes the fact extractors based on the output from a compiler. The fact extractors listed here use the output from GCC because GCC is open source software. These fact extractors are significantly simpler than the compiler front end. However, these fact extractors are extensions to a compiler, not stand alone software systems.

Technique	Refers-To Relation Features
Scanner only	MultiLex [CC00] [CC03], Cscope [Csc04], AWK [AKW79]
Compiler front end	Sniff [Bis92][BKMS95], GENOA [Dev92], CIA [CNR90], A* [LR95], Rigi [MK88], Reprise [RW91], FAMIX [SoC99], TkSee/SN [TkS03]
Using output from a compiler (GCC [GCC02])	CPPX[CPP02], GCC_XML [GX], gccXfront [HMP03], XOgastan [APMV03]

Table 2.3 Fact extractors surveyed

2.1.2 Attribute Grammars

Attribute grammars are a specification of computations and dependence based on a formal calculus introduced by Knuth [Knu68]. They are well known as a formal technique for compiler construction. Most AG generators use an abstract syntax tree as an intermediate representation. Each symbol on the tree is associated with some (semantic) attributes. The semantic specifications in AGs are defined as attribute computations (attribution rules) associated with production rules of context free grammar. The value of an attribute cannot be accessed outside the context of the production rule. Thus a production rule can be considered as a specification unit, where an *inherited attribute* is an input and a *synthesized attribute* is an output of the context of the production rule. The computation of attributes has no side effects and is independent of parsing actions. They provide a notation for specifying

relationships among computations on ASTs. The programmers describe only individual computations and where they are invoked, and a tool deduces both a traversal strategy and a strategy for storing computed values.

The definition of an attribute grammar starts with the definition of an underlying *context-free grammar*. A context-free grammar is G = (N, T, P, Z) where N is the set of *non-terminals*, T is the set of *terminals*, P is the set of *productions*, and $Z \in N$ is the *start symbol*. The set $V = N \cup T$ is called the *vocabulary*. Each production $p \in P$ has the form p: $X \rightarrow \alpha$ where $X \in N$ and $\alpha \in V^*$.

An attribute grammar augments a context-free grammar by adding attributes and attribute computations. It is a quadruple AG = (G, A, R, B), where

G is a context-free grammar,

A is a set of attributes associated with each symbol in V,

R is a finite set of attribute computations

B is a finite set of plain computations.

The tree structure defined by the context-free grammar is then decorated. Each production describes a context consisting of a node and its children (if any). Attribute values decorate the nodes, attribute computations specify whom these values are related to, and plain computations extract information for other process.

Example 2.1 describes primitive expressions. It defines four non-terminal node types (*Expr*, *Add*, *Sub*, and *Const*) and 3 terminal node types (*Integer*, '+', and '-'). The children of the node type *Add* have the selector names *Lop*, '+', and *Rop*. Attribute computations are mainly written as assignments of target language expressions to attributes and are enclosed in { } brackets. Then attribute computations describe the evaluation of expressions. Attribute definitions are in enclosed in [ ] brackets. The node types *Expr* and *Integer* define one attribute named *Value* of type *INTEGER*. The subtypes *Add*, *Sub*, *Const*, and *Zero* inherit the attribute *Value* from *Expr*.

At a tree node, the attributes of this node and the attributes of the children are accessible. Attributes of the current node or of the left-hand side of a rule are denoted just by their name. The subtype *Zero* inherits the computation of the attribute *Value* from the base type *Expr*, whereas the node types *Add*, *Sub*, and *Const* overwrite it with node type specific computations. The value for the attribute of the node type *Integer* has to be provided by a scanner and parser.

```
Expr = [Value: INTEGER] { Value := 0; }

Add = Lop: Expr '+' Rop: Expr { Value := Lop.Value + Rop.Value; }

Sub = Lop: Expr '-' Rop: Expr { Value := Lop.Value – Rop.Value; }

Const = Integer { Value := Integer.Value; }

Zero = Integer : [Value: INTEGER]
```

Example 2.1 Attribute Grammar

In 1982, the Attribute Grammar for Pascal [ZKH82] and Ada [UDP+82] were published. Both of these grammars were developed for the GAG system, a generator that accepts attribute grammar and produces a Pascal program to carry out attribute evaluation. Pascal requires 63 pages, Ada 365 pages. Even the authors structured the specification in the best way they can, these specifications are hard to understand for human readers. The development in Attribute Grammars then focuses on enabling readable specification and generating efficient code. The extensions in the former are relevant to this thesis and target mostly at the following three areas.

First, adding a remote (attribute) access. In the tradition AG only local dependencies are allowed. A remote (attribute) access, a reference to an attribute on a remote tree node, is used as a tool to overcome part of the "localness" problem. For example, an attribute instance may need to be propagated through a sequence of nodes in corresponding to a number of production rules. Such a propagation sequence can be avoided by introducing a remote access to the propagated attribute. In other words, specification by remote access seems to be simpler than using lengthy attribute propagation (copy from one node to another).

Second, handling circular dependency or non-tree structures. These structures appear in many important tasks like symbol processing and data-flow analysis. For these tasks, current generative techniques cannot generate codes and specific algorithms have to be hand coded.

Third, allowing values of attributes to be changed. In traditional AGs, the values of attributes remain unchanged, which is quite inconvenient for many compiling tasks. Therefore, some extensions to AGs allow an attribute change its internal data. For instance, a symbol table attribute needs to change its internal value when an operation such as insertion is performed on it. An attribute with this property is called state transitional, and the operation on a state-transitional attribute is called an action. Some AG systems (e.g. Lido) adopt these concepts by providing ad-hoc constructs. For example, many stems support explicit (action) dependencies by treating them as special attribution rules.

To give a whole picture of the extensions to AGs, we first list common extensions to AGs. Table 2.4 shows the common extensions in the current AG systems. These systems are categorized into modularity, object-oriented, remote access, and collective computing. Modularity allows users to define the attributes and attribution rules of a symbol in more than one file. Inheritance construct lets users define the attribution rules of a symbol by composing from those of other symbols. An upward remote attribute denoted X.a accesses the nearest ancestor of symbol X and attribute a. Constituents [KHZ82][KW92] represents a set of attributes in the descendants of a symbol. A chaining [KHZ82][KW92] is a left-to-right attribute. A bucket brigade [JD84] is a bi-directional chaining. A target language's expression [Hed92][Gro89] is an expression evaluated into a reference to a tree node in accessing a remote attribute. List attribution assigns the inherited attribute of each son.

Second, besides implementing new extensions in AG system, the notations used in AGs are also extended over the years. For improving modularity, composable AGs [FMY92] are built from some component AGs, each of which models a particular sub-domain. The interconnections of component AGs are through input/output attributes specified by a glue grammar. Modular AGs [DC90] consists of a number of patterns associated with a set of templates. A template specifies the attribute rules to be generated for each matching production rule.

Category	Approach	AG systems
Modularity	Separate specification	ALADIN/GAG[KHZ82], Lido/Eli [KW92], SSL[RT89], OLGA[JP91] [JP97], Ag[Gro89][Gro92]
	Piped AGs	OLGA[JP91]
	Textual matching on productions	Modular AGs [DC90]
	Component grammar	Composable AGs [FMY92], Rewritable AGs [EH04]
Object-oriented	Multiple inheritance	Lido/Eli[KW92], Ag[Gro89][Gro92]
	Single inheritance	TOOLS [KNP88] [Los91], Door AGs [Hed92]
Remote access	Upward attribute	ALADIN/GAG [KHZ82], Lido/Eli [KW92][GHL+92], SSL [RT89], OLGA [JP91]
	Constituent	ALADIN/GAG [KHZ82], Lido/Eli [KW92]
	Pass variable	TOOLS [KNP88][Kos91]
	Target's expression	Door AGs [Hed92], Ag[Gro89] [Gro92]
Collective Computing	Chaining	ALADIN/GAG [KHZ82], Lido/Eli [KW92]
	List attribution	OLGA [JP91]
	Bucket brigade	Regular Right-part AGs [JD84]
	Scan	Scan Grammar [Rep92]

Table 2.4 Common extensions to Attribute Grammar

Some AGs also employs the concept of hierarchy, which in essence is similar to modularity. Attribute Coupled Grammars [Gie88] provides more higher level AG compositions: A compilation is decomposed into a number of AGs, each of which reads an attributed tree and generates an attributed tree. This is an AG extension for multi-pass compiling. Higher-Order AGs [TC90][VSK90] start from another direction: promoting abstract syntax tree to "first class citizens." The model allows an attribute to be an abstract syntax tree called non-terminal attribute (NTA), which can be attributed again.

Some AGs allow more flexible attribute access. Circular AGs [Jon90] allow circular-attribute dependency. The power of fixed-point attribute evaluation can elegantly solve a collection of attributes involved in a dependency circle. In Conditional AGs [Boy96], attribution rules may have

guards. Rules are active only when their guards are satisfied. They can express well-behaved computations that involve circular attribute dependency.

Object-oriented notations have been employed in AGs after Object-Oriented technique gains popularity. OOAG [SK90] is an approach that integrates AGs and object-oriented programming. The static semantic is specified in Ags, the dynamic semantics is defined using message passing, which may cause side effects. Multiple AG inheritance [MLA+00] allows an AG to inherit the specifications from ancestors: adding or overriding specifications from ancestors.

2.1.2.1 Problems with Attribute Grammars

Although AGs have some success in compiler construction and domain specific languages, for example, AG software tools listed in Table 2.4, using AGs in semantic analysis of production-quality compilers is still not widely accepted. As for extracting semantic facts, the Refers-to relation in particular, the problems with AGs are three-fold.

1. Attribute Grammars do not formalize the rules for the Refers-to relation. Instead, they treat the Refers-to relation as a decoration of the parse tree. Data structures and functions similar to those in programming languages are used during to decorate the tree. Only the traversal of the parse tree and intermediate storage is formalized and computed by Attribute Grammar tools.

2. It is hard to validate the Attribute Grammars against language standards. In Attribute Grammars, rules for the Refers-to relation are implemented by a mixture of context free grammar, copying and transfer of attribute and hand coded program. It is hard to collate the rules in the language standards and specification in Attribute Grammars, let alone proving they are equivalent.

3. Attribute Grammars are complex, especially when scaling up to real world programming languages. So far, few production-quality compilers are written using Attribute Grammars, even though the whole semantic analysis of Ada was specified in Attribute Grammar in 1982 [UDP+82].

Due to the above reasons, Attribute Grammars are not widely used in fact extraction.

2.1.2.2 Program transformation based on Attribute Grammars

Program transformation systems can also extract the Refers-to relation. Similar to Attribute Grammars, program transformation systems extract the Refers-to relation as a series of transformation (decoration) of the parser tree [GCD03] [BKMV03]. The program transformation systems surveyed in this thesis include TXL [Cor06] [DCMS02] and Stratego/XT [Vis04].

2.2 Data models for facts in the source code

A *data model* is a collection of high-level data description constructs that hide many low-level storage details [RG00]. For facts extracted from source code, the data model tells us 1) what facts are represented and 2) how to represent them. The documents that address these issues are often referred to as *schema*. Similar to database research, the data models of facts are classified based on their *conceptual model* and *storage model* [Cox]. The conceptual model describes the high level abstraction of the facts. For example, the relational data model of facts describes how the facts (lexical, syntactical and semantic) are represented as relations. These relations contain information about *entities*, such as types, variables, expressions, and about relationships, such as the Refers-to relation between the uses of variables and their declarations. The storage model specifies the storage details. For example, Swagkit [Swa02] represents facts as relations in conceptual model, but stores these relations in a text file using TA format [Hol97a].

2.2.1 Conceptual models for extracted facts

As described in conceptual models, extracted facts have been represented using relational [RG00], objected-oriented [RG00], deductive [RG00] and graph-based data model [CMR92]. The object-oriented model is one example of navigational data model. Other navigational data models are the network model (e.g., used in IDS and IDMS), the hierarchical model (e.g., used in IBM's IMS DBMS), XML DOM data model and so on.

This thesis uses the relational model. In this model, data are stored in relations, which can be thought of as a set of records. A relation has its name, the name of each field, and the type of each field. The operations defined on relations include set operations and relational operations [RG00].

2.2.2 Storage models for extracted facts

The storage models describe the detail about implementation such as whether to use text files or database and what database to use. One conceptual model can be implemented by different storage model. For example, the relational model can be implemented as text files, relational database and Prolog [OK90] [Pro03] rules.

Table 2.5 lists the conceptual model and storage model used in some reverse engineering tools. Comparing with these tools, the data model for ARA is flexible. That is, its conceptual model is relational and its storage model is the storage model used by the software tool that executes the ARA formulas. For example, if we use relational database management system to execute the ARA formulas, then the storage model for ARA is relational.

System	Conceptual model	Storage model
CIA [CGK+95]	ER	Relational
PBS [Holt]	Relational	TA format (ASCII)
Rigi [MOT93]	User-defined domain model	Graph and RSF (ASCII)
Jupiter [CC01] [Cox]	Tagged document	Tagged document
Harmonia [BG00]	Graph (parse tree)	XML document
cppML[MK]	Graph (Parse tree)	XML document
JavaML [Bad00]	Graph (Parse tree)	XML document
ARA	Relational data model	Model used by the software

Table 2.5 The data models for reverse engineering tools

2.2.3 Schemas

Schemas are the documentation that describes the conceptual and storage model of extracted facts. Schemas can be language-specific, that is, applicable to only one language or language-independent, applicable to a set of languages. The Bauhaus schema [KGW98], for instance, models C and a subset of Ada in one joint schema. The Datrix [Bel01] and Columbus [FBMG01] schemas both support a form of generalization across languages that are similar to C++, including C, C++ and Java.

Most schemas for facts are a description document for the form of the data, in terms of a set of entities with attributes and relationships that prescribe the form of the instance data. The schemas aiming to represent all entities and relations in the source code tend to be similar. For example, the independently developed Columbus and Datrix schemas have a lot in common since they are both derived from the C++ grammar, although they differ in their terminology and details.

Datrix Schema is often used during the writing of the thesis and therefore is briefly discussed in the following.

2.2.3.1 Datrix Schema

Datrix Schema specifies how to represent a program as an Abstract Semantic Graph (ASG). An ASG is essential an AST embedded with semantic information. The edges in ASGs are typed and represent different types of relationship between program entities. Datrix Schema includes the following facts from the source code.

1. Lexical facts

The Datrix schema records the line and column position of most items in the AST. File names are stored in special nodes within the AST, at the point where the file is included.

2. Syntactical facts

The Datrix schema has representation for C++ templates, types, functions, and statements. These representations largely follow the AST of the source code.

3. Semantic facts

The Datrix schema has representation of typing information, naming and resolution.

Program entities (such as functions, variables, data types, and source files) are represented as entities, and the relationships between these entities (such as function calls, uses of variables, and instantiations of data types) are represented as relations. Each entity may also have associated attributes that describe properties such as the entity's name or line number.

The Datrix team at Bell Canada implemented a front end that parses C/C++/Java and emits information in VCG (for visualization) [San01], and TA [Hol97a]. In the first half of 2001, the team of Thomas Dean and Andrew Malton built CPPX (C++ Extractor) [DMH01] that analyzes GCC ASGs and generates corresponding Datrix ASGs. Since the data is represented in TA, mathematically speaking, the data is a typed, directed, attributed graph, which we will call simply a typed graph. We can use an entity/relation (E/R) diagram to specify the set of all graphs that a legal Datrix parser could produce.

2.3 Application of relational algebra in software engineering

Relational algebra is widely used in software engineering. Among these applications, program analysis, software repository exploration and software architecture recovery and repair are closely related to ARA. The following gives some detail.

2.3.1 Program analysis

In program analysis, relational algebra has been used in many applications including relation lifting, checking of design rules, cycle analysis, impact analysis, detecting unused components and dead code, detecting similar code and similar classes, points-to analysis. In these applications, relational algebra plays the role of calculation engine for reachability analysis, graph pattern matching, transitive closure, shortest paths and etc.

Ullman [Ull89] first suggested formulating data-flow analysis as database queries. Reps [Rep94] used a deductive database for demand-driven inter-procedural data-flow analysis. Jedd [LH04b] is a Java language extension that rewrites pointer analysis as relational algebra formulas and then solves these formulas using a fast algorithm called BDD. Recently, Lam [LWL+05] describes a database framework that simplifies the development of context-sensitive program analyses. This framework rewrites a large number of analyses as database queries, including C and Java pointer alias analyses, finding buffer overruns and format strings in C programs, Java type inference, and reflection analyses and detecting numerous vulnerabilities in Java web applications.

Relational algebra is also used in detecting graph pattern that are associated with design patterns and design problems of software systems. For example, Beyer et al [BN04] [BNL03] is able to detect circular inheritance among other design defects. Other existing tools also capable of detecting graph patterns include Grok [Hol98], RPA [FKO98], and RelView [BLM02], GraphLog [CM90] and Prolog [OK90] [Pro03].

Many researchers applied relational methods to automatic detection of implementation patterns, object-oriented design patterns, architectural styles, potential design problems, code clones, inductive inference of design patterns [AFC98] [FH00] [MS95] [SSC96].

Transitive closure computation is an extension to relational algebra. It has been the ingredient of some of the above analyses, but is also crucial for dead code and change impact analysis [CGK98] [FKO98]. Computing the difference between two graphs is necessary for checking the conformance of the as-built architecture to the as-designed architecture [FH00] [FKO98] [MNS01], [SSC96]. Another application of relational methods is the lifting and lowering of relations [FH00] [FKO98] to get new abstraction levels, and the calculation of software metrics (e.g., [KW99][MS95]).

Calculation of relations is also important for program analyses like points-to analysis [BLQ+03], and for the implementation of general graph algorithms. Jedd [LH04b] and Lam [LWL+05] rewrite points-to analysis for Java and C as relational database queries and then solve the queries using BDD engines.

2.3.2 Software repository exploration

Software repository exploration tools help users to search the software repository using graphical browsing, a query language or both [LSW01]. Graphical browsing, such as Source navigator [SoN03] usually includes package browsing, class browsing, function browsing and so on [JV03]. Query languages for software repository are often based on SQL, PROLOG and graph theory.

2.3.3 Software architecture recovery and repair

Software architecture is the high level view of components of a software system and the relationship between these components. The Refers-to relation plays important roles in software architecture

recovery [Hol99] [MOTU93] [SCHC99] and software architecture repair [FHM97] [TGLH02] [TH99].

In summary of application of relational algebra in software engineering, the examples of program analysis, software repository exploration and software architecture recovery and repair are listed in Table.

Application	Example
Program analysis	data-flow analysis [Ull89] [Rep94], pointer analysis [BLQ+03] [LH04b] [LWL+05], code defects [LWL+05], design problems [BN04], design pattern [AFC98] [FH00] [MS95] [SSC96], dead code and change impact analysis [CGK98] [FKO98], lifting and lowering of relations [FH00] [FKO98]
Software repository exploration (query languages)	SQL [CGK+95] [KC98], relational algebra [Holt], PQL [Jar98], Jquery [JV03], Jupiter [CC01], ASTLOG [Cre97], Lclint [EGHT94], SCA [PP96]
Software architecture recovery and repair	software architecture recovery [Hol99] [MOTU93] [SCHC99], software architecture repair [FHM97] [TGLH02] [TH99], conformance of the as-built architecture to the as-designed architecture [FH00] [FKO98] [MNS01] [SSC96]

Table 2.6 Examples of application of relational algebra in software engineering

2.4 Relational algebra software tools

The mathematical notation in this dissertation comes from Grok [Hol02], a relational algebra calculator. There are other relational calculators, such as the work on RPA (Relation Partition Algebra) [PS99] and the Relview language [Beh99][BKU96]. A related approach, GReQL, supports queries on graphs. It would be interesting to explore using GReQL where we have used Grok. It would also be interesting to try using a graph grammar system, such as PROGRES [Sch98][SWZ95] in expressing queries such as given in this dissertation. PROGRES has the advantage of supporting visual. Mendelzon's GraphLog system also supports visual querying of graphs [CM90].

In principle, the formulas in this dissertation could be written in SQL, but our experience indicates that such SQL queries are both clumsy to write, and slow to execute, compared with the Grok approach. PROLOG is closer in approach to Grok.

Aho and Ullman argued that an operator for transitive closure is needed for many database applications, which is not supported in relational algebra and calculus [AU79] until SQL99. Crocopat (Beyer et al [BNL03]) provides a language that is as powerful as the previous approaches, but adds a convenient operator for transitive closure; and the interpreter is efficient for general purpose relational computation, because it is based on BDD technology. In addition, Crocopat is relational calculator for n-ary relations, while Grok is for binary relations only.

In summary, Table 2.7 lists these operations and software tools that support these operations.

Category	**Tools**	**Optimizations**
Relational algebra calculator	Grok [Hol02], RPA [PS22], RelView [BBM98] [Beh99] [BKU96], RELAX[MG00]	Specially designed searching and sorting algorithm
RDBMS	Oracle, DB2, …	Rewrite, index
Prolog	PROLOG	Unification
Deductive database	Datalog, graphLog [CM90]	
Graph database	GreQL [Sch98][SWZ95]	
BDD	Jedd[LH04b], bddbddb[LWL+05], Crocopat[BNL03]	Machine learning, profiler…

Table 2.7 Relational algebra software tools

Grok is chosen as the notation for this dissertation because formulas written in Grok tends to be more concise, elegant and insightful. We discovered some interesting facts about programming languages based on these formulas (see Chapter 8). In ARA for C++, we need trinary relations, which is not part of pure Grok. However, we believe the elegance and insight in the Grok formulas is more important to the whole thesis. As a result, Grok is used in the thesis with a minor extension of the composition of n-ary relations.

.

Chapter 3
Completeness of fact extraction

3.1 Introduction

Software reverse engineering extracts and presents information about existing software systems. A key part of this activity is automated by fact extractors which input source code (or other artifacts) and produce facts about the code. These facts can be thought of as rows in a relational data base table, or as edges in a graph. For example, the fact (call, P, Q), could mean that function P calls function Q.

The SWAG group at the University of Waterloo has been involved in developing a number of fact extractors, including CPPX (C++ Extractor) [DMH01][CPP02]. CPPX consists of the GCC front end together with the CPPX graph transformer, shown here as boxes and arrows drawn with solid lines. The GCC front end transforms the source program into a corresponding Abstract Syntax Graph (ASG); see Figure 3.1. The ASG is an abstraction of the program's syntax tree decorated with edges that correspond to resolution of references to declarations and with attributes representing information such as line numbers.

Ordinarily, the back end of GCC proceeds to translate the ASG to assembly language. CPPX operates by replacing GCC's back end by a graph transformer, called cppx (written in lower case to emphasize that it is only part of the CPPX fact extractor). The cppx transformation produces another version of the ASG, which is based on the Datrix schema [Bel01][HHL+00] for representing facts about C or C++ programs. The Datrix schema is designed to be convenient for reverse engineering purposes.

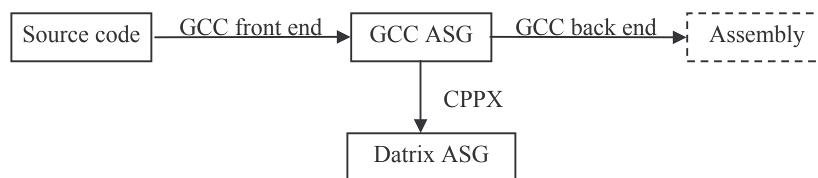

Figure 3.1: CPPX fact extractor.

There has been considerable research on validating fact extractors [MNL96] [AT98] [SHE20] [BG97]. In this chapter we record the approach we used toward validating the CPPX extractor. More generally, we give a method for validating that an extractor is semantically complete, that is, that its extracted facts contain enough information to recover a program with the same behavior as the original source program.

The rest of this chapter is organized as follows. Section 3.2 provides details about compiler phases and ASGs. Section 3.3 provides background information on factbases, schemas and exchange formats. Section 3.4 defines four increasingly detailed levels of facts that an extractor may produce and introduces the concept of relative completenss of extractors. Section 3.5 gives a method for validating the semantic completeness of a fact extractor and explains how we applied this method to CPPX. Section 3.6 discusses the cppx graph transformer. Section 3.7 lists problems in CPPX that were detected as a result of applying our validation method. Finally, section 3.8 summarizes the research and proposes future work.

3.2 Compiler phases and ASGs

In a typical compiler [AU77][App98], the source code is initially preprocessed, parsed, and subjected to semantic analysis. In the case of GCC, these steps are carried out by its front end; see Figure 3.1. Following these initial steps, the assembly or machine code is generated, optimized and emitted. These final steps are carried out by the compiler's back end.

The result of parsing is a parse tree. Lexical details of the source, such as spacing, comments, preprocessor directives, do not appear in the tree. This tree is then simplified and made more convenient for further processing. The simplified tree is called the Abstract Syntax Tree (AST) [AU77][App98].

The semantic analysis phase of compilation decorates the AST by adding semantic information such as types of identifiers, declaration locations and overload resolution. The decorated AST is called the Abstract Semantic Graph (ASG) [Bel01]. After the insertion of semantic information, the syntactic structure may be simplified further:

The nodes of the ASG represent source program entities including types, classes, methods, statements, expressions, and so on down to the lowest level of constants and variable references. The edges represent relationships between them. There are two kinds of edges in the ASG: tree edges and semantic edges.

Tree edges give the tree structure of the ASG. They represent containment in the source syntax. For example, a declaration is contained by its scope, a declared identifier by its declaration, a variable reference by an expression involving it, and a conditional expression by the if statement which it controls. Semantic edges (non-tree edges) represent semantic connections, such as typing and the resolution of scoped names. For example, semantic edges connect the operands of an expression to their declarations and an instance declaration to its class type.

3.3 Schemas and interchange formats

An extractor such as CPPX can be used for a variety of reverse engineering purposes, so its output (the Datrix ASG factbase) should be available in a well documented, accessible format. The format

should determine a concrete syntax, so the ASG can be conveniently input by other reverse engineering tools such as visualizers and analyzers.

A number of exchange formats have been proposed for exchange of reverse engineering factbases, including GXL [Win01][GXL02] (based on XML), TA [Hol97a], RSF [Won98], EER/GRAL [EWB+96] and Grax [EWB+98] [EKW98]. CPPX generates TA by default, with an option to emit GXL. Besides determining a concrete syntax, exchange formats such as TA and GXL allow the user to specify a schema or data model. The schema constrains the relationships between facts in the factbase and is used to give an interpretation of those facts. For example, the schema may constrain Call edges to connect only Function nodes, and its interpretation may indicate that each call in the source code is to have a corresponding Call edge in the factbase. The key benefit of schemas, and not just a fixed exchange format, is that this allows the exchange format to be broadly used, across a set of applications. The user of the format creates a special schema to handle the data of interest, such as the data generated by CPPX.

Examples of fact exchange schemas include the Datrix schema, the Columbus schema [FBT+02] [FBM+01][FSG04] and the Dagstuhl Middle-Level Model [Let02]. CPPX uses the Datrix schema. (Technically speaking, CPPX uses a slightly modified version of the Datrix schema. To simplify the presentation in this chapter, we will refer to these both as simply the "Datrix schema".)

The Datrix Schema specifies how to represent a C or C++ program as an Abstract Semantic Graph (ASG). An ASG is essentially an AST with embedded semantic information. The edges in ASGs are typed and represent different kinds of relationships between program entities. Datrix Schema includes the following facts from the source code.

1. Lexical facts

The Datrix schema records the line and column position of most items in the AST. File names are stored in special nodes within the AST, at the point where the file is included.

2. Syntactical facts

The Datrix schema has representations for C++ templates, types, functions, and statements. These representations largely follow the AST of the source code.

3. Semantic facts

The Datrix schema has representations of typing information, naming and resolution.

Program entities (such as functions, variables, data types, and source files) are represented as entities, and the relationships between these entities (such as function calls, uses of variables, and instantiations of data types) are represented as relations. Each entity may also have associated attributes that describe properties such as the entity’s name or line number.

The Datrix team at Bell Canada implemented a front end that parses C/C++/Java and emits information in VCG (for visualization) [San01], and TA [Hol97a]. In the first half of 2001, the team of Thomas Dean and Andrew Malton built CPPX (C++ Extractor) [DMH01] that analyzes GCC ASGs and generates corresponding Datrix ASGs. An example of such ASGs is provided in Section 3.6.

3.4 Completeness of fact extractors

An extractor is analogous to a compiler in that it inputs source code and translates it to data in a very different form. In the case of a compiler, the target data is assembly or machine language, for use in linking and execution. In the case of an extractor, the target data is facts about the source, for use in reverse engineering.

The extracted factbase may include only high level information, such as interactions between global entities such as functions and classes, or may also contain detailed information down to the level of statements and expressions.

3.4.1 Four levels of completeness

It is useful to characterize the extracted factbase in terms of how complete it is. At the most inclusive extreme, the factbase can be source complete, meaning that it is possible to recover the exact source program, byte for byte, including comments and white space, from the factbase. Most extractors including CPPX are not source complete, because information such as white space is not usually needed and would bloat the factbase with unwanted detail.

A fact extractor E inputs the original source program p0 and produces factbase g0, i.e., g0 = E(p0); see Table 3.1. In this table, FE and BE are the Front End and Back End of a compiler. Sem maps a program to its semantics

In Table 3.1 we define four levels of completeness for an extractor. These levels are a generalization of source completeness as defined by Dean et al. [DMH01]. At each level, completeness is defined by whether the extracted factbase retains enough information to answer a certain question. Figure 3.2 illustrates how the questions in levels 1 to 3 might be addressed by means of testing.

Level	Question	Definition
1. Source complete	Are original and recovered source programs p_0 and p_1 identical, byte for byte, including comments and spacing?	$p_1 = R(g_0)$
2. Syntax complete	Are original and recovered syntax trees t_0 and t_1 identical?	$t_0 = FE(p_0)$ $t_1 = FE(p_1)$
3. Compiler complete	Are original and recovered assembly code a_0 and a_1 identical?	$a_0 = BE(t_0)$ $a_1 = BE(t_1)$
4. Semantically complete	Are original and recovered behaviors s_0 and s_1 equivalent?	s_0=Sem(p_0) s_1=Sem(p_1)

Table 3.1 Four levels of completeness for a fact extractor

3.4.2 Hierarchy of completeness

From top to bottom in Table 3.1, or from left to right in Figure 3.2, completeness becomes weaker, i.e., the levels form a completeness hierarchy in which less information needs to be retained in the factbase as the level number increases.

