ALGORITHM AND IMPLEMENTATION FOR EXTRACTING SEMANTIC INFORMATION FROM LEGACY APPLICATION CODE

By

SANGEETHA SHEKAR

A THESIS PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2003
Copyright 2003

by

Sangeetha Shekar
To Prashant and my Mother
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Joachim Hammer, for giving me the opportunity to work on this topic under his supervision. Without his continuous guidance and constant encouragement this thesis would not have been possible. I also want to thank Dr. Mark S. Schmalz and Dr. Raymond Issa for being on my supervisory committee and for their invaluable suggestions throughout this project. I would like to thank all my colleagues in SEEK, especially Nikhil, Huanqing, Oguzhan, and Laura, who assisted me in my thesis. I would also like to thank Sharon Grant for making the Database Center a nice work environment.

I am grateful to my family, especially my mother, for her constant encouragement and support in every decision I made towards shaping my career. I would also like to thank Prashant for always being there for me through my many ups and downs in the past two years and for being such an understanding friend.

Most importantly, I would like to thank God for always taking care of me and helping me come this far.

I would like to acknowledge the National Science Foundation for supporting this research under grant numbers CMS-0075407 and CMS-0122193.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>x</td>
</tr>
</tbody>
</table>

## CHAPTER

1 INTRODUCTION ...........................................................................................................1

1.1 Motivation .............................................................................................................2
1.2 Solution Approaches .........................................................................................4
1.3 Challenges and Contributions ...........................................................................5
1.4 Organization of Thesis ......................................................................................7

2 RELATED RESEARCH .................................................................................................8

2.1 Program Comprehension ......................................................................................9
2.2 Lexical and Syntactic Analysis .........................................................................11
2.3 Control Flow Analysis .......................................................................................12
2.4 Data Flow Analysis ..........................................................................................13
2.5 Program Dependence Graphs ..............................................................................14
2.6 Program Slicing ................................................................................................14
2.7 Business Rule Extraction ..................................................................................16
2.8 Cliché Recognition ............................................................................................18
2.9 Pattern Matching ...............................................................................................19

3 SEMANTIC ANALYSIS ALGORITHM ........................................................................20

3.1 Algorithm Design ................................................................................................23
  3.1.1 Heuristics Used ............................................................................................24
  3.1.2 Semantic Analysis Algorithm Steps ..........................................................29
3.2 Java Semantic Analyzer .....................................................................................38
4 IMPLEMENTATION OF THE JAVA SEMANTIC ANALYZER..............................42
  4.1 Implementation Details ............................................................................. 42
  4.2 Illustrative Example .................................................................................. 51

5 QUALITATIVE EVALUATION OF THE JAVA SEMANTIC ANALYZER PROTOTYPE.....................................................................................................................59

6 CONCLUSION..............................................................................................................69
  6.1 Contributions ................................................................................................. 70
  6.2 Limitations ..................................................................................................... 71
    6.2.1 Extraction of Context Meaning ............................................................... 71
    6.2.2 Semantic Meaning of Functions .............................................................. 72
  6.3 Future Work .................................................................................................... 73
    6.3.1 Class Hierarchy Extraction ................................................................. 73
    6.3.2 Improvements to the Algorithm ............................................................. 73

APPENDIX

A GRAMMAR USED FOR THE 'C' CODE SEMANTIC ANALYZER ...............75
B GRAMMAR USED FOR THE JAVA SEMANTIC ANALYZER .........................81
C TEST CODE LISTING.........................................................................................87
D REDUCED SOURCE CODE GENERATED BY JAVA PATTERN MATCHER .......90
E AST FOR THE TEST CODE..............................................................................93
F SEMANTIC ANALYSIS RESULTS OUTPUT ...............................................101
LIST OF REFERENCES ....................................................................................104
BIOGRAPHICAL SKETCH ...............................................................................108
<table>
<thead>
<tr>
<th>Table</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1 Information maintained by the pre-slicer for slicing variables</td>
<td>53</td>
</tr>
<tr>
<td>4-2 Signatures of methods defined in the source file maintained by the pre-slicer</td>
<td>53</td>
</tr>
<tr>
<td>4-3 Semantic knowledge extracted for slicing variable $t_{\text{finish}}$</td>
<td>55</td>
</tr>
<tr>
<td>4-4 Semantic information gathered slicing variable $t$</td>
<td>57</td>
</tr>
<tr>
<td>4-5 Semantic information for variable $t_{\text{finish}}$ after the merge operation</td>
<td>58</td>
</tr>
<tr>
<td>Figure</td>
<td>page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>2-1 Program slicer driven by input criteria.</td>
<td>16</td>
</tr>
<tr>
<td>3-1 Conceptual build-time architecture of SEEK’s knowledge extraction algorithm.</td>
<td>20</td>
</tr>
<tr>
<td>3-2 Semantic analysis implementation steps.</td>
<td>32</td>
</tr>
<tr>
<td>3-3 Generation of an AST for either C or Java code.</td>
<td>35</td>
</tr>
<tr>
<td>3-4 Substeps executed inside the analyzer module.</td>
<td>37</td>
</tr>
<tr>
<td>3-5 Substeps executed inside the Java SA analyzer module.</td>
<td>40</td>
</tr>
<tr>
<td>4-1 Semantic Analyzer code block diagram.</td>
<td>43</td>
</tr>
<tr>
<td>4-2 Java Pattern Matcher code block diagram.</td>
<td>45</td>
</tr>
<tr>
<td>4-3 Java Pattern Matcher data structures.</td>
<td>46</td>
</tr>
<tr>
<td>4-4 Methods and data members of <code>FunctionsDefined</code> class.</td>
<td>48</td>
</tr>
<tr>
<td>4-5 Semantic analysis results data structure.</td>
<td>50</td>
</tr>
<tr>
<td>4-6 Reduced AST generated by the code slicer for slicing variable <code>tfinish</code>.</td>
<td>54</td>
</tr>
<tr>
<td>4-7 Screen snapshot of the ambiguity resolver user interface.</td>
<td>56</td>
</tr>
<tr>
<td>5-1 Code fragment depicting the types of parameters that can be passed to a <code>resultSet</code> <code>get</code> method.</td>
<td>60</td>
</tr>
<tr>
<td>5-2 SQL query composed using the string concatenation operator (+).</td>
<td>61</td>
</tr>
<tr>
<td>5-3 Code fragment demonstrating indirect output statements.</td>
<td>62</td>
</tr>
<tr>
<td>5-4 Code fragment demonstrating context meaning of variables.</td>
<td>64</td>
</tr>
<tr>
<td>5-5 Business rules involving method invocations on slicing variables.</td>
<td>65</td>
</tr>
<tr>
<td>5-6 Code fragment showing slicing variable <code>tstart</code> passed to two functions.</td>
<td>66</td>
</tr>
</tbody>
</table>
As the need for enterprises to participate in large business networks (e.g., supply chains) increases, the need to optimize these networks to ensure profitability becomes greater. However, due to the heterogeneities of the underlying legacy information systems, existing integration techniques fall short in enabling the automated sharing of data among participating enterprises. Current techniques require manual effort and significant programmatic set-up. This necessitates the development of more automated solutions to enable scalable extraction of knowledge resident in legacy systems of a business network, to support efficient sharing. Given the fact that an application is a rich source for semantic information including business rules, in this thesis we have developed algorithms and methodologies to extract semantic knowledge extraction from legacy application code.

Despite the fact that much effort has been invested in areas of program comprehension and in researching techniques to extract business rules from source code, no
comprehensive solution has existed before this work. In our research, we have developed an automated approach for extracting semantic knowledge from legacy application code. Our methodology integrates and improves upon existing techniques, including program slicing, program dependence graphs and pattern matching, and advances the state-of-the-art in many ways, most importantly to reduce dependency on human input and to remove some of the other limitations.

The semantic knowledge extracted from the legacy application code contains information about the application specific meaning of entities and their attributes as well as business rules and constraints. Once extracted, this semantic knowledge is important to the schema matching and wrapper generation processes. In addition, this methodology can be applied, for example, to improving legacy application code and updating the documentation for the source code.

This thesis presents an overview of our approach. Evidence to demonstrate the extraction power and features of this approach is presented using the prototype that has been developed in our Scalable Extraction of Enterprise Knowledge (SEEK) testbed in the Database Research and Development Center at the University of Florida.
CHAPTER 1
INTRODUCTION

In the current era of E-Commerce, factors such as increased customizability of products, rapid delivery, and online ordering or purchasing have greatly intensified the competition in the market but have left enterprises to deal with the problems arising out of the customer centric approach. For example, the high degree of variability in work orders or demands in combination with the need for rapid delivery limits the ability of a single enterprise to mass produce a certain product and thereby limits its ability to bring uniformity to its production. Enterprises are unable to mass-produce products, leading to increased costs of operation and low profit margins. This justifies the need for a production in a supply chain and extensive enterprise collaboration. An enterprise or business network is comprised of several individual enterprises or participants that collaborate in order to achieve a common goal (e.g., produce goods or services with small lead times and variable demand). Recent research has led to an increased understanding of the importance of coordination among subcontractors and suppliers in a business network (Ballard and Howell 1997, Koskela and Vrijhoef 1999). Hence, there is a requirement for decision or negotiation support tools to improve the productivity of an enterprise network by improving the user’s ability to co-ordinate, plan, and respond to dynamically changing conditions (O’Brien et al. 1995).

The utility and success of such tools and systems greatly depend on their ability to support interoperability among heterogeneous systems (Wiederhold 1992). Currently, the time and investment involved in integrating such heterogeneous systems that help an
enterprise network to achieve a common goal are significant stumbling blocks. Data and knowledge integration among systems in a supply chain requires a great deal of programmatic set up and human hours with limited code reusability. There is a need to develop a toolkit that can semi-automatically discover enterprise knowledge from enterprise sources and use this knowledge to configure itself and act as a software or “glue-ware” between the legacy sources. The SEEK\(^1\) project (Scalable Extraction of Enterprise Knowledge) that is currently underway at the Database Research and Development Center at the University of Florida is directed at developing methodologies to overcome some of the problems of assembling knowledge resident in numerous legacy information systems (Hammer et al. 2002a, 2002b, 2002c).

### 1.1 Motivation

A legacy source is defined as a complex stand-alone system with poor or outdated documentation of the data and application code. Frequently, the original designer(s) of such a data source are not available to provide information about design and semantics. A typical enterprise network has contractors and sub-contractors that use such legacy sources to manage their data and internal processes. The data present in these legacy sources are an important input to decision making at the project level. However, a large number of firms collaborating on a project imply a higher degree of physical and semantic heterogeneity in their legacy systems due to a number of reasons stated below. Thus, developers of enterprise-level decisions support tools are faced with four practical difficulties related to accessing and retrieving data from the underlying legacy source.

\(^1\) This project is supported by National Science Foundation under grant numbers CMS-0075407 and CMS-0122193.
The first problem faced by enterprise-level decisions support tools is that the firms can use various internal data storage, retrieval and representation methods. Some firms might use professional database management systems while some others might use simple flat files to store and represent their data. There are many interfaces including SQL or other proprietary languages that a firm may use to manipulate its data. Some firms might manually access the data at the system level. Due to such high degrees of physical heterogeneity, retrieval of similar information from different participating firms amounts to a significant overhead including extensive study about the data stored in each firm, detection of approach used by the firm to retrieve data, and translation of queries to manipulate the data into the corresponding database schema and query language used by the firm.

The second problem is heterogeneity among terminologies of the participating firms. The fact that a supply chain usually comprises firms working in the same, or closely related, domains does not rule out variability in the associated vocabulary or terminology. For example, firms working in a construction supply chain environment might use Task, Activity, Work-Item to refer to an individual component of the overall project. Although all these terms have the same meaning, it is important to be able to recognize that. In addition, data fields may have been added over time that have names that provide little insight into what these fields actually represent. This semantic heterogeneity manifests itself at various levels of abstraction, including the application code that may have business rules encoded therein, making it important to establish relationships between the known and unknown terms to help resolve semantic heterogeneities.
Another important problem when accessing enterprise code is that of preventing loss of data and unauthorized access—hence the access mechanism should not compromise on privacy of the participating firm’s data and business model. It is logical to assume that a firm can restrict sharing of enterprise data and business rules even among other participating firms. It is therefore important to be able to develop third party tools that have access to the participating firm’s data and application code to extract semantic information but at the same time assure the firm of the privacy of any information extracted from its code and data.

Lastly, the existing solutions require extensive human intervention and input with limited code reusability. This makes the knowledge extraction process tedious and cost inefficient.

Thus, it is necessary to build scalable data access and extraction technology that have the following desirable properties:

- Automates the knowledge extraction process as much as possible.
- Must be easily configurable through high level specifications.
- Reduces the amount of code that must be written by reusing components.

1.2 Solution Approaches

The role of the SEEK system is to act as an intermediary between the legacy data and the decision support tool. Based on the discussion in the previous section, it is crucial to develop methodologies and algorithms to facilitate discovery and extraction of knowledge from legacy sources. SEEK has a build-time component (data reverse engineering) and a run-time component (query translation). In this thesis we focus exclusively on the build-time component, which operates in three distinct phases. In general, SEEK (Hammer et al. 2002a) operates as a three-step process:
SEEK generates a detailed description of the legacy source including entities, relationships, application-specific meanings of the entities and relationships, business rules. The Database Reverse Engineering (DRE) algorithm extracts the underlying database conceptual schema while the Semantic Analyzer (SA) extracts application-specific meanings of the entities, attributes, and the business rules used by the firm. We collectively refer to this information as *enterprise knowledge*.

2. The semantically enhanced legacy source schema must be mapped onto the domain model (DM) used by the application(s) that want(s) to access the legacy source. This is done using a schema mapping process that produces the mapping rules between the legacy source schema and the application domain model.

3. The extracted legacy schema and the mapping rules provide the input to the wrapper generator, which produces the source wrapper. The source wrapper at run-time translates queries from the application domain model to the legacy source schema.

This thesis mainly focuses on the process and related technologies highlighted in phase 1 above. Specifically, we focus on developing robust and extendable algorithms to extract semantic information from application code written for a legacy database. We will refer to this process of mining business rules and application-specific meanings of entities and attributes from application code as *semantic analysis*. The application-specific meanings of the entities and attributes and business rules discovered by the Semantic Analyzer (SA), when combined with the underlying schema and constraints generated by the data reverse engineering module, give a comprehensive understanding of the firm’s data model.

### 1.3 Challenges and Contributions

Formally, semantic analysis can be defined as the application of analytical techniques to one or more source code files to elicit semantic information (e.g., application-specific meanings of entities and their attributes and business logic) to provide a complete understanding of the firm’s business model. There are numerous challenges in the process of extracting semantic information from source code files with respect to the objectives of SEEK; these include but are not limited to the following:
Most of the application code written for databases is written in high-level languages like C, C++, Java, etc. The semantic information to be gathered may be dispersed across one or more files. Thus the analysis is not limited to a single file. Several passes over the source code files and careful integration of the semantic information thus gathered is required.

The SA may not always have access or permissions to all the source code files. The accuracy and the correctness of the semantic information generated should not be affected by the lack of input. Even partial or incomplete semantic information is still an important input to the schema matcher in phase 3.

High-level languages, especially object oriented languages like C++ and Java, have powerful features such as inheritance and operator overloading, which if not taken into account, would generate incomplete and potentially incorrect semantic information. Thus, the SA has to be able to recognize overloaded operators, base and derived classes, etc. thereby making the semantic analysis algorithm intricate and complex.

Due to maintenance operations, the source code and the underlying database are often modified to suit the changing business needs. Frequently, attributes with non-descriptive, even misleading names may be added to relations. The associated semantics for this attribute may be split up among many statements that may not be physically contiguous in the source code file. The challenge here is to develop a semantic analysis algorithm that discovers the application-specific meaning of attributes of the underlying relations and captures all the business rules.

Human intervention in the form of comments by domain experts is typically necessary. See, for example, Huang et al. (1996) where the SA merely extracts all the lines of code which directly represent business rules. The task of presenting the business rule in a language independent format is left to the user. Such an approach is inefficient, incomplete, and not scalable. We present all the semantic information gathered about an attribute or entity in a comprehensive fashion with the business logic encoded in a XML document.

The semantic analysis approach should be general enough to work with any application code with minimal parameter configuration.

The most important contribution of this thesis is a detailed description of the SA architecture and algorithms for procedural languages such as C, as well as object oriented languages such as Java. Our design has addressed and solved each one of the challenges stated above. This thesis also highlights the main features of the SA and proves that our design is scalable and robust.
1.4 Organization of Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of the related research in the field of semantic information extraction from application code and business rules extraction in particular. Chapter 3 provides a description of the SA architecture and semantic analysis algorithms used for procedural and object oriented languages. Chapter 4 is dedicated to describing the implementation details of SA using the Java version as our basis for the explanations, and Chapter 5 highlights the power of the Java SA in terms of what features of the Java language it captures. Finally, Chapter 6 concludes the thesis with a summary of our accomplishments and issues to be considered in the future.
CHAPTER 2
RELATED RESEARCH

Over the past decade, much research has been done to overcome the heterogeneity at various levels of abstraction such as work on sharing architectures and languages (Sheth and Larson 1990), mediation (Ullman 1997) and source wrappers (Hammer et al. 1997a, 1997b). Wrapper technology (Nestorov et al. 1997) especially plays an important role in light of the rising popularity of cooperative autonomous systems. Different approaches to develop a mediator system have also been described in (Ashish and Knoblock 1997, Gruser et al. 1998, Nestorov et al. 1997). Data mining (Huang et al. 1996) uses a combination of machine learning, statistical analysis, modeling techniques, and database technology, to discover patterns and relationships in data. The preceding approaches require detailed knowledge of the internal database schema, business rules, and constraints used to represent the firm’s business model.