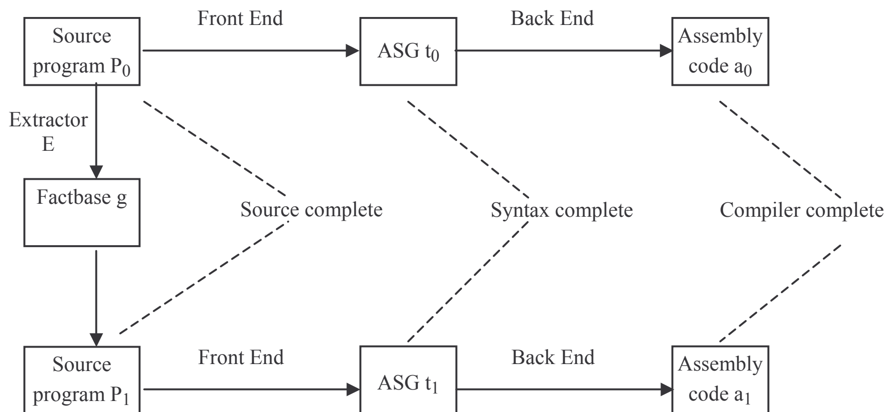

Figure 3.2: Validating three kinds of completeness

If an extractor is source complete, it is also syntax complete, because the extracted factbase g_0 can be transformed back to the original source program, from which the original syntax tree can be derived. By a similar argument, source completeness also implies compiler completeness and semantic completeness.

Syntax completeness implies both compiler completeness and semantic completeness because the syntax tree incorporates all the information about a program that is needed for code generation. Assuming the compiler is correct, compiler completeness implies semantic completeness because if original source and the recovered source have identical generated assembly language then they must have the same semantics.

We have left the definition of syntax completeness somewhat ambiguous, allowing it to be based on either the context free parse tree or on the ASG, whichever is most convenient for a given purpose.

3.4.3 Semantic completeness

The lowest level in Table 3.1 is semantic completeness, which retains information about program behavior. This level is particularly interesting because this information is needed by many reverse engineering tasks. This is the level of completeness that CPPX (supposedly) attains.

In level 4 of Table 3.1, we have assumed that there is a semantics function Sem, which maps a source program to a representation of its behavior. Unfortunately, for production languages such as C and C++, such a function has not been formally defined. Furthermore, the equality of semantics is undecidable: In general we can't check if two programs p_0 and p_1 have the same behavior. Since semantic completeness asks whether p_0 and p_1 are semantically equivalent, it seems that determining if an extractor is semantically complete must be quite a challenge! We take up that challenge in the last half of this chapter.

Many extractors do not extract enough information to satisfy any of the levels in Table 3.1. For example, the CFX extractor [FHK+97] only extracts information at the level of functions and global variables along with the interactions among them. This level is sufficient for certain reverse engineering analyses such as recovery of architecture, but is insufficient for analysis involving function bodies. (We might call this "architecture complete" if we assume that architectural design recovery is based only on functions and global variables.)

CPPX's goal was to retain as much useful information as possible from the GCC ASG, and more information than is retained by most existing fact extractors. As a result, we decided to try to make CPPX semantically complete.

3.4.4 Relative completeness

While the four kinds of completeness we have described are important, they are not the only possible kinds of extractor completeness. Consider the possibility that a use-def graph [ASU86] is needed to detect dead code. We define an extractor to be use-def complete if its created factbase contains enough information to produce a use-def graph.

More generally, we will now introduce the formal concept of relative completeness among translators and extractors, as follows. Suppose a source program p is translated by transformation T

to produce T(p). For example, T might be the front end FE of a compiler or might be an extractor for use-def graphs. Program p is also transformed by extractor E to produce information E(p). We define that E is at least as complete as T if the information produced by E can be further processed to create the information created by T. (see Figure 3.3.) Formally, we define this as follows:

Definition 1. If there exists a function F such that for all p

$$F(E(p)) = T(p)$$

then we say E is at least as complete as T. We write this as:

$$E \geq_c T$$

See Figure 3.3 for illustration of T, E and F.

Although Definition 1 is intuitively appealing, in actual practice, as illustrated by Figure 3.2, we used the following definition involving a recovery transformation R:

Definition 2. If there exists a function R such that for all p

$$T(R(E(p))) = T(p)$$

then we say E is at least as complete as T. We write this as:

$$E \geq_c T$$

This second definition states that E is more complete than T if E's output E(p) can be recovered back to R(E(p)) which T processes to output equivalent to T(p). Fortunately, the two definitions are equivalent, as we will now show.

Proposition. Definitions 1 and 2 are equivalent.

We will now prove this proposition. It is obvious that if R exists, F also exists, because F can be defined in terms of R as

$$F(e) = T(R(e)).$$

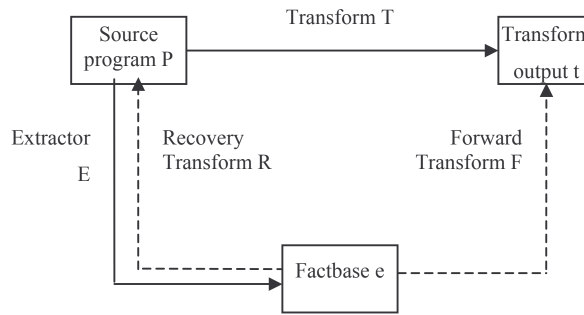

Figure 3.3: Relative completeness of E and T.

Now consider the converse: suppose F exists such that F(E(p)) = T(p). We show that R exists such that T(R(E(p)) = T(p). Suppose T_1(t) is a left inverse of T, that is, T_1(T(p)) = p. Then let R (e) = T_1(F(e)), and we have T(R(E(p)) = T(T_1(F(E(p)) = T(T_1(T(p)) = T(p) as required. Such T_1 exists since it only needs to be defined on the range of T. End of proof.

Aside. Note that if F, E, and T are programs, then R can also be a program: that is, R is computable. Given an extraction e, it is sufficient (for R) to generate programs p in the source language and test them successively until one be found for which T(p) = e. However, this is not very efficient! In practice R examines e, which is a data structure, and constructs p from the information therein.

Although we have explained the concept of relative completeness in terms of extractors and translators, this concept is purely mathematical, and can be applied to any functions. The relative completeness operator (c forms a lattice. Its top element is ID (the identity function):

$$\forall\ T \bullet ID \geq_c T$$

which means that no transform can preserve more information than does the identity function. Its least elements are any constant function K:

$$\forall T \bullet T \geq_c K$$

which means that translating the input program to a constant, such as the null string, loses all information about the input.

The four hierarchical levels of completeness (source, syntax, compiler and semantic completeness) can be defined in terms of relative completeness. For example, extractor E is compiler complete for compiler C if E $\geq_c$ C. E is ASG complete for front end FE if E $\geq_c$ FE. E is source complete for compiler C if E $\geq_c$ ID, which means that the facts extracted by E can be used to reconstruct the source program.

With this discussion of completeness behind us, we are ready to explain how we validated CPPX's semantic completeness.

3.5 Validating CPPX's semantic completeness

This section describes the method we used to validate the semantic completeness of CPPX. Perhaps the most obvious approach to validate CPPX is to run it against a large test suite of source programs, and to check that the extracted factbase for each test is correct. This approach is very expensive, as it requires extensive manual work to do the checking or to create correct factbases for comparison to CPPX generated factbases. As we will explain, our approach to validation of CPPX uses a test suite of programs, but avoids this manual step.

The structure of CPPX was introduced using Figure 3.4. (Recall that CPPX consists of the front end of GCC together with the cppx graph transformation.) Figure 2.4 expands Figure 2.1 by adding a transformation called RCCPX (Reverse CPPX), which recovers a source program p_1 from CPPX's extracted ASG g_0. The figure shows recovered source p_1 being compiled by GCC's front end and back end to produce assembly code a_1.

If all phases in the diagram are working correctly and CPPX is compiler complete, then the assembly language a_0 from any original source p_0 will be the same as its recovered assembly language a_1 from recovered source p_1. As is explained below, our validation method for CPPX works by checking that a_0 and a_1 are identical for a suite of test source programs.

3.5.1 Why are assembly codes a_0 and a_1 identical?

Suppose we run a source program p_0 through the phases shown in Figure 3.4. If we find that a_0 and a_1 are identical, what can we conclude? It is tempting to conclude that this implies that cppx is semantics complete. However, this might not be the case. If GCC's back end is severely buggy and acts as a constant function K, producing null output for all input, then a_0 and a_1 will always be identical, regardless of the actions of CPPX. Since GCC is widely used in production, it is clear that its back end is not severely buggy. In fact, we can safely assume that its back end and front end together reliably generate correct assembly output for most input source programs.

As we will explain below, the recovery transformation RCPPX is written to be simple and easy to make correct, so we can expect that it is reasonably reliable. So, we can conclude that if there is a failure in one of the phases shown in Figure 3.4, the failure will probably occur in the least reliable phase, namely, in the CPPX transformation phase.

Even if CPPX is semantically complete, it would be wrong to expect a_0 and a_1 to be identical. After all, GCC is an optimizing compiler, and the smallest change in its input, as p_0 varies to p_1, may produce a change from a_0 to a_1 while maintaining the same semantics. So, it may come as a surprise to learn that when we run test source programs though the configuration in Figure 3.4, a_0 and a_1 turn

out to be identical (ignoring the cases when we encounter a bug in CPPX). We will now answer the question: Why are a_0 and a_1 identical?

Although GCC's back end is extremely complex, it is deterministic, i.e., for the same input it always generates the same output assembly code. So, if the GCC ASGs t_0 and t_1, for the top and bottom lines in Figure 3.2, are the same, then a_0 and a_1 will be identical. So the question becomes: Can we expect the GCC ASGs to be the same for source programs p_0 and p_1?

To our mild disappointment, we discovered that GCC's front end does some low level program transformations.

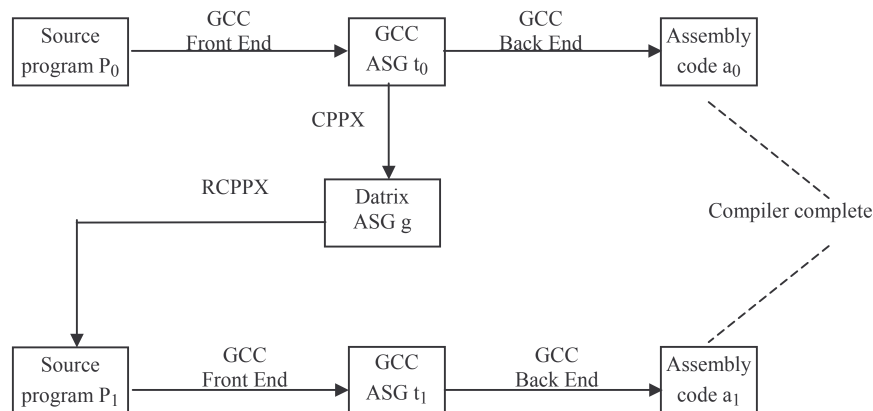

Figure 3.4 Validating Completeness of CPPX.

For example, it rewrites the integer comparison j >= 2 into j > 1. Because of this, one might conclude that it is unlikely that the two GCC ASGs t_0 and t_1 can be identical. It turns out that this conclusion is not warranted for the following reason.

Apparently, the purpose of these rewrites is to produce a standard or canonical form for a few special constructs, and that these rewrites are idempotent, i.e., doing them a second time has no effect. If we denote this rewriting as function f, we can expect or hope that, $f(p_0) = f(p_1)$. What we observed was that, with CPPX working properly, the two ASGs were effectively the same, i.e., they were treated the same by the back end. The idempotent property of the GCC front end together with the deterministic nature of the GCC back end yields the overall property that, when CPPX is operating correctly, a_0 and a_1 are identical.

After our initial validation of CPPX was largely finished, we ran tests to see if ASGs t_0 and t_1 were the same. We found that they were very similar but not identical. They differed in their information about file and line location. They also differed due to renaming of local variables, as described below. These differences do not affect the generated code. Probably they could be eliminated by deleting file and line information and by optionally disabling the renaming of local variables, but we have not tried this.

In short, CPPX is almost ASG-complete, but our method of validation only shows the weaker property of compiler completeness. Compiler completeness seems to be more "surprising" and satisfying as its definition doesn't rely on the internal details of the compiler.

3.5.2 Suite of test programs

In our validation of CPPX, we used a test suite of C and C++ programs derived from two sources. The first part of this suite consists of a set of small programs that were designed to show how source programs are represented when using the Datrix schema. The second part consists of the suite of C and C++ programs used to test the GCC compiler. Combined, these two parts provide an initial reasonable coverage of the features of C and C++. (Note that we have thus far concentrated on validating CPPX on C rather than on C++ input.)

3.5.3 The resolution problem

As we have explained, CPPX generates an ASG in which tree edges represent the containment structure and semantic edges give connections in the tree. One key use of semantic connections to represent references to declarations. For example, in the following program, there is a reference from the final x on line 4 to the local declaration of x (line 3), not to the global declaration of x (line 1).

```
1       int x;
2       int f() {
3               int x;
4               x++;
5       }
```

In the corresponding ASG, there is an edge from the node for the final x to the node for the declaration of the local x and not to the global declaration. This edge is redundant in the sense that the scope rules of the programming language imply which declaration the final x is referencing. Despite this redundancy, such edges are important because they allow the resolution problem (the sometimes difficult problem of resolving appropriate references) to be reliably solved once, in the extractor. Indeed, one of the reasons CPPX is based on GCC is to take advantage of the fact that GCC's ASG provides a reliable solution to this problem.

Now consider the possibility that CPPX might incorrectly solve the resolution problem, for example, by generating the reference edge from the final x to the global x instead of to the local x. Unfortunately, this error would not be detected by our validation method for CPPX, because in the

recovered source program p1, there is no record of this edge and hence no evidence of the error. Although CPPX creates this edge, RCPPX deletes it when recovering the source program.

We devised the following approach to allow our validation method to check that resolution edges in the factbase are correct: local (non-external) identifiers are systematically renamed to be unique, by suffixing additional characters. For the above example, the recovered program p1 would have this form:

```
1    int x;
2    int f() {
3        int x_2;
4        x_2++;
5    }
```

This renaming is done in way that avoids clashing with any global identifiers which might have a similar form. If the global had been called 'x_2' then this would of course not be chosen as the new name of the local. With this suffixing convention, if CPPX generates incorrect reference edges, the resulting semantic errors will be reflected in the recovered code. As a result, our validation method effectively checks that reference edges are correct, and that the CPPX has correctly solved the resolution problem.

3.6 RCPPX: Recovering source from facts

As can be seen in Figure 3.4, to use our validation method, we need a recovery transformation, namely we need RCPPX. RCPPX inputs a Datrix ASG in the TA exchange format and outputs the equivalent C/C++ source program. TA is a relational notation, whose underlying data model is much like a Relational Data Base. In TA, there is a set of triples, recording the source, target and type of each edge in a graph. In this case, the graph is the Datrix ASG. As well, there are triples that represent attributes. In TA, this information is encoded as an ASCII stream, stored in a flat file.

The following is a tiny C compilation unit, which will serve as an example:

```
/* A tiny C program */
int x;
void glup (int y) {
    x = y + 1;
}
```

The CPPX extractor translates this program to TA notation (simplified here for presentation purposes) as follows:

```
FACT TUPLE :
```

$INSTANCE @0 cFunction

cRefersTo @2 @3

$INSTANCE @2 cNameRef

$INSTANCE @3 cObject

$INSTANCE @4 cFormalObject

$INSTANCE @5 cBlock

$INSTANCE @7 cBuiltInType

cRefersTo @9 @4

$INSTANCE @9 cNameRef

$INSTANCE @10 cBuiltInType

contain @40 @2

$INSTANCE @15 cBuiltInType

cInstance @3 @10

contain @40 @55

contain @55 @9

contain @55 @66

contain @0 @5

cInstance @0 @7

cInstance @4 @10

contain @0 @4

cInstance @66 @10

contain @5 @40

$INSTANCE @40 cOperator

$INSTANCE @55 cOperator

$INSTANCE @66 cLiteral

contain @577 @3

contain @577 @0

$INSTANCE @577 cScopeCompil

FACT ATTRIBUTE :

@0 { name = glup }

@2 { name = x }

@3 { name = x }

@4 { name = y }

@7 { name = void }

@9 { name = y }

@10 { name = int }

@40 { op = asgn-eq }

@55 { op = bplus }

@66 { value = 1 }

This TA represents a graph, which is diagrammed in Figure 3.5. By comparing the source program with its image in TA (or with the diagram in Figure 3.5), it can be seen that CPPX preserves the structural information, but deletes comments and layout. Syntax in the source code is replaced by explicit relationships in TA.

It is the job of RCPPX to bridge the gap from TA back to source code. RCPPX does this in three steps, first by a renaming script, second by transforming to nested syntax, and third by two structural transformations implemented in TXL [CDM+02].

3.6.1 Suffixing script

This first step modifies program identifiers as explained in Section 3.5.3 above, in order to validate the extractor's computation of semantic edges. The result in the TA for the sample program is to replace

@4 { name = y }

by

@4 { name = y_1 }

and similarly for node @9.

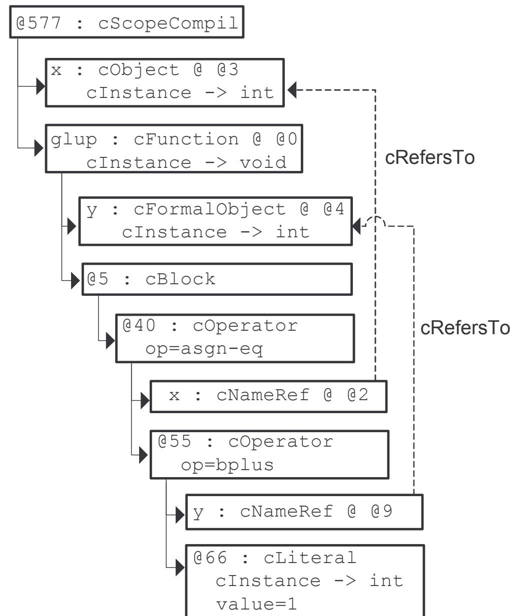

Figure 3.5: TA from example C program as a diagram

3.6.2 Nested syntax

In the second step, the ASG TA is translated to nested syntax. This was done with the Grok relational calculator [Hol02] [Hol98][Wu04]. Grok stores and manipulates a relational model using algebraic operators, and can read and write TA. In order to reveal syntactic structure, Grok's operation showtree outputs the relational model in a form in which tree edges are represented implicitly by nesting of braces, { and }. In the Datrix schema the tree edges are contain and cInstance. Our example represented in showtree format exhibits the tree structure once again represented by syntax, as seen here:

```
@577 : cScopeCompil
{       x : cObject @ @3
                ( cInstance -> int @ @10 )
        glup : cFunction @ @0
                ( cInstance -> void @ @7 )
        {  y_1 : cFormalObject @ @4
                ( cInstance -> int @ @10 )
                @5 : cBlock
```

```
{ @40 : cOperator { op=asgn-eq }
        { x : cNameRef @ @2
                ( cRefersTo -> x @ @3 )
                @55 : cOperator { op=bplus }
                        { y_1 : cNameRef @ @9
                                ( cRefersTo -> y_1 @ @4 )
                                @66 : cLiteral
                                        { value=1 }
                                ( cInstance -> int @ @10 )
                        }
                }
        }
}
}
```

3.6.3 TXL scripts

The final step of RCPPX is carried out by two sets of TXL scripts [DCM+02]. TXL is a source transformation system based on context-free structure of input and output.

The first TXL script reads its input in the format shown above. This format is structurally similar to the original program (TXL understands programs), rather than to the fact base (TXL does not understand data bases). The key idea is to use syntax (relative position in the token stream) to represent structural information, instead of using explicit relational edges.

TXL is well suited to translation from input according to one grammar into output according to a second grammar. TXL first inputs a union grammar [DCM+02] that encompasses the grammars of both the input and the output languages. Driven by a command script, the TXL processor repeatedly manipulates the parse tree of the input, until the tree assumes the form specified by the output grammar. Then TXL outputs the result as a source program, which satisfies the output grammar.

RCPPX drives TXL with a first set of scripts, which start by reading the ASG as represented in showtree format. These scripts carry out a pattern of simple local rewrite transformations in which subtrees are locally manipulated. For example, an input showtree expression of the form

```
cOperator { op = bplus }
{ E1 E2 }
```

is transformed to this C/C++ expression

```
(E1 + E2)
```

Here is the TXL rule that carries out the transformation:

```
replace [operator]
        cOperator { op = bplus }
        { E1 [expression] E2 [expression] }
by
        (E1 + E2)
```

This replaces each operator in the program of the form

```
cOperator { op = bplus } { E1 E2 }
```

with

```
(E1 + E2)
```

Next, RCPPX uses a second set of TXL scripts to carry out local rotation transformations, which change the order of entities. For example, an input showtree fragment of the form

```
p : cObject
{       cPtrType
        {  int : cBuiltInType
        }
}
```

is transformed to a C/C++ fragment of the form

```
int *p;
```

This replaces cPtrType by a star (*) and "rotates" the fragment so that type int precedes the star and the star precedes identifier p. As well, the transformation deletes unwanted punctuation and keywords, and adds a semicolon. Here is the TXL rule that does the replacement and the renaming:

```
replace [object]
        cObject  cPtrType
        { T [type] }
        D [declarator]
        construct ND [declarator]
                * D
```

by

cObject T ND

In this rule, T corresponds to a type, such as int, and D corresponds to an identifier, such as p. The rule constructs ND as a temporary value consisting of a star preceding D. The result of the transformation is

cObject T ND

whose value is

cObject T * D

This rule is executed repeatedly and recursively to replace each cPtrType by a star and to rearrange terms. The transformations that rewrite the ASG in TA and showtree notation to source programs in C/C++ notation could have been written in a language such as PERL or C. However, they are much easier to write and debug in a special syntactic transformation language such as TXL. Since they are straightforward, and are done in very high level notations, we have confidence that they are carried out reliably.

3.7 Errors found in CPPX

Before we developed the validation method described in this chapter, CPPX had been used to extract facts from a number of large systems, including PostgreSQL, which consists of about 400,000 lines of source code. Because of this application of CPPX we knew that it was useable, but also that it had various bugs. This knowledge motivated us to develop the validation method described in this chapter.

Here are three examples of the kinds of bugs we found in CPPX using our validation method.

1) Headers of for loops. A for loop header, e.g.,

```
for (i = 1; i < 10; i ++)
```

has, at most, three fields. For example, the third field is omitted here:

```
for (i = 1; i < 10)
```

We found that when CPPX produced facts for a header with omitted fields, these facts did not specify which fields were missing.

2) Details about data types. We found that CPPX did not correctly handle certain details about data types such as struct and enum type.

3) Missing identifiers. We found that CPPX sometimes omitted certain identifiers, such as names of structs.

We expect to find and correct more bugs in CPPX as our test suite grows and becomes more comprehensive.

3.8 Conclusions and future work

This chapter presents a hierarchy of completeness for fact extractors and introduces the concept of relative completeness of extractors. It gives a new method for validating a fact extractor. The method avoids manual checking of generated fact bases. It works by recovering a version of the source program from the extracted factbase and compiling that version.

Given certain properties of a compiler (idempotent front end and deterministic back end), the method is able to validate semantic completeness of the extractor by checking that the generated assembly language is identical for the original and recovered source programs. This method was applied to the CPPX extractor. The correctness of name resolution in the factbase was checked by the artifice of suffixing a number (a node key) to local identifiers.

In our application of this method to CPPX, we created a recovery transformation RCPPX. We did this using high level tools including TXL and Grok, so RCPPX is small and sufficiently reliable. Our validation of CPPX has thus far been limited to source programs written in C; we expect to extend this work to C++ programs in the future. Our validation test suite has been limited to programs of modest size, to allow us to monitor results as needed. We plan to scale up these tests into a fully automatic testing framework. We have an experimental Java extractor, but thus far it does not produce facts at the level of statements and expressions. When it is extended to extract these facts and to be semantically complete, we will be able to apply our validation method.

Our work involved two major surprises. First, we were surprised to discover that the assembly code for an original source program and for the version of the source program recovered from its extracted factbase were identical. Second, we were surprised to find a mechanical test for a special case of a generally undecidable problem. In particular, our method tests the semantic equivalence of two programs, by using properties of the programs and the compiler. Combining these two surprises, we developed a method for an automatic validation for semantic completeness of a fact extractor.

Chapter 4
Algebraic Refers-to Analysis (ARA) as a new approach to fact extraction

Part 2 of this thesis consists of this chapter (Chapter 4) through and including Chapter 8. This Part presents a new approach to fact extraction called Algebraic Refers-to Analysis (ARA). Part 2 is organized as follows.

- Chapter 4 introduces ARA as a general approach to formalizing fact extraction, along with background information about traditional approaches.
- Chapters 5, 6 and 7 apply ARA to the C, Java and C++ languages, respectively.
- Chapter 8 summarizes the case studies that apply ARA to seven languages, validates ARA against language standards and clarifies several issues related to the Refers-to relation.

The present chapter (Chapter 4) is organized into the following sections.

4.1 **Traditional approaches.** This section surveys how fact extraction has traditionally been done.

4.2 **Motivation.** This section gives our motivation for developing a new approach to fact extraction.

4.3 **Mathematical notation.** This section introduces the algebraic notation for manipulating relations, which underlies our approach.

4.4 **Program definition**. This section defines the Refers-to relation using concepts in set and relation theory.

4.5 **Introduction to ARA.** ARA has two pipelines, three-step pipeline for the simplest languages and multi-phase pipeline for practical languages such as C and Java. This section introduces the three-step pipeline using ALGOL 60-like scoping rules as an example. At the end of the section, multi-phase pipelines are introduced.

4.6 **Validation of ARA**. This section explains that ARA is based on a particular language standard and can be validated to actually implement that standard.

4.7 **Chapter summary**. This section summarizes the chapter.

4.1 Traditional approaches to fact extraction

In Reverse Engineering, *facts* are essential information about the source code. *Fact extraction* is the process of analyzing the source code and exacting appropriate facts. This section surveys current approaches to fact extraction and explains their drawbacks.

4.1.1 Facts and their classification

Facts in the source code fall into three groups, *lexical facts*, *syntactical facts* and *semantic facts*. Lexical facts are about tokens, separators, layout, and compilation units. Syntactical facts include features of the language specified by the language's grammar (usually context-free) and are often stored in a parse tree or an Abstract Syntax Tree (AST).

Semantic facts are either *static semantic facts* and *dynamic semantic facts*. Static semantic facts include the Refers-to relation (the terms name resolution and name analysis are also used to refer to similar facts) and type information. Dynamic semantic facts include points-to relation, runtime polymorphism and so on. In the field of reverse engineering, these facts are usually represented as entities and relations among them. In general, static semantic facts are decided at compile time while dynamic semantic facts are decided at run-time. Therefore, fact extractors, which are similar to compiler front ends, only extract static semantic facts.

*Schema*s, e.g. the *Datrix Schema* [Bel01] are the documents that determine the form of the facts in the source code as entities and relations. Table 4.1 shows examples of entities in a program (adapted from the Datrix Schema). As shown in the table, occurrences of identifiers and blocks in the source code are lexical facts. The data type definitions, variables, function definitions, labels, expressions and statements are syntactic facts.

Type	Entity Type	Description
Lexical	Occurrence	Occurrence of an identifier
	Block	Global block, function body, data type definition, etc
Syntactic	Data Type	Including built-in type, array type, pointer type, etc
	Variable	Instance of a data type
	Function	Function definition (with a body) or function declaration
	Label	Labels in the program
	Expression	Including binary expression, unary expression , etc.
	Statement	Including if, switch, loop and other statements

Table 4.1 Example entities in source code

Table 4.2 shows some relations, also adapted from the Datrix Schema. The relations in rows 1 through 6 are lexical facts and syntactic facts. The Refers-to relation belongs to semantic facts. We will revisit them when we present ARA.

4.1.2 Importance of Refers-to relation

The Refers-to relation represents semantic facts, more specifically, static semantic facts. This relation is important to fact extraction for the following reasons.

1) References, as formalized by the Refers-to relation, are a fundamental concept in programming languages. Programming language specifications, textbooks and critics of programming languages go to some length to discuss this concept. And, extracting the Refers-to relation is an important task in semantic analysis phase of the compiler [App98].
2) We can derive other static semantic facts from the Refers-to relation and lexical and syntactical facts. The derivation can be written as relational algebra formulas and executed on many existing software tools such as RDBMS and relational algebra calculators.
3) The Refers-to relation as extracted from the code for a software system is important in understanding the architecture of the systems [BHB99] [FHC01]. And, it can be used to recover and repair the software architecture of large software systems such as the Linux operating system [FHC01].

Type	Relation	Description
Lexical	HasName (HN)	The relation between the occurrence and the identifier
Syntactic	Inherits (I)	Relation between a class and its base class
	Denotes (T)	Relation between an occurrence and the block it denotes
	Contains (C)	Relation between a block and its members
	Specifies (SP)	Relation between type specifier and declarator
	Qualifies (Q)	Relationship between the qualifying occurrence and the qualified occurrence
Semantic	Instance-Of	Type-instance relationship between two elements
	Points-To	Relationship between a pointer and the object it points to
	Refers-to (R)	Relationship between an referential occurrence and the definitional occurrence it refers to

Table 4.2 Example of relations in source code

4.1.3 Traditional approaches to extracting Refers-to relation

Traditionally, there have been two approaches to specifying the extraction of Refers-to relation, namely: (1) *ad hoc* methods and (2) Attribute Grammars, each having some significant drawbacks. These two approaches will now be discussed.