Industrial legacy database applications often have tens of thousands of lines of application code that maintain and manipulate stored data. The application code evolves over several generations of developers; original developers of the code may have left the project. Documentation for the legacy database application may be poor and outdated. The internal database schema may have been modified hastily, to accommodate new concepts without too much emphasis on design principles. As a result, the new relations and attributes could have non-intuitive and non-descriptive names. Therefore, not only is it important to extract the underlying database schema and the conceptual structure, but also to discover application specific meanings of the entities and relations. It is also
important to note that the relevant information about the underlying concepts and their meaning is usually distributed throughout the legacy database application.

The process of extracting data and knowledge from a legacy application code logically precedes the process of understanding it. As discussed in the previous chapter, this collection or extraction process is non-trivial and may require multiple passes over source code files. Generally speaking, semantic information is present at more than one location in the code and if not carefully composed and collected much of the semantics may be lost. So a key task for the SEEK Semantic Analyzer (SA) is to recover these semantics and business rules that provide vital information about the system and allow mapping between the system and the domain model. The problem of extracting knowledge from application code is an important one. Major research efforts that attempt to answer this problem include program comprehension, control and data flow analysis algorithms, program slicing, cliché recognition and pattern matching. We summarize the state-of-the-art in the each of these areas below.

2.1 Program Comprehension

An important trend in knowledge discovery research is program analysis or program comprehension. Program comprehension typically involves reading documentation and scanning the source code to better understand program functionality and impact of proposed program modifications, leading to a close association with reverse engineering. The other objective of program comprehension is design recovery. Program comprehension takes advantage not only of source code but also other sources like inline comments in the code, mnemonic variable names, and domain knowledge. Implementation emphasis is more on the recovery of the design decisions and their
rationale. Since a firm’s way of doing business is expressed by its software systems, business process re-engineering and program comprehension are also closely linked.

Several major theoretical program comprehension models have been proposed in the literature. Among the more important ones are Shneiderman and Mayer’s (1979) model of program comprehension and Soloway and Ehrlich’s (1984) model. Shneiderman and Mayer view comprehension as a process of converting source code to an internal semantic form. The conversion can be achieved only with the help of the expert user or programmer’s semantic and syntactic knowledge. The first step requires the expert user to be able to intelligently guess the program’s purpose. In the next step, the model requires the programmer to then identify low-level structures such as familiar algorithms for sorting, searching and other groups of statements. Finally when a clear understanding of the program’s purpose is reached, it is represented in some syntax independent form.

Soloway and Ehrlich’s (1984) model on the other hand divides the knowledge base and the assimilation process differently. In Soloway and Ehrlich’s terminology, to understand a program is to recover the intention behind it. Goals denote intentions and plans denote techniques to realize these intentions. In other words, a plan is a set of rewrite rules that covert goals to sub goals and ultimately to program code. The knowledge base in this model includes programming language semantics, goal knowledge, and plan knowledge. Therefore at the very least the user should have a good understanding of the language in which the code was written, the user’s set of possible meanings for the computational goals, and an encoding of the solutions to problems the user has solved and understood before. Experimental studies proved that Soloway and
Ehrlich’s model can easily discover and express low-level concepts but can not accurately capture the high-level semantics of a program.

While both methods described above were theoretically strong, they suffer from similar drawbacks - both rely heavily on user or human input and both have a low degree of automation of the program comprehension process. The above disadvantages make it virtually unacceptable to design the SA on the basis of these models. Since our SA is designed to achieve total automation with minimal user input.

2.2 Lexical and Syntactic Analysis

Different methods have been proposed in the literature to automate the program comprehension process. They range from simple methods such as textual or lexical analysis to increasingly complex approaches that capture the control and data flow paths in a program.

Lexical analysis is defined as the process of decomposing a sequence of characters in the program’s source code file into its constituent lexical units. Once lexical analysis has been performed, various useful representations of the program are available. At the least, lexical analysis tells us the number of unique identifiers defined in the program. Halstead (1977) devised a metric to measure the difficulty in program comprehension based on the number of unique identifiers in a program.

The next logical step in automating program comprehension is syntactic analysis. Usually, the language properties are expressed formally as a context free grammar. The grammars themselves are described in a stylized notation called Backus Naur Form (Backus 1959) in which the program parts are defined by rules and in terms of their constituents. Once the grammar of a language is known, a parser can be easily constructed.
Traditionally the results of semantic analysis are represented in an *Abstract Syntax Tree* (AST). An AST is similar to a parsing diagram, which is used to show how a natural language sentence is decomposed into its constituents but without extraneous details such as punctuation. Therefore, an AST contains the details that relate to the program’s meaning. AST generation has many advantages, the most obvious being that it can be traversed using any standard tree traversal algorithm. It also forms the basis of several program comprehension techniques. Such techniques can be as simple as a high-level query expressed in terms of the node types in an AST. The tree traversal algorithm then interprets the query, traverses the tree until it arrives at the appropriate node, and delivers the requested information. More complicated approaches to program comprehension include control flow and data flow analysis.

### 2.3 Control Flow Analysis

Once the AST of a program has been constructed, it is possible to perform Control Flow Analysis (Hecht 1977) (CFA) on it. There are two major types of the CFA – *Interprocedural* and *Intraprocedural* analysis. Interprocedural analysis determines the calling relationship among program units while intraprocedural analysis determines the order in which statements are executed within these program units. Together they construct a *Control Flow Graph* (CFG).

Interprocedural analysis first identifies *basic blocks* in the program. A basic block is a collection of statements such that control can only flow in at the top and leave at the bottom either using a conditional or unconditional branch. These basic blocks are then represented as nodes in the CFG. Forward or backward arcs that represent a branch or a loop respectively indicate the flow of control. The CFG need not be constructed separately. It can be directly constructed on the AST by traversing the tree once to
determine the basic blocks. These blocks can then be connected using control flow arcs that represent a conditional or unconditional branch.

Intraprocedural analysis is the process of determining which routines invoke which others. This information is usually maintained in a call graph with each routine connected with downward arcs to all the sub-routine it calls. In the absence of procedure parameters and pointers, the call graph can also be maintained directly using the AST. However, when analyzing programs written in high-level languages like C, C++, Java etc., procedure parameters, pointers, and polymorphism may prevent us from knowing which routine or method was being invoked until run-time. A conservative solution proposed by (Larsen and Harrold 1996) connects such call nodes to all possible routines that may be invoked, making the analysis unnecessarily exhaustive. In SEEK, we are interested in both interprocedural and intraprocedural analysis but need to be able to perform control flow analysis, even when dynamic binding occurs.

2.4 Data Flow Analysis

In our SEEK SA, it is important to able to retrieve and understand the definition and usage of a variable. A variable is customarily defined when it appears on the left hand side of an assignment statement. The use of a variable, however, is indicated when the variable’s value is referenced by another statement, for example, when it appears as a function parameter or as an operand in an arithmetic expression. Data Flow Analysis (Hecht 1977) (DFA) is concerned with tracing a variable’s use from its point of definition. Like CFA, DFA also annotates the AST with arcs that connect the node where the variable is defined to nodes where the variable is used. While interprocedural analysis is straightforward, intraprocedural analysis may pose several problems, for example, when a procedure is called with a pointer argument, which in turn is passed on to another
procedure with a different name or alias. The SEEK SA has to be able to trace such procedure calls with aliases; hence DFA in its present form will not completely solve the problem at hand, namely, extraction of semantic knowledge from application code.

2.5 Program Dependence Graphs

A Program Dependence Graph (Horwitz and Reps 1992) (PDG) is a DAG whose vertices are assignment statements or predicates of an if-then-else or while constructs. Different edges represent control and data flow dependencies. Control flow edges are labeled true or false depending on whether they enter a then block or an else block of the code. In other words, a PDG is a CFG and DFG integrated in one graph which has several advantages including a more structural approach to program comprehension.

The SEEK SA’s primary objective is to be able to extract semantic knowledge from source code. This goal of extracting meaning for some interesting program variables is different from the goal of program comprehension techniques using PDG’s. Therefore, the construction of a PDG that represents even the minute details for the entire source code file may very well turn out to be wasteful exercise. It is important to investigate techniques that attempt to reduce the size of the source code under consideration by retaining only those statements that have the variable of interest in them. Using these techniques to reduce the size of the source code under consideration might be a necessary first step before generating the PDG.

2.6 Program Slicing

Slicing was introduced by Weiser (1981) and has served an important basis for various program comprehension techniques. Weiser (1981) defines the “slice of a program for a particular variable at a particular line in the source code as that part of the code that is
responsible for giving a value to the variable at that point in the code”. The idea behind slicing is to retrieve the code segment that has a direct impact on the concerned variables and nothing else. Starting at a given point in the program, program slicing automatically retrieves all relevant code statements containing control and/or data flow dependencies.

Figure 2-1 shows the various steps that have to be performed before the program slicing can proceed as outlined by Cimitile et al. (1995). The source code is sent as an input to a lexical analyzer and parser, which generate the AST. The control and data flow analyzers annotate the AST with the control flow and data dependency arcs. The program slicer requires three inputs:

- **slicing criteria**
- **direction of slicing**
- annotated abstract syntax tree

which contains the control and data flow dependencies on it. Traditionally, the slicing criteria (Huang et al. 1996) of a program \( P \) comprises a pair \( <i, V> \) where \( i \) is a program statement in \( P \), and \( V \) is a set of variables referred to in statement \( i \). The other input to the program slicer is the direction of slicing, which could be either forwards or backwards.

*Forward slicing* examines all statements between statement \( i \) and the end of the program. *Backward slicing* examines all statements before statement \( i \) until the first statement in the program.

Although slicing seems to a suitable solution with respect to SEEK SA’s objectives, it often produces slices that are nearly as large as the source code itself. This is especially true for programs that serve as application code for legacy systems where every variable in the code might be a potential slicing variable. Large slices translate to poor extraction of enterprise knowledge, specifically business rules. Huang et al. (1996) describe an
interesting approach to solving the problem of \textit{Business Rule Extraction} (BRE) from legacy code.

![Program Slicer Diagram](image)

\textbf{Figure 2-1 Program slicer driven by input criteria}

\textbf{2.7 Business Rule Extraction}

Legacy software systems typically contain business logic that has been encoded in the software for over many years. Business rules are also subject to change as markets and technology changes. When an update occurs in the company’s business model, the corresponding sections of the code must be changed in order to update the business rule(s). In the course of time and with increasing updates, software programmers tend to focus on updating the code and not the documentation. Therefore, the situation where the up-to-date business logic is available in the code and through no other source, including the programmer’s documentation of the code, may very well arise. BRE therefore is an important problem and is a focus of this research. The requirements of any BRE engine include faithful representation in its current and most up to date form of the business
rules as in the legacy software, and the ability to represent the extracted business rules in a language independent, easily communicable, and domain-specific form with all program variables replaced by their appropriate semantic meaning.

Huang et al. (1996) define a business rule as a function, constraint or transformation rule of an application’s inputs to outputs. Formally, a business rule $R$ can be expressed as a program segment $F$ that transforms a set of input variables $I$ to a set of output variable $O$. Mathematically, this can be represented as $O = F(I)$. The first step in BRE is the identification of important variables in the code that belong to set $O$ or $I$. Huang et al. (1996) propose a heuristic for identifying these variables. The authors claim only the overall system input and output variables could be members of these two sets. These variables are called the domain variables, which in turn are the slicing variables. The direction of slicing is decided based on the following heuristic: If the slicing variable appears in an output (input) statement the direction of slicing is fixed as backwards (forwards) as it is likely that the business rules of interest will be at some point above in the code.

Huang et al.’s (1996) approach successfully extracts business rules from the code, but presents the business rules in language dependant code to the end user. Sometimes, the business rules extracted may involve specific and intricate features of the language that might not easily understood by a managerial level employee. The SEEK SA on the other hand not only aims at extracting all the business rules from the source code but also representing the enterprise knowledge extracted in a language independent, and easily exchangeable format.
Sneed and Erdos (1996) adopt an entirely different approach for BRE. They argue that business rules are encoded in the form of assignments, results, arguments, and conditions as:

\[
\text{<result> ← assignment (arguments)} \quad \text{IF (conditions)}
\]

Their BRE algorithm works as follows: first, the assignment statements are captured along with their location. Next, the conditions that trigger the assignments are captured by representing the decision logic in the code in a tree structure. Therefore the Sneed and Erdos approach reduces the source code to a partial program that only contains statements that affect the values of variables on the left hand side of assignment statements. The algorithm leaves many questions unanswered and makes costly assumptions, including the supposition that the expert user knows which variables are interesting, or that all variables in the code have meaningful names. Additionally, the analyst must have some idea of critical business data. The biggest problem of the above described method is that it does not provide any mechanism to actually accomplish the reduction of code. Clearly this places the above assumptions in conflict with the goals of SEEK SA.

2.8 Cliché Recognition

Cliché recognition is an extension of static program analysis. It involves searching the program text for common programming patterns or idioms. An example of a cliché is a pattern describing loops that perform linear search. Several research tools provide cliché libraries (Willis 1994), which are automatically searched for in source code. Cliché recognition promises to be a powerful tool due to the abstraction power it provides. However, it remains a challenging research problem to solve, as there are many ways to
program even simple patterns such as a loop performing a linear search. Moreover, the linear search could be on any data structure of any type (e.g., on arrays of type \texttt{int}, or a linked list, etc.). Cliché recognition does not have the power to parameterize the data structure being searched or the type of value being searched for.

### 2.9 Pattern Matching

Pattern matching identifies interesting patterns code patterns and their dependencies. For example, conditional control structures such as \texttt{if..then..else} or \texttt{case} statements may encode business rules, whereas type declarations and class/structure definitions can provide information about the names, data types and structure of concepts as represented in the source code. Paul and Prakash (1994) have implemented a pattern matcher by transforming source code and templates constructed from pre-selected patterns into AST’s. Paul and Prakash’s (1994) approach has several advantages. Most important among them are the fact that patterns can be encoded in an extended version of the underlying language and the pattern matching process is syntax directed rather than character based. Unlike cliché recognition, the pattern matching approach proposed herein does not suffer from the drawback of not being able to parameterize the data structure and data types involved. Paul and Prakash (1994) propose a scheme of using wild cards in pattern templates to solve this problem. When coupled with program slicing and program dependency graphs, pattern matching promises to be a valuable tool for extracting semantic information.

The remainder of this thesis describes the SEEK SA architecture and provides a stepwise description of the semantic analysis algorithm used to extract application-specific semantics and business rules from legacy source code.
CHAPTER 3
SEMANTIC ANALYSIS ALGORITHM

A conceptual overview of the SEEK knowledge extraction architecture, which represents the build time component, is shown in Figure 3-1. SEEK applies Data Reverse Engineering (DRE) and Schema Matching (SM) processes to legacy databases, in order to produce a source wrapper for a legacy source. This source wrapper will be used by another component (not shown in Figure 3-1) for communication and exchange of information with the legacy source (run-time). It is assumed that the legacy source uses a database management system for storing and managing its enterprise data or knowledge.

![Figure 3-1. Conceptual build-time architecture of SEEK’s knowledge extraction algorithm](image)

First, SEEK generates a detailed description of the legacy source, including entities, relationships, application-specific meanings of the entities and relationships, business
rules, data formatting and reporting constraints, etc. We collectively refer to this information as *enterprise knowledge*. The extracted enterprise knowledge forms a knowledgebase that serves as input for the subsequent steps outlined below. In order to extract this enterprise knowledge, the DRE module shown on the left of Figure 3-1 connects to the underlying DBMS to extract schema information (most data sources support at least some form of Call-Level Interface such as JDBC). The schema information from the database is semantically enhanced using clues extracted by the semantic analyzer from available application code, business reports, and, in the future, perhaps other electronically available information that may encode business data such as e-mail correspondence, corporate memos, etc. It has been our experience (through discussions with representatives from the construction and manufacturing domains) that such application code exists and can be made available electronically.

Second, the semantically enhanced legacy source schema must be mapped into the domain model (DM) used by the application(s) that want(s) to access the legacy source. This is done using a schema matching process that produces the mapping rules between the legacy source schema and the application domain model. In addition to the domain model, the schema matching module also needs access to the domain ontology (DO) describing the model.

Finally, the extracted legacy schema and the mapping rules provide the input to the wrapper generator (not shown), which produces the source wrapper.

The three preceding steps can be formalized as follows. At a high level, let a legacy source $L$ be denoted by the tuple $L = (DB_L, S_L, D_L, Q_L)$, where $DB_L$ denotes the legacy database, $S_L$ denotes its schema, $D_L$ the data and $Q_L$ a set of queries that can be answered
by $DB_L$. Note, the legacy database need not be a relational database, but can include text, flat file databases, and hierarchically formatted information. $S_L$ is expressed by the data model $DM_L$.