4.1.3.1 Ad hoc methods

Most programming standards specify the Refers-to relation informally, for example, ISO/IEC standard for C++[C++03]. These standards communicate the rules for the language at the level of abstraction in an ad hoc manner. Their informal rules are the foundation for designing and teaching the language, building compilers, and creating test suites. Unfortunately, there is no software tool that

can implement a fact extractor directly from these rules. Programmers have to implement the fact extractor manually, which is time consuming and error prone.

4.1.3.2 Attribute Grammars

Attribute Grammars [Knu68] were introduced in 1968 and are viewed as a rigorous formalization for compiler construction. Chapter 2, on related work, details the development of Attribute Grammars over the years. While Attribute Grammars have been formalized and shown to have promise in compiler construction, they are generally not been used in extracting the Refers-to relation for three reasons.

1) Attribute Grammars do not formalize the rules for the Refers-to relation. Instead, they treat the Refers-to relation as a decoration of the parse tree. Data structures and functions similar to those in programming languages are used during to decorate the tree. Only the traversal of the parse tree and intermediate storage is formalized and computed by Attribute Grammar tools.
2) It is hard to validate the Attribute Grammars against language standards. In Attribute Grammars, rules for the Refers-to relation are implemented by a mixture of context free grammar, copying and transfer of attribute and hand coded program. It is hard to collate the rules in the language standards and specification in Attribute Grammars, let alone proving they are equivalent.
3) Attribute Grammars are complex, especially when scaling up to real world programming languages. So far, few production-quality compilers are written using Attribute Grammars, even though the whole semantic analysis of Ada was specified in Attribute Grammar in 1982 [UDP+82].

Considering these three reasons, Attribute Grammars seem not to be convenient for specifying fact extraction.

This completes our survey of traditional approaches to fact extraction. We conclude that current approaches cannot conveniently formalize the extraction of the Refers-to relation. In considering these approaches and their shortcomings, we decided to develop a new approach.

4.2 Motivation toward a relational approach to fact extension

My search for a new approach to fact extraction is motivated by the relational data model [Cod70]. Introduced by E. Codd in 1970, this model is very simple yet powerful. As described in [RG00],

> A database is a collection of one or more relations, where each relation is a table with rows and columns. … The major advantages of the relational model over the older data models are its simple data

representation and the ease with which even complex queries can be expressed.

But what makes the relational model relevant in this thesis is that the relational model was introduced to remedy shortcomings similar to the shortcomings of Attribute Grammars. The data models widely used before relational model are the hierarchical model (e.g., used in IBM's IMS DBMS) and the network model (e.g. used in IDS and IDMS). In these models, data are stored in tree structures or lattices. Queries to the database of these models are solved by navigating the trees or lattices [Bac73], which is in essence similar to a decorated parse tree (which is specified in Attribute Grammars).

Since its introduction to the database management systems, which is two years later than Attribute Grammars were introduced, the relational model has laid solid foundation for the theory of databases and has dominated the market. More importantly, with extensions such as transitive closure and other operations, the relational data model has been used to solve a wide range of problems in reverse engineering and program analysis, including code repository exploration [FHK+97] [Aca96], software architecture recovery and repair [BHB99] [FHC01], pointer analysis [WL04] and code defect detection [LWL+05].

Motivated by the above research, the author invented Algebraic Refers-to Analysis (ARA). ARA uses a whole new diagram in specifying fact extraction. It differs from Attribute Grammars in three ways. First, in Attribute Grammar, fact extraction is a process of decorating the parse tree. In ARA, fact extraction is a query to find pairs of occurrences of identifiers that satisfy certain conditions. For example, one of the conditions requires that two occurrences of identifier have the same name.

Second, Attribute Grammars specify the transportation of data on the parser tree, while ARA represents facts in the source code as relations and stores them in a relational database or an ASCII file.

Finally, Attribute Grammars require programmers to write code to implement the rules on the Refers-to relation. ARA translates these rules into relational algebra formulas and runs them on relational database or other existing software tools. Because relational algebra is closely related to predicate logic [RG00], ARA formulas are similar to rules written in natural language and therefore are easy to be verified against language standards. In addition, many software tools including relational database management systems support ARA formulas. As a result, the major parts of ARA require no coding.

Since we will be using a relational approach as a basis for fact extraction, we will now introduce mathematical notation for representing and manipulating relations.

4.3 Mathematical notation: Relational algebra

This section introduces the notation for relational algebra that is used in this thesis. The notation is in general compliant with from Grok [Hol98] as extended with N-ary relational operations. In this

dissertation, a tuple of elements x, y, ... and z is written as <x, y, ... z>. A path of nodes x, y, ... to z is written as x-y-...z.

4.3.1 Binary relational algebra

Binary relations are sets of 2-tuples. Binary relational algebra [Tar41] is the relational algebra on binary relations. All set operations (Union, Intersection, Difference) are applicable on binary relations. In addition, this dissertation uses the following operations.

The *set identity* of set S, *id S*, is the relation $\{<x, x> \mid x \in S\}$. In the Grok relational calculator, this relation is denoted as ID (S).

The *relation inverse* of relation R, R^{-1}, is the relation $\{<y, x> \mid <x, y> \in R\}$. If edge x-y represents the tuple <x, y> in relation R, then edge y-x represents the tuple <y, x> in R^{-1}.

The *set projection* of relation R on set S, *S . R*, is the set $\{y \mid x \in S \wedge <x, y> \in R\}$. Similarly, *R . S* is the set $\{ x \mid y \in S \wedge <x, y> \in R\}$.

The *relational composition* of relations S and T, *S o T*, is the relation $\{<x, y> \mid \exists z, <x, z> \in S \wedge <z, y> \in T\}$. The relational composition can be visualized as Figure 4.1.Note that relational composition of relation S and T is the set of starting and end points of paths (e.g. 1-2-3) composed of one edge in relation S (e.g. 1-2) and one edge in relation T (e.g. 2-3).

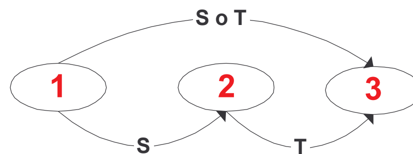

Figure 4.1 Relational composition

For convenience, we write R o R as R^2, R o R o R as R^3 and so on. R^2 is the set of the starting point and end point of paths composed by two edges in R, and R^3 is the set of the starting point and end point of paths composed by three edges in R.

In addition, suppose that S is a set and R is a binary relation and that we want to compute the composition of the identity relation of S and R. We can write

(id S) o R

or simply

S o R

and conversely R o S.

In reverse engineering, we only deal with finite relations and it is guaranteed that the fixed point can be reached when certain operations are repeated for sufficiently many times. In this dissertation, two kinds of fixed point operation are used, Transitive closure and reflexive transitive closure. The *transitive closure* of Relation R, R^+ is defined as

$$R^+ \equiv R \cup R^2 \cup R^3 \cup \ldots \text{until a fixed point is reached.}$$

The *reflexive transitive closure*, R* is defined as ID $\cup$ R^+ where ID is the identity relation of all elements in R. In terms of the directed graph, R* is the set of starting and end points of paths composed of zero or more edges in R. R^+ is the set of starting and end points of paths composed of one or more edges in R.

4.3.2 N-ary relational algebra

This dissertation only uses one n-ary relational algebra operation: join, more precisely *natural join* [RG00]. Suppose we have relations S <w, x, y> and T <x, y, z>. The natural join of S and T is written as

$$R \langle w, x, y, z \rangle = S \langle w, x, y \rangle \otimes T \langle x, y, z \rangle$$

It means

$$R = \{\langle w, x, y, z \rangle \mid \langle w, x, y \rangle \in S \wedge \langle x, y, z \rangle \in T\}$$

In addition to relational algebra, the thesis also uses the standard set operations such as Union ($\cup$), intersection ($\cap$), cross-product ($\times$) and difference ($-$). We have summarized the relational algebra notation used in this dissertation. Now, we begin to present ARA.

4.4 Problem definition

ARA defines the extraction of the Refers-to relation as a query to find a set of pairs of occurrences of identifier <x, y> that satisfy four *Binding Conditions,* which will be introduced in Section 4.5. This problem definition is drawn heavily from the ISO C++ language standard [C++03] and Java language specification [GJSB05]. And the thesis will show that this definition is suitable for other statically scoped languages such as C, Fortran, Pascal and Ada.

4.5 ARA pipelines

ARA extracts the Refers-to relation using two pipelines, the three-step pipeline for the simplest languages and multi-phase pipeline for practical languages such as C and Java. This section focuses on the three-step pipeline but will briefly introduce the multi-phase pipeline at the end.

4.5.1 Three-step pipeline

As explained in Chapter 1, for simplest languages, ARA extracts the Refers-to relation in three sequential steps, Basic Fact Extraction, Normalization and Binding. We call the three steps the three-step pipeline (see Figure 4.2).

Figure 4.2 The three-step pipeline

We use Example 4.1 to explain the three-step pipeline. For convenience, the occurrences and blocks are assigned unique numbers. To simplify the presentation, we use the scoping rule similar to AlGOL 60. That is, an occurrence can refer to a declaration in an enclosing blocks and the declaration in the innermost enclosing block is used if there are more than one declaration in the enclosing blocks. For example, occurrence 5 can refer to occurrence 1 and 4. But occurrence 4 is in the innermost enclosing block of occurrence 5, so occurrence 5 refers to occurrence 4. Occurrence 8 can only refer to occurrence 1. Based on ARA's definition of fact extraction, our goal is to compute the set of pairs {<5, 4>, <8, 1>}.

int x_1;
foo_2( ){$_3$
    int x_4;
    x_5;
$_3$}
bar_6( ){$_7$
    x_8;
$_7$}

Example 4.1 C program

4.5.1.1 Basic Fact Extraction

In ARA, the extraction of the Refers-to relation is a query to find occurrences that satisfy the Binding Conditions. To solve this query using relational algebra, ARA first collects the useful lexical and syntactic facts and stores them as relations. This step is called Basic Fact Extraction.

The facts in Example 4.1 can be represented as relations as shown in Table 4.3. The lexical facts are represented as relation *HasName*. The occurrence 1 has name of “x”, therefore, <1, “x”> is an element of relation *HasName*.

The syntactical facts are stored in two relations, *HasKind* and *Contains*. The occurrence 0 is the global block, i.e. the occurrence 0 has the kind of block and <0, “block”> belongs to relation *HasKind*. The occurrence 1 is a definitional occurrence of variable x, so it has the kind of “def_var” and <1, “def_var”> is an element of relation *HasKind*. The occurrence 5 is a referential occurrence of variable “x”, and <5, “ref_var”> belongs to relation *HasKind*. The relation *Contains* is the relation between a block and entities immediately contained in it. The global block (block 0) contains occurrence 1, therefore, <0, 1> belongs to relation *Contains*.

Relation	Content
HasName (HN)	<1, “x”>, <2, “foo”>, <4, “x”>, <5, “x”>, …
HasKind (HK)	<0, “block”>, <1, “def_var”>, <2, “def_fun”>, <3, “block”>, …
Contains (C)	<0, 1>, <0, 2>, <0, 3>, <0, 6>, <3, 4>, <3, 5>, …

Table 4.3 Facts in Example 4.1.

The relations in Table 4.3 are all binary relations. Such relations can be depicted as a directed graph, which will be used often in the rest of the thesis. We can draw a binary relation as a directed graph. The graph for HasName for Example 4.1 is Figure 4.3. As we can see, occurrences 1, 4, 5 and 8 have the name of “x”. Occurrence 2 has the name of “foo”. Occurrence 6 has the name of “bar”.

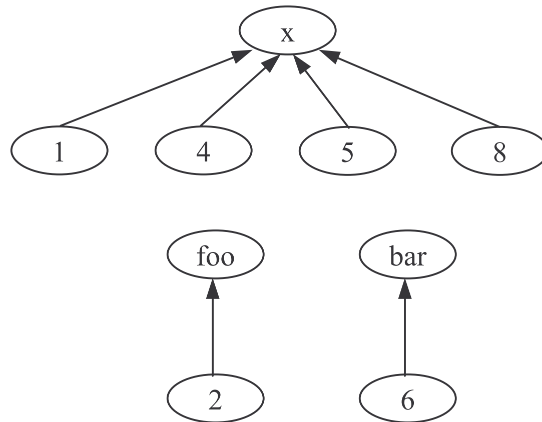

Figure 4.3 HasName for Example 4.1

The second relation in Basic Facts (Table 4.3) is HasKind (HK), the binary relation between an occurrence and its kind. HasKind for Example 4.1 is shown in Figure 4.4. It shows that occurrences 1 and 4 are definitional occurrences and occurrences 5 and 8 are referential occurrences.

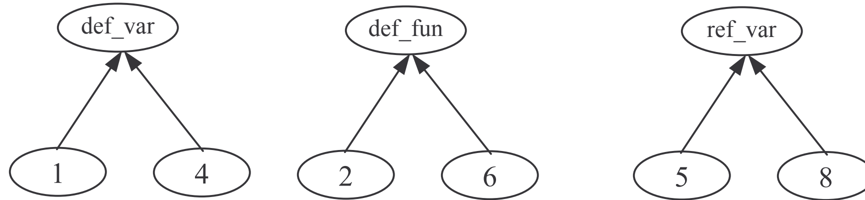

Figure 4.4 HasKind for Example 4.1

The third and last relation in Basic Facts (Table 4.3) is Contains (C). It is a binary relation between a block and the blocks or occurrences that it contains directly. Contains for Example 4.1 is Figure 4.5. It shows the containment relationship between blocks and occurrences. For example block 0 contains occurrence 1 and block 2. With a parser generator such as Yacc, we can create a parser that extracts these three relations straightforwardly.

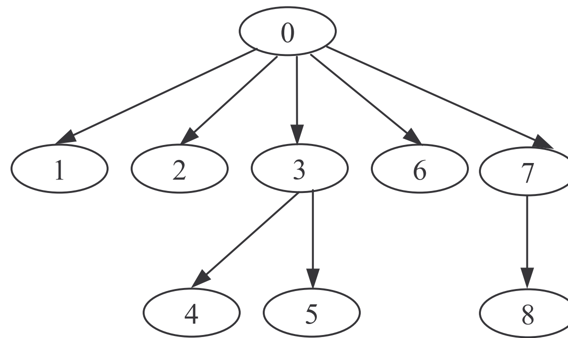

Figure 4.5 Contains for Example 4.1

4.5.1.2 Normalization

Normalization converts facts (Basic Facts) from being language specific into a form in which an (almost) standard set of binding transformations can create the Refer-to relation. Normalization uses a set of relational algebra formulas to convert the Basic Facts into the relations SameName(N), CompatibleKind(K), Visibility(V) and Hiding(H). These four relations are called the *Normalized Facts* and are defined in almost the same way for all languages. Each language has its own rules as to what kinds of occurrence are compatible, what regions of code text can be referred to by an occurrence and so on. ARA translates these rules as relational algebra formulas and uses them in this step.

SameName (N). N is the set of pairs of nodes that have the same name. This relation can be constructed by the following relational algebra formula.

$$N = HN \text{ o } HN^{-1}$$

where HN is relation HasName (see Figure 4.3). The formula means that the relation SameName (N) is the set of starting and end points of paths composed of one edge in HN and one edge in the reverse of HN. For example, occurrences 4 and 5 have the same name. In Figure 4.3, they are connected with one edge in HN (4-x) and one edge in the reverse of HN (x-5).

CompatibleKind (K). We compute K, the relation of occurrences whose kinds are compatible, as follows.

U_1 = HK . “ref_var”

D_1 = HK . “def_var”

U_2 = HK . “ref_fun”

D_2 = HK . "def_fun"

$K = U_1 \times D_1 \cup U_2 \times D_2$

Using set projection, set U_1 is computed as the set of all referential occurrences of variable. In Example 4.1, elements of U_1 include occurrences 5 and 8. Similarly, set D_1 is the set of all definitional occurrences of variable. In Example 4.1, elements of D_1 include occurrences 1 and 4. The cross product $U_1 \times D_1$ of these sets gives all pairs of referential occurrence of variable and definitional occurrence. For example, <5, 1>, <5, 3> and so on. Similarly, U_2 is the set of function calls and D_2 is the set of function definitions.

Visibility (V). We compute the relation V of <x, y> such that y is visible to x. In Example 4.1, we assume that an occurrence can see declarations in an enclosing block, which is similar to ALGOL-60's scoping rule. The formula is the following.

$V = P^+ \circ C$

where relation P is the inverse of relation Contains (C) (see Figure 4.5). The formula means that the relation Visibility is the set of starting and end points of paths composed of one or more P edges and one Contains edge.

Suppose that occurrence x is in block b_1, then the definitional occurrence is in block b_1, or the block b_2 that contains block b_1, or the block b_3 that contains b_2, etc. In Figure 4.5, occurrence 5 is in block 3, its definition is supposed to be in block 3 or block 0. From occurrence 5, we can reach these blocks via one or more P edges, e.g., 5-3 and 5-3-0. From these blocks, we can reach the occurrences in the block via one Contains edge, e.g. 5-3-4, 5-3-0-1. Therefore, paths of one or more P edges and one Contains edge connect an occurrence with its potential match. The formula

$V = P^+ \circ C$

ignores the name and kind of the occurrences, because these are separately handled in formulas for SameName (N) and CompatibleKind (K).

Hiding (H). A pair <x, y> in H means that if another occurrence z can refer to both x and y, then x hides y and z should refer to x. In Example 4.1, the declarations in the inner block can hide the occurrences in the outer block. For example, occurrence 4 can hide occurrence 1. H is computed as follows.

$H = P \circ P^+ \circ C$

The formula means that occurrence x can hide occurrence y if they are the starting and end points of paths composed of two or more P edges and one Contains edge. Suppose that occurrence x is in block b_1, then x can hide definitional occurrences in block b_2 that contains b_1 or block b_3 that contains b_2, …, etc. In Figure 4.5, occurrence 4 is in block 3 and it can hide occurrences in block 0. From occurrence 4, we can reach block 0 via two or more P edges, I.e. 4-3-0. From these blocks, we can reach the occurrences in them via one Contains edge, e.g., 4-3-0-1 and occurrence 4 can hide occurrence 1. Therefore, an occurrence is connected to the occurrence it can hide by a path of two or more P edges and one Contains edge. Again, we can ignore name and kind of occurrence in the formula because they are specified in formulas for SameName and CompatibleKind.

4.5.1.3 Binding

Once Normalized Facts have been computed, we can compute the Refers-to relation. ARA defines the Refers-to relation as the set of pairs that satisfy the four Binding Conditions, as presented in section 4.4. In Binding, we construct four relations B_1, B_2, B_3, B_4, where B_1 satisfies Binding Condition 1, B_2 satisfies Binding Conditions 1 and 2, B_3 satisfies Binding Conditions 1, 2 and 3 and finally B_4 satisfies all four Binding Conditions. We will consider the four relations B_1, B_2, B_3, B_4 one by one.

Computing B_1. B_1 requires that occurrences x and y have the same name. In Normalization (Section 4.5.2), we have computed the relation SameName (N), which is the set of pairs of occurrences that have the same name. Therefore B_1 is simply N:

$$B_1 = N$$

In Example 4.1, occurrences 1, 4, 5 and 8 have the same name of "x". Therefore, there are 16 tuples in the relation SameName. It is obvious that some tuples are redundant. For example, <4, 4> because the occurrence cannot refer to itself. Therefore, further refinement is needed.

Computing B_2. B_2 requires that x and y have the same name and the kinds of x and y are compatible. From Normalization, we have the relation *CompatibleKind* (K) that consists of all pairs of occurrences whose kinds are compatible. Therefore, B_2 is the intersection of B_1 and K.

$$B_2 = B_1 \cap K$$

The intersection of B_1 and K removes from B_1 the pairs whose kinds are not compatible, e.g. <4, 4> and <1, 4> because both occurrences are definitional occurrences. B_2 for Example 4.1 is {<5, 1>, <5, 4>, <8, 1>, <8, 4>}.

Computing B_3. B_3 requires that x and y have the same name, that the kinds of x and y are compatible and that y is visible to x. In Normalization, we have computed the relation Visibility (V).

A pair <x, y> of Visibility means that y is visible to x. For Example 4.1, Visibility (V) is {<5, 1>, <5, 2>, <5, 4>, <5, 6>, <8, 1>, <8, 2>, <8, 6>}. ARA also defines the relation B_3 as the set of pairs that satisfy Binding Conditions 1, 2 and 3. Therefore,

$$B_3 = B_2 \cap V$$

It removed redundant tuples <x, y> where y is not visible to x. For example, <8, 4> should be removed from B_2 because occurrence 4 is not visible to occurrence 8. In other words, it is impossible for the definition of occurrence 8 to reside in block 3 which is not an enclosing block of occurrence 8. At this stage, we are close to solving the Refers-to relation and B_3 for Example 3.1 is Figure 4.6. Occurrence 8 is matched with occurrence 1, which is correct. However, occurrences 5 has two matches, <5, 1> and <5, 4>. The last formula is to remove the redundant tuples.

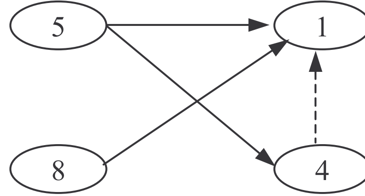

Figure 4.6 Relation B_3 (solid arrows) and an edge of H (dashed arrows)

Computing B_4. B_4 requires that y is not hidden by other occurrences besides satisfying Binding Conditions 1, 2 and 3. In Example 4.1, both occurrences 1 and 4 have the same name and compatible kind with occurrence 5 and they are both visible to occurrence 5. However, occurrence 4 hides occurrence 1 and occurrence 5 refers to occurrence 4, not occurrence 1. In Normalization, we have computed the relation Hiding (H). For Example 4.1, H is {<4, 1>}. By the definition of relational composition,

$$B_3 \text{ o } H = \{\langle x, y\rangle \mid \exists z \langle x, z\rangle \in B_3 \wedge \langle z, y\rangle \in H\}$$

In plain English, the relation composition of B_3 and H is the set of pairs <x, y> in B_3 such that y is hidden by some occurrence z. By removing such pairs from B_3, we can obtain pairs that satisfy all four Binding Conditions. Suppose B_3 is the set of pairs that satisfy all four Binding Conditions and R is the Refers-to relation, then

$$B_4 = B_3 - B_3 \text{ o } H$$

The result of composition B_3 o H is the set of pairs <x, y> such that there is a pair <x, z> in B_3 and occurrence z can hide occurrence y. According to the simplified C language rules, these pairs should be removed. In Figure 4.6, such pairs are the starting and end points of a path of one B_3 edge and one H edge, e.g., <5, 1>. We remove these pairs from B_3 and obtain the Refers-to relation, which is shown in Figure 4.7. The extraction of Refers-to relation is complete.

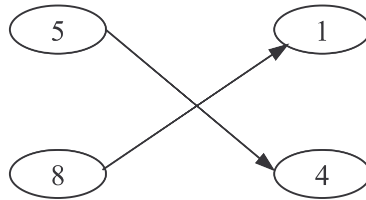

Figure 4.7 The Refers-to Relation for Example 4.1

4.5.2 Tables and graphs used for summarizing the three-step pipeline

ARA is much more concise than informal specification and Attribute Grammars. For example, we can summarize the three-step pipeline for languages similar to ALGOL-60 as Table 4.4.

Step	Relation	Definition
Basic Fact Extraction	HasName	$HN = \{<x, y> \mid \text{occurrences x has name y}\}$
	HasKind	$HK = \{<x, y> \mid \text{occurrence x has kind y}\}$
	Contains	$C = \{<x, y> \mid \text{block x directly contains occurrence or block y}\}$
Normalization	SameName	$N = HN \text{ o } HN^{-1}$
	Compatible-Kind	$U_1 = HK$. “ref_var” $D_1 = HK$. “def_var” $U_2 = HK$. “ref_fun” $D_2 = HK$. “def_fun” $K = U_1 \times D_1 \cup U_2 \times D_2$
	Visibility	$V = P^{+} \text{ o } C$
	Hiding	$H = P \text{ o } P^{+} \text{ o } C$
Binding	Condition 1	$B_1 = N$
	Condition 2	$B_2 = B_1 \cap K$
	Condition 3	$B_3 = B_2 \cap V$
	Condition 4	$B_4 = B_3 - B_3 \text{ o } H$

Table 4.4 Summary of three-step pipeline for ALGOL-60 like languages

For convenience, we often depict ARA pipeline as a data flow graph like Figure 4.8. This graph shows two kinds of information. First, it shows the relations (facts) created in each step of the pipeline. In Figure 4.8, Basic Fact Extraction creates three relations, HasName, HasKind and Contains. Normalization creates four relations, SameName, CompatibleKind, Visibility and Hiding. Finally, Binding computes B_1, B_2, B_3, B_4 and B_4 is the Refers-to relation. Second, it shows how the relations are used in the pipeline. For example, the arrow between relation HasName and relation SameName indicates that the relation SameName is computed using relation HasName as input.

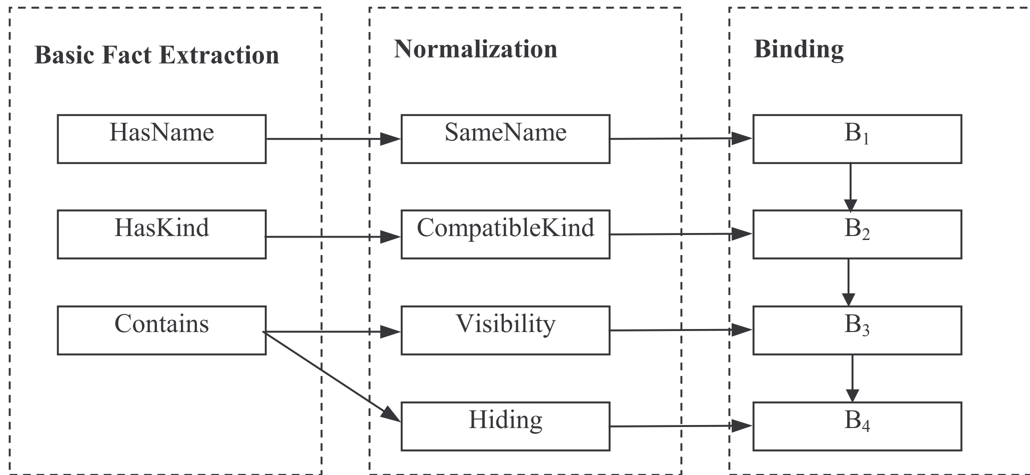

Figure 4.8 Data flow graph of the three-step pipeline

4.5.3 Multi-phase pipelines

During the writing of the thesis, seven languages, including C, CPP (C Preprocessor), Java, C++, Fortran, Pascal and Ada, were studied. As explained in Chapter 1, for these languages, we need multi-phase pipeline (see Figure 4.9).

Each phase in the multi-phase pipeline resolve a subset of occurrences such as all occurrences in the import declarations (in Java) or all occurrences in the ordinary statements. The union of results from phases is the Refers-to relation.

Within each phase, there can be at most two stages. Stage 1 resolves the unqualified occurrences such as occurrence of x in x++. Stage 2 resolves the qualified occurrences such as occurrence y in x.y. Each stage has three steps, Basic Fact Extraction, Normalization and Binding, as we have seen in the three-step pipeline. The result from the previous phases (stages) can be used by the current phase (or stage). Figure 4.9 implies that each stage has its own parser. But in an actual implementation, it is better to use one parser to extract the Basic Facts that are used in several phases.

We will give more detail of multi-phase ARA pipeline in Chapters 5, 6 and 7.

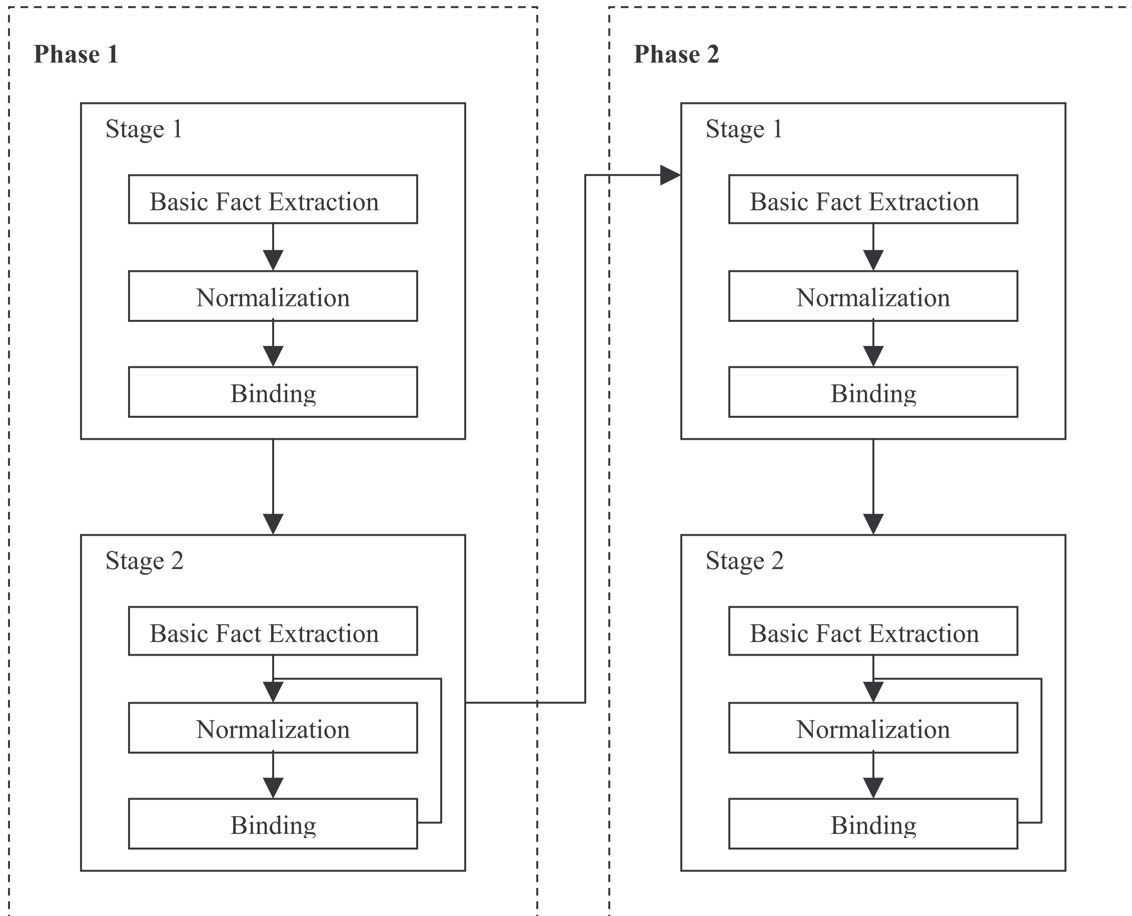

Figure 4.9 Multi-phase pipeline

4.6 Validation of ARA

This section answers the question how ARA is shown to be correct. Validation of ARA means to demonstrate that ARA complies with the corresponding language standard such as ISO C++ standard [C++03] and Java language specification [GJSB05]. For each rule in the language standard that deals with the Refers-to relation, we show that the rule is correctly transcribed into ARA. For example, the following is Item 4 of Clause 6.2.1 of ISO C standard.