We also define an application via the tuple $A = (S_A, Q_A, D_A)$, where $S_A$ denotes the schema used by the application and $Q_A$ denotes a collection of queries written against that schema. The symbol $D_A$ denotes data that is expressed in the context of the application. We assume that the application schema is described by a domain model and its corresponding ontology (as shown in Figure 3-1). For simplicity, we further assume that the application query format is specific to a given application domain but invariant across legacy sources for that domain. Let a "legacy source wrapper" $W$ be comprised of a query transformation

$$f_W^Q : Q_A \mapsto Q_L$$  \hspace{1cm} (3-1)

and a data transformation

$$f_W^D : D_L \mapsto D_A,$$  \hspace{1cm} (3-2)

where the $Q$’s and $D$’s are constrained by the corresponding schemas.

The SEEK knowledge extraction process shown in Figure 3-1 can now be stated as follows. Given $S_A$ and $Q_A$ for an application wishing to access legacy database $DB_L$, let schema $S_L$ be unknown. Assuming that we have access to the legacy database $DB_L$ as well as to application code $C_L$ accessing $DB_L$, we first infer $S_L$ by analyzing $DB_L$ and $C_L$, then use $S_L$ to infer a set of mapping rules $M$ between $S_L$ and $S_A$, which are used by a wrapper generator $WGen$ to produce $(f_W^Q, f_W^D)$. In short:

$$DRE: (DB_L, C_L, ) \mapsto S_L$$  \hspace{1cm} (3-4)

$$SM : (S_L, S_A) \mapsto M$$  \hspace{1cm} (3-5)
**WGen**: \((Q_A, M) \mapsto (f^Q_W, f^D_W)\)

Thus, the DRE algorithm (Equation 3-4) is comprised of schema extraction (SE) and semantic analysis (SA). This thesis will concentrate on the semantic analysis process by analyzing application code \(C_L\), thereby providing vital clues for inferring \(S_L\). The implementation and experimental evaluation of the DRE algorithm have been carried out are described in (Hammer et al. 2002b) and hence will not be dealt with in detail in this thesis.

The following section focuses on the semantic analyzer algorithm. It first provides the reader with the intuition behind the design of the semantic analyzer and then proceeds to outlines the SA algorithm.

### 3.1 Algorithm Design

The objective of the application code analysis is threefold:

- Augment entities extracted with domain semantics.
- Extract queries that help validate the existence of relationships among entities.
- Identify business rules and constraints not explicitly stored in the database, but which may be important to the wrapper generator or application program accessing legacy source \(L\).

Our approach to code analysis is based on code mining, as well as a combination of program slicing (Weiser 1981) and pattern matching (Paul and Prakash 1994). However our fundamental goal is broader than that described in the literature by Huang et al. (1996). Not only do we want to extract business rules and constraints, we also want to discover application-specific meanings of the underlying entities and attributes in the legacy database. Hence the heuristics used by our algorithms are different from the heuristics proposed by Huang et al. (1996) and are tailored to SEEK’s objectives. The following section lists the heuristics that form the basis of the SA algorithm.
3.1.1 Heuristics Used

The semantic analysis algorithm is based on several observations based on the general nature of legacy application code. Whether the application code is written for a client side application like an online ordering system or for resource management by an enterprise (e.g., a product re-order system manipulated by the employees), database application code always has queries embedded. The data retrieved or manipulated by queries is displayed to the end user (client or enterprise employee) in a pre-defined format. Both the queries and the output statements contain rich semantic information.

Heuristic 1. Application code typically has report generation modules or statements that display the results of queries executed on the underlying database.

Typically, output statements display one or more variables and/or contain one or more format strings. A format string is defined as a sequence of alphanumerical characters and escape sequences within quotes. An escape sequence is a backslash character and followed by a sequence of alphanumerical characters (e.g., \n, \t etc), which in combination indicate how to align and format the output. For example, in the statement

```java
System.out.println(“\n Task cost:” + v);
```

the substring “\n Task cost: ” represents the format string. The escape sequence “\n” specifies that the output should begin on a new line.

Heuristic 2. The format string in an input/output statement, if present, describes the displayed variable.

In other words, to discover the semantic meaning of a variable \( v \) in the source code, we have to look for an output (input) statements in which the variable \( v \) is displayed (accepted). Sometimes the format string that contains semantic information about the
display variable \( v \) and the output statement that actually displays the variable \( v \) may be split among two or more statements. Consider following statements:

```java
System.out.println("\n Task cost:");
System.out.println("\t" + v);
```

Let us call the first output statement with the format string as \( s1 \) and the second output statement that actually prints the value of the variable \( s2 \). Notice that, \( s1 \) and \( s2 \) can be separated by an arbitrary number of statements. In such a case, we would have to look backwards in the code from statement \( s2 \) for an output statement that prints no variables but a text string only. The text string contains the context meaning or clues about the application specific meaning of the variable. A classic example of this situation in database application code is the set of statements that display the results of a SELECT query in a matrix or tabular format. The matrix title and the column headers contain important clues about the application specific meanings of variables displayed in the individual columns of the matrix.

**Heuristic 3.** If an output statement \( s1 \) displaying variable \( v \) has no format string (and therefore no semantics for variable \( v \) that can be extracted from \( s \)), then the semantic meaning or context meaning of \( v \) may be the format string of another output statement \( s2 \) that only has a format string and displays no variables. Examining statements in the code backwards from \( s1 \) can lead to output statement \( s2 \) that contains the context meaning of \( v \).

It is logical to assume that variable \( v \) should have been declared and defined at some point in the code before it is used in an output statement. Therefore if a statement \( s \) assigns a value to \( v \) and \( s \) is a statement that retrieves a particular column value from the result set of a query \( q \), then \( v \)'s semantics can be associated to a particular column in \( q \) in the database.
Heuristic 4. If a statement \( s \) assigns a value to variable \( v \), and \( s \) retrieves a value of a column \( c \) of table \( t \) from the result set of a query \( q \), we can associate \( v \)'s semantics with column \( c \) of table \( t \).

As Erdos and Sneed (1996) observed, business logic is encoded either as assignment statements or conditional statements like \( \text{if..then..else, switch..case} \), etc. or a combination of them. Mathematical formulae translate into assignment statements while decision logic translates into conditional statements.

Heuristic 5a. If variable \( v \) is part of an assignment statement \( s \) (i.e. appears either on the left hand side or is used in the right hand side of the assignment statement), then statement \( s \) represents a mathematical formula involving variable \( v \).

Heuristic 5b. If variable \( v \) appears in the condition expression of an \( \text{if..then..else} \) or \( \text{switch..case} \) or any other conditional statement \( s \), then \( s \) represents a business rule involving variable \( v \).

Typically, in legacy application code the statements that are of interest to us are distributed throughout the application code. Hence, extracting semantic information for a variable \( v \) may amount to making one full pass over the legacy application code. Additionally, a fairly large subset of variables declared in the source code appear either in input, or output, or in database statements. Let us denote this subset of variables using the set \( V \). We refer to the statements that extract individual column values from the result set of a query and those statements that execute the queries on the database as database statements.

If we attempt to mine semantic information for all the variables in set \( V \) in parallel, and in one single pass over the code, we face the risk of extracting either incomplete or
potentially incorrect information due to the complexity of the process of extracting
semantic knowledge. Hence, by limiting the number of passes of the source code to one,
although the run-time complexity of the algorithm decreases, the correctness of the result
may be jeopardized, which is not desirable.

Since the emphasis in SEEK is not so much on run-time efficiency, but rather on
completeness and correctness, we adapt Weiser’s (1981) program slicing approach to
mine semantic information from application code. The SEEK SA aims at augmenting
entities and attributes in the database schema with their application-specific meanings. As
already discussed, output (input) statements provide us with the semantic meaning of the
displayed variable. Variables that appear on the left hand side of database statements can
be mapped to a particular column and table accessed in the query. Hence, it is reasonable
to state that the variables that appear in input/output or on database statements should be
traced throughout the application code. We will call these variables *slicing variables*.

As we described in Section 2, program slicing generates a reduced source code that
only contains statements that use or modify the slicing variable. Slicing is performed by
making a single pass over the source code and examining every statement in the code.
Only those statements that contain the slicing variables are retained in the reduced source
code.

**Heuristic 6.** The set of slicing variables includes variables that appear in input, output
or database statements. This is the set of variables that will provide the maximum
semantic knowledge about the underlying legacy database.

**Heuristic 7.** Slicing is performed once for each slicing variable to generate a reduced
source code that only contains statements that modify or use the slicing variable.
The program slicing routine takes three inputs in addition to the source code itself:

- slicing variable
- direction of slicing
- constraint of termination condition for slicing.

So far, we have discussed how to compose the set of slicing variables. The direction of slicing for a given slicing variable can be decided based on whether the slicing variable appears in an input, output or database statement. If the slicing variable appears in an input statement, it is logical to surmise that the value of the variable being accepted from the user will be used in statements below the current input statement. Hence, the statements of interest are below the input statement and the direction of slicing can be fixed as forward. On the other hand, if the slicing variable appears in an output statement, then the statements that define and assign values to that variable will appear above the current output statement in the code. Hence the direction of slicing is fixed as backward.

The third kind of slicing variables are those that appear in database statements. Since these statements assign a value to the slicing variable, it is reasonable to assume that all statements that modify or manipulate this slicing variable or related statement will be below the current database statement in the code, with the exception of the SQL query itself. In this case, neither forward nor backward slicing will suffice. Therefore, we adopt a combination of forward and backward slicing techniques, which we call recursive slicing to generate the reduced code. Recursive slicing is a three step process that proceeds as follows:

1. Perform backward slicing from the current database statement retaining all statements that use or modify the slicing variable, stopping only when an SQL SELECT query has been encountered in the code.
2. Append all statements below the current database statement in the code, to the program slice generated in step 1.
3. Finally, perform forward slicing from current database statement retaining only those statements that alter or use the slicing variable. This generates the final program slice. The default termination condition for slicing whether forward, backward or recursive is the function or class scope. In other words, slicing is automatically terminated at the point when the slicing variable goes out of scope. We summarize these insights in the final four heuristics.

**Heuristic 8a.** The direction of slicing is fixed as forward is if the slicing variable appears in an input statement and therefore only statements below this input statement in the source code, which contain the slicing variable, will be part of the program slice generated.

**Heuristic 8b.** The direction of slicing is fixed as backward if the slicing variable appears in an output statement and therefore only statements above this output statement in the source code, which contain the slicing variable, will be part of the program slice generated.

**Heuristic 8c.** If the slicing variable appears in a database related statement slicing must be performed recursively. The search for statements in the forward direction, that are part of the program slice, is bounded by the occurrence of an SQL SELECT query.

**Heuristic 9.** The termination criterion for slicing is determined by the scope of a given variable variable. In other words slicing terminated at the point where the slicing variable goes out of scope.

The following section describes the steps of the semantic analyzer algorithm in detail.

### 3.1.2 Semantic Analysis Algorithm Steps

Application code for legacy database systems is typically written in high-level languages like C, C++, Java etc. In this thesis, we discuss the implementation of C and
Java semantic analyzers. Not only does the C semantic analyzer serve as a good example of how to implement an SA for a procedural language such as C, it also serves as a learning experience before proceeding to design and implement a semantic analyzer for object oriented languages like Java. The lessons learned from implementing the C semantic analyzer are useful in building the Java semantic analyzer for the following reasons:

• The language grammar for statements like the `if..then..else`, `switch..case`, and assignment statements are similar in C and Java. Thus the business rule extraction strategy used in the C semantic analyzer can be reused in the Java semantic analyzer.

• Queries that are embedded in legacy application code are written in SQL both in C and Java. Hence the module that analyzes queries need not be re-designed for the Java SA.

We now describe the six-step semantic analysis algorithm pictured in Figure 3-2.

Semantic analysis begins by invoking the AST generator that uses the source code as input and generates an AST as output. Next, the pre-slicer module identifies the slicing variables by traversing the AST. Since the identification of the slicing variables logically precedes the actual program-slicing step, we call this module the ‘pre-slicer’. The code-slicer module, as the name suggests, generates the program slice corresponding to that slicing variable by retaining only those statements that contain the slicing variable. The primary objective of the analyzer module is to extract all the semantic information including data type, column and variable name, business rules, etc. corresponding to the slicing variable from the reduced AST. The analyzer module stores the semantic information extracted into appropriate data structures used to generate semantic analysis (result) reports. Once semantic analysis has been performed on all slicing variables, the semantic analysis results data structure is examined to see if there is any slicing variable
for which the analyzer was not able to clearly ascertain the semantic meaning of the slicing variable. Therefore, if an ambiguity in the meaning of a slicing variable is detected, the ambiguity resolver module is invoked. The ambiguity resolver presents all the semantic information extracted for the slicing variable to the user and accepts the semantic meaning of the slicing variable from the expert user. Finally the result generator module compiles the semantic analysis results, generates a report that serves as an input to the knowledge encoder. We describe each of these six steps in detail, as follows:

Step 1: AST generation for the application code. The SA process begins with the generation of an abstract syntax tree (AST) for the legacy application code. The following discussion references Figure 3-3, which is an expansion of the AST Generator representation shown in Figure 3-2. In Figure 3-3, the process flow on the left side is specific to building ASTs for C code, and the flow on the right side is for developing ASTs for Java code.

The AST generator for C code consists of two major components: the lexical analyzer and the parser. The lexical analyzer for application code written in C reads the source code line-by-line and breaks it up into tokens. The C parser reads in these tokens and builds an AST for the source code in accordance with language grammar (see Appendix A for a listing of the grammar for the C code that is accepted by the semantic analyzer). The above approach works well for procedural languages such as C. However, when applied directly to object oriented languages (e.g., Java), it greatly increases the complexity of the problem due to issues such as ambiguity induced by multiple inheritance, diversity resulting from specialization of classes and objects, etc.
Figure 3-2. Semantic analysis implementation steps

As more application code is written in Java, it becomes necessary to develop an algorithm to infer semantic information from Java code. As previously implied, the grammar of an object-oriented language is complex when compared with procedural languages like the C language. Building a Java lexical analyzer and parser would require the parser to look ahead multiple tokens before applying the appropriate production rule. Thus, building a Java parser from scratch does not seem like a feasible solution. Instead, tools like *lex* or *yacc* can be employed to do the parsing. These tools generate N-ary AST’s. N-ary trees, unlike binary trees, are difficult to navigate using standard tree traversal algorithms. Our objective in the AST generation is to be able to extract and associate the meaning of selected partitions of application code with program variables.
For example, format strings in input/output statements contain semantic information that can be associated with the variables in the input/output statement. This program variable in turn may be associated with a column of a table in the underlying legacy database. Standard Java language grammar does not put the format string information on the AST, since that would defeat the purpose of generating AST’s for the application code.

The above reasons justify the need for an alternate approach for analyzing Java code. Our Java AST builder (depicted on the right-hand side of Figure 3-3) has four major components, the first of which is a code decomposer. In object oriented languages like Java its possible that more than one class has been defined in the same source code file. The semantic analysis algorithm, which is based on the heuristics described above, takes a source code file that has just one class or file scope. Therefore, the objective of the Java source code decomposer is to decompose the source code into as many files as there are classes defined in it. It splits the original source code into a number of files, one per class, and then passes these files one by one to the pattern matcher. The objective of the pattern matcher module is twofold. First, it reduces the size of the application code being analyzed. Second, while generating the reduced application code file, it performs selected text replacements that facilitate easier parsing of the reduced source code. The pattern matcher works as follows: It scans the source code line by line looking for patterns such as System.out.println that indicate output statements or ResultSet that indicate JDBC statements. Upon finding such a pattern, it replaces the pattern with an appropriate pre-designated string. After this text replacement has been performed, the statement is closer in syntax to that of a procedural language. The replacement string is
chosen based on the grammar of this Java like procedural language. For example, in the following line of code:

```java
System.out.println("Task Start Date" + aValue);
```

the pattern `System.out.println` is replaced with `printf`, and following line is generated in a reduced source code file:

```java
printf("Task Start Date" + aValue);
```

After one pass of the application code, the pattern matcher generates a reduced source code file that contains only JDBC and output statements, which more closely resemble a procedural language. Appendix B provides a listing of the grammar production rules for this C-like language. In writing a lexical analyzer and parser for this reduced source code, we can re-use most of our C lexical analyzer and parser. The lexical analyzer reads the reduced source code line by line and supplies tokens to the parser that builds an AST in accordance with the Java language grammar.

**Step 2: Pre-slicer.** The pre-slicer identifies the set of slicing variables i.e., the set of variables that appear in input, output and database statements as described in Heuristic 7. The pre-slicer performs a pre-order traversal of the AST and examines every node corresponding to an input, output and database statement, searching the subtree of these nodes and adding all the variables in the subtree to the set of slicing variables. The pre-slicer extracts the signature (name of function, return type, number of parameter, and data types of all the parameters) of all functions defined in the source code file. Steps 3 through 5 are performed for every variable in the set of slicing variables. After analysis has been performed on all the slicing variables, Step 6 is invoked.
Step 3: Code slicer. The code slicer traverses the AST in pre-order and retains only those nodes that contain the slicing variable in their sub-tree. Each time the code slicer encounters a statement node, it searches the subtree of the statement node for the occurrence of the slicing variable. If the slicing variable is present, the code-slicer pushes the statement node (and therefore its subtree) onto a stack. After traversing all the nodes in the AST, the code-slicer pops out the nodes in the stack two at a time, connects them using the left child-right sibling notation of N-ary trees, and pushes the resulting binary tree back on to the stack. Finally, the code slicer is left with just one binary tree in the stack that corresponds to the reduced AST or the program slice for the given slicing variable. The reduced AST is sent as an input to the Analyzer.
Step 4: Analyzer. Figure 3-4 shows a flowchart containing the sub-steps executed by the analyzer module. The analyzer traverses the reduced AST and extracts semantic knowledge for a given slicing variable. The data type extractor searches the reduced AST for a ‘dcln’ node to learn the data type of the slicing variable. The semantic meaning extractor searches the reduced AST for ‘print’ or ‘scanf’ nodes. These nodes contain the mapping information from the text string to the identifier. Thus, we can extract the contextual meaning of the identifier from the text string. The column and table name extractor searches the reduced AST for an ‘embSQL’ node to discover the mapping between the slicing variable and a corresponding column name and table name in the database. The business rules extractor scans the reduced AST looking for ‘if’, ‘switch’, ‘assign’ nodes that correspond to business rules involving the slicing variable.