"... the scope of one entity (the inner scope) will be a strict subset of the scope of the other entity (the outer scope). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is hidden (and not visible) within the inner scope" (6.2.1(4)).

Using set theory and relational algebra, this rule is transcribed into the following formulas.

$$V = P^{+} \text{ o } C$$

$$H = P \text{ o } P^{+} \text{ o } C$$

We claim that transcription such as this one are direct enough and clear enough that it is obvious that they are correct.

For practical programming languages such as the seven languages studied in this thesis (C, Java, C++, CPP, Fortran, Pascal and Ada), ARA uses multi-phase pipeline, which is composed of phases, stages and three-step pipelines. In general, the validation of these pipelines consists of showing that the following three claims are satisfied.

1. Phases and stages must be designed to match the language standard.
2. Basic Facts must represent the source code according to the language standard.
3. Formulas for Normalization and Binding must follow the language standard.

In Section 8.2, we will give more details about validation and how these claims are shon to be satisfied.

4.7 Chapter summary

This chapter gave an overview of ARA. It started with a brief survey of traditional specification techniques for extracting the Refers-to relation, including informal specification and Attribute Grammars and points out the drawbacks of these approaches.

Motivated by the relational data model and developments in program comprehension and reverse engineering, Algebraic Refers-to Analysis (ARA) was invented. ARA formulated the extraction of the Refers-to relation as a query to find occurrences of identifier that satisfy the Binding Conditions. The query was solved with either three-step pipeline or multi-phase pipeline.

The three-step pipeline has three sequential steps, Basic Fact Extraction, Normalization and Binding. Using a simple language with ALGOL-60 like scoping rule, the Basic Fact Extraction and formulas used in Normalization and Binding were introduced. The data flow of ARA pipeline was presented.

Chapter 5 ARA for C

This chapter presents an example of the multi-phase pipeline, ARA pipeline for the C language (ARA for C). ARA for C has one phase. Within the phase, there are two stages. Stage one resolves unqualified occurrences. Stage two resolves qualified occurrences. Each stage is a three-step pipeline.

Validation of ARA for C means to show that ARA for C satisfies the ISO C standard [C99]. For each rule in the ISO C standard that is related to the Refers-to relation, we show that the rule is correctly transcribed into phases (or stages or formulas) of ARA for C. Section 8.2 will discuss the validation in more detail.

This chapter is organized as follows. First, Section 5.1 introduces some basic concepts about the C language and its Refers-To relation. Then the overview of ARA for C is presented in Section 5.2. The stages are explained in Sections 5.3 and 5.4. Finally, Section 5.5 concludes the chapter with a short summary.

5.1 C syntax related to Refers-to relation

The current standard for the C programming language is ISO/IEC 9899:1999 [C99], commonly referred to as C99. In this standard, syntax related to Refers-to relation includes the different kinds of occurrences, scopes in C and rules for visibility and name hiding.

5.1.1 Occurrences in C

Recall that in the Refers-to relation, an occurrence can play one of the two roles, the referential occurrence and the definitional occurrence. A definitional occurrence introduces a new entity (label, type, variable, function, …) into the program. A referential occurrence refers to an entity introduced by a definitional occurrence.

We can distinguish the definitional occurrence from the referential occurrence by the syntactic context. For example, occurrence x in

```
int x;
```

is a definitional occurrence. Occurrence y in

```
y++;
```

is a referential occurrence.

Each occurrence has its kind that is determined by the syntactic context in most cases. Occurrence x in the above defines a variable, therefore its kind is definitional occurrence of variable. Occurrence y in the above example refers to a variable and its kind is referential occurrence of variable.

In the C language, the kind of an occurrence can be:

1. definitional (or referential) occurrence of a label;
2. definitional (or referential) occurrence of the tags;
3. definitional (or referential) occurrence of the members of structures or unions;
4. definitional (or referential) of all other identifiers, called ordinary identifiers.

Sometimes syntactic context of an occurrence is not sufficient to decide its kind. For example,

```
T * x;
```

This is called syntactic ambiguity, which will be discussed in Chapter 8. In this chapter, we assume that we can decide the kind of occurrence by syntactic context.

Finally, consider the qualification of referential occurrence. A referential occurrence can appear alone or connected with other occurrences by member access operator (.) or points-to operator (->). The occurrence following these operators are called *qualified occurrence*. Otherwise, it is called an *unqualified occurrence*. For example, occurrences y and z in x.y.z are qualified occurrences. Occurrences x and y in x++ and y.z are unqualified occurrences.

The Refers-to relation of qualified occurrences and unqualified occurrence are computed in different ways. An unqualified occurrence can be resolved without the knowledge of the Refers-to relation of other referential occurrences. This is not the case for qualified occurrences. Consider the following example.

struct A_1 $\{_2$
        struct B_3 $\{_4$
                int z_5;
        $_4\}$ y_6;
$_2\}$ x_7;

$x_8.y_9.z_{10}$ ++;

We first resolve occurrence 8 because it is an unqualified occurrence and match occurrence 8 with occurrence 7. Then, we know that the match for occurrence 9 is in block 2 (inside struct A). So, we search block 2 for match for occurrence 9 and find occurrence 6. Again, we know that match for occurrence 10 is in block 4 (inside struct B) and finally we match occurrence 10 with occurrence 5.

As a result, ARA for C has one phase. Within this phase there are two stages. Stage 1 resolves unqualified occurrences. Stage 2 resolves qualified occurrences (see Figure 5.1).

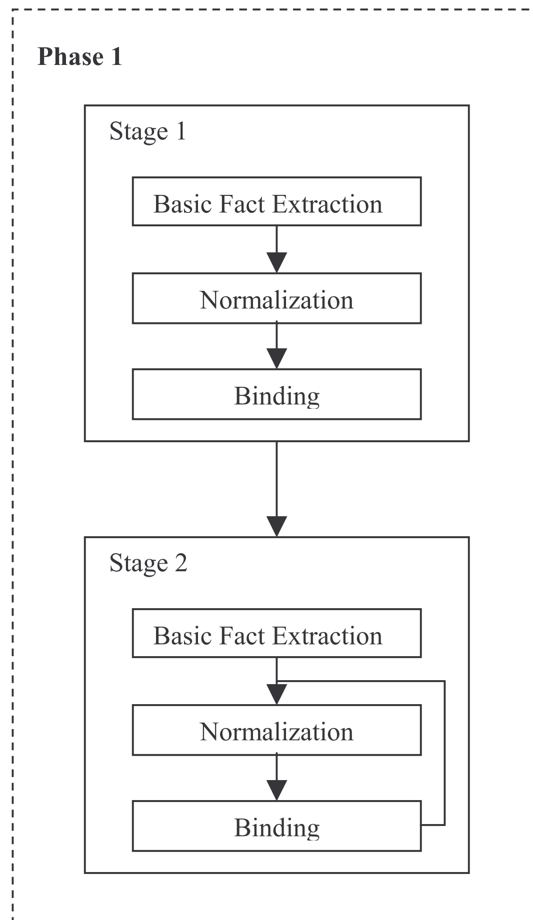

Figure 5.1 ARA for C

5.1.2 Scopes in the C language

The C standard [C99] lists four kinds of scopes: file scope, local scope, function scope and function prototype scope. Among them, the last two scopes are simple and used less often. Only labels depend on the function scope, which encloses the each function definition. The function prototype scope encloses the function prototype. File scope and local scope are detailed in the following.

5.1.2.1 File scope

File scope includes the entire translation unit and is often referred to as the global scope. Definitions that are outside of any block or a list of parameters are considered inside the global scope (file scope). A definition in the global scope is visible to all referential occurrences appearing after it.

A special case of the global scope is the tag of structure. Consider the following program.

```
struct A {
        struct B {
                …
        };
};
```

The structure A is considered inside the global scope because it is outside any block. However, the C language specifies that structure B is visible in structure A and in the global scope. To avoid confusions, it specifies that B and A must have different names.

5.1.2.2 Local scope

A local scope is a portion of program text contained within a block (function definition, compound statement and structure definition). Declarations in a local scope are visible to a referential occurrence in the same scope or in a nested scope and declarations in the local scope must be declared before it is used.

5.1.3 Visibility and name hiding

In C language, containment relation, the order of occurrence and qualification are used to decide the visibility of a definition. Also, containment relation also decides name hiding. We will give details in Sections 5.3 and 5.4.

5.2 Overview of ARA for C

ARA for C has one phase. Within the phase, there are two stages (see Figure 5.2). Stage one is in fact a three-step pipeline (see Chapter 4) of Basic Fact Extraction, Normalization and Binding. Stage two is also a three-step pipeline with a loop. Suppose we have the following occurrences in the program,

```
x.y.z
a.b
```

Stage one resolves unqualified occurrences x and a. Then, the first loop of Stage two resolves qualified occurrences y and b. Finally, the second loop of Stage two resolves qualified occurrence z. In general, if we have n qualified occurrences following one unqualified occurrence, Stage two loops

n times (see Figure 5.2). In Section 5.3, we explain Stage one, resolving unqualified occurrences. In Section 5.4, we explain Stage two, resolving qualified occurrences.

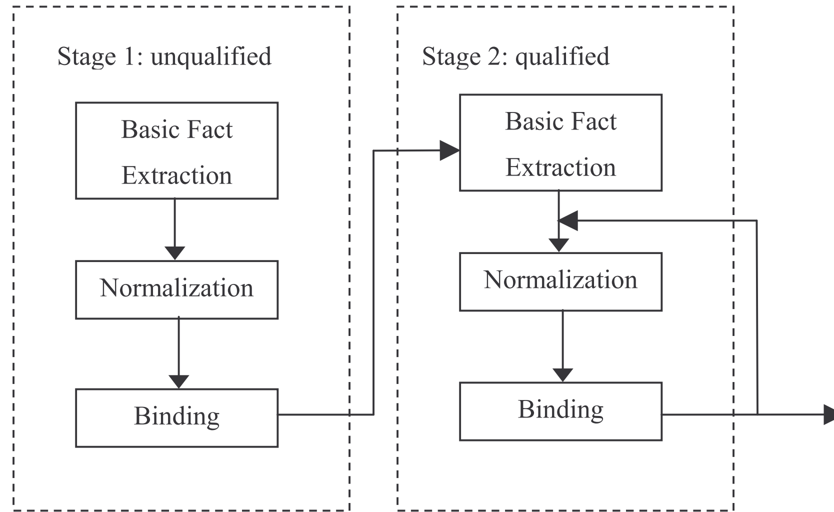

Figure 5.2 Stages in ARA for C

5.3 Stage one: Resolving unqualified occurrences

The formulas for resolving unqualified occurrences in C program are similar to formulas introduced in Section 4.5. Therefore, we only explain this stage briefly and readers can find detail in Section 4.5.

5.3.1 Basic Fact Extraction

The Basic Facts needed to resolve unqualified occurrences are stored in four relations: *HasName* (HN), *HasKind* (HK), *Follows* (F) and *Contains* (C), and their definitions are listed in Table 5.1.

5.3.2 Normalization

Normalization converts Basic Facts into Normalized Facts, namely relations *SameName* (N), *CompatibleKind* (K), *Visibility* (V) and *Hiding* (H).

Relation	Explanation
HasName (HN)	{<x, y> | occurrences x has name y}
HasKind (HK)	{<x, y> | occurrence x has kind y}
Contains (C)	{<x, y> | occurrence block x directly contains an occurrence or block y}
Follows (F)	{<x, y> | occurrence x follows occurrence y immediately}

Table 5.1 Basic Facts for unqualified occurrences

5.3.2.1 Formulas for relations SameName (N) and CompatibleKind (K)

The formulas for relations SameName (N) and CompatibleKind (K) are the following, which have been explained in Section 4.5.

$N = HN \circ HN^{-1}$

R_1 = HK . ”referential occurrence of ordinary identifier”

D_1 = HK . “definitional occurrence of ordinary identifier”

R_2 = HK . ”referential occurrence of tag”

D_2 = HK . “definitional occurrence of tag”

$K = R_1 \times D_1 \cup R_2 \times D_2$

5.3.2.2 Formulas for relation Visibility (V)

The relation Visibility for unqualified occurrences is decided by two rules:

1) an unqualified occurrence can see definitions in the same block or an outer block;

2) a name must be defined before being referred to.

As discussed in Section 4.5, rule 1 is transcribed into the following formula.

$P^{+} \circ C$

We use relation Follows (F) to construct the formula for rule 2. A pair <x, y> of relation Follows means that occurrence x follows y immediately. Then, a pair <x, y> of the transitive closure of Follows means that x follows y, immediately or transitively. Combining rules 1 and 2 by intersection, we get the following formula for relation Visibility (V) for unqualified occurrences.

$$V = P^+ \circ C \cap F^+$$

5.3.2.3 Formulas for the Hiding (H) Relation

The formula for relation Hiding is the following, which has been explained in Section 4.5.

$$H = P \circ P^+ \circ C$$

5.3.3 Binding Formulas

As introduced in Section 4.5, the formulas for Binding are the following.

$$B_1 = N$$

$$B_2 = B_1 \cap K$$

$$B_3 = B_2 \cap V$$

$$B_4 = B_3 - B_3 \circ H$$

where B_4 is the Refers-to relation (R).

5.4 Stage two: Resolving qualified occurrences

Stage two is also a three-step pipeline but with a loop. Each loop resolves a layer of qualified occurrences until all of them are resolved. Suppose we have x.y.z and a.b in the program. After the unqualified occurrences are resolved, we need two loops, one for y and b and the other for z.

Figure 5.3 shows the data flow of Stage two. There are three steps in the stage, Basic Fact Extraction, Normalization and Binding. First, Basic Fact Extraction (the dashed box on the left) creates the relations that are needed in resolving unqualified occurrences, including the *HasName* (HN), *HasKind* (HK), *Denotes* (T), *Qualifies* (Q) and *Contains* (C) relations. The Normalization (the dashed box in the middle) use the *HasName* (HN) relation to create the SameName(N) relation, the *HasKind* (HK) relation for the *CompatibleKind* (K) relation, the relations *Denotes* (T), *Specifies* (SP), *Qualifies* (Q) and *Contains* (C) and relation R from the previous loop (and Stage 1) for the Visibility (V) relation. The Hiding (H) relation is an empty relation in for qualified occurrences. The Normalized Facts are then used in Binding (the dashed box on the right) to compute the Refers-To relation (R). There are four formulas in Binding and Relations B_1, B_2, B_3 and B_4 are respective results.

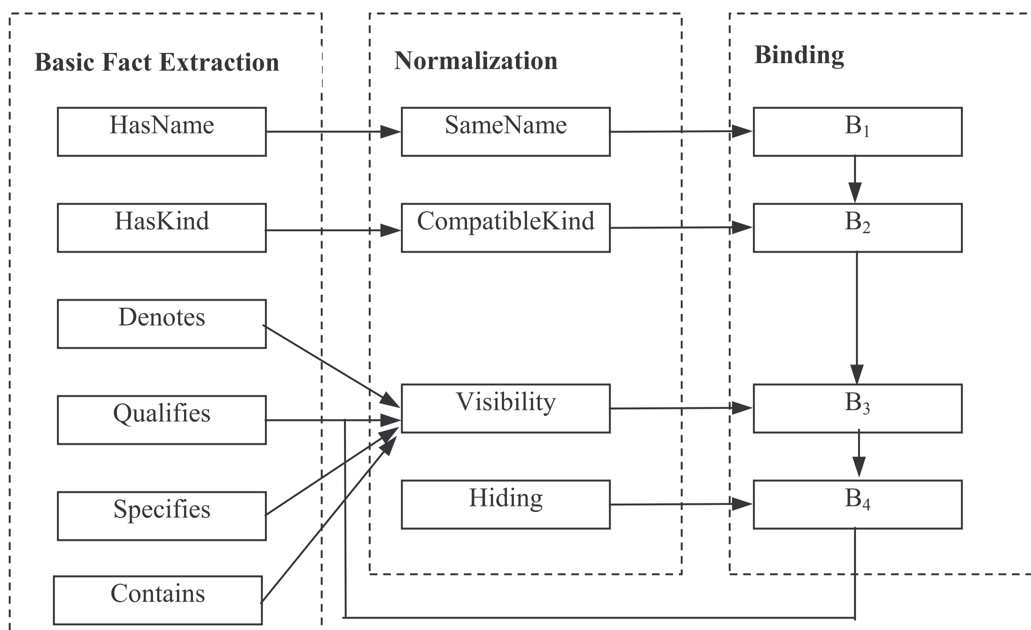

Figure 5.3 Data flow graph for Stage two

5.4.1 Basic Fact Extraction

Basic Facts needed in resolving qualified occurrences are listed in Table 5.2.

Relation	Explanation
HasName (HN)	{<x, y> | occurrences x has name y}
HasKind (HK)	{<x, y> | occurrence x has kind y}
Contains (C)	{<x, y> | occurrence block x directly contains an occurrence or block y}
Follows (F)	{<x, y> | occurrence x follows occurrence y immediately}
Denotes (T)	Example: struct A_1 $\{_2\ \ _2\}$ // $\langle 1, 2\rangle \in T$
Specifies (SP)	Example: struct A_1 a_2; // $\langle 1, 2\rangle \in SP$
Qualifies (Q)	Example: $x_1 . y_2$ // $\langle 1, 2\rangle \in Q$

Table 5.2 Basic Facts for Stage two

There are three ways to define a structure in the C language, illustrated below as Examples 5.1, 5.2 and 5.3.

struct A_1 {$_2$ int x_3; $_2$};

struct A_4 a_5;

a_6. x_7;

Example 5.1

For Example 5.2, relations Denotes, Specifies and Qualifies are the following.

Denotes (T) = {<1, 2>}

Specifies (SP) = {<4, 5>}

Qualifies (Q) = {<6, 7>}.

struct B_1 {$_2$ int x_3; $_2$} b_4;

b_5 . x_6;

Example 5.2

For Example 5.2, the relations are the following.

Denotes (T) = {<1, 2>},

Specifies (SP) = {<1, 4>},

Qualifies (Q) = {<5, 6>}.

struct {$_1$ int x_2; $_1$} c_3;

c_4 . x_5;

Example 5.3

For Example 5.3, the relations are the following.

Denotes (T) = {<3, 1>}

Qualifies (Q) = {<4, 5>}.

The remaining relations in Basic Facts have been introduced before and are not discussed in detail. Now, consider the Normalization of the Basic Facts.

5.4.2 Normalization

Normalization formulas convert Basic Facts into Normalized Facts, i.e. relations *SameName* (N), *CompatibleKind* (K), *Visibility* (V) and *Hiding* (H). Note that the formulas are based on member access operator (.). But with minor changes, the formulas also work for pointer operator (->).

5.4.2.1 Formulas for relation SameName (N) and CompatibleKind (K)

$N = HN \circ HN^{-1}$

R_1 = HK . ”qualified occurrence”

D_1 = HK . “definitional occurrence of ordinary identifier”

R_2 = HK . ”qualified occurrence”

D_2= HK . “definitional occurrence of tag”

$K = R_1 \times D_1 \cup R_2 \times D_2$

5.4.2.2 Formulas for relation Visibility (V)

Because there are three ways to define the structures, the formula for relation Visibility (V) has three terms, for Examples 5.1, 5.2 and 5.3, respectively.

First, consider Example 5.1. Relations Denotes (T), Specifies (SP), Qualifies (Q) are shown in Figure 5.4. Relation R is the Refers-to relation computed by the previous phase. In Example, occurrence 4 resolves to occurrence 1 and occurrence 6 to occurrence 5.

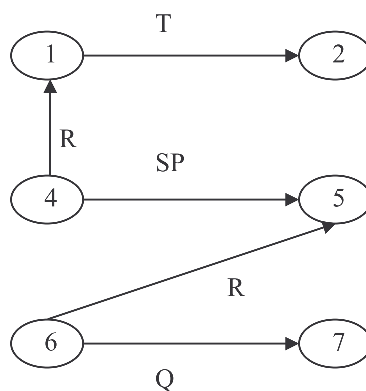

Figure 5.4 Relations for Example 5.1

The qualified occurrence we need to resolve is occurrence 7 and definitions visible to occurrence 7 are in block 2. On Figure 5.4, they are the starting and end points of the path from occurrence 7 to occurrence 2. Therefore, Visibility (V) for Example 5.1 is

$$Q^{-1} \text{ o } R \text{ o } SP^{-1} \text{ o } R \text{ o } T \text{ o } C$$

Second, consider Example 5.2. Relations Denotes (T), Specifies (SP), Qualifies (Q) are shown in Figure 5.5. Relation R is the Refers-to relation computed by the previous phase. In Example, occurrence 5 resolves to occurrence 4. Again, we can find the definitions visible to occurrence 6 by constructing a path from occurrence 6 to definitions in block. Therefore, the formula is the following.

$$Q^{-1} \text{ o } R \text{ o } SP^{-1} \text{ o } T \text{ o } C$$

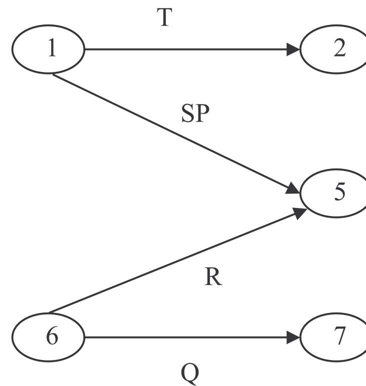

Figure 5.5 Relations for Example 5.2

Finally, consider Example 5.3. Relations Denotes (T), Specifies (SP), Qualifies (Q) are shown in Figure 5.6. Relation R is the Refers-to relation computed by the previous phase. In Example, occurrence 4 resolves to occurrence 3. The formula for Visibility is the following.

$$Q^{-1} \text{ o } R \text{ o } T \text{ o } C$$

Putting everything together, we have the formula for Visibility as the following.

$$V = Q^{-1} \text{ o } R \text{ o } (ID \cup SP^{-1} \cup SP^{-1} \text{ o } R) \text{ o } T \text{ o } C$$

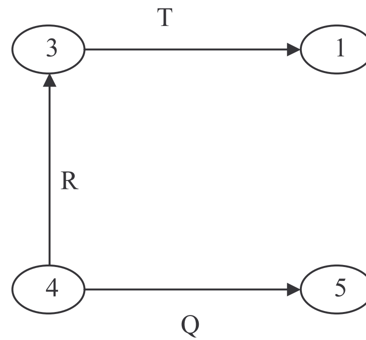

Figure 5.6 Relations for Example 5.3

5.4.2.3 Formulas for the Hiding (H) Relation

In resolving qualified occurrences, we only search one block. According to the One Definition Rule, the relation Hiding (H) should be empty.

$$H = \varnothing$$

5.4.3 Binding Formulas

The Binding Formulas for resolving qualified occurrences are the same as Binding Formulas for resolving unqualified occurrences (see Section 5.3).

5.5 Chapter summary

This chapter used ARA for C as an example and presented the two stages of a multi-phase pipeline. ARA for C has one phase. Within the phase, there are two stages. Stage one is a three-step pipeline that resolves unqualified occurrences. Stage two is a three-step pipeline with a loop that resolves qualified occurrences.

This chapter also explained in detail the formulas for resolving qualified occurrences. It explained the three ways of defining structures and presents formulas for the three variations.

Chapter 6
ARA for Java

This chapter uses ARA for the Java language (ARA for Java) as an example to explain how to compute the Refers-to relation in an Object-Oriented language. ARA for Java is a multi-phase ARA pipeline (see Chapter 4) and this chapter also details how the output of previous phases is used in the current phase. Another interesting fact about the Java language is that its specification [GJSB05] is written in the similar fashion as ARA, which is discussed in Section 6.1.

Validation of ARA for Java means to show that ARA for Java satisfies the Java language specification [GJSB05]. For each rule in the Java language specification that is related to the Refers-to relation, we demonstrate that the rule is correctly transcribed into phases (or stages or formulas) of ARA for Java. Section 8.2 will discuss the validation in more detail.

The chapter is organized as follows. Section 6.1 explains Java syntax related to the extraction of the Refers-To relation. Sections 6.2, 6.3 and 6.4 explain the ARA for Java. Section 6.5 concludes the chapter.

6.1 Java syntax related to Refers-to relation

This section introduces Java syntax related to Refers-to relation, including the kinds of occurrences, scopes and rules about visibility and name hiding. It also introduces an interesting fact about Java language specification. That is, the language specification is written in the similar way as ARA.

6.1.1 Occurrences in Java

Occurrences in Java play two roles in the Refers-to relation. The definitional occurrences introduce entities (package, class, function, variable) into the program. The referential occurrence refers to entities defined by definitional occurrences.

There are two kinds of referential occurrences, qualified occurrences and unqualified occurrences. Recall from Chapter 5 that the Refers-to relation of qualified occurrences depend on the Refers-to relation of unqualified occurrences and other qualified occurrences. In Java, even the Refers-to relation of unqualified occurrences may depend on the Refers-to relation of other occurrences. For example, the import statement

```
import java.io.*;
```

introduces definitions in the package java.io into the translation unit. Therefore, we must first resolve the name java.io before we resolve the unqualified occurrences.

Table 6.1 lists occurrences in different statements. For example, the first row is occurrences in package declaration. All source files (.java files) in the same package start with the same package declaration. The second row is occurrences in import statements, which introduce definitions into current translation unit.

Table 6.1 also shows the order of resolving these occurrences. The first row is package declaration and the occurrences (qualified and unqualified) are resolved first. This is because by default, a definition is visible to all files within the same package. The last row is occurrences in ordinary statement. That includes all occurrences that are not mentioned in the first four rows of Table 6.1. In ARA for Java, each row in Table 6.1 is implemented as a phase.

Statements	Use	Examples
Package declaration	Merging separate parts of the same package	Package MyPackage;
Import declaration	Introducing definitions into translation unit	Import java.io.*;
Class declaration	Determining class hierarchy	class Derived extends Base { }
Template	Resolving template-related types and variables	<**T** extends Base> **T** f(class<**T**> a);
Ordinary Statements	Resolving labels, types and objects (variables, functions)	**MyClass** p; **Abc**++;

Table 6.1 Occurrences in Java

6.1.2 Scopes in Java

There are three kinds of important scopes in Java, package scope, local scope and class scope. The package scope and local scope are similar to global scope and local scope that are commonly referred to in C and C++ languages. The class scope is similar to the block scope for C structures. But the difference is that one class scope (derived class) can inherits definitions in another class scope (base class).

6.1.3 Visibility and name hiding

Java uses containment relation, import statements, inheritance hierarchy and qualification to decide visibility and name hiding. These rules are detailed in Sections 6.3 and 6.4. The interesting fact about Java is that its specification [GJSB05] is written in the same fashion as ARA.

Java Specification explains the visibility of definition in Section 6.3. For example,

> The scope of a local variable declaration in a block is the rest of block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable statements.

The name hiding rules are divided into three categories, shadowing, hiding and obscuring. Shadowing is mainly about imports and enclosing blocks while hiding only applies to members of subclasses. Obscuring resolves certain syntactic ambiguities. However, all of these are specified similar to ARA's relation Hiding. For example,

> A declaration d of a type named n shadows the declarations of any other types named n that are in scope at the point where d occurs throughout the scope of d. …
>
> A variable will be chosen in preference to a type, …

Appendix A will list the Java rules for visibility and name hiding and their location in the Java specification. In addition, C++ standard [C++03] also uses the similar way to specify its rules. These specification show that although ARA uses mathematical formulas, it is also close to the way people talk about the Refers-to relation and therefore easy to understand.

6.2 Overview of ARA for Java

ARA for Java is a multi-phase ARA pipeline (see Chapter 4). We need multiple phases for two reasons. First, as in C, there are qualified occurrences and resolving these occurrences requires multiple phases. Second, package name, import declarations, class names must be resolved before we can extract some Basic Facts such as inheritance (see Table 6.1). Therefore, occurrences in package statements, import statement and class declarations are resolved. Then, occurrences in other statements are resolved. We leave the discussion of occurrences in template to Chapter 8, where templates in other languages are also discussed.

Within each phase, there are two stages, Stage 1 for resolving unqualified occurrences and Stage 2 for resolving qualified occurrences. Each stage has three steps including Basic Fact Extraction, Normalization and Binding. Among all phases, the phase that resolves occurrences in ordinary statement uses all Basic Facts and results from all previous phases and therefore, is most difficult. The rest of the chapter explains this phase. Section 6.3 explains Stage 1 of this phase. Section 6.4 explains Stage 2 of this phase.

6.3 Stage One: Resolving unqualified occurrences

This section explains how ARA resolves unqualified occurrences in ordinary statements. By ordinary statements, we mean that the statement is not a package statement, import statement or a class definition. ARA resolves these occurrences in three steps, Basic Fact Extraction, Normalization and Binding. Figure 6.1 illustrates the data flow of these steps. First, in Basic Fact Extraction (the dashed box on the left) creates the relations that are needed in resolving unqualified occurrences, including *HasName* (HN), *HasKind* (HK), *Contains* (C), *Inherits* (I) and *GroupImports* (GM) and *SingleImports* (SM). The Normalization (dashed box in the middle) uses *HasName* (HN) creates relation *SameName* (N), *HasKind* (HK) for *CompatibleKind* (K), *Contains* (C), *Inherits* (I), *GroupImports* (GM) and *SingleImports* (SM) for the Visibility (V) relation, and *Contains*(C), *Inherits* (I), *GroupImports* (GM) and *SingleImports* (SM) for the *Hiding* (H) relation. Finally, Binding (dashed box on the right) formulas 1, 2, 3 and 4 uses the Normalized Facts to compute the Refers-To relation (B). Relations B_1, B_2, B_3 and B_4 are respective results of the formulas.

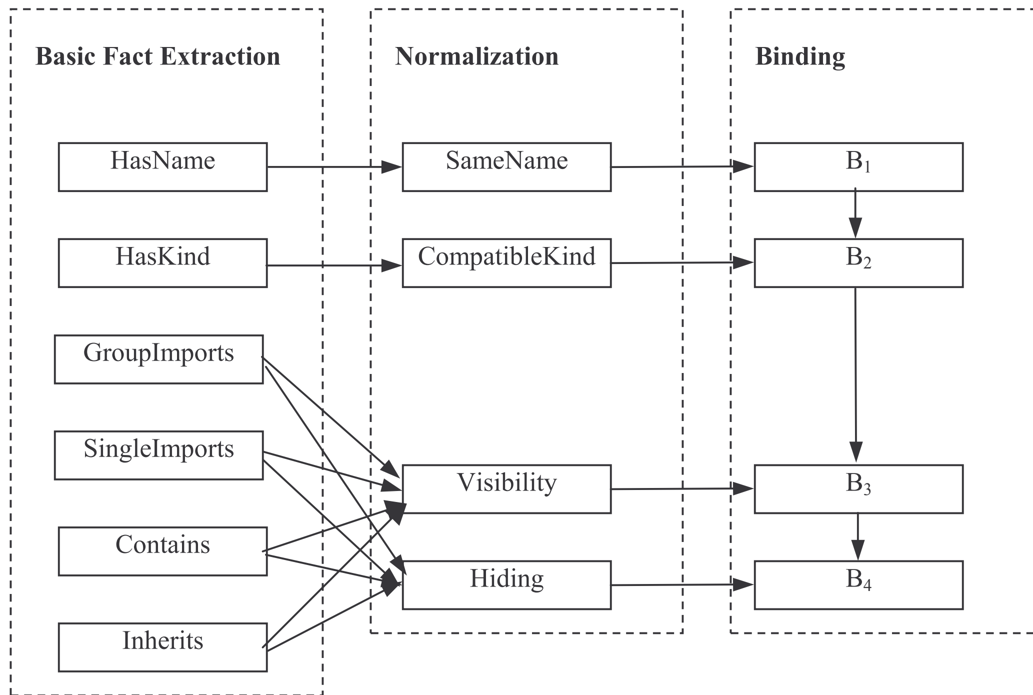

Figure 6.1 Data flow graph for resolving unqualified occurrences

Now, we use Example 6.1 to explain the three steps of resolving unqualified occurrences.

class $Base_1$ $\{_2$

int x_3;

void f_4( ) $\{_5$ $_5\}$

```
2}
class Sub6 extends Base7{8
        int x9, y10;
        void g11( ) {12
                x13++;
                y14++;
                f15( );
        12}
8}
```

Example 6.1. A Java program

In Example 6.1, class Sub extends class Base and inherits occurrence 3 and occurrence 4. The occurrence 3 is shadowed in the class Sub by occurrence 9. Finally, occurrence 11 accesses occurrence 9, occurrence 10 and occurrence 4.

6.3.1 Basic Fact Extraction

The Basic Facts for ARA include four relations, *HasName* (HN), *HasKind* (HK), *Contains* (C), *Inherits* (I), *GroupImports* (GM) and *SingleImports* (SM) (see Table 6.2).

Relation	Explanation
HasName (HN)	{<x, y> | occurrences x and y have the same name}
HasKind (HK)	{<x, y> | x is an occurrence of kind y}
Contains (C)	{<x, y> | block x contains block or occurrence y directly}
Inherits (I)	{<x, y> | blocks x inherits definitions in block y}
SingleImports (SM)	Importing individual class, e.g. import java.util.List;
GroupImport (GM)	Importing classes in a package, e.g. import java.util.*;

Table 6.2 Basic Facts for unqualified occurrences

Basic Fact Extraction is implemented by a simple parser that has no symbol tables. Relation Inherits (I) and Imports (M) in Table 6.2 are first constructed by the parser and then combined with the results from the previous phases. Consider the following example.

```
Class Base1 {2 … 2}
```

Class $Derived_3$ extends $Base_4$ $\{_5 \ldots _5\}$

Suppose that parser constructs a relation called RawInherits (RI), which has a tuple <3, 4>. This relation is not convenient and what we need is the Relation Inherits between blocks. From Table 6.1, we know that occurrence 4 has been resolved to occurrence 1 before we resolve unqualified occurrences. The Refers-to relation from previous phases is relation R. In addition, relation Denotes (T) tells us that occurrence 1 denotes block 2 and occurrence 3 denotes block 5. Therefore, the following formula creates the relation Inherits that is between blocks.

$$I = T^{-1} \circ RI \circ R \circ T$$

Relations *GroupImports* (GM) and *SingleImports* (SM) are constructed in the similar way.

6.3.2 Normalization

Normalization formulas convert the Basic Facts (relations *HasName* , *HasKind*, *Contains*, *Inherits, GroupImport, SingleImport*) into Normalized Facts (relations *SameName*, *CompatibleKind*, *Visibility* and *Hiding*).

6.3.2.1 The Formula for SameName (N)

The formula is

$$N = HN \circ HN^{-1}$$

6.3.2.2 The Formulas for CompatibleKind (K)

Formulas for CompitableKind (K) are the following.

R_1 = HK . "unqualified referential occurrence of ordinary identifier"

R_2 = HK. "unqualified referential occurrence of package"

R_3 = HK. "unqualified referential occurrence of class"

D_1 = HK . "definitional occurrence of ordinary identifier"

D_2 = HK. "definitional occurrence of package"

D_3 = HK. "definitional occurrence of class"

$$K = R_1 \times D_1 \cup R_2 \times D_2 \cup R_3 \times D_3$$

6.3.2.3 Formulas for Visibility (V).

Three kinds of occurrences are visible to an unqualified occurrence x in class A:

1) occurrences in the same block or an outer block,
2) occurrences that are member of super-classes,
3) occurrences that are imported.

1. Occurrences in the same or an outer block

Figure 6.2 is the graph of relations Contains (C) and Inherits (I) for Example 6.1. Finding the occurrences in the same or an outer block is equivalent to finding a path of one or more *Parent* (P) edges and one *Contains* (C) edge, where relation *Parent* (P) is the inverse of *Contains* (C). For example, occurrence 14 can see occurrence 10 because of the path 14-12-8-10. The formula finding these occurrences is

$$P^{+} \circ C$$

2. Occurrences that are member of super-classes

Finding the occurrences that are members of super-classes is equivalent to finding a path of one of more Parent (P) edges, one or more Inherits (I) edges and one Contains (C) edges. For example, occurrence 15 can see occurrence 4 because of the path 15-12-8-2-4. The formula computing these occurrences is

$$P^{+} \circ I^{+} \circ C$$

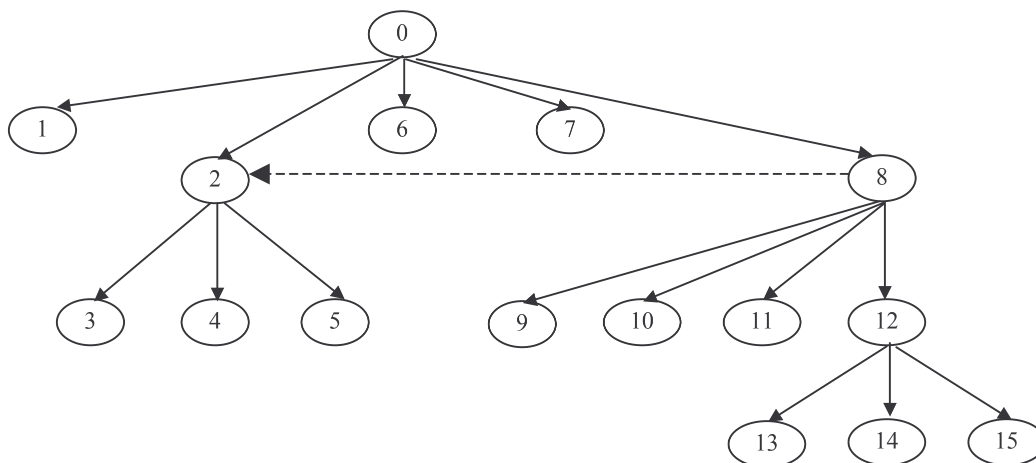

Figure 6.2 Relation Contains (C) and Inherits (I) for Example 6.1

The solid arrows represent Contains I edges and the dashed arrows represent Inherits (I) edges.

3. Occurernces that are imported

In Java, we can import a single class, e.g. import java.util.List. Relation SingleImport (SM) is the relation between the global scope and the individual classes it imports. Because classes in the global scope are visible from the enclosed scopes, we have the following formula for classes that are imported individually.

P^+ o SM

In addition, we can also import all classes in a package, e.g. import java.util.*. Relation GroupImport (GM) is the relation between the global scope and the package it imports. The formula for imported package is the following.

P^+ o GM o C

All together, we have the following formula for relation Visibility.

$V = P^+ \text{ o } I^* \text{ o } C \cup P^+ \text{ o } SM \cup P^+ \text{ o } GM \text{ o } C$

6.3.2.4 Formulas For Hiding (H)

In ARA, the relation Hiding (H) is used to resolve name conflicts. The sources of name conflicts are

1. Java's single name space;
2. Individual Visibility Rules, i.e., containment, inheritance;
3. Interaction between Visibility Rules, i.e. containment vs. inheritance and containment and using directives.

The following are the formulas for resolving these conflicts.

Source 1. Java's single name space

Java has one single name space and disambiguate by the kind of the occurrence. For example, a class name can be hidden by the name of an object or function declared in the same scope [Java section 6.3.2]. The following formula is equivalent to such a rule.

ObjSet = { objects in the program }

TypeSet = { types in the program }

$H = ObjSet \times TypeSet$

The cross product of ObjSet and TypeSet means that an object takes precedence over a type. We can have similar formulas for precedence between the imported definitions and user definitions as well.

Java also specifies the precedence among imported occurrences (see Appendix A). For example, *single-type-import* hides *type-import-on-demand*. Therefore, we have formulas like the following.

SingleTypeSet = {single-type-import}

TypeImportOnDemand = {type-import-on-demand}

H = SingleTypeSet × TypeImportOnDemand

Source 2. Conflict caused by individual Visibility rules

First, what if there are multiple definitions of the same name in the enclosing blocks? Like the C language, Java specifies that the definition in the inner block has the precedence over the definitions in the outer blocks. So, ARA reuses the formula for C, which is the following.

$P \circ P^+ \circ C$

Second, what if there are multiple definitions of the same name in the base classes. On the inheritance hierarchy, the definition in the derived class can hide the definitions in the base classes. Suppose that occurrence x is in class A and occurrence y is in class B and B is a base class of A. On the diagram of Contains (C) and Inherits (I) like Figure 5.2, such pairs <x, y> are the starting and end points of one *Parent* (P) edge, one or more *Inherits* (I) edges and one *Contains* (C) edge. For example, occurrence 9 can hide occurrence 3 because of the path of 9-8-2-3. The formula for these pairs is

$P \circ I^+ \circ C$

Source 3. Conflicts caused by interaction between Visibility rules

What if we find multiple definitions by both Containment hierarchy and Inheritance hierarchy? The inherited members are deemed as own and therefore have precedence over the definitions in the enclosing blocks. Using the diagram of Contains (C) and Inherits (I) like Figure 5.2, we can find these pairs of by finding the paths composed of one *Parent* (P) edge, one or more inverse of *Inherits* (I) edges, one or more *Parent* (P) edges and one *Contains* (C) edge. The formula for these pairs is

$P \circ (I^{-1})^+ \circ P^+ \circ C$

$= (I^+ \circ C)^{-1} \circ P^+ \circ C$

As we can see, it is the composition of the inverse of (I^+ o C), which is the second part of formula for 2) and (P^+ o C), which is the second part of the formula for 1). That is, the occurrences found by Inherits (I) relation take precedence over the occurrences found by *Contains* (C) relation. Similarly, the locally defined occurrences can hide the imported occurrences. Therefore, we have

$$P^+ \text{ o } SM \cup P^+ \text{ o } GM \text{ o } C$$

In summary, the formulas for the Hiding relation are the following (rules in source one are simplified).

ObjSet = {objects in the program}

TypeSet = {types in the program}

$$H = ObjSet \times TypeSet$$
$$\cup\ P \text{ o } P^+ \text{ o } C$$
$$\cup\ P \text{ o } I^+ \text{ o } C$$
$$\cup\ (I^+ \text{ o } C)^{-1} \text{ o } P^+ \text{ o } C$$
$$\cup\ P^+ \text{ o } SM$$
$$\cup\ P^+ \text{ o } GM \text{ o } C$$

6.3.3 Binding

The Binding formulas remain the same for Java. They are the following.

$$B_1 = N$$

$$B_2 = B_1 \cap K$$

$$B_3 = B_2 \cap V$$

$$B = B_4 = B_3 - B_3 \text{ o } H$$

6.4 Stage Two: Resolving qualified occurrences

This section introduces the ARA formulas for qualified occurrences. Again, ARA extracts Refers-To relation for qualified occurrences in three steps, Basic Fact Extraction, Normalization and Binding. Figure 6.3 illustrates the data flow for these three steps. First, Basic Fact Extraction (the dashed box on the left) creates five relations as Basic Facts, including *HasName* (HN), *HasKind(*HK), *Qualifies* (Q), *Denotes* (T), Specifies (SP), *Contains* (C) , *Inherits* (I) and R from the previous step. The Normalization (the dashed box in the middle) converts the Basic Facts into the Normalized Facts. The *Hiding* (H) relation for qualified occurrences is the empty relation. Finally, Binding Formulas 1, 2, 3

and 4 (the dashed box on the right) compute the Refers-To relation (R). Relations B_1, B_2, B_3 and B_4 are the respective results for these formulas. The Normalization and Binding loop until all qualified occurrences are resolved.

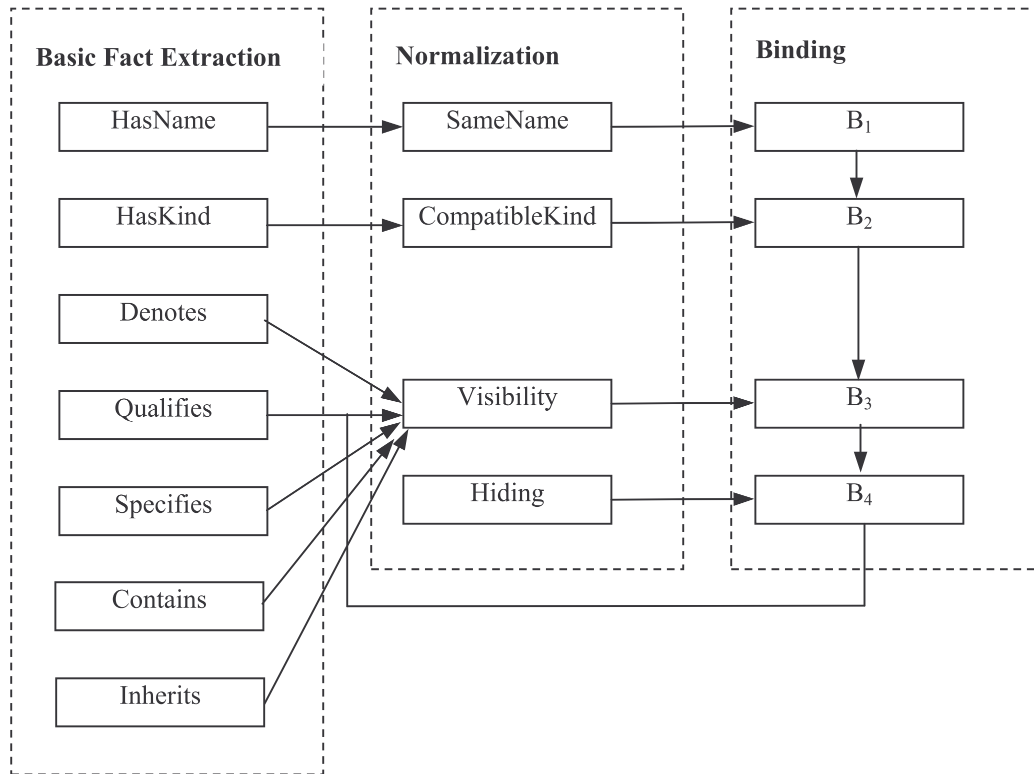

Figure 6.3 Data flow graph for qualified occurrences

6.4.1 Basic Fact Extraction

Table 6.3 lists the Basic Facts we need to resolve qualified occurrences in Java.

6.4.2 Normalization

Normalization formulas convert the Basic Facts (relations *HasName* , *HasKind*, *Contains*, *Inherits*) into four relations (*SameName*, *CompatibleKind*, *Visibility* and *Hiding*).

1. Formulas for SameName (N)

$$N = HN \text{ o } HN^{-1}$$

Relation	Explanation
HasName (HN)	{<x, y> | x is an occurrence named y}
HasKind (HK)	{<x, y> | x is an occurrence of kind y}
Qualifies (Q)	{<x, y> | x::y in the program}
Specifies (SP)	{<x, y> | x y; in the program }
Denotes (T)	{<x, y> | x denotes block y }
Contains (C)	{<x, y> | x, y are blocks, x directly contains y}
Inherits (I)	{<x, y> | x, y are blocks, x inherits y}

Table 6.3 Basic Facts for qualified occurrences.

2. Formulas for CompitableKind (K)

R_1 = HK . "qualified referential occurrence of ordinary identifier"

R_2 = HK. "qualified referential occurrence of package"

R_2 = HK. "qualified referential occurrence of class"

D_1 = HK . "definitional occurrence of ordinary identifier"

D_2 = HK. "definitional occurrence of package"

D_2 = HK. "definitional occurrence of class"

$K = R_1 \times D_1 \cup R_2 \times D_2 \cup R_3 \times D_3$

3. Formula for Visibility (V):
A qualified occurrence can see all members of the nominated class and members of the base classes of the nominated class. Based on formulas for the C language (see Section 5.4), we can find all members of the nominated class using the following formula:

$$Q^{-1} \text{ o } R \text{ o } SP^{-1} \text{ o } R \text{ o } T \text{ o } C$$

We also know from the previous section, we can find members of base classes using formula

$$I^{+} \text{ o } C$$

Therefore, the *Visibility* (V) relation for qualified occurrences can be computed by

$$V = Q^{-1} \text{ o } R \text{ o } SP^{-1} \text{ o } R \text{ o } T \text{ o } I^{*} \text{ o } C$$

4) Formulas for Hiding (H)
Because of polymorphism in the Java language, it is impossible to decide the Hiding relation for qualified occurrences at compile time. However, ARA can provide all candidates for further analysis to choose the correct fields or methods. Therefore, the relation Hiding (H) for qualified occurrences is the empty relation. That is,

$$H = \varnothing$$

6.4.3 Binding

Binding formulas for qualified occurrences are the same as the Binding formulas for unqualified occurrences.

6.5 Chapter summary

Java is an Object-Oriented programming language. The Refers-To relation is decided by qualification, containment and class hierarchy. This chapter showed that ARA was able to handle Object-Oriented languages.

An interesting fact about Java is that the Java specification is written in the similar fashion as ARA. In addition, C++ specification also adopts the similar way of specification. This showed that ARA was not just a mathematical approach but also the way people talk about the Refers-to relation.

Chapter 7
ARA for C++

C++ is widely known to be complex in many aspects. Features such as multiple inheritance and namespace using-directive make the Refers-to relation in C++ program difficult to extract. This chapter shows that ARA can scale up to handle C++ with the multi-phase ARA pipeline. It also presents an interesting discovery about C++, which is that we need and trinary relations in computing the Refers-to relation for C++, while we only need binary relations for C, Java, CPP, Fortran, Pascal and Ada.

Validation of ARA for C++ means to demonstrate that ARA for C++ satisfies the ISO C++ standard [C++03]. For each rule in the ISO C++ standard that is related to the Refers-to relation, we show that the rule is correctly transcribed into phases (or stages or formulas) of ARA for C++. Section 8.2 will discuss the validation in more detail.

The chapter is organized as follows. Section 7.1 introduces C++ syntax related to the Refers-to relation, including classification of occurrences and the relations that determine the Refers-to relation. Then, section 7.2 gives an overview of ARA pipeline for C++. Section 7.3 uses unqualified occurrences as an example to explain formulas for extracting the Refers-To relations. Section 7.4 concludes this chapter.

7.1 C++ syntax related to the Refers-to relation

C++ syntax related to the Refers-to relation includes the syntax of occurrences and scopes in C++. In addition, C++ uses sub-object lattice to determine name hiding in multiple inheritance. We briefly introduce these concepts based on ISO C++ standard [C++03] [Str01].

7.1.1 Occurrences in C++

In C++ syntax, an occurrence (of an identifier) can appear alone or connected with other occurrences by the scope operator (::), member access operator (.) or points-to operator (->) to form a *qualified name*, such as x::y::z or x.y.z. In such cases, we say that occurrence x qualifies occurrence y and occurrence y qualifies occurrence z. An occurrence that is a part of a qualified name is called *qualified occurrence*. Otherwise, the occurrence is an *unqualified occurrence*.

Recall that in the Refers-to relation, an occurrence can play one of the two roles, the referential occurrence and the definitional occurrence. A definitional occurrence introduces a new entity (type, object, function, namespace, template, …) into the program. A referential occurrence refers to an entity introduced by a definitional occurrence.

The referential occurrences in C++ must be resolved in phases. To resolve references to class members, we need facts about the class hierarchy. To understand the class hierarchy, we must resolve the referential occurrences in class declarations such as

```
class Derived : public Base {  …  };
```

In this example, the occurrence of identifier *Base* must be resolved to another occurrence of *Base* that actually declares the base class. In general, occurrences in class declarations must be resolved before resolving references to class members.

Table 7.1 lists the referential occurrences in various statements. These occurrences must be resolved in different phases and in the order in Table 7.1. The bold font indicates the occurrences that are resolved in the phase.

The first row in Table 7.1 shows occurrences in namespace declaration. The namespace N is declared in two parts, one containing variable x, the other containing variable y. To merge the two parts into one declaration, we compute the Refers-to relation of all namespace declarations (the two N's) before we resolve other occurrences.

After namespace declaration, we should resolve occurrences in using-directive and using-declaration (N and N::x in the second row), and continue until all occurrences (listed in Table 7.1) are resolved.

Note that the statement "abc ++" appears twice in Table 7.1, in rows 5 and 7. In row 5, "abc++" is inside a template definition. In row 7, "abc ++" is just an ordinary statement. The two statements are resolved in different phases.

7.1.2 Scopes in C++

The C++ standard [C++03] lists five kinds of scopes: local scope, namespace scope, class scope, function scope and function prototype scope. Among them, the last two scopes are simple and used less often. Only labels depend on the function scope, which encloses the each function definition. The function prototype scope encloses the function prototype. The local scope, namespace scope and class scope are detailed in the following.

Statements	Use	Examples
Namespace declaration	Merging separate parts of the same namespace	namespace **N** { int x;} namespace **N** {int y;}
using-directive and using-declaration	Making a namespace or a declaration visible	using namespace **N**; using **N:x**;
Class declaration	Determining class hierarchy	Class **Derived** : public **Base** { … };
class member definition	Merging class member defined outside the class body	Void **MyClass** :: **MyFunc**( ) { … }
Template definition	Resolving template-independent types and variables	Template<class T> void func(T x) { **abc**++; …}
Template instantiation	Instantiating templates	Queue<**int**> * pq;
Ordinary statements	Resolving labels, types (struct, class, union, enum) and objects (variables, functions)	**MyClass** * p; **Abc**++;

Table 7.1 C++ occurrences

7.1.2.1 Local scope

A local scope is a portion of program text contained within a block (function definition or compound statement). Declarations in a local scope are visible to a referential occurrence in the same scope or in a nested scope. Besides the declarations residing in the local scope, using-declarations and using directives can introduce declarations from other scope. For example,

```
void f ( ) {
        using N:x;   //N::x is now visible.
}
```

Same as the C language, declarations in the local scope must be declared before it is used.

7.1.2.2 Namespace scope

In essence, a namespace is a block with a name. The outermost namespace scope of a program is called global scope or global namespace. For example,

```
int x;                //x in the global namespace
namespace N {
        int x;        //x in the namespace N
}
int y;                //y in the global namespace
namespace N {
        int y;        //y in the namespace N
}
```

As shown in the example, the definition of a namespace (namespace N in the example) may not be contiguous.

Declarations in a namespace can be referred in three ways. First, declarations in a namespace are visible to referential occurrences in the same or nested scopes. Second, declarations in a namespace can be referred to by qualification. For example, N::x. And finally, they can be referred to by using declaration and using directives. Consider the following example.

```
void f( ){
        using namespace N;          //N::x is visible now
        x++;
}
```

The namespace N contains declaration of variable x. The using-directive "using namespace N" in function f means that the member of namespace N are visible from within function f.

7.1.2.3 Class scope

A class has its own member variables and member functions. In C++, member variables (functions) can be either static or instance. A static member is shared by all objects of the same class. The compiler creates one copy of static member per class. An instance member belongs to each object. The compiler creates a copy of non-static members for each object of the class.

Besides accessing its own members, a class can *inherit* some members from another class. The class that inherits is called the *derived class* and the class that is inherited is called the *base class*.

When class D inherits class B, the compiler creates a separate copy of members of class B for each object of class D. There are two kinds of base class in C++, *virtual base class* and *non-virtual base class*. An example of virtual base class is

```
class V {/* … */};
class A : virtual public V {/* … */};
```

An example of non-virtual base class is

```
class L {/* … */};
class A : public L {/* … */};
```

In C++, one derived class can have several base classes. This is called *multiple inheritance,* where one derived class inherits multiple base classes. Consider an example of multiple inheritance of non-virtual base class (Example 7.1).

```
class L {/* … */};
class A : public L {/* … */};
class B : public L {/* … */ };
class C: public A, B {/* … */};
```

Example 7.1 Non-virtual base class

Both class A and class B inherit class L. An object of class A (or B) has a sub-object of class L. Class C inherits both class A and class B. An object of class C has a sub-object of class A and a sub-object of class B. Because class L is non-virtual base class of class A and class B, an object of class C has two sub-object of class L, one from the sub-object of class A, the other from the sub-object of class B.

The virtual base class acts differently in multiple inheritance. For example (Example 7.2),

```
class V {/* … */};
class A : virtual public V {/* … */};
class B : virtual public V {/* … */ };
class C: public A, B {/* … */};
```

Example 7.2 Virtual base class

Again, class A and class B inherit class V and class C inherits class A and class B. However, class V is a virtual base class, an object of class C has only one sub-object of class V.

The declarations in a class scope can be referred to in four ways. First, declarations in a class scope can be referred to from within the class scope and nested local scopes. Second, a derived class can

refer to declarations in the base classes. Third, they can be referred to by qualification. In particular, we use scope operator (::) to access class members, member access operator(.) to access instance member. If we use a pointer to an object, we access the members using arrow operator (->). And finally, they can be access by using declaration.

7.1.3 Sub-object lattices

The sub-object lattice is a graph used to decide the relation Hiding (H) among classes when there is multiple inheritance[C++03]. Here are examples of sub-or non-virtual and virtual base class adapted from [C++03]. First, consider the non-virtual base class.

```
class L { … };
class A : public L { … };
class B : public L { … };
class C : public A, public B { … };
```

In the example, each object of class A has a sub-object of class L. So does each object of class B. An object of class C has a sub-object of class A and a sub-object of class B and therefore has two sub-objects of class L, one from class A and the other from class B. The sub-object lattice is Figure 7.1. now suppose that both class L and class B have a member called x. B::x can hide L::x from within class B, but not within class C because class C inherits both class A and B, which gives it two copies of class L.

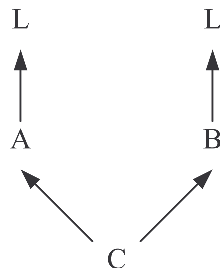

Figure 7.1 Sub-object lattice for non-virtual base class

Now consider virtual base class.

```
class V { … };
class A : public virtual V { .. };
class B : public virtual V { … };
```

class C : public A, public B { … };

In this example, an object of class A has a sub-object of virtual base class V. Similarly, an object of class B has a sub-object of virtual base class V. An object of class C has one sub-object of class A and one sub-object of class B. However, because class V is virtual base class, class C has only one sub-object of class V. The sub-object lattice for this example is Figure 6.2. Suppose that both class V and class B have a member x. B:x can hide V::x from within class B and class C, because this time the object of class C has only one copy of class V.

It is impossible to formalize the operations on the lattice as relational algebra. However, ARA can still use the sub-object lattice by constructing the sub-object lattice using the algorithm used in current compiler and representing the graph as a relation called *SameSubObject* (ST). Suppose x, y and z are classes, then ST is defined as the following.

ST = {<x, y, z> | x and y have the same sub-object of z}.

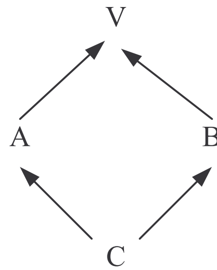

Figure 7.2 Sub-object lattice for virtual base class

In Figure 7.1, a sub-object of C has two copies of sub-object of L, while a sub-object of A has only one copy of sub-object of L. Therefore,

<C, A, L> ∉ ST

On the other hand, in Figure 7.2, a sub-object of C, A and B all have one copy of sub-object of V. Therefore,

<C, A, V> ∈ ST

<C, B, V> ∈ ST

How to use the relation SameSubobject (ST) in deciding the relation Hiding (H)? Suppose that x, y and z are classes and <x, y, z> ∈ ST, then members of class y can hide members of class z from class x.