Besides extracting the data type, meaning, business rules and database association of the slicing variable, the analyzer also checks to see if the slicing variable is passed to a function as a parameter. If so, then the analyzer invokes the function call tracer. The function call tracer executes the following three steps:

1. Records the name of function to which the variable is passed and the parameter position.
2. Sets a flag indicating that a merge of the semantic knowledge discovered for the formal and actual parameters would be required after semantic analysis has been performed on all slicing variables for this file.
3. Adds the formal parameter corresponding to this slicing variable to the set of slicing variables gathered by the pre-slicer for this file.

It is important to note that unless the formal and actual parameter results are merged, the knowledge discovered about a single semantic entity will exist in two separate semantic analysis records. The three steps executed by the function call tracer are necessary for the following reason: The formal parameter may not be in the set of slicing
variables identified by the pre-slicer. In that case, if the function call tracer did not add
the formal parameter to the set of slicing variables, the associated business rule(s) may
never be discovered. Therefore, the semantic information extracted for the actual
parameter may be incomplete or potentially incorrect. Situations where the business rules
are abstracted into individual functions are common both in procedural and object-
oriented languages.

Figure 3-4. Substeps executed inside the analyzer module

**Step 5: Ambiguity resolver.** The ambiguity resolver’s primary function is to check
the semantic information discovered for every slicing variable to see if there is any
ambiguity in the knowledge extracted. The ambiguity resolver detects an ambiguity if
the meaning of the slicing variable is unknown, but the analyzer has been able to extract a
possible or context meaning of the slicing variable as described in Heuristic 3. The ambiguity resolver displays all the semantic knowledge discovered for the slicing variable including the possible or contextual meaning in a user interface and asks the user to enter the meaning of the slicing variable given all this information. This is the only step in the entire semantic analysis algorithm that requires user input.

**Step 6: Result generator.** The result generator has the following dual functionality. First, it merges the semantic knowledge extracted for the formal and actual parameters in a function call. Second, it replaces the slicing variables in the business rules with their application-specific meanings, thereby converting the business rules extracted into a source code-independent format. The merge algorithm executed by the result generator has $O(N^2)$ complexity, since it that iterates through $N$ semantic analysis result records checking every record with the remaining $N-1$ records to see if they represent a pair of formal and actual parameter records that need to be merged. Finally, the result generator writes all the discovered semantic knowledge to a file.

At the end of this six-step semantic analysis algorithm, control is returned to the schema extractor in the DRE algorithm. In the next section, we describe the Java semantic analyzer and justify the need for a more elaborate analyzer and result generator.

### 3.2 Java Semantic Analyzer

Most application code for databases written today is written in Java, making it important to verify that the SA algorithm is able to mine semantic information from application code written both in procedural languages such as C and in object-oriented languages like Java. Java is an object-oriented language with powerful features like inheritance, operator overloading and polymorphism. This means that methods can be invoked on objects either defined in Java’s extensive Application Program Interface
(API) or on objects that may be user defined. Alternatively, the function call may be defined a base class higher than given object in the inheritance hierarchy. The semantic analysis algorithm presented in the previous section cannot handle such cases.

In order to take in account all the above-mentioned features of Java, we redesigned the analyzer and the result generator module of the semantic analyzer. Figure 3-5 depicts an enlarged view of the analyzer module and outlines the sub-steps executed inside the analyzer module.

The sequence of sub-steps executed inside the analyzer module remain unchanged in most cases. However, if the slicing variable is passed to a function as a parameter, then the steps executed in the Java SA result generator module are different. It becomes important to determine whether the method was invoked on an object or is simply a call to a function defined in the same file, or in the base class. If the method is invoked on an object, the definition of the method is not present in the source code file under analysis. In this case, the source code decomposer ensures that the input to the Java SA is a file that has only one class scope. If the method was not invoked on an object, one of three cases can occur:

1. The definition of the method is present in the same file; or
2. The definition of the method is present in the base class; or
3. It is a call to a method in the Java library.

We will now analyze each of the three cases above with respect to their implications on the semantic analysis algorithm.
Case one generates two possibilities. Initially, if the method invoked is defined in the same file, the same file function call tracer is invoked, which is identical to the function tracer in Step 5 of the semantic analysis algorithm described in the previous section. However, if the method is not invoked on an object and the method name is not present in the list of methods defined in this file, then we can determine if it is a call to a method defined in the base class as follows: we check to see if the class we are analyzing is derived from any other class. If the class is not derived from any class, we can conclusively state that the method being invoked is a call to method in the Java API. If the class is indeed derived from another class, the possibility of the method being defined in the base class exists. Hence, we invoke the different file function call tracer which executes the following three steps:
1. Records the name of function to which the variable is passed and the parameter position.
2. It sets a flag indicating that a merge of the semantic knowledge discovered for the formal and actual parameters would be required after semantic analysis has been performed on all source code files.
3. Finally, it adds the name of the function, and the parameter position of this slicing variable, and the name of the object on which this method is invoked (in this case the base class name) to the global set of slicing variables. The set of slicing variables for every source code file except the first one is the union of the set of slicing variables discovered by the pre-slicer for that individual file and the global set of slicing variables.

The case when a method is invoked on an object reduces to the case where the definition of the method is not present in the same file and can be handled in the exact same fashion by invoking the different file function call tracer.

The SA result generator has to be modified to support integration of semantic knowledge extracted in the analysis of multiple source code files. If for a particular slicing variable result record, the flag that indicates that merge is required across different semantic analysis result files has been means that additional semantic knowledge about the same physical entity is present in another results file that was generated by analyzing a different source code file. The class name tells us which result file to examine. The method name and the parameter position point to a particular result record in that file, whose results should be integrated with the current result record under consideration.

With the aforementioned changes and additions to the semantic analysis algorithm, the SA is able to extract semantic information from source code written in Java. In the following chapters we describe the implementation details of Java SA prototype and illustrate the major steps of semantic analysis using an example.
CHAPTER 4
IMPLEMENTATION OF THE JAVA SEMANTIC ANALYZER

In the previous chapter we presented the intuition behind the semantic analyzer design and described the steps of the algorithm. In this chapter, we describe the implementational details of the current Java SA prototype. The current version aims at extracting semantic information from application code and at tracing function calls with the same source code file. It also assumes that the file input has only program or class scope. The SA prototype is implemented using the Java SDK 1.3 from Sun Microsystems. The prototype was tested with application code written in Java. In this chapter we use italics to introduce new concepts and highlight slicing variable names. Nodes in the AST are represented by placing the node name in italics, within single quotes (e.g., ‘embSQL’). Class names, methods, data members of classes, and built-in data types are highlighted using italicized Courier font (e.g., SAResults). Code statements and fragments are represented using the Courier font.

4.1 Implementation Details

Figure 4-1 shows the code block diagram of the SA prototype. The driver method for the semantic analyzer is the main method of the class javalexicalAnalyzer. The main method accepts the name of the source code file to be analyzed as a command line argument, then invokes the Java Pattern Matcher and passes the name of the source code file to it as a parameter. The Pattern Matcher module generates a new reduced source code file by replacing pre-defined patterns with suitable text, and then returns control to the main method of the class javalexicalAnalyzer that invokes the lexical
analyzer and parser on the reduced code file. The parser generates an AST, which is an object of type `LinkedBinaryTree`, for the reduced code file. The driver program next invokes a series of methods defined in the `LinkedBinaryTree` class, which represent the major steps in the semantic analysis algorithm. The pre-slicer method returns a set of slicing variables. The code slicer and analyzer methods are invoked on the AST for each slicing variable which is passed as a parameter to both methods. Finally, the result generator method saves the extracted semantic knowledge to the `SAResults` data structure.

![Figure 4-1. Semantic Analyzer code block diagram](image)

We next outline the implementation details of each module in our Java SA prototype as described in Figure 3-2.

**SA-1: AST generator.** The main method of the class `javalexicalAnalyzer` invokes the `generateReducedCode` method in the class `javaPatternMatcher` as shown in Figure 4-1. The Java Pattern Matcher scans the source code file looking for
pre-defined patterns or pre-specified pattern generators. Pre-defined patterns include output, declaration, and JDBC patterns. For example, the text string

`System.out.println` is a pre-defined output pattern. JDBC patterns include database connectivity statements and query execution statements, methods and objects. They are stored in the class `JDBCPatterns`. Similarly the output statement patterns are stored in the `outputPatterns` data structure, as shown in Figure 4-2.

If the Pattern Matcher encounters a pre-defined pattern, it performs appropriate text substitutions and stores the modified source code file. In object-oriented languages like Java, objects can be instantiated and methods invoked on these objects. A method invocation on an object may have the same functionality as one of the pre-defined patterns. Hence it is important to be able to trace such method invocations on objects and replace them with appropriate text. The object, on which the method is invoked, is referred to as a pre-defined pattern generator. The Pattern Matcher adds the object instance and method combination to the list of pre-defined patterns. For example, consider that the following statements:

```java
PrintWriter p = new PrintWriter(System.out);
p.println("Task End Date");
```

are functionally equivalent to the statement:

```java
System.out.println("Task End Date");
```

Here, `p` is an instance of the object of type `PrintWriter`. The object `PrintWriter` is the pattern generator and `p.println` is an output pattern we henceforth search for in the sourcecode. We append `p.println` to the `outputPatternStrings` array in the class `outputPatterns`. Therefore, when the Java Pattern Matcher reads the line:
p.println("Task End Date");

it recognizes that p.println is a pre-defined output pattern and re-writes the line to
the modified file as:

printf("Task End Date");

![Java Pattern Matcher code block diagram](image)

Figure 4-2. Java Pattern Matcher code block diagram

The goal of the Pattern Matcher is to generate a reduced source code file that is closer
to a procedural language such as C. Hence all declaration statements involving the `new`
operator have to be re-written in a C-like declaration statement without the `new` operator.
The Pattern Matcher uses the `dataTypesSupported` class to identify lines in the
source code that declare objects of pre-defined or built-in data types. The `stringValues` data structure maintains the value of the string variables at every point
in the code. The Pattern Matcher uses this data structure to regenerate queries that have
been composed in several stages as a combination of string variables and text strings using the overloaded addition operator (+) for strings.

Figure 4-3 lists the data members and the methods defined for each of the four data structures used by the Pattern Matcher. The DataTypesSupported class uses the method addDefaultDataTypes to add built-in data types like int, float, boolean etc. to the DataTypes array. The JDBCPatternType array in the class JDBCPatterns class stores the type of the JDBC pattern used to distinguish query execution statement patterns and resultSet get methods. The rest of the data members and methods of the data structures in Figure 4-3 are self-explanatory.

The Java Lexical Analyzer reads the reduced source code file generated by the Java Pattern Matcher and tokenizes it. The tokens are sent to the Java parser that applies the
appropriate production rule from the language grammar and generates a sub-tree which corresponds to that statement. Therefore, the root node of the sub-tree corresponds to a statement in the code and has additional information including the actual starting and ending lines and column numbers of the source code statements. The parser pushes these sub-trees onto a stack as it generates them. After the Parser has parsed the last line in the reduced source code, it begins to construct the AST. The sub-trees are popped two at a time from the stack and connected using the left child-right sibling representation of a N-ary tree as a binary tree. The resulting binary tree is pushed back onto stack and this operation is repeated till there is only one tree left in the stack. This binary tree represents the AST of the modified source code.

**SA-2: Pre-slicer.** This step is defined as a method in the class `LinkedBinaryTree` as shown in Figure 4-1. The method performs a pre-order traversal of the AST, marking nodes it has visited while trying to identify a list of slicing variables. When it encounters a `printf`, `embSQL`, or `scanf` node that corresponds to an output, SQL or input statement respectively in the code, it performs a pre-order traversal of this statement node. If it finds an `identifier` node in the sub-tree which corresponds to the occurrence of a variable in that statement, it appends the `identifier` node’s left child, which has the actual variable name, to the list of slicing variables. The list of slicing variables is maintained as an array of `String` in memory. Lastly, the pre-slicer marks the identifier node as visited.

The other task that the pre-slicer accomplishes is to compose a list of methods defined in the source file. If the pre-slicer encounters a `function` node, it traverses the sub-tree of the `function` node and it appends the name of the method, number of parameters, return
type of the function, and parameter list to the *FunctionsDefined* data structure shown in Figure 4-4. The data members and methods of this class are self-explanatory.

<table>
<thead>
<tr>
<th>FunctionsDefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>NameOfFunction: String</td>
</tr>
<tr>
<td>NumberOfParameters: int</td>
</tr>
<tr>
<td>DataTypeOfParams: array String</td>
</tr>
<tr>
<td>NameOfParams: array String</td>
</tr>
</tbody>
</table>

FunctionsDefined()
setFunctionName(String s)
setDataType(String s)
setParamName(String s)

Figure 4-4. Methods and data members of *FunctionsDefined* class

**SA-3: Code slicer.** This step is implemented as a method in the class *LinkedBinaryTree* as shown in Figure 4-1. The method performs a pre-order traversal of the AST and examines every node in the tree that corresponds to a statement. If the slicing variable is one of the nodes in the sub-tree of the statement node, then the code slicer takes the statement node and disconnects it from its parent and sibling nodes in the tree and pushes it into a stack. At the end of the pre-order walk of the entire AST, the stack contains only those statements nodes that contain the slicing variable. A reduced AST is constructed using the same approach as the Java Parser in step 1 uses to construct the AST. This reduced AST is also an object of type *LinkedBinaryTree* and a reference to its root node is passed to the analyzer module.

**SA-4: Analyzer.** The analyzer module is also implemented as a method in the class *LinkedBinaryTree*. While traversing the reduced AST, if the analyzer encounters a ‘*dcln*’ node, which corresponds to a declaration of a variable in the source code, it
extracts the data type of the variable and saves it to the \textit{Datatype} data member of the \textit{SAResults} data structure. If the analyzer encounters either an ‘assign’, ‘if’, or ‘switch’ node on the reduced AST, which correspond to either a assignment statement, \texttt{if..then..else} statement, or a \texttt{switch} statement respectively, it executes the two steps described below to extract the corresponding business rule. First, using the line and column numbers stored in the statement node, it retrieves the statements corresponding to this node in the reduced AST from the source code file, and assigns it to the \textit{BusinessRules} data member of the \textit{SAResults} data structure. Second, every occurrence of the variable name in the business rule is replaced by its meaning. The step transforms the business rule extracted into a code-independent format. ‘\texttt{embSQL}’ nodes contain the mapping information from an identifier name to corresponding column and table name in the database.

The \textit{SAResults} data structure shown in Figure 4-5 stores semantic knowledge extracted for each slicing variable. The meaning and business rules are defined as an array of \texttt{String} as there may be more than one meaning or business rule that can be associated with a slicing variable. If the slicing variable is passed to a method as a parameter, then the name of the function and parameter position is respectively saved in the \textit{SAResults} data structure in \texttt{ToFuncName} and \texttt{ToFuncParamPosition} data members. A slicing variable may be passed as a parameter to more than one function. Hence both \texttt{ToFuncName} and \texttt{ToFuncParamPosition} are defined as arrays.

If the slicing variable itself is defined in the parameter list of a function definition, then the name of the function and parameter position are stored in \texttt{FuncName} and \texttt{FuncParamPosition} data members of the \textit{SAResults} data structure. \texttt{Alias} is an
array of *String*, used to store the formal parameter variable names corresponding to a variable. The rest of the members of the Semantic Analysis Results data structure are self-explanatory.

<table>
<thead>
<tr>
<th>SAResults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variablename: String</td>
</tr>
<tr>
<td>Alias: array String</td>
</tr>
<tr>
<td>AliasPos: int</td>
</tr>
<tr>
<td>Datatype: String</td>
</tr>
<tr>
<td>TableName: String</td>
</tr>
<tr>
<td>ColumnName: String</td>
</tr>
<tr>
<td>Meaning: array String</td>
</tr>
<tr>
<td>MeaningPos: int</td>
</tr>
<tr>
<td>PossibleMeaning: String</td>
</tr>
<tr>
<td>BusinessRules: array String</td>
</tr>
<tr>
<td>BusinessRulePos: int</td>
</tr>
<tr>
<td>IsVarParam: boolean</td>
</tr>
<tr>
<td>FuncName: array String</td>
</tr>
<tr>
<td>FuncCount: int</td>
</tr>
<tr>
<td>FuncParamPosition: array int</td>
</tr>
<tr>
<td>IsVarPassedParam: boolean</td>
</tr>
<tr>
<td>ToFuncName: array String</td>
</tr>
<tr>
<td>ToFuncCount: int</td>
</tr>
<tr>
<td>ToFuncParamPosition: array int</td>
</tr>
</tbody>
</table>

**Figure 4-5. Semantic analysis results data structure**

**SA-5: Ambiguity resolver.** If the meaning of a variable is not known at the end of Step 5, we present the information gathered about the slicing variable including the data type, column and table name in the data base, business rules, and the context or possible meaning of the variable in a Java swing interface. The user is prompted to enter the meaning of the variable given this information. The meaning entered by the user is saved to the `SAResults` data structure.
**SA-6: Result generator.** The primary objective of the result generator is to iterate through all the records of the *SAResults* data structure and merge the records corresponding to the formal and actual parameter. Two records $i$ and $j$ in the array of *SAResults* result records are merged only if the *ToFuncName* field of $i$ is identical to the *FuncName* field of $j$, the *ToFuncParamPosition* field of $i$ is identical to the *FuncParamPosition* of $j$, and both *isVaramParam* of $j$ and *isVarPassedParam* of $i$ are both true. This condition verifies that record $i$ corresponds to the actual parameter and record $j$ corresponds to the formal parameter.

The variable name corresponding to entry $j$ is saved as an alias of the variable corresponding to entry $i$. In the next section, we illustrate the SA process.

**4.2 Illustrative Example**

We herein employ the source code listed in Appendix C to simulate the Java SA prototype stepwise. The test code has been written in Java SDK version 1.3 from Sun Microsystems, and is specific to a manufacturing domain database that contains queries and business rules that would typically be embedded in application code written for manufacturing domain databases. The test code first establishes a JDBC connection to the underlying Oracle database. After the connection has been established, a query is executed on the underlying database to extract the project start date, project finish date and cost for a certain project with name ‘Avalon’. The code also contains a business rule which checks to see if the total project cost is over a certain threshold, and if so offers a 10% discount for such projects. The project cost in the underlying database is updated to reflect the discount. The task start date, finish date and unit cost for all the tasks of this project that have the name ‘Tiles’ are extracted. For each task, the task unit cost is raised
by 20% if the number of days between the start and end of the task is less than ten. Also
the code ensures that the start and end of the individual tasks are well within the project
start and end dates. We now simulate the various steps in semantic analysis for this given
test code.

**Step 1: AST generation.** The Java Pattern Matcher generates the reduced source code
as listed in Appendix D. The Java lexical analyzer and parser construct the AST for this
reduced source code file. The AST of the reduced source code is as listed in Appendix E.
Each line in the AST represents a node in the AST and the number of periods in the
beginning of each line of the AST, denotes the level of that node in the AST. The N-ary
tree corresponding to this AST can be visualized by taking a mirror image of the tree
printed in this format and inverting it.

**Step 2: Pre-slicer.** As described in the previous section, the pre-slicer’s task is two-
fold. First, it generates a list of slicing variables. Second, it maintains a list of all methods
defined in the source file and their signatures. Table 4-1 shows the information
maintained by the pre-slicer for slicing variables. Table 4-2 highlights the information
maintained by the pre-slicer for methods defined in the same source file.

Steps 3 through 6 are executed for each slicing variable. We will illustrate steps 3
through 6 for slicing variable \textit{tfinish}.

**Step 3: Code slicer.** The code slicer generates the reduced AST as shown in Figure 4-
6. The reduced AST is constructed by retaining only those statement nodes in the original
AST in which the slicing variable \textit{tfinish} occurs some where in the sub-tree of that
statement node.
Table 4-1. Information maintained by the pre-slicer for slicing variables

<table>
<thead>
<tr>
<th>Slicing Variable</th>
<th>Type of Statement</th>
<th>Direction of Slicing</th>
<th>Text String (only for print nodes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pfinish</td>
<td>Output</td>
<td>Backwards</td>
<td>&quot;Project Finish Date for Avalon &quot;</td>
</tr>
<tr>
<td>pcost</td>
<td>database</td>
<td>Recursive</td>
<td>-----</td>
</tr>
<tr>
<td>tfinish</td>
<td>output</td>
<td>Backwards</td>
<td>-----</td>
</tr>
</tbody>
</table>

Table 4-2. Signatures of methods defined in the source file maintained by the pre-slicer

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Return Type</th>
<th>Number of Parameters</th>
<th>Parameter List</th>
</tr>
</thead>
<tbody>
<tr>
<td>CheckDuration</td>
<td>float</td>
<td>3</td>
<td>date, date, float</td>
</tr>
<tr>
<td>checkifValidDate</td>
<td>void</td>
<td>1</td>
<td>date</td>
</tr>
</tbody>
</table>

**Step 4: Analyzer.** The analyzer traverses the reduced AST and extracts semantic information for the slicing variable *tfinish*. The information extracted by the analyzer is shown in Table 4-3. The analyzer stores the semantic knowledge extracted in the *SAResults* data structure.

**Step 5: Ambiguity resolver.** If the meaning of the slicing variable is not known at the end of Step 5, the ambiguity resolver is invoked. The ambiguity resolver presents the semantic information extracted for the slicing variable, along with any possible or context meaning to the expert user, and accepts the meaning of the slicing variable *tfinish* from the user. Figure 4-7 shows a screen snapshot of the ambiguity resolver user interface.

**Step 6: Result generator.** The result generator detects that a merge will be required to integrate the semantic knowledge discovered for the slicing variable *tfinish* as it has been passed to another method in the source code. The *SAResults* record corresponding to the formal parameter is found by searching for a *SAResults* record that has the same value in the fields corresponding to the function name and function parameter position as

---

1 The “Is variable passed as parameter” field is set to “yes.”
the slicing variable has in its ToFuncName and ToFuncParamPosition fields.

Table 4-4 shows the semantic information extracted for the formal parameter \( t \). Table 4-5 shows the semantic information for the variable \( t\text{finish} \) after the semantic knowledge specific to formal and actual parameters have been merged.

---REDUCED AST---

```
program
dcln(2)
  . <identifier>(1)
  . . Date(0)
  . = (2)
  . . <identifier>(1)
  . . . rhscall(2)
  . . . . <identifier>(1)
  . . . . . getDate(0)
  . . . . <string>(1)
  . . . . . "Task_Finish_Date"(0)
  assign(2)
  . . <identifier>(1)
  . . . tcost(0)
  . . . rhscall(4)
  . . . . <identifier>(1)
  . . . . . checkDuration(0)
  . . <identifier>(1)
  . . . tstart(0)
  . . <identifier>(1)
  . . . tfinish(0)
```

---REDUCED AST---

```
if(2)
  . or(2)
  . . <(2)
  . . . rhscall(1)
  . . . . <identifier>(1)
  . . . . . tstart.get(0)
  . . . . rhscall(1)
  . . . . . <identifier>(1)
  . . . . . . pstart.get(0)
  . . . . > (2)
  . . . rhscall(1)
  . . . . <identifier>(1)
  . . . . . <identifier>(1)
  . . . . . . tfinish.get(0)
  . . . . . rhscall(1)
  . . . . . . <identifier>(1)
  . . . . . . . pfinish.get(0)
  . . . . . . block(1)
  . . . . emptyprintf(1)
  . . . . . <string>(1)
  . . . . . . "The task start and finish dates have to be within the project start and finish dates"(0)
```

Figure 4-6. Reduced AST generated by the code slicer for slicing variable \( t\text{finish} \)
Table 4-3. Semantic knowledge extracted for slicing variable *tfinish*

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Tfinish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Date</td>
</tr>
<tr>
<td>Alias</td>
<td>----</td>
</tr>
<tr>
<td>Table Name</td>
<td>MSP_Tasks</td>
</tr>
<tr>
<td>Column Name</td>
<td>Task_Finish_Date</td>
</tr>
<tr>
<td>Meaning</td>
<td>----</td>
</tr>
<tr>
<td>Possible Meaning</td>
<td>Finish Date of Task Start Date of Task Unit Cost for Task</td>
</tr>
<tr>
<td>Is variable defined as a function parameter</td>
<td>No</td>
</tr>
<tr>
<td>Function Name</td>
<td>----</td>
</tr>
<tr>
<td>Function Parameter Position</td>
<td>----</td>
</tr>
<tr>
<td>Is variable passed as parameter</td>
<td>Yes</td>
</tr>
<tr>
<td>To Function Name</td>
<td>CheckDuration</td>
</tr>
<tr>
<td>To Function Parameter Position</td>
<td>2</td>
</tr>
</tbody>
</table>

**Business Rules**

```java
if ((tstart.getDate() < pstart.getDate()) || (tfinish.getDate() > pfinish.getDate()))
{
    System.out.println("The task start and finish dates have to be within the project start and finish dates");
}
```
Figure 4-7. Screen snapshot of the ambiguity resolver user interface
Table 4-4. Semantic information gathered slicing variable $t$

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Date</td>
</tr>
<tr>
<td>Alias</td>
<td>----</td>
</tr>
<tr>
<td>Table Name</td>
<td>----</td>
</tr>
<tr>
<td>Column Name</td>
<td>----</td>
</tr>
<tr>
<td>Meaning</td>
<td>----</td>
</tr>
<tr>
<td>Possible Meaning</td>
<td>----</td>
</tr>
<tr>
<td>Is variable defined as a function parameter</td>
<td>Yes</td>
</tr>
<tr>
<td>Function Name</td>
<td>CheckDuration</td>
</tr>
<tr>
<td>Function Parameter Position</td>
<td>2</td>
</tr>
<tr>
<td>Is variable passed as parameter</td>
<td>No</td>
</tr>
<tr>
<td>To Function Name</td>
<td>----</td>
</tr>
<tr>
<td>To Function Parameter</td>
<td>----</td>
</tr>
<tr>
<td>Position</td>
<td></td>
</tr>
<tr>
<td>Business Rules</td>
<td>if (s.getDate() - t.getDate() &lt; 10)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>revisedcost = f + f * 20/100;</td>
</tr>
<tr>
<td></td>
<td>System.out.println(&quot;Estimated New Task Unit Cost : &quot;, revisedcost);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>revisedcost = f;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Table 4-5. Semantic information for variable \textit{tfinish} after the merge operation

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>tfinish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>Date</td>
</tr>
<tr>
<td>Alias</td>
<td>t</td>
</tr>
<tr>
<td>Table Name</td>
<td>MSP_Tasks</td>
</tr>
<tr>
<td>Column Name</td>
<td>Task_Finish_Date</td>
</tr>
<tr>
<td>Meaning</td>
<td>Task End Date</td>
</tr>
</tbody>
</table>
| Business Rules  | if (((start.getDate() < pstart.getDate()) || (tfinish.getDate() > pfinish.getDate()))
            {
                System.out.println("The task start and finish dates have to be within the project start and finish dates");
            }
            if (s.getDate() - t.getDate() < 10)
            {
                revisedcost = f + f * 20/100;
                System.out.println("Estimated New Task Unit Cost : "+ revisedcost);
            }
            else
            {
                revisedcost = f;
            } |
CHAPTER 5
QUALITATIVE EVALUATION OF THE JAVA SEMANTIC ANALYZER PROTOTYPE

The previous chapter describes the implementation details of the Java SA prototype. In this chapter, we use code fragments from the source code listed in Appendix C to highlight and demonstrate important features of the Java programming language that the Java SA prototype can accurately capture.

In Java, the tuples that satisfy the selection criteria of an SQL SELECT query are returned in a resultSet object. The Java Database Connectivity (JDBC) Application Program Interface (API) [33] provides several get methods for resultSet objects to extract individual column values from a tuple in the resultSet. The parameter of a resultSet get method can either be a string or an integer. The string parameter has to a column name from the SELECT query column list while the integer parameter has to be an integer between zero and the number of columns in the SELECT query minus one. The two scenarios in Figure 5-1 highlight the types of parameters that can be passed to a resultSet get method.

**SA Feature 1.** The Java SA can accurately extract the table name and column name from a SQL SELECT query that corresponds to the slicing variable even if the column number (instead of the column name) was specified as the parameter in the resultSet get method.
### Scenario A:

```java
String query = "SELECT Task_Start_Date, Task_Finish_Date, Task_UnitCost FROM MSP_Tasks WHERE Task_Name = 'Tiles";
ResultSet rset = stmt.executeQuery(query);
Date tstart = rset.getDate("Task_Start_Date");
```

### Scenario B:

```java
String query = "SELECT Proj_Start_Date + 1, Project_Finish_Date -1, Project_Cost FROM MSP_Projects WHERE Proj_Name = 'Avalon"; 
ResultSet rset = stmt.executeQuery(query);
Date pstart = rset.getDate(0);
```

Figure 5-1. Code fragment depicting the types of parameters that can be passed to a `resultSet get` method

In Scenario A, in Figure 5-1, the Java SA extracts the column name that the slicing variable `tstart` corresponds to by extracting the string parameter sent to the `resultSet get` method. If the `resultSet get` method parameter is an integer, the Java SA extracts the corresponding column name by moving `n` levels down to the right in the subtree corresponding to the column list of the SQL SELECT query.

An SQL SELECT query’s column list is defined as list of comma separated mathematical expressions in the language grammar. Scenario B in Figure 5-1 is an example of a SELECT query where the column names are used in mathematical expressions instead of being specified directly.

**SA Feature 2.** The Java SA can map the slicing variable to the corresponding column name in the SQL SELECT query even if the column name is embedded in a complex mathematical expression.

The Java SA determines the column name corresponding to the variable `pstart` in two steps. First, it locates the first child of the SELECT query ‘`columnlist`’ node. This node represents the subtree corresponding to the mathematical expression.
Proj_Start_Date + 1. In the second step, the Java SA accurately identifies the column name by searching for a previous undeclared identifier in the sub-tree of the mathematical expression. This strategy ensures that the Java SA can always extract the column name without getting confused by the presence of other variables, integers and operands in the expression.

A powerful feature of object-oriented languages such as Java is operator overloading. A classic example of overloading in Java is the addition (+) operator for strings. In Java, queries are executed by passing an SQL query as a parameter of type of `String` to either the `execute` or `executeQuery` methods, which are defined for `Statement` and `PreparedStatement` objects. The query string itself can be composed in several stages using the string concatenation (+) operator as shown in the code fragment in Figure 5-2.

**SA Feature 3.** The Java SA can capture the semantics of the string concatenation (+) operator.

```java
stmt.executeUpdate("UPDATE MSP_Tasks SET Task_UnitCost = " + tcost + "
WHERE Task_Start_Date = " + tstart + " AND Task_Finish_Date = " + tfinish + "");
```

**Figure 5-2.** SQL query composed using the string concatenation operator (+)

The Java SA enables this feature by monitoring the value of string variables at every point in the code. Therefore, the Java SA regenerates an SQL query composed in stages using the string concatenation operator by simply substituting the string variable with its value at that point in the code.

In Java, output methods like `print` and `println` accept a string parameter and display the string content. This makes it possible to have a situation where an output
statement displays only string variables and no format or text strings in the same statement. The string variables in turn may have been assigned values in a series of one or more assignment statements prior to their use in the output statement. We define such output statements **indirect output statements**.

**SA Feature 4.** The Java SA can capture semantics hidden in indirect output statements.

Figure 5-3 depicts an example of an indirect output statement. The Java SA discovers the meaning of the variable `pstart`, which might not have been extracted if this feature was not built into the Java SA. Semantic information hidden behind indirect output statements are extracted by parsing the right hand side of all assignment statements whose left hand side is a string variable.

```java
String displayString;
displayString = "Project Start Date " + pstart;
System.out.println(displayString);
```

Figure 5-3. Code fragment demonstrating indirect output statements

The format string in a Java output statement in Java is a combination of text that contains the semantic meaning of the output variable and escape sequences used to position or align the output. In some situations however, it is necessary to split the format string between two or more output statements. One output statement has the semantic meaning of the output variable and the other has the escape sequences for alignment of the output variables on the standard output. A common example of such a situation in code occurs when displaying data stored in an object or array in a tabular format. Rich semantic clues are embedded in the output statements that display the title or heading of each column or the table itself. These format strings of such output statements contain
clues to the meaning of the output variables in the given context. Hence we define it as the context meaning of the output variable. This is especially important when the format string corresponding to the output variable is made up only of escape sequences that shed little light on the meaning of the variable.

**SA Feature 5.** The Java SA can extract context meanings (if any) for variables.

When the Java SA encounters an output statement with no format string, the Java SA examines statements before the output statement until it encounters an output statement that only has a format string and displays no variable. The Java SA extracts this as the possible meaning of the variable and presents the information as a guideline to the expert user, to enable him/her resolve any ambiguities. The result of this search for possible meaning is not affected by the presence of any number of statements in between. For example, consider the code fragment shown in Figure 5-4. The Java SA cannot extract any meaning for the variables \( tfinish \), \( tstart \) and \( tcost \). However, the semantic clues embedded in the output statement that serves as a title for the tabular display of data is captured by the Java SA as the context meaning of these variables (notice, the Java SA intelligently disregards output statements that have a format string made up of non-alphanumeric characters only). The Java SA extracts the string 'Finish Date of Task Start Date of Task Unit Cost for Task’ as the context or possible meaning for the variables \( tfinish \), \( tstart \) and \( tcost \).