7.1.4 Other issues related to the Refers-to relation

First, we defer the discussion of the resolution of syntactic ambiguity and template instantiation until Chapter 8 because these topics are related to many languages. In Chapter 8, we will address these topics in the context of all languages studied in this thesis.

Second, in C++, if an unqualified occurrence represents a function call and we cannot find its definition based on containment, inheritance and using directives, then we should look up the namespaces associated with the arguments of the function call and search for its definition. This is called *argument-dependent name lookup*[C++03]. We can implement argument-dependent name lookup with formulas similar to formulas for qualified occurrences. Due to limit on space, argument-dependent name lookup is not discussed.

Finally, overload resolution and access control happen after the Refers-to relation is computed. They can be done based on the Refers-to relation and other syntactical facts.

7.2 Overview Of ARA For C++

ARA for C++ is a multi-phase pipeline (see Chapter 4). It resolves the referential occurrences in the program in phases. Each phase resolves a subset of the referential occurrences and the union of the results from all phases is the Refers-to relation for the program.

7.2.1 Phases in ARA for C++

Phases and their order in ARA for C++ are decided based on Table 7.1. Each row of Table 7.1 is mapped into ARA as a phase. We defer the discussion of templates to Chapter 8.

7.2.2 Inside each phase of ARA for C++: Trinary hiding relation

Each phase of ARA for C++ consists of three steps: Basic Fact Extraction, Normalization and Binding. We will show the detail of these steps in next section. However, what is unique in ARA for C++ is that its relation Hiding is trinary. A tuple <x, y, z> of the relation Hiding (H) means that occurrence y hides occurrence z from occurrence x.

Why can't we still use the binary relation Hiding as we do in ARA for C and Java? Consider Example 7.3 based on the sub-object lattice in Figure 7.1.

class L_1 $\{_2$ void f_3 ( ); $_2\}$;

class A_4 : public L_5 {$_6$

void f_7 ( );

void g_8 ( ) {$_9$ f_{10} ( ); $_9$}

$_6$};

class B_{11} : public L_{12} {$_{13}$ /* … */ $_{13}$};

class C_{14}: public A_{11}, B_{12} {$_{13}$

void h_{14} ( ) {$_{15}$ f_{16} ( ); $_{15}$}

$_{13}$};

Example 7.3 An example of non-virtual base class

Class A inherits class L and occurrence 7 (function f in the derived class A) overrides occurrence 3 (function f in base class L). As a result, occurrence 10 (function call to f) refers to occurrence 7. Now, consider the Refers-to relation for occurrence 16. An object of class C has two copies of sub-object of L and therefore, two copies of occurrence 3. Occurrence 7 only overrides occurrence 3 in class A, but not the occurrence 3 in class B. Therefore, the program is ambiguous and we do not know whether occurrence 16 should refer to occurrence 3 or occurrence 7.

Now, consider Example 7.4 based on sub-object lattice in Figure 7.2. Class V is a virtual base class.

class V_1 {$_2$ void f_3 ( ); $_2$};

class A_4 : public V_5 {$_6$

void f_7 ( );

void g_8 ( ) {$_9$ f_{10} ( ); $_9$}

$_6$};

class B_{11} : public V_{12} {$_{13}$ /* … */ $_{13}$};

class C_{14}: public A_{11}, B_{12} {$_{13}$

void h_{14} ( ) {$_{15}$ f_{16} ( ); $_{15}$}

$_{13}$};

Example 7.4 An example of virtual base class

According to Figure 7.2, the sub-object of class A, B and C all has one copy of sub-object of V. Therefore, occurrence 7 can hide occurrence from both occurrence 10 and occurrence 16. In short, if there exists multiple inheritance, saying that x hides y is not sufficient. We must say that x hides y from z. Similarly, we use trinary relation Hiding if there are using directives.

Now, we have completed the overview of ARA for C++. Next, we use unqualified occurrences in ordinary statements (see Table 7.1) as an example to illustrate the detail of one phase of ARA for C++. Other phases, which are simpler, are summarized in Appendix B.

7.3 Resolving unqualified occurrences in ordinary statements

As an example of ARA phases for C++, this section introduces the Basic Fact Extraction, Normalization and Binding for unqualified occurrences in ordinary statements (Figure 7.3).

7.3.1 Basic Fact Extraction

Basic Facts needed in resolving unqualified occurrences are listed in Table 7.2.

Relation	Explanation
HasName (HN)	{<x, y> | occurrences x has name y}
HasKind (HK)	{<x, y> | occurrence x has kind y}
Follows (F)	{<x, y> | occurrence x follows occurrence y directly
Contains (C)	{<x, y> | block x directly contains occurrence of identifier or block y}
Inherits (I)	{<x, y> | block x directly inherits definitions in block y}
Uses (S)	{<x, y> | block x directly uses definitions in block y}
SameSubobject (ST)	{<x, y, z> | x and y have the same sub-objects of z on the sub-object lattice}

Table 7.2 Basic Facts needed in resolving unqualified occurrences in ordinary statements

The Basic Facts come from three sources. That is

1. Relations HasName, HasKind and Follows come the parser;
2. Relations Contains, Inherits and Uses also come from the parser but results from previous phases will also be merged with them in Normalization.
3. Relation SameSubobject is created by a program that computes sub-object lattice from the relation Inherits.

As we can see, Basic Facts are a very small portion of facts the parser extracts for compilation. They are also straightforward to extract. Now, we use examples to describe the extraction of the Basic Facts, relations Contains and Inherits in particular.

7.3.1.1 Extracting relation Contains

ARA assigns a unique number to occurrences and blocks. The occurrences appearing inside a block is considered as contained by the block. There are three special cases. First, the function parameters are considered as part of the block. For example, ARA considers the parameter x in the following function definition contained inside the block 3.

void f_1 (int x_2) $\{_3$ /* … */ $_3\}$

Second, the scope of a loop or if-statement. In the following for-loop, the declaration of i should be treated as if it is inside block 2.

for (int i_1 =0; ;) {$_2$ /* …*/ $_2$}

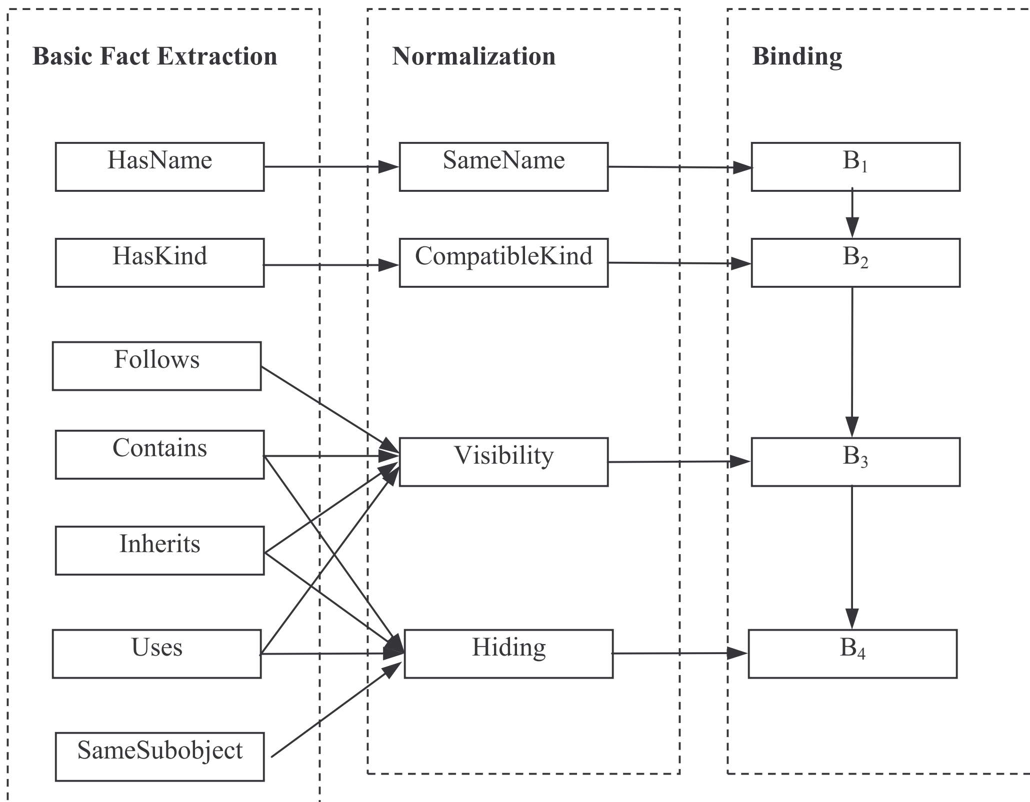

Figure 7.3 Data Flow for resolving unqualified occurrences

And, the declaration of x should be treated as if it is inside block 2 and block 2 includes both branches of the if-statement.

if (int x_1= 0) {$_2$

$_2$}

else {$_2$

$_2$}

Third, the injected class name. A class-name is inserted into the scope of the class. This is called class name injection [C++03, Clause 9]. Therefore, a synonym for the class name A (numbered 3) is inserted into block 2.

class A_1 $\{_2$ /* … */ $_2\}$;

7.3.1.2 Merging results from previous phases

The parser in Basic Fact Extraction only parses the code and has no symbol tables. Therefore, we need merge results from the previous phases into relations Contains, Inherits and Uses, which has been discussed in Section 6.3.

7.3.2 Normalization

Normalization formulas convert the Basic Facts into Normalized Facts (relations SameName, CompatibleKind, Visibility and Hiding). In addition, we also show how the output from other phases is used in Normalization when we explain formulas for relation Visibility.

7.3.2.1 Formula for SameName (N)

Formula for the SameName (N) relation in C++ is the same as those in C and Java. It is

$N = HN \circ HN^{-1}$

7.3.2.2 Formulas for CompatibleKind (K)

In most cases, we can decide the compatible kind for a referential occurrence. For example, occurrence x in x++ should match a variable. But there are cases where we cannot decide the compatible kind in C++. For example, sizeof (x). ARA for C++ creates a catch-all kind for such cases. So, formulas for CompitableKind (K) are the following.

R_0 = HK . “referential occurrence for type and object”

D_0 = HK. {“definitional occurrence for type”, “definitional occurrence for object”}

R_1 = HK . “referential occurrence for type”

D_1 = HK. “definitional occurrence for type”

R_2 = HK . “referential occurrence for object”

D_2 = HK. “definitional occurrence for object”

$K = R_0 \times D_0 \cup R_1 \times D_1 \cup R_2 \times D_2$

7.3.2.3 Formulas for Visibility (V)

Based on rules summarized in Section 7.1.2, the following three kinds of occurrences are visible.

1) Occurrences in the same and enclosing blocks,

2) Occurrences in the super-classes,

3) Occurrences in the namespace nominated in using-directives.

1. Occurrences in the same and enclosing blocks

As in the C language, unqualified occurrences in C++ can see the definitions in the outer blocks (which can be local scope, class scope or namespace scope). We can find these definitions by the following formula:

$$P^+ \circ C$$

In addition, a variable or function that is not a class member must be defined before its use. Suppose the set Members (M) contains members of all classes and the set E contains all occurrences. The cross product, $E \times M$, gives all combination of an entity and a member. The formula

$$F^+ \cup (E \times M)$$

can find all definitions that are either a name declared before or a member declaration of some class. It guarantees that members can be accessed everywhere in the class. Therefore, the following formula returns a relation between an unqualified occurrence and definitions that it can see based on containment relationship.

$$P^+ \circ C \cap (F^+ \cup (E \times M))$$

2. Occurrences in the super-classes

In the C++ language, an unqualified occurrence in class A can see the members of class A and its base classes.

```
class A1 {2  int x3;  2};
class B4 : public A5 {6  int x7;
        void f8 ( ) {9
                x10 ++;
        9}
6};
class C11 : public A12 {13  13};
```

class D_{14} : public B_{15}, public C_{16} $\{_{17}$

void $g_{18}()$ $\{_{19}$

x_{20}++;

$_{19}\}$

$_{17}\}$;

Example 7.4 (Adapted from ISO/IEC C++ 10.1(4))

The relation Inherits (I) records the class hierarchy. For Example 7.4, Basic Fact Extraction records the class definitions as I = {<4, 5>, <11, 12>, <14, 15>, <14, 16>}. But It is much more convenient to use if the relation Inherits is between blocks. That is, for Example 7.4, I = {<6, 2>, <10, 2>, <17, 6>, <17, 10>}. This substitution is done by relation compositions of relation Inherits, Denotes, and output from previous phase (see Table 7.1).

Figure 7.4 is the graph of relations Contains (C) and Inherits (I) for Example 7.4. On this diagram, finding members of class A and its base classes is equivalent to having a path from the occurrence to its enclosing block following the inverse Contains (C) edge, then to base classes via zero or more Inherits (I) edges, and finally to the occurrences in the base classes following Contains (C) edge. In Figure 7.4, the solid arrows represent Contains (C) edges (reverse Parent (P) edges) and dashed arrows represent Inherits (I) edges. Occurrence 20 can see occurrence 3 because there is a path 20-19-17-6-2-3.

Therefore, the *Visibility* (V) relation based on inheritance can be found by the following formulas:

$$P^{+} \circ I^{*} \circ C$$

3. Occurrences in the namespace nominated in using-directives

If block A has a Using-directive "using namespace B", then members in namespace B are visible to occurrences in block A. On the graph of relations Contains (C) and Using (S), this is equivalent to a path from an occurrence to block A following the inverse of Contains (C) edges and from block A to namespace B following one or more Uses (U) edges. Consider Example 7.5.

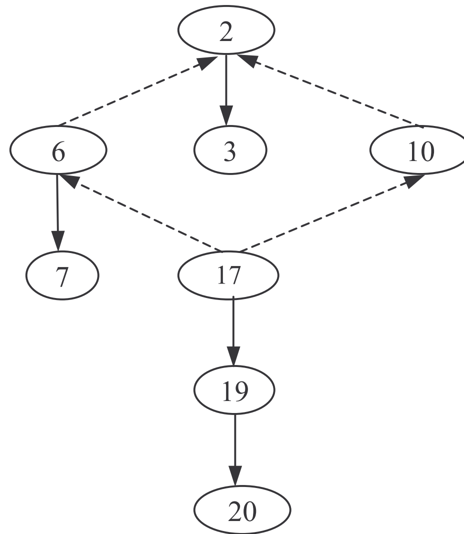

Figure 7.4 Relation Parent (P) and Inherits (I) for Example 7.4

namespace A_1 $\{_2$ int x_3; $_2\}$

namespace B_4 $\{_5$

    using namespace A_6;

    x_7;

$_5\}$

namespace C_8 $\{_9$

    int x_{10};

    void f_{11}( ) $\{_{12}$

        using namespace B_{13};

        $x_{14} = 7$;

    $_{12}\}$

$_9\}$

Example 7.5 An example for namespace

The Relations Contains (C) and Uses (U) for Example 7.5 are shown in Figure 7.5. The solid arrows represent Contains (C) edges (reverse of Parent (P) edges) and dashed arrows represent Uses (S)

edges. Occurrence 14(x) in block 12 can see occurrence 3 in block 2 because block 12 uses block 5 and block 5 uses block 2. I.e. there is a path 14-12-5-2-3.

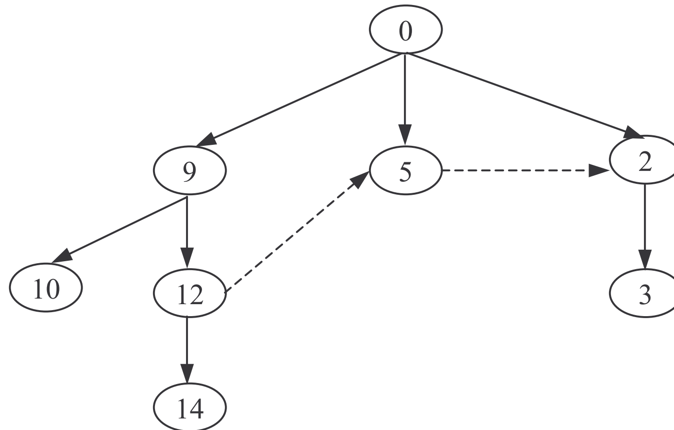

Figure 7.5 Relation Parent (P) and Uses (S) in Example 7.5

Therefore, We can find visible definitions based on using-directives by the following formula.

$$P^+ \text{ o } S^* \text{ o } C$$

Putting formulas for containment, inheritance and using directives together, we have the formula for Visibility (V) as the following.

$$V = (P^+ \text{ o } C \cup P^+ \text{ o } I^* \text{ o } C \cup P^+ \text{ o } S^* \text{ o } C) \cap (F^+ \cup (E \times M))$$

7.3.2.4 Formulas for Hiding (H) Relation

The Hiding (H) relation for C++ is a trinary relation. It is defined as

$$H = \{\langle x, y, z\rangle \mid x, y, z \text{ are occurrences, } y \text{ can hide } z \text{ from } x\}$$

The relation Hiding is used to resolve the name conflicts. The sources of name conflicts are

1. C++'s single name space;
2. Individual Visibility Rules, i.e., containment, inheritance;
3. Interaction between Visibility Rules, i.e. containment vs. inheritance and containment and using directives.

Source 1. C++'s single name space

C++ has one single name space and ISO/IEC specifies the precedence among entities in clause 3.3.7 [C++03]. For example, a class name or enumeration name can be hidden by the name of an object, function, or enumerator declared in the same scope. The following formula is equivalent to such a rule.

D_1 = HK. "definitional occurrence for type"

D_2 = HK. "definitional occurrence for object"

$H = E \times D_2 \times D_1$

where E is the set of all entities. The cross product gives the precedence and the cross product of E makes the relation trinary.

Source 2. Relation Hiding for individual Visibility Rules

First, C++ solves name conflicts in enclosing block by specifying that the definitions in the inner block can hide the definitions in outer blocks. It is the same as rules in C and Java. So, we can use the same formulas but add one more dimension. Suppose P is the *Parent* relation and E is the entity relation for all occurrences in the program. Then, the relation Hiding (H) based on Containment is the following.

$H(x, y, z) = E \times (P \circ P^+ \circ C)$

where E is the set of all entities. The formula in the parenthesis is the same as the formulas in the extensions of ARA for the C language. The cross product of E makes the relation trinary.

Second, C++ uses the sub-object lattice to resolve name conflicts caused by inheritance. In ARA, the sub-object lattice is represented by the relation SameSubobject (ST). Recall that the relation SameSubobject (ST) is defined as

ST: {<x, y, z> | x and y have the same sub-object of z on the sub-object lattice}

If $<x, y, z> \in ST$, then member declarations of class y can hide member declarations of class z from class x. For clarity, we use upper case letters for classes and lower case letters for members. Then the relation Hiding (H) based on Inheritance is the following.

$H(x, y, z) = ST(X, Y, Z) \otimes C(X, x) \otimes C(Y, y) \otimes C(Z, z)$

The compositions ($\otimes$) substitute the classes in the relation ST with the their members and create the relation Hiding for resolving name conflicts caused by inheritance.

Source 3. Relation Hiding for different Visibility rules

First, consider the conflict between containment and using directives. C++ specifies that the names in the nominated namespace appear as if they were declared n the nearest enclosing namespace with contains both the using directive and the nominated namespace [C++03, Clause 7.3.4]. For convenience, we use upper case letters for namespaces, and low case letters for occurrences. Suppose we define a relation NearestCommon (NC) as

$$NC = \{<X, Y, Z> \mid Y \text{ is the nearest common enclosing block of } X \text{ and } Z\}.$$

The relation NearestCommon (NC) can be computed from the relation Contains (C) using relational algebra. And suppose we expand the relation Using (S) to trinary and define a relation S3 as

$$S3 = \{<X, Y, Z> \mid <X, Z> \in S\}.$$

Then, $NC \cap S3$ is $\{<X, Y, Z> \mid X$ uses Z and members in Z appear as if they are in $Y\}$. Therefore, members in any block that is enclosed by Y can hide members in Z, according to C++ rules for enclosing blocks. Then, suppose the Hiding relation at the block level is HB, then

$$NS = NC \cap S3$$

$$HB\ (X, W, Z) = NS\ (X, Y, Z) \otimes C^{+}\ (W, Y)$$

Where C is the relation Contains (C). Substituting namespaces with their members, we get the relation Hiding for resolving conflicts between containment and using directives.

$$H\ (x, y, z) = HB(X, Y, Z) \otimes C\ (X, x) \otimes C(Y, y) \otimes C(Z, z)$$

Second, consider the conflict between inheritance and containment. Because the inherited members are considered its own members, they take precedence over the definitions in the enclosing blocks. Therefore, we can use the same formula in ARA for Java. Of course, we need add one more dimension.

$$H = E \times ((I^{+} \text{ o } C)^{-1} \text{ o } P^{+} \text{ o } C)$$

7.3.3 Binding Formulas

In ARA pipeline for C++, the first three Binding Formulas remain unchanged from Binding Formulas for Java (see Chapter 6). They are

$$B_1 = N$$

$$B_2 = B_1 \cap K$$

$B_3 = B_2 \cap V$

Because the relation Hiding is trinary in C++, we need to adjust the fourth Binding Formula. Suppose that B_4 is the result of Binding Formula 4 and must satisfy all Conditions. First, we introduce relation R as

$$R = \{<x, y> \mid \exists z <x, z> \in B_3 \wedge <x, z, y> \in H\}.$$

In plain English, R consists of pairs <x, y> of B_3 that some definition z can hide y. We can compute R using the definition of n-ary relational composition.

$$R (x, y) = B_3 (x, z) \otimes H (x, z, y)$$

By removing R from B_3, we get the fourth and the last Binding formula in Algebraic Dependence Analysis. Therefore, the Binding Formulas for C++ are the following.

$B_1 = N$

$B_2 = B_1 \cap K$

$B_3 = B_2 \cap V$

$B_4 = B_3 - R \quad \text{where} \quad R (x, y) = B_3 (x, z) \otimes H (x, z, y)$

This section uses an example to explain the Basic Fact Extraction, Normalization and Binding for resolving unqualified occurrences in ordinary statements. Based on Chapter 5 and 6, it is not difficult to write the formulas for qualified occurrences in ordinary statements.

Formulas for other occurrences in Table 7.1 are similar but much simpler because they are resolved in earlier phases and therefore fewer facts (relations) are involved.

7.4 Chapter Summary

The C++ language is known to be complex in many aspects including its Refers-to relation. The occurrences in C++ programs are resolved in phases as listed in Table 7.1. The ARA for C++ has two major extensions. First, relation SameSubObject (ST) is introduced to represent the sub-object lattice. Second, the relation Hiding (H) is extended to trinary relation in ARA for C++. That is, a tuple <x, y, z> of the relation Hiding (H) means that occurrence y hides occurrence z from occurrence x.

As an example of the ARA phase for C++, the resolution of unqualified occurrence in ordinary statements is discussed and the Basic Fact Extraction, Normalization and Binding are presented.

Chapter 8

Advantages of ARA, applications of ARA and miscellaneous

This chapter discusses three topics. First, it presents the three advantages of ARA over the existing approaches. The advantages are 1) ARA formulas are elegant, concise and insightful, 2) ARA is mathematically validated and 3) ARA formulas are supported by existing software tools. The validation is discussed in detail with examples. The three advantages are presented in three sections, Sections 8.1, 8.2 and 8.3.

Second, this chapter discusses the applications of ARA in reverse engineering by listing the wide range of applications of the Refers-to relation that ARA produces. The applications are presented in Section 8.4.

Finally, Section 8.5 discusses some miscellaneous issues such as the resolution of syntactic ambiguities, the handling of macros and templates and features not supported by ARA such as runtime polymorphism.

8.1 ARA is elegant, concise and insightful

During the writing of this thesis, multi-phase pipelines for seven widely used languages, namely C, CPP, C++, Java, FORTRAN, Pascal and Ada are designed. Among them, ARA for C, Java and C++ is presented in Chapters 5, 6 and 7. As we can be seen in Table 8.1, these languages generally use qualification, containment, the order of occurrences, inheritance, importing and other features to determine the Refers-To relation. In this section, we summarize the ARA phases, Basic Fact Extraction, Normalized and Binding in the seven languages and show that ARA is elegant, concise and insightful.

8.1.1 Phases and stages makes ARA concise

The Refers-to relation of some occurrences may depend on the Refers-to relation of others. For example, the Refers-to relation of occurrence y in the qualified name x::y depends on the Refers-to relation of occurrence x.

ARA introduces multi-phase pipeline and uses phases and stages to untangle the dependence. As shown in Figure 8.1, a multi-phase pipeline consists of a series of phases, each dealing with one category of occurrences, for example, class names or namespace name. Within each phase, there are two stages, Stage one for unqualified occurrences and Stage two for qualified occurrences. Each stage is a three-step pipeline of Basic Fact Extraction, Normalization and Binding. In other words, the multi-phase pipeline is a combination of several three-step pipelines. As a result, although multi-

phase pipelines resolve much more difficult programming languages than the three-step pipeline, they are only slightly complicated than the three-step pipeline.

Language	Standard	Refers-To Relation Features
C	ISO/IEC 9899:1999 [C99]	Qualification, containment, order of occurrences
CPP	ISO/IEC 9899:1999 [C99]	Order of occurrences
Java	Java Language Specification, 3rd Ed. [GJSB05]	Qualification, containment, single inheritance, import
C++	ISO/IEC 14882:2003 [C++03]	Qualification, containment, order of occurrences, multiple inheritance, namespace using-directives, namespace alias, argument-dependent name lookup
Fortran	ISO/IEC 1539-1:2004 [For04]	Containment, uses clause, contains clause, interface clause
Pascal	ISO/IEC 10206: 1990(E) [Pas90]	Qualification, containment, order of occurrences, uses clause, import/export clause
Ada	ISO/IEC 8652:1995(E) [Ada95]	Qualification, containment, order of occurrences, single inheritance (Ada95 only), redefine

Table 8.1 Seven languages in case studies.

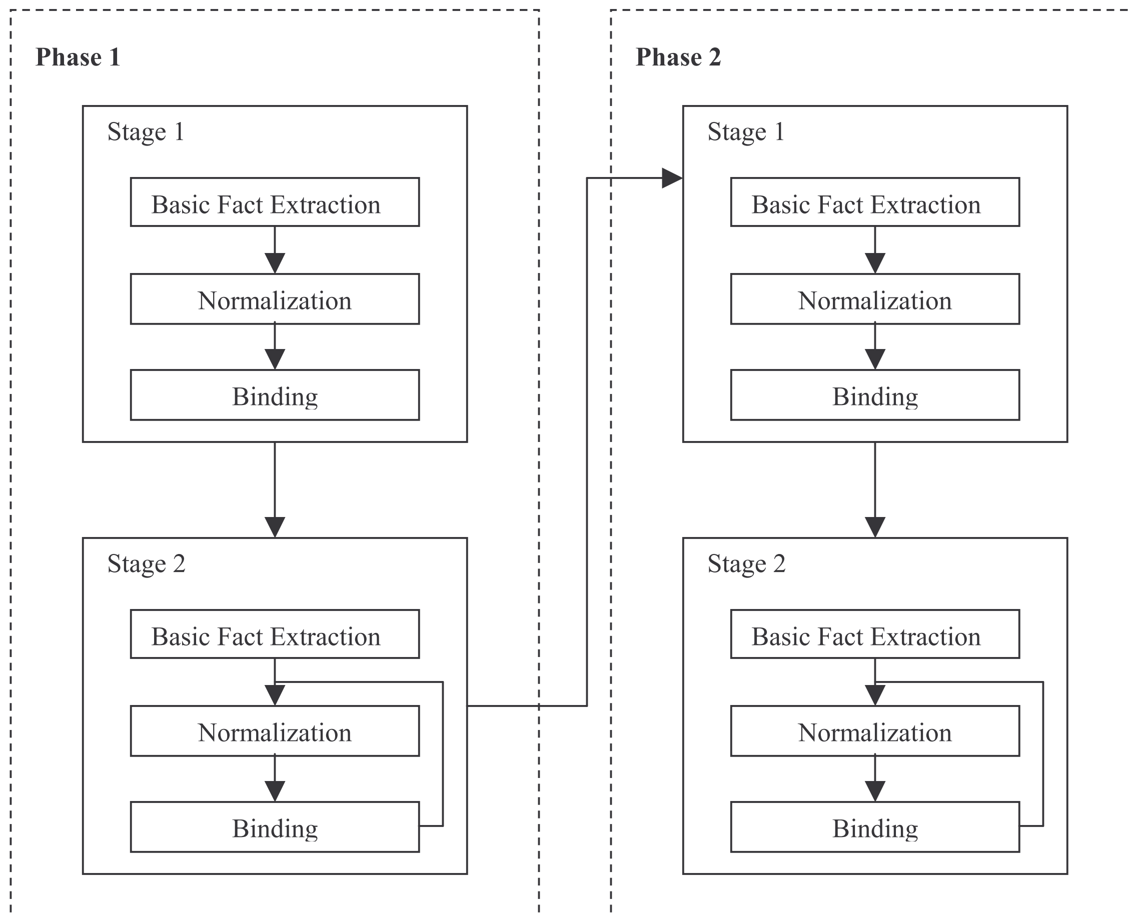

Figure 8.1 Multi-phase pipeline

8.1.2 ARA only uses a small number of Basic Fact Extraction

Recall that Basic Fact Extraction is the process of extracting the facts ARA needs from the source code, usually done by a parser and scanner. ARA needs only a small portion of lexical and syntactic facts in the source code. We can see this from chapters 5, 6 and 7. All the Basic Facts are represented as relations and store them in one place, either a relational database or a text file.

8.1.3 Normalization formulas are elegant and insightful

The Normalization formulas take care of the language-specific rules about the Refers-to relation and converts the Basic Facts into the Normalized Facts. The formulas for Visibility (V) and for Hiding

(H) are elegant and reveal some insightful facts of the programming languages, as detailed in the following two sub-sections.