Java provides a rich set of data types and methods that can be invoked on objects of these built-in data types. This increases the expressive power of the language and allows developers to use any combination of these objects and methods to manipulate and compare variables.
System.out.println("Finish Date of Task Start Date of Task Unit Cost for Task");
System.out.println("---------------------------------------------------------");
while (rset.next())
{
    Date tstart = rset.getDate("Task_Start_Date");
    Date tfinish = rset.getDate("Task_Finish_Date");
    float tcost = rset.getFloat("Task_UnitCost");
    tcost = checkDuration(tstart, tfinish, tcost);
    stmt.executeUpdate("UPDATE MSP_Tasks SET Task_UnitCost = " + tcost + "
    WHERE Task_Start_Date = '" + tstart + "' AND Task_Finish_Date = '" +
    tfinish + "'");
    System.out.print(tfinish);
    System.out.print("\t" + tstart);
    System.out.println("\t" + tcost);
}

Figure 5-4. Code fragment demonstrating context meaning of variables

SA Feature 6. The Java SA can capture business rules involving method invocations on variables.

Since the Java parser treats all method invocations on objects as a simple function call (it ignores the fact that the method was invoked on an object), the Java SA parses the method name to learn if the method was in fact invoked on a pre-defined variable or object. If this approach was not adopted, then the business rule in Figure 5-5 would not have been discovered for slicing variable \textit{tstart}.

The central idea behind object-oriented languages like Java is to encapsulate all the data manipulation statements into individual function such that each function has a specific functionality. Consequently, the application code written in these languages will contain a sequence of function calls with variables being passed to these functions as parameters. Therefore, the semantic knowledge (business rules and meaning) for a single physical entity or variable may potentially be distributed among several functions. Tracing each of these function calls would generate a comprehensive report of the semantics of the slicing variable.
if ((tstart.getDate() < pstart.getDate()) ||
    (tfinish.getDate() > pfinish.getDate())) {
    System.out.println("The task start and finish dates have
to be within the project start and finish dates");
}

Figure 5-5. Business rules involving method invocations on slicing variables

**SA Feature 7a.** Function calls are traced i.e. if a slicing variable is passed as a parameter to a method defined within the same file, then the semantic information gathered for the formal and actual parameter is integrated.

The Java SA captures parameter passing and traces function calls by recording the name of each function that a variable is passed to along with the rest of the semantic knowledge discovered for that variable.

**SA Feature 7b.** The same variable may be passed to more than one function as a parameter. The Java SA can capture and integrate the semantic knowledge extracted for the actual parameter and all its associated formal parameters.

In the code fragment shown is Figure 5-6, the Java SA traces the slicing variable `tstart` to two different methods `checkDuration` and `checkIfValidDate` and merges the semantic knowledge extracted for the actual parameter `tstart` and both its associated formal parameters `s` and `i`.

Figure 5-7 demonstrates another interesting scenario where the slicing variable `tstart` is passed to function `checkDuration` and its value is received in formal parameter `s`. The variable `s` is in turn passed to another function `checkIfValidDate` and its value received in variable `i`. The same variable is passed from one function to another in a chain of function calls, a situation we term *parameter chaining*. Parameter chaining occurs
when a variable passed as a parameter to function $f_1$ is passed again from function $f_1$ to another function $f_2$ as a parameter. The Java SA can recognize and integrate semantic information extracted in such situations.

```java
checkifValidDate(tstart);
tcost = checkDuration(tstart, tfinish, tcost);

public static float checkDuration(Date s, Date t, float f)
{
    .
    .
}
public static void checkifValidDate(Date i)
{
    .
    .
    .
}
```

Figure 5-6. Code fragment showing slicing variable $t_{start}$ is passed to two functions

**SA Feature 7c.** The Java SA can capture parameter chaining.

Parameter chaining is captured using a sophisticated merge algorithm in the result generator module of the Java SA. The current Java SA prototype can extract semantic knowledge from application code that directs its output to the standard output, which is one of the many ways to display data in Java. However, if we wanted our Java SA to be able to extract semantic information from Java Servlets, we would have to do the following:

- Add a new pattern to the Java Pattern Matcher to identify and modify output statements in Java Servlets. Output statements in Java Servlets have the format string and output variables embedded inside HTML source code.
- Plug-in HTML parsers into the Pattern Matcher to extract the format string embedded in HTML source code and re-write the output statement like a regular output statement.
tcost = checkDuration(tstart, tfinish, tcost);

public static float checkDuration(Date s, Date t, float f)
{
    checkifValidDate(s);
}

public static void checkifValidDate(Date i)
{
}

Figure 5-7. Code fragment showing parameter chaining

The rest of the semantic analyzer modules need not be modified to capture the semantic information from Java Servlets since the Java Pattern Matcher would have generated a modified source code file according to the grammar listed in Appendix B.

**SA Feature 8.** The Java SA prototype design is extensible and can capture semantics from new Java technologies by plugging in appropriate patterns and parsers into the Pattern Matcher with minimal modification to the actual semantic analyzer modules. This approach is used to extract semantic information does not have to be re-engineered for each time a different kind of input source code has to be analyzed.

We have highlighted some of the important features of the Java SA prototype that clearly demonstrate that the Java SA can extract semantic information from application code written in Java with minimal user input. Not only can the Java SA capture application-specific meanings of entities and attributes, it can also extract business rules dispersed in the application code. As demonstrated, the Java SA is able to capture the semantics of overload operators and parameter chaining. The strength of the Java SA
prototype lies in its extensible and modular design, making a useful and easily maintainable toolkit.

In the next chapter, we summarize our efforts in mining semantic information from application source code and evaluate it against the objectives of SEEK. We also list some of the limitations of the approach used to extract semantic information from application code.
CHAPTER 6
CONCLUSION

Semantic analysis and program comprehension of application code has been an important research topic for more than two decades. Despite extensive previous efforts, a truly comprehensive solution for mining semantic knowledge from application code has remained elusive. Several proposals that approach closely related problems like program comprehension and code improvement exhibit severe shortcomings such as inability to trace procedure or functions calls. The substantial published work on this problem also remains theoretical, with very few implemented systems present. Also, many authors suggest semi-automatic methods to discover business rules from application code written in languages like COBOL. However, there has been no comprehensive effort in the area of business rules extraction to develop a fully automatic discovery of business rules from application code written in any high-level language.

This thesis has provided a general solution for the semantic analysis problem for application code written for relational databases. Our algorithm examines the application code using a combination of several program comprehension techniques and extracts semantic information that is explicitly or implicitly present in the application code. The semantic knowledge extracted is documented and can be used for various purposes such as schema matching and wrapper generation, code improvement, code documentation effort etc. We have manually tested our approach with application code written in ANSI C and Java to validate our semantic analysis algorithm and to estimate how much user
input is required. The following section lists the contribution of this work and the last section discusses possible future enhancements.

### 6.1 Contributions

The most important contributions of this work are the following. First, a broad survey of existing program comprehension and semantic knowledge extraction techniques was presented in Chapter 2. This overview not only updates us with the knowledge of different approaches, but also provides a significant guidance while developing the SA algorithm.

The second major contribution is the design and implementation of a semantic analysis algorithm, which imposes minimum restrictions on the input (application code), is as general as possible in design, and extracts the maximum possible knowledge possible from all the code files, with minimal external intervention.

Third, a different and new approach is presented for mining the context meaning of variables that appear in the application code. Fourth, an approach is presented on how to map a particular column of a table in the underlying database to its application-specific meaning that is extracted from the source code. The fifth and major contribution is the approach used to extract business rules from application code and present them in a code-independent format.

The most significant contribution of the semantic analysis algorithm is its readily extensible design. The algorithm can be easily configured and extended to mine semantic information from a new Java programming language technology by simply plugging the corresponding modules to the pattern matcher, which is a preliminary step in the semantic analysis algorithm. Only minimal changes to the core semantic analysis algorithm and modules are required. It is also important to note that the semantic analysis algorithm
proposed can be used to mine application code written in procedural as well as object-oriented languages. If a source code in a language different from Java or ANSI C is presented to the SA, only a new pattern matcher module will have to be plugged in. Also, the complexity of the semantic analysis algorithm does not increase exponentially with the features of the language being analyzed. For example, the Java SA algorithm complexity both in terms of run-time and algorithm design does not increase significantly (by a factor of $N$) with the features like polymorphism, inheritance and operator-overloading etc that it has to capture.

One of the more significant aspects of the prototype we have built is that is highly automatic and does not require human intervention except in one phase when the user might be asked to resolve any ambiguity in the semantic knowledge extracted. The system is also easy to use and the results are well documented. Another vital feature is the choice of tools. The implementation is in Java, due to the popularity and portability.

Though the preliminary experimental results of the SA prototype are highly encouraging and its development in the context of wrapper generation and the knowledge extraction module in SEEK extremely valuable, there are some shortcomings in the current approach. For example, the process of knowledge extraction from application code could be enhanced with some future work. The following subsection discusses some limitations of the current SA prototype and Section 6.3 presents possible future enhancements.

### 6.2 Limitations

#### 6.2.1 Extraction of Context Meaning

When the semantic analyzer cannot find a format string in the input or output statement that can be associated with the slicing variable, it proceeds to search for a
context meaning of the slicing variable in the code. The approach used to extract the
context meaning simply searches for output statements in the code prior to the current
statement that displays no variables but has a format string. The semantic analyzer
extracts this format string as the context meaning of the slicing variable. However, this
algorithm may generate incorrect, potentially misleading results in some cases, especially
if the application code is poorly written and maintained. Consider the following
statements written in ANSI C:

\[
\text{printf(“Recalulation of the project cost”);} \\
\text{scanf(“%d”, &cost);} \\
\]

The first output statement’s format string is not connected to the following input
statement that accepts the value of the cost. However, the present semantic analyzer
prototype will extract the string “Recalulation of the project cost” as
the context meaning for the variable cost. This may mislead the user into believing that
the variable cost actually corresponds to the project cost.

6.2.2 Semantic Meaning of Functions

In both procedural and object-oriented languages, software developers are encouraged
to write individual functions that implement a specific functionality or feature. Hence, the
in the driver program will contain a series of simple function calls. This style of
programming also ensures that modifications if any to that feature need be made only at
one place in the code. Application code for databases usually follows this design
philosophy rather closely. Therefore, it is possible to encounter an assignment statement
in the application code, where the right hand side of the assignment is a call to a function,
and the left hand side of the assignment statement is the slicing variable. Although the
present semantic analyzer extracts this assignment statement as a business rule corresponding to the slicing variable, little is learned from extracting the assignment statement as the functionality of the operation or function being invoked is not known. In such situations, a significant amount of semantic knowledge may remain undiscovered.

6.3 Future Work

6.3.1 Class Hierarchy Extraction

A powerful feature of object-oriented languages like Java is inheritance. Typically application code written for database applications is well-designed for later re-use and extension of the application. Often the application code also consists of several class files that form an inheritance hierarchy. In order to be able to capture parameter passing to methods defined in other source files, a preliminary and necessary first step would be to extract the inheritance hierarchy of all the classes that comprise the application code. This inheritance hierarchy alone, if discovered, can accurately answer questions if the method being invoked has been previously defined in some base class in the inheritance hierarchy.

A preliminary solution proposed to solve the above described problem would be to construct an N-ary tree, where each node in the tree represents an object in the inheritance hierarchy. Each node would also contain the signatures of all the methods defined in that class file. A node is attached as a child of the parent node if it derives from the parent node. Therefore, a traversal of this tree will quickly tell us what classes the present class under analysis is derived from.

6.3.2 Improvements to the Algorithm

Currently our semantic analysis algorithm puts a restriction on the format of the output statements, since the semantic analyzer can only analyze output statements that direct
their output to the standard output. However, output can be directed to file or displayed in HTML format, methods that are very frequently used in application code. It is important therefore to extend the semantic analysis algorithm to capture semantic knowledge from such statements.

Another area of improvement is the representation of the business rules extracted. It is important to leverage existing technology or to develop our own model to represent business rules extracted from application code in a completely code independent format, which can be easily understood by people outside of the code development community, and such that it can be easily exchanged in the form of e-mails and memos. Finally, although semantic analysis is part of a build-time activity, it will be interesting to conduct further performance analysis experiments especially for large application code files and make the prototype more efficient.
### APPENDIX A
GRAMMAR USED FOR THE ‘C’ CODE SEMANTIC ANALYZER