8.1.3.1 Formulas for Visibility and what they reveal

To compute the relation Visibility, we need to consider the location of the occurrence, the commands that introducing declarations to other scopes, and qualification. Although the wording varies from one language standard to another, the rules in the seven languages studied in the thesis follow the classification. Table 8.3 lists the formulas for Visibility, which are explained in the following.

These formulas show the elegance of relational algebra. For example, we can find the potential matches for an occurrence in the enclosing blocks using the following formula.

$$P^+ \text{ o } C$$

And we can find the potential matches for an occurrence in the base classes using the following formulas.

$$I^+ \text{ o } C$$

The two formulas use different Basic Facts (Contains and Inherits) but follow the same pattern.

In addition, these formulas can be combined easily. For example, ARA formulas for containment and ordering of occurrences can be combined by intersection as the following.

$$P^+ \text{ o } C \cap F^+$$

This formula is used in C, C++, Ada and Pascal.

In addition, the formulas for Visibility also reveal the following interesting facts about programming languages. First, the location of an occurrence means what scope the occurrence is in (Containment) and in what order the occurrences appear. The first three rows show the formulas related to containment. That is, declarations in the same scope, declarations in outer scope and declarations in any enclosing scope. Order of occurrences is often used in Visibility formulas. The fourth row is the formula for all previous declarations. In cases where order of occurrences is ignored, for example, export list in Pascal, class members in C++ and Java, we can use the formula similar to the fifth row.

Second, there are many commands that can introduce declarations into other scopes in the seven language studied in the thesis. The commands can be non-transitive (e.g. use clauses in Fortran, Pascal, Ada, import statement in Java, using-declaration in C++) or transitive (e.g. inheritance in Java and C++, using-directive in C++). The formulas for these commands are in the sixth through eighth row.

Finally, except CPP and Fortran, all languages use qualification. Qualification can make the members in the nominated scope and/or members imported (/inherited) members of the nominated scope visible. Last three rows of Table 8.3 are the formulas for qualification.

Category	Rules	Formulas
Containment	Declarations in the **same scope** (variable in Fortran)	P o C
	Declarations in **outer scope** (subroutine in Fortran)	P o P o C
	Declarations in **any enclosing** scope (C, C++, Java, Pascal, Ada)	P^+ o C
Order	**All Previous** declarations (C, C++, Java, Pascal, Ada, CPP)	F^+
	Declared **anywhere** (e.g. class members in C++, Java)	E × M
Import/ Inherit/ Using	Imports members in **one** scope (Java, Ada, Fortran, Pascal)	P^+ o GM o C
	Inherits members in **all base classes** (C++, Java)	P^+ o I^+ o C
	Member in scopes nominated by using-directive and using-directives are **transitive** (C++)	P^+ o S^+ o C
Qualification	Qualification only (C)	Q^{-1} o R o SP o R o T o C
	Qualification and importing members in **one scope** (Pascal, Ada)	Q^{-1} o R o SP o R o T o GM o C
	Qualification and **all base classes** (C++, Java)	Q^{-1} o R o SP o R o T o I^+ o C

Table 8.2 Formulas for Visibility (V) for different languages

8.1.3.2 Formulas for Hiding (H) and what they reveal

Formulas for the relation Hiding are used to resolve name conflicts (see Table 8.4). They have the same elegance and conciseness we have seen in formulas for Visibility (V). In addition, they reveal the following interesting facts about programming languages.

First, the first row of Table 8.4 is the formula for resolving name conflicts if the kind of an occurrence cannot be decided by the context. In C++, this rule is called hiding. In Java, this rule is called obscure.

Second, rows 2 through 5 are formulas for resolving name conflicts caused by single Visibility rules. For example, if declarations in any enclosing scope are visible (row 3 in Table 8.3), then the declaration in the innermost scope should be chosen (row 2 in Table 8.4). We can see that the conditions such as “the innermost”, “most specific” and “most recently” correspond to the transitive closure of some relation.

Third, last three rows of Table 8.4 are formulas for resolving conflicts between different Visibility rules, e.g. inheritance vs. containment. The formulas are often the composition of two Visibility rules. For example, the seventh row in Table 8.4 is the composition of the third and seventh row in Table 8.3 (Visibility formulas).

Finally, the last row shows that the relation Hiding in C++ is trinary, while the relation Hiding in other languages are binary. It reveals the mathematics behind the facts that the C++ language is more complicated than others.

Sources		Formula
Undecided kind	Definition of object **hides** definition of type (Java, C++)	D_1 = HK. “type_def” D_2 = HK. “object_def” $D_2 \times D_1$
Single Visibility	Declaration in the **innermost** scope (C, Java, C++, Pascal, Ada)	$P \circ P^+ \circ C$
	Most specific member declaration (Java, C++)	$P \circ I^+ \circ C$
	Most recently declaration (CPP)	F^+
	sub-object lattice (C++)	ST
Visibility Interaction	With-clause **hides** enclosing scope (Fortran)	$(W \circ C)^{-1} \circ P \circ C$ where W is with-clause
	Inherited member **hides** declaration in enclosing scope (Java, C++)	$(I^+ \circ C)^{-1} \circ P^+ \circ C$
	Using-directive and enclosing blocks (C++)	HB (X, W, Z) $= NS(X, Y, Z) \otimes P^+ (W, Y)$

Table 8.3 Formulas for Hiding (H) for different languages

8.1.4 Binding formulas are the same for most languages

Binding is the last step of the computation of the Refers-to relation. Previous steps, Basic Fact Extraction and Normalization, have removed most of language-specific parts of the computation.

Therefore, there are only two forms of Binding, one for C, Java, CPP, Fortran, Pascal and Ada, and the other for C++ (see Table 8.5). In addition, the Binding formulas for C++ use one trinary relation while the Binding formulas for other languages use binary relations only. It again shows that the C++ language is more complicated than others.

Language	Binding formulas
C++	$B_1 = N$ $B_2 = B_1 \cap K$ $B_3 = B_2 \cap V$ $B_4 = B_3 - R$, where $R(x, y) = B_3(x, z) \otimes H(x, z, y)$
C, Java, CPP, Fortran, Pascal, Ada	$B_1 = N$ $B_2 = B_1 \cap K$ $B_3 = B_2 \cap V$ $B_4 = B_3 - B_3 \circ H$

Table 8.4 Binding formulas for different languages

8.2 Validating ARA

In this thesis, validation of ARA for a given programming language (e.g. ARA for C) means to show that ARA satisfies the standards for the programming language (e.g. ISO standard for C).

As we have shown, for most programming languages, ARA is a multi-phase pipeline (see Figure 8.1). It consists of several phases, each dealing with names in certain category, for example, names of classes. Inside each phase, there are two stages. Stage 1 resolves the unqualified occurrences. Stage 2 resolves the qualified occurrences. Each stage is a three-step pipeline, consisting of Basic Fact Extraction, Normalization and Binding.

To validate ASA, that is, to validate its a multi-phase pipeline, we need to show that it satisfies the language standard. To do this we must validate the following claims:

1. Phases and stages must satisfy the language standard;
2. Basic Facts must represent the source code according to the language standard;
3. Formulas for Normalization and Binding must satisfy the language standard.

In the following subsections (8.2.2, 8.2.3, 8.2.4), we will show how these claims are validated.

8.2.1 Validating claim 1: Phases and stages must satisfy the language standard

In an ASA multi-phase pipeline, a phase resolves one category of names such as class names. Within each phase, there are two stages, one for the unqualified occurrences, the other for the qualified occurrences. The order of phases and stages is strictly based on the language standards.

Briefly consider the order of phases in C++ (see Appendix B). We must show that the order of these phases satisfies ISO C++ language standard [C++03]. For example, why the Phase 1, resolving namespace declarations precedes Phase 2, resolving namespace member definitions? We can find the reason in the language standard. In C++, namespaces can be declared in parts (Clause 7.3.1 in ISO C++ standard [C++03]). This requires the ARA to identify all parts of a namespace N before it can find the members of N. We must demonstrate that this identification is correctly transcribed from the standard to the ARA formulas. Therefore, the design of Phase 1 and Phase 2 satisfies the language standard.

The order of stages is validated by showing that the order in the ASA corresponds to the ordering specified by the standards. For example, unqualified names need to be dealt with before qualified names. To resolve qualified occurrences y and z in the qualified name x::y::z, ARA first resolves y, then the result of resolving y is fed back to Normalization, and finally z is resolved. This loop satisfies the rules in Clause 3.4.3 of the C++ language standard [C++03].

In summary, we validate the order of ASA phases and stages by showing that their order satisfies the language standards.

8.2.2 Validating claim 2: Basic Facts must represent the source code according to the language standard

Inside each stage of the multi-phase pipeline is a three-step pipeline. Basic Fact Extraction is the first step of these three steps. It extracts facts needed in later steps and phases as relations (Basic Facts). It is straightforward to validate that most of these relation (Basic Facts) are, such as HasName and HasKind are correctly extracted.

However, the language standards contain a few special cases of Basic Facts, mostly related to the Contains relation. We inspect the Basic Fact Extractor (implemented with a parser generator such asYacc or Bison) to ensure these special cases are handled. Consider the if statement in C++. Item 3 of Clause 6.4 in ISO C++ standard specifies that the variables declared in condition of the if statement are visible in both branches. As a result, in the following example variable x is visible in both blocks 2 and 3. I.e., x is contained in both block 2 and 3.

if (int x_1= 0) $\{_2$

$_2\}$

else $\{_3$

$_3\}$

In designing the Basic Fact Extractor, we ensure that the Contains relation include pairs <2, 1> and <3, 1>.

By carefully reading the language standards and the code for Basic Fact Extractor, it is not difficult to collect all special cases and validate Basic Fact Extractor.

8.2.3 Validating claim 3: Formulas for Normalization and Binding must satisfy the language standard

In three-step pipeline, Normalization converts Basic Facts into Normalized Facts and then Binding computes the Refers-to relation from the Normalized Facts. These two steps consist of a series of relational algebra formulas. We validate these formulas with mathematical proof. In other words, for each rule related to Normalization and Binding, we show that the rule is equivalent to one or more formulas based on set theory and relational algebra.

Consider the following rule in ISO C language standard [C99]. Item 4 of Clause 6.2.1 says:

"…If an identifier designates two different entities in the same namespace, the scopes might overlap. If so, the scope of one entity (the inner scope) will be a strict subset of the scope of the other entity (the outer scope). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is hidden (and not visible) within the inner scope" (6.2.1(4)).

This rule is equivalent to the following:

1. y is visible to x: y is declared in a block that encloses x, directly or indirectly;
2. x can hide y: y is declared in a block that enclosed the block where x is declared.

Recall that the relation Contains (C) is the set of pair <x, y> such that x contains y directly. Its reverse P is the set of pair <x, y> such that x is contained in y directly. The transitive closure P^+ is the set of pair <x, y> such that x is contained in y directly or indirectly. By the definition of relational composition, rule 1 is equivalent to the following relational algebra formula.

1. $V = P^+ \text{ o } C$

Now consider rule 2. The block y that enclosed the block where x is declared can be expressed as the composition of P and P^+. Therefore, rule 2 is equivalent to

2. $H = P \text{ o } P^+ \text{ o } C$

The proofs are easier to understand when presented with graphs, which is our way to present these formulas in Chapters 4, 5, 6 and 7. In summary, this section shows that ARA is validated mathematically to satisfy the language standards.

8.3 All major components of ARA are supported by existing software

The three major components of ARA, Basic Fact Extraction, Normalization and Binding are supported by existing software tools. Basic Fact Extraction is implemented using Yacc, Bison or other parser generators. Basic Facts are straightforward to extract. For example, assigning a unique number to occurrences. Normalization and Binding needs no coding because relational database management systems, relational algebra calculators, prolog and many existing software tools can execute the ARA formulas. For example, we can write the formula for SameName (N)

$N = HN \circ HN^{-1}$

as the following SQL statement.

```
SELECT T1.occurrence, T2.occurrence
    FROM HasName T1, HasName T2
    WHERE T1.name = T2.name
```

As another example, we can write the formula for CompatibleKind (K)

U = HK . “ref”

D = HK . “def”

$K = U \times D$

as the following SQL statement.

```
SELECT T1.occurrence, T2.occurrence
    FROM HasKind T1, HasKind T2
    WHERE T1.HasKind = “ref”
        AND
        T2.HasKind = “def”
```

Other relational algebra tools (see Chapter 2) such as Crocopat (Beyer et al [BNL03]) provides a language that is as powerful as the previous approaches, but adds a convenient operator for transitive closure; and the interpreter is efficient for general purpose relational computation, because it is based on BDD technology. In addition, Crocopat is relational calculator for n-ary relations, while Grok is for binary relations only.

Grok is chosen as the notation for this dissertation because formulas written in Grok tends to be more concise, elegant and insightful. We discovered some interesting facts about programming

languages based on these formulas. In ARA for C++, we need trinary relations, which is not part of pure Grok. However, we believe the elegance and insight in the Grok formulas is more important to the whole thesis. As a result, Grok is used in the thesis with a minor extension of the composition of n-ary relations.

In order to show the efficiency of existing software, ARA formulas are tested on Grok, a relational algebra calculator. Finally, Binary Decision Diagram (BDD) can also be used to execute the ARA formulas (see Chapter 2).

The test has two steps. First, we build the Basic Fact Extractor by inserting printf statement into GCC parser. The formulas for Normalization and Binding are written as a Grok script. Second, the fact extractor is used to extract Refers-To relation from open source software. The execution time of ARA is then compared with CPPX because CPPX is a fair representation of fact extractor and has successfully extracted facts from many software systems including DB2.

Table 8.6 shows the results. For example, OpenSSH has roughly 70K line of code, the build time is 47.3 seconds, CPPX execution time is 2 minutes 28.5 seconds, and ARA execution time is 1 minute 46.7 seconds. Overall, ARA performance is satisfactory, comparable to CPPX. Note that CPPX seems to take longer time because it outputs more relations. Tests are done on one Intel ® Pentium ® IV 1.6 GHz CPU and 1GB memory.

System	KLOC	Build	CPPX	ARA
OpenSSH	70	47.3s	2m28.5s	1m46.7s
PostgreSQL	520	2m27.3s	10m23.4s	6m37.2s
Linux kernel, 2.0	5100	25m1.5s	188m24.6s	46m25.7s
Koffice	960	5m46.6s	25m53.2s	12m1.7s

Table 8.5 Test results on open source software

8.4 ARA has a wide range of applications in reverse engineering

ARA has a wide range of applications in reverse engineering because it produces the Refers-to relation, which is crucial for understanding the semantics of a program. We can summarize the application of the Refers-to relation in reverse engineering as follows.

1. Software architecture recovery and repair

The Refers-to relation as extracted from the code is the basis for understanding the architecture of the systems [BHB99] [FHC01]. According to survey of literature, it is also used to recover and repair the software architecture of large software systems [FHC01].

2. Detection of design pattern instances

Design patterns are the abstract description of an object-oriented design and design rationale. The Refers-to relation can be used to identify the design patterns in software systems. Many tools have been developed or extended for the automatic detection of design patterns instances, e.g., Pat [KP96], the tool of Antoniol et al. [AFC98], SPOOL [KSR+99].

3. Detection of design problems

Using the Refers-to relation, we can detect design pattern instance and detect potential design problems such as cyclic references. Many tools are created to detail theses defects, including Crocopat [BNL03], Hy+ [MS95], RPA [FKO98], and Grok [FH00].

4. Calculation of design metrics

A large number of design metrics can be defined in simple and language independent way based on facts about software, the Refers-to relation in particular [ML02]. Using relational algebra tools automates the calculation of metric values without coding.

5. Abstraction of design models

As we can see, it is usually convenient to extract low-level relationships directly from the source code. However, the low-level relationship of a large software system consists of too many nodes and edges. So, it is necessary to derive the relationships between high-level entities from these low-level relationships. This is called lifting and is a common application of relational calculators [FKO98].

6. Derivation of other relationships

Many important relationships between entities in the software can be derived from the Refers-to relation using relational algebra. For example, Ullman [Ull89] first suggested formulating data-flow analysis as database queries based on the Refers-to relation and other facts. Reps [Rep94] used a deductive database for demand-driven inter-procedural data-flow analysis. Jedd [LH04b] formulates the points-to relation as relational algebra formulas of the Refers-to relation and other facts and then solves these formulas using a fast algorithm called BDD.

8.5 Miscellaneous

In this section, we comment on some miscellaneous issues.

8.5.1 Resolution of syntactic ambiguities

There are syntactic ambiguities in many programming language. One type of ambiguity occurs when an occurrence can potentially be interpreted as more than one kind of entity (e.g. both type and variable). The typedef names in C is an example of such ambiguity. Currently, parsers use two ways to disambiguate, both related to the Refers-to relation.

First, the parser creates a symbol table and uses it to decide the kind of an occurrence. This approach is often used for resolving typedef names in C.

Second, the parser puts the ambiguous occurrences in one group, calling it ambiguous occurrences, and matches the group with declarations and then reclassifies the kind of these occurrences. Java compilers [GJSB05] use this approach.

Both approaches can be used with ARA. To use the first approach, we can insert the code for disambiguation in the Basic Fact Extraction (parser). To use the second approach, we can add another phase to ARA, where only the Refers-to relation for the ambiguous occurrences is computed. The relation HasKind is adjusted for these occurrences afterwards.

8.5.2 Macros, templates and generics

The languages studied in the thesis support generic programming with macros, templates and generics, which are expanded or instantiated by the compiler. The Refers-to relation is also needed during the expansion (instantiation). In Section 8.1, the ARA formulas for CPP (C Preprocessor, responsible for expanding macros) have been introduced briefly.

Templates and generics in other languages are more complex because the syntax of templates can be ambiguous and there are different ways of instantiation (e.g. expansion in C++ and erasure in Java). However, the computation of the Refers-to relation for templates and generics is similar to ordinary occurrences. For example, in C++ [C++03, 14.6], occurrences inside the template definition should be resolved as the ordinary occurrences. Then templates are instantiated, the source code is changed and all templates are replaced. The occurrences that are resolved to template parameters are resolved again, in the context of modified source code and using the same rules for resolving other occurrences. So, ARA is able to compute the Refers-to relation of templates as long as we provide the lexical and syntactic facts before and after the instantiation of templates.

8.5.3 Features not supported by current ARA

So far, ARA can only analyze lexical facts and syntactical facts in the program. Information about run-time behavior such as points-to analysis, type casting and stack allocation is not incorporated into ARA. As a result, the current ARA formulas cannot handle runtime polymorphism (e.g., virtual function in C++). By the same token, ARA cannot handle dynamically scoped languages such as LISP.

However, points-to analysis and other analysis can also be expressed as relational algebra [LH04b] [LWL+05]. It is interesting future work to see whether two fields of research can be merged and whether runtime polymorphism and dynamically scoped language can be solved by relational algebra.

8.6 Chapter summary

So far, we have designed ARA pipelines for seven statically typed languages, including C, CPP, Java, C++, Fortran, Pascal and Ada. This chapter summarized the advantages of ARA over the existing approaches. It also presented the applications of ARA in reverse engineering. Finally, this chapter clarified some issues related to resolution of ambiguities, replacement of macros and templates, and features not supported by current ARA.

Chapter 9 Conclusions

This dissertation consists of two parts, each of which deals with a formal approach to fact extraction. The first part presents the level of completeness of a fact extractor, a new approach to characterizing the amount of information a fact extractor extracts from the source code. The advantage of this approach is that we can write a script to test the level of completeness of a fact extractor automatically. With the help of tools such as TXL, the scripts are in general straightforward.

To validate this approach, scripts to test the compiler completeness of CPPX, a real fact extractor, were created. The cases where CPPX violates compiler completeness were found. The tests also showed that compiler completeness is the most practical level of completeness for most fact extractors.

The second part of the thesis presents Algebraic Refers-to Analysis (ARA) as a new approach to fact extraction, with an emphasis on the Refers-to relations and its application to various programming languages.

ARA introduces a new paradigm. It formulates the extraction of the Refers-to relation as a query to find a relation of occurrences. It represents the facts in the source code as relations and transcribes the rules in language standards into relational algebra formulas. During the writing of the thesis, ARA pipelines for seven languages including C, C++, Java, CPP, Fortran, Pascal and Ada are designed. The prototype fact extractor for the C language is created.

Validating ARA means to demonstrate that ARA pipelines satisfy the programming language standards such as ISO C++ standard. In other words, we show that ARA phases (stages and formulas) are correctly transcribed from the rules in the language standard.

Comparing the existing approaches such as Attribute Grammar, ARA has the following advantages. First, ARA formulas are concise, elegant and more importantly, insightful. As a result, we have some interesting discovery about the programming languages. Second, ARA is validated mathematically, which is more reliable than exhaustive testing. Finally, ARA formulas are supported by existing software tools such as database management systems and relational calculators.

In the rest of the chapter, the overall contributions of the thesis and future work are discussed.

9.1 Contributions to fact extraction

This section highlights the contributions this dissertation makes to fact extraction

1. Inventing the concept of hierarchy of completeness and the automatic testing of completeness

Part 1 of the dissertation begins by introducing the concept of completeness of fact extractor. Fact extraction has been a research topic for years. As I began my study, there were various fact extractors. However, validating these fact extractors is not a simple task because it is undecidable whether two programs are equivalent and therefore there does not exist a general algorithm to decide whether the extracted facts have the same meaning as the original source code.

The completeness of a fact extractor is a measure for determining whether a fact extractor is accurate enough to be useful in reverse engineering. We can classify fact extractors into different levels based on their completeness. We can also validate how far a fact extractor is from achieving certain level of completeness. To do this, we design a series of transformation to convert the extracted facts back to the source code and then compare the assembly code generated from the converted code and the original code. If the two versions of assembly code are identical, then we know that the fact extractor is complete.

2. Use of the relational data model in fact extraction

Part 2 of the dissertation has results highlighted here and in the following points 3 and 4. Using Attribute Grammars, people have been modeling the extraction of the Refers-to relation as a process of decorating the parse tree. This approach has succeeded in compiler construction but is time consuming and error prone for reverse engineering. More importantly, Attribute Grammars only formalize the transfer of data on the parse tree, not the extraction of Refers-to relation itself. The rules on the Refers-to relation are implemented as small hand-coded functions or pseudo code.

ARA, which is presented as part 2 of this dissertation, formulates the extraction of the Refers-to relation as a query to find a relation of occurrences. This introduces the relational data model to the extraction of the Refers-to relation. It enables ARA to represent all the facts it needs as relation and stores them in one place, either a database or a text file. It also enables ARA to transcribe the rules in the language standards as relational algebra formulas.

3. Inventing Algebraic Refers-to Relation Analysis (ARA)

This dissertation documents the invention of ARA and ARA formulas for seven practical languages namely C, CPP, Java, C++, Fortran, Pascal and Ada. Based on the language standards, an ARA pipeline consists of a series of phases. Each phase resolves a subset of occurrences in the program, for example, occurrences in the import declaration. Each phase is divided into at most two stages. The first stage resolves the unqualified occurrences. The second stage resolves the qualified occurrences. In each stage, there are three steps, Basic Fact Extraction, Normalization and Binding.

ARA follows the language standards closely. The major components of the ARA pipeline, Basic Fact Extraction, Normalization and Binding are transcribed from the language standards. Concepts such as the Binding Conditions, Visibility and Hiding are inspired by the language standards for Java and C++. In addition, the phases and stages in ARA are created based on the dependency between occurrences, which can be inferred from the language standard straightforwardly.

4. Discovery of some interesting facts of programming languages

During the design of ARA, some interesting aspects of programming languages are discovered. For example, we only need binary relations in ARA for C, CPP, Java, Fortran, Pascal and Ada. But we need both binary relation and trinary relation in ARA for C++.

Considering the whole dissertation, the levels of completeness of fact extractor and ARA are fundamental to the maturity of the research field of reverse engineering. The completeness of fact extractor provides a declarative approach to testing fact extractors. ARA formulates fact extraction in an innovative way and gives relational algebraic formulas of extracting the Refers-to relation that are supported by many existing software tools. Implementing the extraction of the Refers-to relation with ARA requires minimum coding and yet the implementation is accurate and verifiable against the language standards. These inventions can help programmers to extract facts they need accurately and quickly and to focus on the analysis they intend to perform on the source code.

9.2 Future work

Possible future work that stems from this dissertation is considered below.

9.2.1 Applying ARA to dynamically scoped languages

ARA is a promising new technique for extracting facts and has bee used in seven statically scoped languages including C, CPP, Java, C++, Fortran, Pascal and Ada. So far, we have not studied the Refers-to relation in a dynamically scoped languages such as LISP.

We believe that it is possible for ARA to handle dynamically scoped languages. As we have seen in this thesis, ARA models fact extraction as a query to find a set of occurrences. As long as we represent the facts as relations and write the query as relational algebra, we can use ARA to solve the problem. Since many problems related to the program's behavior at run time have been formulated as relational algebra problems, we believe that it is an interesting and promising future work to extract facts from dynamically scoped languages using ARA.

9.2.2 Using ARA in program transformation

Program transformation techniques are used in a many areas of software engineering ranging from program synthesis, via program optimization and program refactoring, to reverse engineering and documentation generation. As discussed in Chapter 2, most program transformation systems analyze the Refers-to relation based on Attribute Grammars, whose drawbacks have been discussed in this thesis.

Incorporating ARA in program transformation systems such as TXL has at least two advantages over the current approach. First, ARA makes program transformation systems more concise and reliable. Comparing to Attribute Grammars, ARA formulas are concise, easy to understand and, as shown in the thesis, are verified against the language standards.

Second, ARA uses relational data model. This data model is easy to extend and therefore is able to provide more facts to program transformation systems.

In summary, incorporating ARA in program transformation is promising future work in ARA.

Appendix A Visibility and name hiding rules in Java

The following are rules for visibility and name hiding for Java. For each visibility rule, we summarize it as declarations and their scope and list the page number for the rule. For name hiding rules (shadowing, hiding, obscuring), we summarize it as declaration that hides and the declaration that is hidden and list the page number.

1. Visibility rules in Java specification [GJSB05].

Declaration	Scope	Pages
Top level package	compilation unit	Page 117
Type imported	compilation unit	Page 117
Member imported by single-static_import	compilation unit	Page 117
Top level type	package + no order	Pages 117, 119
Member	current class and nested type	Page 118
Parameter of type parameter	Method	Page 118
Type parameter	body + type parameter section (no order)	Page 118
Local variable	block + order	Page 118
Local in for loop	the whole loop(forinit, expression forupdate)	Page 118

Table 1. Visibility rules in Java specification [GJSB05]

2. Name hiding rules in Java specification [GJSB05].

2.1 Shadowing

That hides	That is hidden	Page
Type	Type already in scope	Page 119
Filed, local	Field, local, … already in scope	Page 119
Method	Method in enclosing scope (class or blocks)	Page 119
Single-type-import	Top level type name in other compilation unit	Page 119
	Type-import-on-demand	Page 119
	Static-import-on-demand	Page 119
Single-static-import	Static-import-on-demand	Page 119
	Top level type name in other compilation unit	Page 119
	Type-import-on-demand	Page 119

Table 2. Rules for shadowing

2.2 Obscuring

That hides	That is hidden	Page
Variable	Type, package	Page 122
Type	Package	Page 122

Table 3. Obscuring rules

2.3 Hiding

That hides	That is hidden	Page
Field in derived (static or instance)	Field in super class or interface (static or instance)	Page 196
Static method	Static method	Page 225
Member class in derived	Member class in super class	Page 237

Table 4. Hiding rules

Appendix B List of formulas in ARA for C++

This appendix gives the formulas for ARA for C++. These are arranged into seven phases, as shown in Table 1. Each phase has two stages. First stage resolves the unqualified occurrences. The second resolves the qualified occurrences. Because they are the same for each phase, the stages are not listed in Table 1.

Phase	Use	Examples
1. Namespace declaration	Merging separate parts of the same namespace	namespace **N** { int x;} namespace **N** {int y;}
2. Namespace member definition	Merging namespace member defined outside the namespace body	void N::f() { }
3. Using-directive and using-declaration	Making a namespace or a declaration visible	using namespace **N**; using **N:x**;
4. Class declaration	Determining class hierarchy	Class **Derived** : public **Base** { … };
5. Class member definition	Merging class member defined outside the class body	Void **MyClass** :: **MyFunc**( ) { … }
6. Template definition	Resolving template-independent types and variables	Template<class T> void func(T x) { **abc**++; …}
7. Template instantiation	Instantiating templates	Queue<**int**> * pq;
8. Ordinary statements	Resolving labels, types (struct, class, union, enum) and objects (variables, functions)	**MyClass** * p; **Abc**++;

Table 1. Phases in ARA for C++

The following are the formulas for resolving unqualified occurrences in each phase. The formulas for resolving qualified occurrences can be derived based on the formulas for resolving unqualified occurrences and are not included in the following tables.

1. Phase for namespace declarations

Step	Relation	Definition
Basic Fact Extraction	HasName	HN = {<x, y> | occurrences x has name y}
	HasKind	HK ={<x, y> | occurrence x has kind y}
Normalization	SameName	$N = HN \circ HN^{-1}$
	Compatible-Kind	R_0 = HK . "namespace name" D_0 = HK. "namespace name" $K = R_0 \times D_0$
	Visibility	$V = E \times E$, where E is the set of all occurrences
	Hiding	$H = \emptyset$
Binding	Condition 1	$B_1 = N$
	Condition 2	$B_2 = B_1 \cap K$
	Condition 3	$B_3 = B_2 \cap V$
	Condition 4	$B_4 = B_3 - B_3 \circ H$

Table 2. Formulas for namespace declaration

2 Phase for using-directives

Step	Relation	Definition
Basic Fact Extraction	*HN, HK*	*Same as previous phase*
	Follows	F= {<x, y> | occurrence x follows occurrence y directly }
	Contains	C = {<x, y> | block x directly contains occurrence of identifier or block y}
	Uses	S = {<x, y> | block x directly uses definitions in block y}
Normalization	*N*	*Same as previous phase*
	Compatible-Kind	R_0 = HK . "occurrence in using-directive or using-declaration" D_0 = HK. {"namespace name", "definitional occurrence of object"} $K = R_0 \times D_0$
	Visibility	$V = (P^+ \circ C \cup P^+ \circ S^* \circ C) \cap (F^+ \cup (E \times M))$
	Hiding	$H = \varnothing$
Binding	B_1, B_2, B_3, B_4	*Same as previous phase*

Table 3. Formulas for using-directives

Note: when there is more than one using-directive in one block, the subsequent using-directives depend on the first using-directive. Therefore, the unqualified occurrence in these using-directives should be resolved during the second stage of the phase, i.e. the stage when ARA resolves qualified occurrences.