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>CProgram</td>
<td>Consts Forwards Dclns Function+ =&gt; &quot;program&quot;;</td>
</tr>
<tr>
<td>Includes</td>
<td>('#include' '&quot;' &lt;filename&gt; '&quot;';)* =&gt; &quot;include&quot;;</td>
</tr>
<tr>
<td>Consts</td>
<td>(Const ';')+ =&gt; &quot;consts&quot;</td>
</tr>
<tr>
<td>Const</td>
<td>'#define' Name =&gt; &quot;const&quot;</td>
</tr>
<tr>
<td>Forwards</td>
<td>(Forward ';')+ =&gt; &quot;forwards&quot;</td>
</tr>
<tr>
<td>Forward</td>
<td>'^' Type Name Params =&gt; &quot;forward&quot;;</td>
</tr>
<tr>
<td>Dclns</td>
<td>(DclnList ';')+ =&gt; &quot;dclns&quot;</td>
</tr>
<tr>
<td>Type</td>
<td>Id;</td>
</tr>
<tr>
<td>DclnList</td>
<td>Type Dcln list ',' =&gt; &quot;dcln&quot;;</td>
</tr>
<tr>
<td>Dcln</td>
<td>Id '=' Expression =&gt; &quot;=&quot;</td>
</tr>
<tr>
<td>Function</td>
<td>Type Name Params '{' Dclns Statement+ '}' =&gt; &quot;function&quot;;</td>
</tr>
<tr>
<td>Params</td>
<td>'{' DclnList ? '}' =&gt; &quot;params&quot;;</td>
</tr>
<tr>
<td>Block</td>
<td>'{' Statement* '}' =&gt; &quot;block&quot;;</td>
</tr>
<tr>
<td>Statement</td>
<td>Assignment ';'</td>
</tr>
<tr>
<td>Assignment</td>
<td>Name '(' (Expression list ')')? ')' =&gt; &quot;call&quot;</td>
</tr>
<tr>
<td>'printf'</td>
<td>'(' String? Expression list ','? ')' =&gt; &quot;print&quot;</td>
</tr>
<tr>
<td>'scanf'</td>
<td>'(' String? Id list ','? ')' =&gt; &quot;scanf&quot;</td>
</tr>
<tr>
<td>'if'</td>
<td>'(' Expression ')'</td>
</tr>
<tr>
<td>'while'</td>
<td>'(' Expression ')'</td>
</tr>
<tr>
<td>'for'</td>
<td>'(' Assignment ';' Expression ';' Assignment ')' Statement</td>
</tr>
<tr>
<td>'do'</td>
<td>Statement 'while' Expression ';'</td>
</tr>
<tr>
<td>'switch'</td>
<td>'(' Term ')' '(' Case+</td>
</tr>
<tr>
<td>SQLprefix SQLstatement SQLterminator?</td>
<td>&quot;embSQL&quot;</td>
</tr>
<tr>
<td>(DclnList ';'+)</td>
<td>&quot;dclns&quot;</td>
</tr>
<tr>
<td>Primary '+'</td>
<td>&quot;++&quot;</td>
</tr>
<tr>
<td>Primary '--'</td>
<td>&quot;--&quot;</td>
</tr>
<tr>
<td>SQLprefix EXEC SQL DBclause?</td>
<td>&quot;beginSQL&quot;</td>
</tr>
<tr>
<td>SQLterminator END-EXEC</td>
<td>&quot;endSQL&quot;</td>
</tr>
<tr>
<td>SQLstatement</td>
<td>'SQLselectone'</td>
</tr>
<tr>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist</td>
<td>&quot;SQLselectone&quot;</td>
</tr>
<tr>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression</td>
<td>&quot;SQLselectone&quot;</td>
</tr>
<tr>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' 'EXISTS' SQLExpression</td>
<td>&quot;SQLselectone&quot;</td>
</tr>
<tr>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' 'NOT EXISTS' SQLExpression</td>
<td>&quot;SQLselectone&quot;</td>
</tr>
<tr>
<td>'SELECT' 'COUNT' '(' '*' ')' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression</td>
<td>&quot;SQLselectonecount&quot;</td>
</tr>
<tr>
<td>'SELECT' 'DISTINCT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression</td>
<td>&quot;SQLselectonedistinct&quot;</td>
</tr>
<tr>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression 'GROUP' 'BY' columnlistgroupby</td>
<td>&quot;SQLselectonegroupby&quot;</td>
</tr>
<tr>
<td>'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression 'ORDER' 'BY' columnlistgroupby</td>
<td>&quot;SQLselectonegroupby&quot;</td>
</tr>
<tr>
<td>'SELECT' columnlist 'FROM' tablelistmod</td>
<td>&quot;SQLselecttwo&quot;</td>
</tr>
<tr>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td>&quot;SQLselecttwo&quot;</td>
</tr>
<tr>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE'</td>
<td>&quot;SQLselecttwo&quot;</td>
</tr>
<tr>
<td>'EXISTS' SQLExpression</td>
<td>=&gt; 'SQLselecttwo'</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' 'NOT EXISTS' SQLExpression</td>
<td>=&gt; 'SQLselecttwocount'</td>
</tr>
<tr>
<td>'SELECT' 'COUNT' '(' '*' ')' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLselecttwodistinct'</td>
</tr>
<tr>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression 'GROUP' 'BY' columnlistgroupby</td>
<td>=&gt; 'SQLselecttwogroupby'</td>
</tr>
<tr>
<td>'SELECT' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression 'ORDER' 'BY' columnlistgroupby</td>
<td>=&gt; 'SQLselecttwogroupby'</td>
</tr>
<tr>
<td>'INSERT' 'INTO' tablelist 'VALUES' '(' hostvariablelist ')'</td>
<td>=&gt; 'SQLinsert'</td>
</tr>
<tr>
<td>'DELETE' Id 'FROM' tablelist 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLdelete'</td>
</tr>
<tr>
<td>'UPDATE' tablelist 'SET' (SQLAssignment ',') list 'WHERE' SQLExpression</td>
<td>=&gt; 'SQLupdate'</td>
</tr>
<tr>
<td>;</td>
<td>=&gt; 'SQLselect'</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>tablelist</th>
<th>=&gt; 'tablelist'</th>
</tr>
</thead>
<tbody>
<tr>
<td>tablelistmod</td>
<td>=&gt; 'tablelist'</td>
</tr>
<tr>
<td>tablename</td>
<td>=&gt; 'tablename'</td>
</tr>
<tr>
<td>columnlist</td>
<td>=&gt; 'columnlist'</td>
</tr>
<tr>
<td>columnlistgroupby</td>
<td>=&gt; 'columnlistgroupby'</td>
</tr>
<tr>
<td>hostvariablelist</td>
<td>=&gt; 'hostvariablelist'</td>
</tr>
<tr>
<td>Variable</td>
<td>=&gt; 'hostvariablelist'</td>
</tr>
<tr>
<td>SQLExpression</td>
<td>=&gt; 'SQLExpression'</td>
</tr>
<tr>
<td>SQLAssignment</td>
<td>=&gt; 'SQLAssignment'</td>
</tr>
<tr>
<td>SQLExpression 'AND' SQLAssignment</td>
<td>=&gt; 'SQLExpression'</td>
</tr>
<tr>
<td>SQLExpression 'OR' SQLAssignment</td>
<td>=&gt; 'SQLExpression'</td>
</tr>
<tr>
<td>SQLAssignment;</td>
<td>=&gt; 'SQLAssignment;'</td>
</tr>
<tr>
<td>SQLAssignment</td>
<td>=&gt; 'SQLAssignment='</td>
</tr>
<tr>
<td>Id '=' Name</td>
<td>=&gt; 'SQLAssignment&gt;'</td>
</tr>
<tr>
<td>Id '&gt;' Name</td>
<td>=&gt; 'SQLAssignment&gt;'</td>
</tr>
<tr>
<td>Id '&lt;' Name</td>
<td>=&gt; 'SQLAssignment&lt;'</td>
</tr>
<tr>
<td>Id '=&gt;' Name</td>
<td>=&gt; 'SQLAssignment=&gt;'</td>
</tr>
<tr>
<td>Id '&lt;=' Name</td>
<td>=&gt; 'SQLAssignment&lt;='</td>
</tr>
<tr>
<td>Id '&lt;&gt;' Name</td>
<td>=&gt; 'SQLAssignment&lt;&gt;</td>
</tr>
<tr>
<td>Id '&lt;&gt;' Name</td>
<td>=&gt; 'SQLAssignment&lt;&gt;'</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>Rule</th>
<th>SQL Assignment Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Id '=' Name '(' (Expression list ',')? ')';</code></td>
<td><code>SQLAssignment=</code></td>
</tr>
<tr>
<td><code>Id '&gt;' Name '(' (Expression list ',')? ')';</code></td>
<td><code>SQLAssignment&gt;</code></td>
</tr>
<tr>
<td><code>Id '&lt;' Name '(' (Expression list ',')? ')';</code></td>
<td><code>SQLAssignment&lt;</code></td>
</tr>
<tr>
<td><code>Id '=&gt;' Name '(' (Expression list ',')? ')';</code></td>
<td><code>SQLAssignment&gt;=</code></td>
</tr>
<tr>
<td><code>Id '&lt;=' Name '(' (Expression list ',')? ')';</code></td>
<td><code>SQLAssignment&lt;=</code></td>
</tr>
<tr>
<td><code>Id '&lt;&gt;' Name '(' (Expression list ',')? ')';</code></td>
<td><code>SQLAssignment&lt;&gt;</code></td>
</tr>
<tr>
<td><code>Id 'LIKE' String</code></td>
<td><code>SQLAssignmentLIKE</code></td>
</tr>
<tr>
<td><code>Id '=' SQLStatement</code></td>
<td><code>SQLAssignment=</code></td>
</tr>
<tr>
<td><code>Id '=' 'ANY' SQLStatement</code></td>
<td><code>SQLAssignment=</code></td>
</tr>
<tr>
<td><code>Id '&gt;' 'ANY' SQLStatement</code></td>
<td><code>SQLAssignment&gt;</code></td>
</tr>
<tr>
<td><code>Id '&lt;' 'ANY' SQLStatement</code></td>
<td><code>SQLAssignment&lt;</code></td>
</tr>
<tr>
<td><code>Id '&lt;=' 'ANY' SQLStatement</code></td>
<td><code>SQLAssignment&lt;=</code></td>
</tr>
<tr>
<td><code>Id '&gt;=' 'ANY' SQLStatement</code></td>
<td><code>SQLAssignment&gt;=</code></td>
</tr>
<tr>
<td><code>Id '&lt;&gt;' 'ANY' SQLStatement</code></td>
<td><code>SQLAssignment&lt;&gt;</code></td>
</tr>
<tr>
<td><code>Id '=' 'IN' SQLStatement</code></td>
<td><code>SQLAssignment=</code></td>
</tr>
<tr>
<td><code>Id '&gt;' 'IN' SQLStatement</code></td>
<td><code>SQLAssignment&gt;</code></td>
</tr>
<tr>
<td><code>Id '&lt;' 'IN' SQLStatement</code></td>
<td><code>SQLAssignment&lt;</code></td>
</tr>
<tr>
<td><code>Id '&lt;=' 'IN' SQLStatement</code></td>
<td><code>SQLAssignment&lt;=</code></td>
</tr>
<tr>
<td><code>Id '&gt;=' 'IN' SQLStatement</code></td>
<td><code>SQLAssignment&gt;=</code></td>
</tr>
<tr>
<td><code>Id '&lt;&gt;' 'IN' SQLStatement</code></td>
<td><code>SQLAssignment&lt;&gt;</code></td>
</tr>
</tbody>
</table>

**DB clause**

<table>
<thead>
<tr>
<th>Rule</th>
<th>SQL Assignment Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BEGIN' 'DECLARE' 'SECTION'</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td><code>'END' 'DECLARE' 'SECTION'</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'WARNING' 'CALL' Name '(' (Expression list ',')? ')';</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'NOT' 'FOUND' 'CALL' Name '(' (Expression list ',')? ')';</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'NOT' 'DO' 'BREAK'</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'NOT' 'DO' 'CONTINUE'</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td><code>COMMIT' 'WORK'</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td><code>WHENEVER' 'SQL' 'ERROR' 'CALL' Name '(' (Expression list ',')? ')';</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td><code>DISCONNECT' 'ALL'</code></td>
<td><code>DBclause</code></td>
</tr>
<tr>
<td>Rule</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td><code>USE</code> Id</td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td><code>CONNECT</code> Name <code>IDENTIFIED</code> <code>BY</code> Name</td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td><code>COMMIT</code></td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td><code>COMMIT</code> <code>WORK</code> <code>RELEASE</code></td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td><code>COMMIT</code> <code>WORK</code></td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td><code>OPEN</code> Name</td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td><code>CLOSE</code> Name</td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td><code>DECLARE</code> Name <code>FOR</code></td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td><code>FETCH</code> Name <code>INTO</code></td>
<td>=&gt; &quot;DBclause&quot;</td>
</tr>
<tr>
<td>hostvariablelist</td>
<td></td>
</tr>
<tr>
<td>Case</td>
<td><code>case</code> <code>&lt;integer&gt;</code> <code>:</code> Block</td>
</tr>
<tr>
<td>Assignment</td>
<td>Id <code>=</code> Expression</td>
</tr>
<tr>
<td></td>
<td>Id <code>+=</code> Expression</td>
</tr>
<tr>
<td></td>
<td>Id <code>=-</code> Expression</td>
</tr>
<tr>
<td>Expression</td>
<td>LExpression <code>?</code> LExpression <code>:</code></td>
</tr>
<tr>
<td></td>
<td>LExpression;</td>
</tr>
<tr>
<td>LExpression</td>
<td>LExpression <code>&amp;&amp;</code> Comparison</td>
</tr>
<tr>
<td></td>
<td>LExpression `</td>
</tr>
<tr>
<td></td>
<td>LExpression <code>~</code> Comparison</td>
</tr>
<tr>
<td></td>
<td>Comparison;</td>
</tr>
<tr>
<td>Term</td>
<td><code>&lt;=</code> Term</td>
</tr>
<tr>
<td></td>
<td><code>==</code> Term</td>
</tr>
<tr>
<td></td>
<td><code>&gt;=</code> Term</td>
</tr>
<tr>
<td></td>
<td>`!=' Term</td>
</tr>
<tr>
<td></td>
<td><code>&lt;</code> Term</td>
</tr>
<tr>
<td></td>
<td><code>&gt;</code> Term</td>
</tr>
<tr>
<td></td>
<td>Term;</td>
</tr>
<tr>
<td>Term</td>
<td><code>+</code> Factor</td>
</tr>
<tr>
<td></td>
<td><code>-</code> Factor</td>
</tr>
<tr>
<td></td>
<td>Factor;</td>
</tr>
<tr>
<td>Factor</td>
<td>Exp <code>*</code> Factor</td>
</tr>
<tr>
<td></td>
<td>Exp <code>/</code> Factor</td>
</tr>
<tr>
<td></td>
<td>Exp <code>%</code> Factor</td>
</tr>
<tr>
<td></td>
<td>Exp ;</td>
</tr>
<tr>
<td>Exp</td>
<td>Primary <code>***</code> Exp</td>
</tr>
<tr>
<td></td>
<td>Primary;</td>
</tr>
<tr>
<td>Primary</td>
<td>`-' Primary</td>
</tr>
<tr>
<td></td>
<td>`+' Primary</td>
</tr>
<tr>
<td></td>
<td>`!' Primary</td>
</tr>
<tr>
<td></td>
<td><code>++</code> Primary</td>
</tr>
<tr>
<td></td>
<td><code>--</code> Primary</td>
</tr>
<tr>
<td></td>
<td>Primary <code>---</code></td>
</tr>
<tr>
<td></td>
<td>Primary <code>+++</code></td>
</tr>
<tr>
<td></td>
<td>Primary <code>--</code></td>
</tr>
<tr>
<td></td>
<td>Atom;</td>
</tr>
<tr>
<td>Atom</td>
<td>→</td>
</tr>
<tr>
<td>------------</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td>Initializer</td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td>Id</td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td>Name</td>
<td>→</td>
</tr>
<tr>
<td>String</td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
</tbody>
</table>
**APPENDIX B**

**GRAMMAR USED FOR THE JAVA SEMANTIC ANALYZER**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Production</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JProgram</td>
<td><code>{ Consts Forwards Dclns Function+ </code>)`</td>
<td>&quot;program&quot;</td>
</tr>
<tr>
<td>Includes</td>
<td>(<code>#include</code> <code>'</code> filename&gt;<code> </code>;`)</td>
<td>&quot;include&quot;</td>
</tr>
<tr>
<td>Consts</td>
<td>(Const <code>;</code>)+</td>
<td>&quot;consts&quot;</td>
</tr>
<tr>
<td>Const</td>
<td><code>#define</code> Name</td>
<td>&quot;const&quot;</td>
</tr>
<tr>
<td>Forwards</td>
<td>(Forward <code>;</code>)+</td>
<td>&quot;forwards&quot;</td>
</tr>
<tr>
<td>Forward</td>
<td><code>^</code> Type Name Params</td>
<td>&quot;forward&quot;</td>
</tr>
<tr>
<td>Dclns</td>
<td>(DclnList <code>;</code>)+</td>
<td>&quot;dclns&quot;</td>
</tr>
<tr>
<td>Type</td>
<td>Id;</td>
<td></td>
</tr>
<tr>
<td>DclnList</td>
<td>AccessLevel <code>static</code>? <code>final</code>? <code>transient</code>? <code>volatile</code>? Type Dcln list <code>;</code></td>
<td>&quot;dcln&quot;</td>
</tr>
<tr>
<td>Dcln</td>
<td>Id <code>=</code> Expression</td>
<td>&quot;=&quot;</td>
</tr>
<tr>
<td>Type</td>
<td>Id;</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Type Type Type Name Params <code>{&quot; Dclns Statement+ </code>}`</td>
<td>&quot;function&quot;</td>
</tr>
<tr>
<td>Params</td>
<td><code>{&quot; DclnList ? </code>}`</td>
<td>&quot;params&quot;</td>
</tr>
<tr>
<td>Block</td>
<td><code>{ </code>Statement+ <code>}</code></td>
<td>&quot;block&quot;</td>
</tr>
<tr>
<td>Statement</td>
<td>Assignment <code>;</code></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td><code>(Expression list </code>;<code>)? </code>)`</td>
<td>&quot;call&quot;</td>
</tr>
<tr>
<td>printf</td>
<td><code>(String)* (Expression)* list </code>+<code>String List</code>+` ')'</td>
<td>&quot;print&quot;</td>
</tr>
<tr>
<td>Emptyprint</td>
<td><code>(Expression)* list </code>+<code>String List</code>+` ')'</td>
<td>&quot;emptyprint&quot;</td>
</tr>
<tr>
<td>Onlyvarprint</td>
<td><code>(Expression) list </code>+<code>String List</code>+` ')'</td>
<td>&quot;onlyvarprint&quot;</td>
</tr>
<tr>
<td>if</td>
<td><code>(Expression)</code> Statement (<code>else</code> Statement)?</td>
<td>&quot;if&quot;</td>
</tr>
<tr>
<td>while</td>
<td><code>(Expression)</code> Statement</td>
<td>&quot;while&quot;</td>
</tr>
<tr>
<td>for</td>
<td><code>(Assignment)</code> Expression <code>;</code></td>
<td>&quot;for&quot;</td>
</tr>
<tr>
<td>Assignment '}'</td>
<td>Statement</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>&quot;for&quot; '(' ';'; ';' ')' =&gt; &quot;for&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'do' Statement 'while' Expression ';' =&gt; &quot;do&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'switch' '{' Term '}'; '{' Case+ 'default'; ':' Block '}' =&gt; &quot;switch&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLprefix SQLstatement SQLterminator? =&gt; &quot;embSQL&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(DclnList ';')* =&gt; &quot;dclns&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'try' '{' Statement* '}' 'catch' '(' Type Id ')' '{' Statement* '}' ';' =&gt; &quot;try&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLprefix SQL? Dbclause? =&gt; &quot;beginSQL&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLterminator END-EXEC =&gt; &quot;endSQL&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQLstatement &quot;SELECT&quot; columnlist 'FROM' tablelistmod</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SQLselecttwo&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SELECT&quot; columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SQLselecttwo&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SELECT&quot; columnlist 'FROM' tablelistmod 'WHERE' 'EXISTS' SQLExpression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SQLselecttwo&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SELECT&quot; columnlist 'FROM' tablelistmod 'WHERE' 'NOT EXISTS' SQLExpression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SQLselecttwo&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SELECT&quot; 'COUNT' '(' '*' ')' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SQLselecttwocount&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SELECT&quot; 'DISTINCT' columnlist 'FROM' tablelistmod 'WHERE' SQLExpression</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SQLselecttwodistinct&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SELECT&quot; columna columnlist 'FROM' tablelistmod 'WHERE' SQLExpression 'GROUP' 'BY' columnlistgroupby</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SQLselectt grouby&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SELECT&quot; columna columnlist 'FROM' tablelistmod 'WHERE' SQLExpression 'ORDER' 'BY' columnlistgroupby</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;SQLselecttwogroupby&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'INSERT' 'INTO' tablelist 'VALUES' '(('hostvariablelist ')')' =&gt; &quot;SQLinsert&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- `DELETE` Id `FROM` tablelist `WHERE` SQLExpression

- `UPDATE` tablelist `SET` (SQLAssignment `,'`) list `WHERE` SQLExpression

- `;` => "SQLselect"

- tablelist -> ( Name list `,`) => "tablelist"

- tablelistmod -> ( tablename list `,`) => "tablelist"

- tablename -> Id Id => "tablename"

- columnlist -> ( Term list `,`) => "columnlist"

- "" ; => "columnlist"

- columnlistgroupby -> ( Name list `,`) => "columnlistgroupby"

- Hostvariablelist -> (Variable list `,`) => "hostvariablelist"

- Variable -> `:` Name ;

- SQLExpression -> SQLExpression `AND` SQLAssignment

- SQLExpression `OR` SQLAssignment

- SQLAssignment ;

- SQLAssignment -> Id `=' Name => "SQLAssignment="

- Id `>' Name => "SQLAssignment>

- Id `<' Name => "SQLAssignment<

- Id `=>' Name => "SQLAssignment>="

- Id `<=' Name => "SQLAssignment<="

- Id `<>' Name => "SQLAssignment<>

- Id `=' (Expression list `,')? Name `=(' (`)``,'`)?

- Id `>' (Expression list `,')? Name `=(' (`)``,'`)?

- Id `<' (Expression list `,')? Name `=(' (`)``,'`)?

- Id `=>' (Expression list `,')? Name `=(' (`)``,'`)?

- Id `<=' (Expression list `,')? Name `=(' (`)``,'`)?

- Id `<>` Name `=(' (`)``,'`)?