3 Phase for class declaration

Step	Relation	Explanation
Basic Fact Extraction	*HN, HK, F, C, S*	*Same as previous phase*
Normalization	*N*	*Same as previous phase*
	Compatible-Kind	R_0 = HK . “referential occurrence in class definition” D_0 = HK . “class name” $K = R_0 \times D_0$
	Visibility	*Same as previous phase*
	Hiding	*Same as previous phase*
Binding	B_1, B_2, B_3, B_4	*Same as previous phase*

Table 4. Formulas for class declarations

4 Occurrences in class member definition

Step	Relation	Explanation
Basic Fact Extraction	*HN, HK, F, C, S*	*Same as previous phase*
Normalization	*N*	*Same as previous phase*
	Compatible-Kind	R_0 = HK . “occurrence in class member definition” D_0 = HK. {“definitional occurrence for type”, “definitional occurrence for object”} $K = R_0 \times D_0$
	Visibility	*Same as previous phase*
	Hiding	*Same as previous phase*
Binding	B_1, B_2, B_3, B_4	*Same as previous phase*

Table 5. Formulas for class member definition

5 Occurrences in template definition

Step	Relation	Explanation
Basic Fact Extraction	*HN, HK, F,* *C, S*	*Same as previous phase*
	Inherits (I)	$\{\langle x, y\rangle \mid \text{block x directly inherits definitions in block y}\}$
	SameSubobject (ST)	$\{\langle x, y, z\rangle \mid \text{x and y have the same sub-objects of z on the sub-object lattice}\}$
	N	*Same as previous phase*
	Compatible- Kind	R_0 = HK . “referential occurrence for type and object inside template” D_0 = HK. {“definitional occurrence for type”, “definitional occurrence for object”} R_1 = HK . “referential occurrence for type inside template” D_1 = HK. “definitional occurrence for type” R_2 = HK . “referential occurrence for object inside template” D_2 = HK. “definitional occurrence for object” $K = R_0 \times D_0 \cup R_1 \times D_1 \cup R_2 \times D_2$
Normalization	Visibility	$V = (P^+ \text{ o } C \cup P^+ \text{ o } I^* \text{ o } C \cup P^+ \text{ o } S^* \text{ o } C) \cap (F^+ \cup (E \times M))$
	Hiding	$NS = NC \cap S3$ $HB (X, W, Z) = NS (X, Y, Z) \otimes C^+ (W, Y)$ $H(x, y, z) = E \times (P \text{ o } P^+ \text{ o } C)$ $\cup\ ST (X, Y, Z) \otimes C (X, x) \otimes C (Y, y) \otimes C (Z, z)$ $\cup\ HB (X, Y, Z) \otimes C (X, x) \otimes C (Y, y) \otimes C (Z, z)$ $\cup\ E \times ((I^+ \text{ o } C)^{-1} \text{ o } P^+ \text{ o } C)$
Binding	B_1, B_2, B_3, B_4	$B_1 = N$ $B_2 = B_1 \cap K$ $B_3 = B_2 \cap V$ $B_4 = B_3 - R$ where $R (x, y) = B_3 (x, z) \otimes H (x, z, y)$

Table 6. Formulas for template definition

6 Occurrences in template instantiation

Step	Relation	Explanation
Basic Fact Extraction	*HN, HK, F, C, S, I, ST*	*Same as previous phase*
Normalization	*N*	*Same as previous phase*
	Compatible-Kind	R_0 = HK . "referential occurrence for type and object referring to template parameter" D_0 = HK. {"definitional occurrence for type", "definitional occurrence for object"} R_1 = HK . "referential occurrence for type referring to template parameter" D_1 = HK. "definitional occurrence for type" R_2 = HK . "referential occurrence for object referring to template parameter" D_2 = HK. "definitional occurrence for object" $K = R_0 \times D_0 \cup R_1 \times D_1 \cup R_2 \times D_2$
	Visibility	*Same as previous phase*
	Hiding	*Same as previous phase*
Binding	B_1, B_2, B_3, B_4	*Same as previous phase*

Table 7. Formulas for occurrences in template instantiation

7 Ordinary statements

Step	Relation	Explanation
Basic Fact Extraction	*HN, HK, F, C, S, I, ST*	*Same as previous phase*
Normalization	*N*	*Same as previous phase*
	Compatible-Kind	R_0 = HK . "referential occurrence for type and object" D_0 = HK. {"definitional occurrence for type", "definitional occurrence for object"} R_1 = HK . "referential occurrence for type" D_1 = HK. "definitional occurrence for type" R_2 = HK . "referential occurrence for object" D_2 = HK. "definitional occurrence for object" $K = R_0 \times D_0 \cup R_1 \times D_1 \cup R_2 \times D_2$
	Visibility	*Same as previous phase*
	Hiding	*Same as previous phase*
Binding	B_1, B_2, B_3, B_4	*Same as previous phase*

Table 8. Formulas for occurrences in ordinary statements

Bibliography

[Aca96] Wiki page for Aca Cia. Website: http://www.program-transformation.org/Transform/AcaCia. Last accessed Oct. 2007.

[Ada95] International Organization for Standardization. ISO/IEC 8652:1995, 1995.

[AFC98] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in Object-Oriented software. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 153–160. IEEE, 1998.

[AKW79] Alfred Aho, Brian Kernighan and Peter Weinberger. AWK – A pattern scanning and processing language. *Software Practice and Experience,* 9(4):267-280, 1979.

[APMV03] Giuliano Antoniol, Massimiliano Di Penta, Gianluca Masone and Umberto Villano. XOgastan: XML-oriented GCC AST analysis and transformation. In *Proceedings of the Third International Workshop on Source Code Analysis and Manipulation (SCAM '03). IEEE*, 2003.

[App98] Andrew Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[ASU86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[AT98] M.N. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proceedings of Fifth Working Conference on Reverse Engineering (WCRE '98)*, pages 30-39, October 1998.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proceedings of Symposium on Principles of Programming Languages*, pages 110–120. ACM, 1979.

[Bac73] Charles W. Bachman. The programmer as navigator. *Communications of the ACM,* Volume 16, Issue 11, pages 653 – 658, 1973.

[Bad00] Greg Badros. JavaML: A Markup language for Java source code. *Computer Networks*, 33(1-6):159-177, June 2000.

[BBM98] Ralf Behnke, Rudolf Berghammer, Erich Meyer, and Peter Schneider. RELVIEW – A system for calculating with relations and relational programming. In Egidio Astesiano, editor, *Proceedings of the 1st InternationalConference on Fundamental Approaches to Software Engineering (FASE 1998)*, LNCS 1382, pages 318–321. Springer-Verlag, 1998.

[Boy96] J. Boyland. Conditional Attribute Grammars. *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 1, pages 73-108, Jan. 1996.

[BDK96] M. van den Brand, A. van Deursen, P. Klint, S. Klusener and E. van der Meulen, Instrial applications of ASF+SDF, In *Algebraic Methodology and Software Technology (AMAST' 96)* pages 9-18, 1996.

[Bel01] Bell Canada. *DATRIX Abstract Semantic Graph Reference Manual, Version 1.4*. Website: http://citeseer.ist.psu.edu/332503.html, last accessed Oct. 2007.

[Beh99] R. Behnke et al. Applications of the RelView system. *Tool support for system specification, development and verification, Advances in Computing Science*, Springer: Wien, pages 33-47, 1999.

[BHB99] Ivan T. Bowman, Richard C. Holt and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering*, pages 555-563. Los Angeles, California, May 1999.

[BG97] Berndt Bellay, Harald Gall. A comparison of four reverse engineering tools. In *Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 2-11, Amsterdam. October 1997.

[BG00] Marat Boshernitsan and Susan L. Graham, Designing an XML-based exchange format for Harmonia. *In Proceedings of Working Conference on Reverse Engineering*, pages. 287-289, Brisbane, Queensland, Australia, November 23-25, 2000.

[Bis92] Walter Bischofberger. Sniff: A pragmatic approach to a C++ programming environment. In *USENIX C++ Conference,* pages 67-82, Portland, Oregon, August 1992.

[BKMS95] Walter Bischofberger, Thomas Kofler, Kai-Uwe Mäetzel and Bruno Schäffer. Computer supported software engineering with beyond-sniff. In *Software Engineering Environments,* pages 145-143, Los Alamitos, California, April 1995.

[BKMV03] Mark van den Brand, Steven Klusener, Leon Moonen, and Jurgen Vinju. Generalized parsing and term rewriting - Semantics directed disambiguation. In *Third Workshop on Language Descriptions Tools and Applications, Electronic Notes in Theoretical Computer Science*, 2003.

[BKU96] R. Berghammer, B. von Karger, and C. Ulke. *Relation- Algebraic analysis of Petri Nets with RELVIEW*, pages 49–69. Springer-Verlag, Berlin, Passau, March 1996.

[BLM02] R. Berghammer, B. Leoniuk, and U. Milanese. Implementation of relational algebra using Binary Decision Diagrams. In *Proc. RelMiCS'01*, LNCS 2561, pages 241–257. Springer, 2002.

[BLQ+03] M. Berndl, O. Lhot´ak, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceeding of Programming Language Design and Implementation*, pages 103–114. ACM, 2003.

[BN04] D. Beyer and A. Noack. Crocopat 2.1 introduction and reference manual. Technical Report CSD-04-1338, University of California, Berkeley, 2004.

[BNL03] D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. *In Proceedings of the 10th IEEE Working Conference on Reverse Engineering*, pages 216–225, Nov. 2003.

[C99] International Organization for Standardization. *Programming languages – C. ISO/IEC 9899:1999*, 1999.

[C++03] International Organization for Standardization. *Programming languages – C++. ISO/IEC 14882:2003*, 2003.

[CC00] Anthony Cox and Charles Clarke. A comparative evaluation of techniques for syntactic level source code analysis. In *7th Asia-pacific Software Engineering conference,* pages 282-289, Singapore, December 2000.

[CC01] Anthony Cox and Charles Clarke. Representing and accessing extracted information. In *proceedings of IEEE International Conference on Software Maintenance (ICSM 2001)* pages 12-21, Nov. 2001.

[CC03] Anthony Cox and Charles Clarke. Syntactic approximation using iteractive lexical analysis. In *Proceedings of the 11th International Workshop on Program Comprehension*, pages 154-163, Portland, Oregon, May 2003.

[CDM+02] James R. Cordy, Thomas R. Dean, Andrew J. Malton and Kevin A. Schneider. Source transformation in software engineering using the TXL transformation system. In *Special Issue on Source Code Analysis and Manipulation, Journal of Information and Software Technology 44,13.* Pages 827-837, October 2002.

[CGK98] Yih-Farn Chen, Emden R. Gansner, Eleftherios Koutsofios. A C++ data model supporting reachability analysis and dead code detection. *IEEE Trans. Software Eng*. 24 (9): 682-694 (1998), 1998.

[CM90] Mariano P. Consens, Alberto O. Mendelzon. GraphLog: A visual formalism for real life recursion, In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 404-416. Nashville, Tennessee, United States 1990.

[CMR92] Mariano P. Consens, Alberto O. Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *International Conference on software Engineering,* pages 138-156, Melbourne, Australia, May 1992.

[CNR90] Yih-Farn Chen, Michael Nishimoto and Chitoor Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering,* 16(3):325-334, March 1990.

[Cod70] Edgar Codd. A relational data model for large shared data banks. *Communications of the ACM,* 13(6):377-387, June 1970.

[Cod82] Edgar Codd. Relational Database: A practical foundation for productivity. *Communications of the ACM, 25(2):* pages 109-117, 1982

[Cor06] J.R. Cordy, 2006. The TXL source transformation language. *Science of Computer Programming* 61,3, pages 190-210, August 2006.

[Cox] Anthony M. Cox. *A Source-based Approach to Representing and Managing Information Extracted by Program Analysis*. PhD thesis. 2002.

[CPP02] CPPX. C++ Source Code Extractor. Website: http//swag.uwaterloo.ca/~cppx. Last accessed Oct. 2007.

[Cre97] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the First Conference on Domain Specific Languages,* pages 229-242, October 1997.

[Csc04] Cscope. Website: http://cscope.sourceforge.net, 2004. Last access Oct. 2007.

[DC90] G. Dueck and G. Cormack. Modular Attribute Grammars. *The Computer Journal*, Vol. 33, No. 2, pages 164-172, 1990.

[DCMS02] Thomas R. Dean, James R. Cordy, Andrew J. Malton and Kevin A. Schneider. Grammar programming in TXL. In *Proceedings of Second IEEE International Workshop on Source Code Analysis and Manipulation*. Montréal, October 2002.

[Dev92] Premkumar Devanbu. GENOA – A customizable, language and front-end independent code analyzer. In *Proceedings of 14th International Conference on Software Engineering,* pages 307-317, Melbourne, Australia, May 1992.

[DMH01] Thomas R. Dean, Andrew J. Malton and Ric Holt. Union schemas as a basis for a C++ extractor. In *Proceedings of Working Conference on Reverse Engineering (WCRE 2001): Working Conference on Reverse Engineering*, Stuttgart, Germany, 2001.

[EBN97] Michael Ernst, Greg Badros and David Notkin. *An Empirical Analysis of C Preprocessor Use*. Computer Science and Engineering UW-CSE-97-04-06, University of Washington, Seattle, Washington, April 1997.

[EGHT94] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: a tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering,* December 1994.

[EH04] T. Ekman and G. Hedin, Rewritable reference attributed grammars. In *European Conference on Object-Oriented Programing (ECOOP'04)*, LNCS 3086, pages 144–169, 2004.

[EKW98] Jürgen Ebert, Bernt Kullbach and Andreas Winter. GraX – An interchange format for reengineering tools. In *Sixth Working Conference on Reverse Engineering,* pages 89-98, Atlanta, Georgia, October 1998.

[EWB+96] Jürgen Ebert, Andreas Winter, Peter Dahm, Angelika Franzke and Roger Süttenbach. Graph based modeling and implementation with EER/GRAL. In Thalheim, B. *15th International Conference on Conceptual Modeling (ER'96), Proceedings*. LNCS 1157, pp. 163-178, Berlin. Springer-Verlag, 1996.

[FBT+02] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen and Tibor Gyimóthy. Columbus - Reverse engineering tool and schema for C++. In *Proceedings of the International Conference on Software Maintenance (ICSM 2002)*, IEEE Computer Society, 2002.

[FBMG01] Rudolf Ferenc, Arpad Beszedes, Ferenc Magyar and Tibor Gyimothy. A short introduction to Columbus/CAN. Technical Report, 2001.

[FH00] H. Fahmy and R. C. Holt. Software architecture transformations. In *Proceedings of 16th International Conference on Software Maintenance,* pages 88–96, IEEE, 2000.

[FHC01] Hoda M. Fahmy, Richard C. Holt and James R. Cordy. Wins and losses of algebraic transformations of software architecture. *Automated Software Engineering (ASE 2001)*, San Diego, California, November 26-29, 2001.

[FHK+97] Pat Finnigan, Richard C. Holt, Ivan Kalas, Scott Kerr, Kostas Kontogiannis, Hausi M¨uller, John Mylopoulos, Steve Perelgut, Martin Stanley, and Kenny. Wong. The software bookshelf. *IBM Systems Journal*, 36(4), November 1997.

[FHM97] Hoda Fahmy, Richard C. Holt, Spiros Mancoridis Repairing software style using graph grammars. In *the IBM Proceedings of the Seventh Centre for Advanced Studies Conference (CASCON'97)*, Toronto, Ontario, Canada, November, 1997.

[FKO98] L. Feijs, R. Krikhaar, and R. Van Ommering. A relational approach to support software architecture analysis. *Software-Practice and Experience*, 28(4):371–400, 1998.

[FMY92] R. Farrow, T. Marlowe, and D. Yellin. Composable Attribute Grammars: Support for modularity in translator design and implementation. In *Proceeding of 19th ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 223-234, New Mexico, USA, Jan. 1992.

[For04] International Organization for Standardization. *ISO/IEC 1539-1:2004, Fortran*. 2004.

[FSH+01] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, Tibor Gyimothy, Towards a standard schema for C/C++, In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, Stuttgart, Germany, Oct. 2001.

[FSG04] Rudolf Ferenc, Istvan Siket and Tibor Gyimothy. Extracting facts from open source software. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 60-69, Chicago, Illinois, September 2004.

[GCC02] GCC. GNU Compiler Collection. Website: http://gcc.gnu.org, 2002.

[GCD03] X. Guo, J. R. Cordy and T. R. Dean. Unique renaming of Java using source transformation. In *Proceedings of the 3rd International Workshop on Source Code Analysis and Manipulation*, pages 151-160, Amsterdam, September 2003.

[GHL+92] R.W. Gray, V.P. Heuring, S.P. Levi, A. M. Sloane, and A.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM 35,* pages 121-131, Feb. 1992.

[Gie88] R. Giegerich. Composition and evaluation of Attribute Coupled Grammars. *Acta Informatica*, Vol. 25, pages 355-423, 1988.

[Gro89] J. Grosch. *Ag-An Attribute Evaluator Generator*. Report No. 16, Compiler Generation Project, University of Karlsruhe, 1989.

[Gro92] Josef Grosch, *Multiple Inheritance in Object-Oriented Attribute Grammars*, Document No. 28, CoCoLab – Datenverarbeitung Höhenweg 6, 77855 Achern Germany Feb. 25, 1992.

[GX] GCC_XML. Website: http://www.gccxml.org/HTML. Last accessed Oct. 2007.

[GXL02] GXL. *The Graph eXchange Language.* Website: http://www.gupro.de/GXL. Last accessed Oct. 2007.

[Hed92] G. Hedin. *Incremental Semantic Analysis*, Dept. Computer Science, Lund University, Sweden, March 1992.

[HHL+00] Ahmed E. Hassan, Richard C. Holt, Bruno Lague, Sebastien Lapierre, and Charles Leduc. E/R schema for the DATRIX C/C++/Java exchange format. In *Proceedings of the 7th Working Conference on reverse Engineering*, pages 284-284, Brisbane, Australia, November 2000.

[HMP03] Mark Hennessy, Brian A. Malloy and James F. Power. gccXfront: Exploiting gcc as a Front End for Program Comprehension Tools via XML/XSLT. *International Workshop on Program Comprehension (IWPC '03)*. 2003.

[Hol02] Richard C. Holt. Introduction to the Grok programming language. Website: http://plg.uwaterloo.ca/~holt/papers/grok-intro.doc. Last accessed Oct. 2007.

[Hol97a] Richard C. Holt. An introduction to TA: The Tuple-Attribute language. Website: http://plg.uwaterloo.ca/~holt/papers/ta-intro.htm. Last accessed Oct. 2007.

[Hol98] Richard C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Proceedings of the 5th Working Conference on Reverse Engineering,* Honolulu, Oct 1998.

[Hol99] Richard C. Holt. Software architecture abstraction and aggregation as algebraic manipulations. In *Proceedings of Centre for Advanced Studies Conference* (*CASCON '99)*, Toronto, November 1999.

[Holt] Richard Holt. Software Bookshelf: Overview and construction. Website: http://www-turing.cs.toronto.edu/pbs/papers/bsbuild.html. Last access Oct. 2007.

[Jar98] Stan Jarzabek. Design of flexible static program analyzers with PQL. *IEEE Transactions on Software Engineering,* 24(3):197-215,March 1998.

[GJSB05] James Gosling, Bill Joy, Guy L. Steele Jr. Gilad Bracha. *The Java Language Specification*, Third Edition, Sun Microsystems, Inc. 2005.

[JD84] R. Jullig, and F. DeRemer. Regular Right-Part Attribute Grammars. *ACM SIGPLAN '84 Symp. On Compiler Construction*, pages 171-178, June 1984.

[Jon90] L. Jones. Efficient evaluation of Circular Attribute Grammars. *ACM Trans. on Programming languages and Systems*, Vol. 12, No. 3 pages 429-462, July 1990.

[JP91] M. Jourdan, and D. Parigot. Internals and externals of the Fnc-2 Attribute Grammar systems. *Attribute Grammars, Applications and Systems,* LNCS No. 545, pages 485-504, Springer-Verlag, 1991.

[JP97] M. Jourdan, D. Parigot. *The FNC-2 System User' s Guide and Reference Manual.* Release 1.19, INRIA Rocquencourt, 1997.

[JV03] D. Janzen and K. de Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd Annual Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, 2003.

[KHZ82] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, LNCS No. 141, Springer-Verlag, 1982.

[KGW98] Rainer Koschke, Jean-Francois Girard and Martin Wurthner. An intermediate representation for integrating reverse engineering analyses. *Presented at Working Conference on Reverse Engineering*, Honolulu, HI, October 1998.

[KHZ82] U. Kastens, B. Hutt, E. Zimmermann. GAG: A practical compiler generator. *Lecture Notes in Computer Science* 141, Springer-Verlag, 1982.

[Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems: The Theory* 2, pages 127-146, June 1968.

[KNP88] K. Koskimies, O. Nurmi, and J. Paakki. The design of a language processor generator. *Software-Practice and Experience*, Vol. 18, No.2 pages 107-135, Feb. 1988.

[Kos91] K. Koskimies. Object-Orientation in Attribute Grammars. *Attribute Grammars, Applications and Systems*, LNCS No. 545, pages 297-329, Springer-Verlag, 1991.

[KP96] C. Kra¨mer and L. Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. *Proc. Third Working Conf. Reverse Eng. (WCRE 1996),* pp.208-215, 1996.

[KSR+99] R.K. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-Based Reverse Engineering of Design Components. *Proc. 21st Int'l Conf. Software Eng. (ICSE 1999),* pp. 226-235, 1999.

[KW92] U. Kastens and W. M. Waite. Modularity and reusability in Attribute Grammars. *Acta Informatica*, Volume 31 , Issue 7 pages 601-627, October 1994.

[KW99] B. Kullbach and A. Winter. Querying as an enabling technology in software reengineering. In *Proc. CSMR,* pages 42–50, 1999.

[Let02] Timothy C. Lethbridge et al. The Dagstuhl Middle Model (DMM) Version 0.005. See http://scgwiki.iam.unibe.ch:8080/Exchange/2. Last accessed May 2007.

[LH04] Yuan Lin and Richard Holt. Software factbase extraction as algebraic transformations: FEAT. In *Proceedings of 1st International Workshop on Software Evolution Transformations (SET 2004)*, page 21-24, Delft, Netherland, November 2004.

[LH04b] O. Lhot´ak and L. Hendren. Jedd: a BDD-based relational extension of Java. *In PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming language Design and Implementation*, pages 158–169, 2004.

[LR95] David Ladd and J. Christopher Ramming. A*: A language for implementing language processors. *IEEE Transactions on Software Engineering*. 21(11):894-901. November 1995.

[LSW01] Carola Lange, Harry Sneed and Andreas Winter: Comparing graph-based program comprehension tools to relational database-based tools. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001),* Toronto, CA, May 2001.

[LWL+05] Monica Lam et al. Context-sensitive program analysis as database queries. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART, Symposium on Principles of Database Systems*. Pages 1-12. 2005.

[MG00] Victor Matos and Rebecca Grasser. RELAX — the relational algebra pocket calculator project. *SIGCSE Bull.* vol. 4, no. 4, pages 40-44, 2000.

[MK00] Evan Mamas and Kostas Kontogiannis. Towards portable source code representations using XML. In *Seventh Working Conference on Reverse Engineering*, pages 172-182, Brisbane, Austrilia, November 2000.

[MK88] Hausi Muller and Karl Klashinsky. Rigi: A system for programming-in-the-large. In *10th International Conference on Software Engineering*, pages 80-86, Singapore, April 1988.

[ML02] T. Mens and M. Lanza. A Graph-Based Metamodel for Object-Oriented Software Metrics. *Electronic Notes in Theoretical Computer Science*, vol. 72, no. 2, 2002.

[MLA+00] M. Mernik, M. Lenic, E. Avdicausevic and V. Zumer. Multiple Attribute Grammar Inheritance. *Informatica*, Vol. 24, No. 3, pages 319-328, June 2000.

[MNL96] Gail C. Murphy, David Notkin, and Erica S.-C. Lan. An empirical study of static call graph exractors. In the *Proceedings of the 18th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pages 90-99, 1996.

[MNS01] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Software Engineering*, 27(4):364–380, 2001.

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley,and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4), pages 181-204, December 1993.

[MS95] A. O. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software – Concepts and Tools*, 16(4):170–182, 1995.

[OK90] R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, USA, 1990.

[Pas90] International Organization for Standardization. *ISO/IEC 10206: 1990(E),* 1990.

[PP96] S. Paul and A. Prakash. A query algebra for program databases. IEEE Trans. *Software Engineering*, 22(3):202–217, 1996.

[Pro03] Swedish Institute of Computer Science. *Quintus Prolog User's Manual*, 2003.

[PS99] André Postma, Marc Stroucken. Applying Relation Partition Algebra for reverse architecting. *Workshop Software-Reengineering*, Bad Honnef, pages 27-28. May 1999.

[Rep92] T. Reps. Scan Grammars: Parallel attribute evaluation via data-parallelism. TR 1120, Computer Sciences Department, University of Wisconsin-Madison, November 1992.

[Rep94] T. Reps. Demand interprocedural program analysis using logic databases. *Applications of Logic Databases*, pages 163–196, 1994.

[RG00] Raghu Ramakrishnan, Johannes Gehrke, *Database Management Systems*, 2nd Ed,McGraw-Hill, pages 28-45, 2000.

[RT89] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, 1989.

[RW91] David Rosenblum and Alexander Wolf. Representing semantically analyzed C++ code with Reprise. In *USENIX C++ Conference,* pages 119-134, Berkeley, California, 1991.

[San01] Georg Sander. VCG Overview. Website: http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html. Last accessed Oct. 2007.

[SCHC99] Susan Sim, Charles Clarke, Ric Holt and Anthony Cox. Browsing and searching software architectures. In *International Conference on Software Maintenance*, pages 381-390, Oxford, England, September 1999.

[Sch98] Andy Schürr, *The PROGRES Language Manual* Version 9.X, Lehrstuhl Informatik III, RWTH Aachen, 1998.

[SHE02] Susan Elliott Sim, Richard C. Holt, Steve M. Easterbrook. On using a benchmark to evaluate C++ extractor. In *Proceedings of International Workshop on Program Comprehension (IWPC '02)*, Paris, June, 2002

[SK90] Yoichi Shinoda and Takuya Katayama. Object-Oriented extension of Attribute Grammars and its implementation using distributed attribute evaluation algorithm. *In Proceedings of the International Workshop on Attribute Grammars and their Applications*, Lecture Note in Computer Science Vol. 461, pages 177-191. Springer-Verlag, 1990.

[SoC99] Software Composition Group, University of Berne. The FAMIX 2.0 Specification, 2.0 edition, Aug. 1999. website: http://www.iam.unibe.ch/~scg/Archive/famoos/FAMIX. Last accessed Oct. 2007.

[SoN03] Source navigator 5.1.4. website: http://sourcenav.sourceforge.net/, Last accessed Oct. 2007.

[SSC96] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proc. ICSE*, pages 387–396. IEEE, 1996.

[Str01] Bjarne Stroustrup, *The C++ programming language* (special edition), Addison-Wesley, Pearson Education, 2001

[Swa02] SwagKit. *The Software Architecture Group (SWAG) Analysis Toolkit*. Website: http://swag.uwaterloo.ca/swagkit, 2002.

[SWZ95] Andy Schürr, Andreas Winter and A. Zündorf, Graph grammar engineering with PROGRES, in *ESEC'95 Proceedings of the 5th European Software Engineering Conference, Lecture Notes in Computer Science*, volume 989, pp. 219-234, Springer-Verlag, Berlin, 1995.

[Tar41] Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, Vol. 6, No. 3, pages 73-89, 1941.

[TC90] T. Teitelbaum and R. Chapman. Higher-Order Attribute Grammars and editing environments. *ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, Pages 197-208, 1990.

[TGLH02] John B. Tran, Michael W. Godfrey, Eric H. S. Lee, and Richard C. Holt. Architectural repair of open source software. In *Proceedings of International Workshop on Program Comprehnesion (IWPC'02)*, 2002.

[TH99] John B. Tran and R. C. Holt. Forward and reverse repair of software architecture. In *Proceeding of Centre for Advanced Studies Conference (CASCON* '99), Toronto, November 1999.

[TkS03] TkSee/SN. *A C++ source code extractor based on Cygnus Source Navigator*. Website: http://www.site.uottowa.ca/~tcl/kbre, 2003. Last accessed Oct. 2007

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md., 1989.

[Vis04] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools and systems in StrategoXT 0.9. Technical Report UU-CS-2004-011, Institute of Information and Computing Sciences, Utrecht University, February 2004.

[VSK90] H. Vogt, S. Swierstra, and M. Kuiper. Higher order Attribute Grammars. *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, Pages 131-45, 1989.

[Win01] Andreas Winter. Exchanging graphs with GXL. In P. Mutzel (ed.) *Graph Drawing - 9th International Symposium, GD 2001*, Vienna, Springer-Verlag. 2001.

[WL04] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[Won98] Kenny Wong. *The Rigi User's Manual - Version 5.4.4*. The Rigi Group, June 1998. See http://www.rigi.csc.uvic.ca/

[Wu04] Jingwei Wu. Jgrok: A query language for reverse engineering. Website: http://swag.uwaterloo.ca/tools.html, 2004. Last accessed Apr. 2007.