- Id `LIKE` String => "SQLAssignmentLIKE"

- Id `=' SQLStatement => "SQLAssignment="

- Id `=' `ANY` SQLStatement => "SQLAssignment="
- \(>\) 'ANY' SQLStatement => "SQLAssignment>
- \(<\) 'ANY' SQLStatement => "SQLAssignment<
- \(<=\) 'ANY' SQLStatement => "SQLAssignment<=
- \(>=\) 'ANY' SQLStatement => "SQLAssignment>="
- \(<>\) 'ANY' SQLStatement => "SQLAssignment<>"
- Id '=' 'ALL' SQLStatement => "SQLAssignment="
- Id '>' 'ALL' SQLStatement => "SQLAssignment>
- Id '<' 'ALL' SQLStatement => "SQLAssignment<
- Id '<=' 'ALL' SQLStatement => "SQLAssignment<="
- Id '>=' 'ALL' SQLStatement => "SQLAssignment>="
- Id '<>' 'ALL' SQLStatement => "SQLAssignment<>"
- Id '=' 'IN' SQLStatement => "SQLAssignment="
- Id '>' 'IN' SQLStatement => "SQLAssignment>
- Id '<' 'IN' SQLStatement => "SQLAssignment<
- Id '<=' 'IN' SQLStatement => "SQLAssignment<="
- Id '>=' 'IN' SQLStatement => "SQLAssignment>="
- Id '<>' 'IN' SQLStatement => "SQLAssignment<>"

DB clause
- 'BEGIN' 'DECLARE' 'SECTION' => "Dbclause"
- 'END' 'DECLARE' 'SECTION' => "Dbclause"
- 'WHENEVER' 'SQL' 'WARNING' 'CALL' Name '(' (Expression list ',')? ')' => "Dbclause"
- 'WHENEVER' 'SQL' 'NOT' 'FOUND' 'CALL' Name '(' (Expression list ',')? ')' => "Dbclause"
- 'WHENEVER' 'SQL' 'NOT' 'FOUND' 'DO' 'BREAK' => "Dbclause"
- 'WHENEVER' 'SQL' 'NOT' 'FOUND' 'DO' 'CONTINUE' => "Dbclause"
- 'COMMIT' 'WORK' => "Dbclause"
- 'COMMIT' 'WORK' 'RELEASE' => "Dbclause"
- 'COMMIT' 'WORK' => "Dbclause"
- 'OPEN' Name => "Dbclause"
<p>| -&gt; | 'CLOSE' Name =&gt; &quot;Dbclause&quot; |
| -&gt; | 'DECLARE' Name 'FOR' =&gt; &quot;Dbclause&quot; |
| -&gt; | 'FETCH' Name 'INTO' hostvariablelist =&gt; &quot;Dbclause&quot; |
| Case | -&gt; | 'case' '&lt;integer&gt;' ':' Block |
| Assignment | -&gt; | Id '=' Expression =&gt; &quot;assign&quot; |
| Assignment | -&gt; | Id '+' '=' Expression =&gt; &quot;assign&quot; |
| Assignment | -&gt; | Id '-' '=' Expression =&gt; &quot;assign&quot; |
| Expression | -&gt; | Lexpression '?' Lexpression ':' =&gt; &quot;?&quot; |
| -&gt; | Lexpression; |
| Lexpression | -&gt; | Lexpression Comparison '&amp;&amp;' =&gt; &quot;and&quot; |
| -&gt; | Lexpression Comparison '||' =&gt; &quot;or&quot; |
| -&gt; | Lexpression Comparison '~' =&gt; &quot;xor&quot; |
| -&gt; | Comparison; |
| Comparison | -&gt; | Term '&lt;=' Term =&gt; &quot;&lt;=&quot; |
| -&gt; | Term '==' Term =&gt; &quot;==&quot; |
| -&gt; | Term '&gt;=' Term =&gt; &quot;&gt;=&quot; |
| -&gt; | Term '!=' Term =&gt; &quot;!=&quot; |
| -&gt; | Term '&lt;' Term =&gt; &quot;&lt;&quot; |
| -&gt; | Term '&gt;' Term =&gt; &quot;&gt;&quot; |
| -&gt; | Term; |
| Term | -&gt; | Term '+' Factor =&gt; &quot;+&quot; |
| -&gt; | Term '-' Factor =&gt; &quot;-&quot; |
| -&gt; | Factor; |
| Factor | -&gt; | Exp '<em>' Factor =&gt; &quot;</em>&quot; |
| -&gt; | Exp '/' Factor =&gt; &quot;/&quot; |
| -&gt; | Exp '%' Factor =&gt; &quot;%&quot; |
| -&gt; | Exp ; |
| Exp | -&gt; | Primary '<em><strong>' Exp =&gt; &quot;</strong></em>&quot; |
| -&gt; | Primary; |
| Primary | -&gt; | '-' Primary =&gt; &quot;-<em>&quot; |
| -&gt; | '+' Primary |
| -&gt; | '!' Primary =&gt; &quot;!</em>&quot; |
| -&gt; | '++' Primary =&gt; &quot;++<em>&quot; |
| -&gt; | '-' Primary =&gt; &quot;-</em>&quot; |
| -&gt; | Primary '++' =&gt; &quot;++<em>&quot; |
| -&gt; | Primary '-' =&gt; &quot;-</em>&quot; |
| -&gt; | Atom; |
| Atom | -&gt; | 'eof' =&gt; &quot;eof&quot; |
| -&gt; | '&lt;integer&gt;/' |
| -&gt; | Id |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-&gt;</td>
<td>'(', Expression ')';</td>
</tr>
</tbody>
</table>
| -> | Name '(', (Expression list '
',')? ')', ';'; |
|   | => "rhscall" |
| Initializer | -> '<integer>' |
|   | -> '&' Name => "&" |
|   | -> '*' Name => "*" |
|   | => 'Name; |
| Name | -> '<identifier>' |
|   | -> '<text>' |
| String | -> '<text>'; |
| AccessLevel | -> 'public' |
|   | -> 'private' |
|   | -> 'protected' |
import java.sql.*;
import java.math.*;

public class TestCode1
{

public static void main(String[] args)
{
    try
    {
        DriverManager.registerDriver(new
        oracle.jdbc.driver.OracleDriver());

        Connection conn = DriverManager.getConnection
        ("jdbc:oracle:thin:@titan:1521:orcl","hamish","tiger");

        Statement stmt = conn.createStatement();

        String query = "SELECT Proj_Start_Date + 1, Project_Finish_Date -
        1, Project_Cost FROM MSP_Projects WHERE Proj_Name = 'Avalon'";

        ResultSet rset = stmt.executeQuery(query);

        Date pstart = rset.getDate(0);
        Date pfinish = rset.getDate(1);
        float pcost = rset.getFloat(2);

        if (checkCost(pcost) > 1000000)
        {
            //Give 10% discount for big budget projects
            pcost = pcost - pcost * 10/100;
            stmt.executeUpdate("UPDATE MSP_Projects SET Project_Cost =
            " + pcost + " WHERE Proj_Name = 'Avalon'");
        }

        String displayString;
        displayString = "Project Start Date " + pstart;
        System.out.println(displayString);
        System.out.println("Project Finish Date for Avalon " + pfinish);

        String query = "SELECT Task_Start_Date, Task_Finish_Date,
        Task_UnitCost FROM MSP_Tasks WHERE Task_Name = 'Tiles'";
        //This query extracts the start and finish date Task Name 'Tiles'
        ResultSet rset = stmt.executeQuery(query);
    }

    String displayString;
    displayString = "Project Start Date " + pstart;
    System.out.println(displayString);
    System.out.println("Project Finish Date for Avalon " + pfinish);

    String query = "SELECT Task_Start_Date, Task_Finish_Date,
    Task_UnitCost FROM MSP_Tasks WHERE Task_Name = 'Tiles'";
    //This query extracts the start and finish date Task Name 'Tiles'
    ResultSet rset = stmt.executeQuery(query);
}
System.out.println("Finish Date of Task Start Date of Task Unit Cost for Task");
System.out.println("---------------------------------------------
------------");

while (rset.next())
{
    Date tstart = rset.getDate("Task_Start_Date");
    Date tfinish = rset.getDate("Task_Finish_Date");
    float tcost = rset.getFloat("Task_UnitCost");

    checkifValidDate(tstart);
    tcost = checkDuration(tstart, tfinish, tcost);

    stmt.executeUpdate("UPDATE MSP_Tasks SET Task_UnitCost = "+ tcost + " WHERE Task_Start_Date = "+ tstart + " AND Task_Finish_Date = "+ tfinish + "");

    System.out.print(tfinish);
    System.out.print("\t" + tstart);
    System.out.println("\t" + tcost);

    if ((tstart.getDate() < pstart.getDate()) ||
        (tfinish.getDate() > pfinish.getDate()))
    {
        System.out.println("The task start and finish dates
have to be within the project start and finish
dates");
    }
}

rset.close();
stmt.close();
conn.close();
}

public static float checkDuration(Date s1, Date t1, float f1)
{
    float revisedcost;
    if ((s1.getDate() - t1.getDate()) < 10)
    {
        // 20 % raise in cost for rush orders
        revisedcost = f1 + f1 * 20/100;
        System.out.println("Estimated New Task Unit Cost : " + revisedcost);
    }
    else
    {
        revisedcost = f1;
    }
}  
return revisedcost;   
}
public static void checkifValidDate(Date i1) 
{
  Date d = new Date();
  d.setYear(1970);
  d.setMonth(1);
  d.setDate(1);

  if (i1.getDate() > d.getDate())
  {
    System.out.println("Invalid Date !");
  }
}
APPENDIX D
REDUCED SOURCE CODE GENERATED BY JAVA PATTERN MATCHER

```java
public static void main(String[] args)
{
    try
    {
        DriverManager.registerDriver(
            oracle.jdbc.driver.OracleDriver());

        Connection conn = DriverManager.getConnection
            ("jdbc:oracle:thin:@titan:1521:orcl","hamish",
             "tiger");

        String query = "SELECT Proj_Start_Date + 1,
                        Project_Finish_Date -1, Project_Cost FROM MSP_Projects
                        WHERE Proj_Name = 'Avalon'";

        String displayString;
        displayString = "Project Start Date " + pstart;
        printf(displayString);
        printf("Project Finish Date for Avalon " + pfinish);

        String query = "SELECT Task_Start_Date, Task_Finish_Date,
                        Task_UnitCost FROM MSP_Tasks WHERE Task_Name = 'Tiles'";
        //This query extracts the start and finish date Task Name 'Tiles'

        printf("Finish Date of Task Start Date of Task Unit Cost
                for Task");
        printf("---
```

```
while (rset.next())
{
    Date tstart = getDate("Task_Start_Date");
    Date tfinish = getDate("Task_Finish_Date");
    float tcost = getFloat("Task_UnitCost");
    checkifValidDate(tstart);
    tcost = checkDuration(tstart, tfinish, tcost);

    printf(tfinish);
    printf("\t" + tstart);
    printf("\t" + tcost);

    if ((tstart.getDate() < pstart.getDate()) ||
        (tfinish.getDate() > pfinish.getDate()))
    {
        printf("The task start and finish dates have to
        be within the project start and finish dates");
    }
}

rset.close();
stmt.close();
conn.close();
}

catch (Exception e)
{
    printf("ERROR : " + e);
    e.printStackTrace(System.out);
}

public static float checkDuration(Date s1, Date t1, float f1)
{
    float revisedcost;
    if (s1.getDate() - t1.getDate() < 10)
    {
        // 20 % raise in cost for rush orders
        revisedcost = f1 + f1 * 20/100;

        printf("Estimated New Task Unit Cost : " + revisedcost);
    }
    else
    {
        revisedcost = f1;
    }
    return revisedcost;
}

public static void checkifValidDate(Date i1)
{
    Date d;
    d.setYear(1970);
    d.setMonth(1);
    d.setDate(1);
    if (i1.getDate() > d.getDate())
    {
        printf("Invalid Date !");
    }
}
APPENDIX E
AST FOR THE TEST CODE

--------- AST ---------
program(7)
  . consts(0)
  . forwards(0)
  . dclns(0)
    . <identifier>(1)
    . . void(0)
    . function(5)
    . . <identifier>(1)
    . . . main(0)
    . . params(1)
    . . . dcln(2)
    . . . . <identifier>(1)
    . . . . . String[](0)
    . . . . <identifier>(1)
    . . . . . args(0)
    . . dclns(0)
    . . try(19)
    . . . call(2)
    . . . . <identifier>(1)
    . . . . . DriverManager.registerDriver(0)
    . . . rhscall(1)
    . . . . . <identifier>(1)
    . . . . . . oracle.jdbc.driver.OracleDriver(0)
    . . . . <identifier>(1)
    . . . . . Connection(0)
    . . . . assign(2)
    . . . . . <identifier>(1)
    . . . . . . conn(0)
    . . . . . rhscall(4)
    . . . . . . <identifier>(1)
    . . . . . . . DriverManager.getConnection(0)
    . . . . . . <string>(1)
    . . . . . . . "jdbc:oracle:thin:@titan:1521:orcl"(0)
    . . . . . . . <string>(1)
    . . . . . . . "hamish"(0)
    . . . . . . <string>(1)
    . . . . . . . "tiger"(0)
    . . . dclns(1)
    . . . dcln(2)
    . . . . <identifier>(1)
    . . . . . String(0)
    . . . . . . = (2)
    . . . . . <identifier>(1)
    . . . . . . . query(0)
    . . . . . . . <string>(1)
"SELECT Proj_Start_Date + 1, Project_Finish_Date - 1,
Project_Cost FROM MSP_Projects WHERE Proj_Name = 'Avalon'"
. . . . . >(2)
. . . . . . rhscall(2)
. . . . . . . <identifier>(1)
. . . . . . . checkCost(0)
. . . . . . . <identifier>(1)
. . . . . . . pcost(0)
. . . . . . . <integer>(1)
. . . . . . . 1000000(0)
. . . . . block(1)
. . . . . . assign(2)
. . . . . . . <identifier>(1)
. . . . . . . pcost(0)
. . . . . . . -(2)
. . . . . . . <identifier>(1)
. . . . . . . pcost(0)
. . . . . . . *(2)
. . . . . . . <identifier>(1)
. . . . . . . pcost(0)
. . . . . . . /(2)
. . . . . . . <integer>(1)
. . . . . . . 10(0)
. . . . . . . <integer>(1)
. . . . . . . 100(0)
. . . . . dclns(1)
. . . . . dcln(2)
. . . . . . <identifier>(1)
. . . . . . String(0)
. . . . . . <identifier>(1)
. . . . . . . displayString(0)
. . . . . assign(2)
. . . . . . <identifier>(1)
. . . . . . . displayString(0)
. . . . . +(2)
. . . . . . <string>(1)
. . . . . . . "Project Start Date "(0)
. . . . . . <identifier>(1)
. . . . . . . pstart(0)
. . . . . onlyvarprintf(1)
. . . . . . <identifier>(1)
. . . . . . . displayString(0)
. . . . . printf(2)
. . . . . . <string>(1)
. . . . . . . "Project Finish Date for Avalon "(0)
. . . . . . <identifier>(1)
. . . . . . . pfinish(0)
. . . . . dclns(1)
. . . . . dcln(2)
. . . . . . <identifier>(1)
. . . . . . String(0)
. . . . . . . =(2)
. . . . . . . <identifier>(1)
. . . . . . . query(0)
. . . . . . . <string>(1)
. . . . . . . . . "SELECT Task_Start_Date, Task_Finish_Date, Task_UnitCost
. . . . . . . . . FROM MSP_Tasks WHERE Task_Name = 'Tiles'"(0)
. . . . . embSQL(3)
. . . . . beginSQL(0)
SQL

columnlist(3)

<identifier>(1)

Task_Start_Date(0)

<identifier>(1)

Task_Finish_Date(0)

<identifier>(1)

Task_UnitCost(0)

tablelist(1)

<identifier>(1)

MSP_Tasks(0)

eendSQL(0)

emptyprintf(1)

<string>(1)

"Finish Date of Task Start Date of Task Unit Cost for Task"

emptyprintf(1)

<string>(1)

"----------------------------------------------------------"

while(2)

rhscall(1)

<identifier>(1)

rset.next(0)

block(7)

dclns(3)

dcln(2)

<identifier>(1)

Date(0)

=(2)

<identifier>(1)

tstart(0)

rhscall(2)

<identifier>(1)

getDate(0)

<string>(1)

"Task_Start_Date"

dcln(2)

<identifier>(1)

Date(0)

=(2)

<identifier>(1)

tfinish(0)

rhscall(2)

<identifier>(1)

getDate(0)

<string>(1)

"Task_Finish_Date"

dcln(2)

<identifier>(1)

float(0)

=(2)

<identifier>(1)

tcost(0)

rhscall(2)

<identifier>(1)

getFloat(0)
"Task_UnitCost"(0)
call(2)
<identifier>(1)
checkValidDate(0)
<identifier>(1)
tstart(0)
assign(2)
<identifier>(1)
tcost(0)
rhscall(4)
<identifier>(1)
checkDuration(0)
<identifier>(1)
tstart(0)
<identifier>(1)
tfinish(0)
<identifier>(1)
tcost(0)
onlyvarprintf(1)
<identifier>(1)
tfinish(0)
printf(2)
<string>(1)
"\t"(0)
<identifier>(1)
tstart(0)
printf(2)
<string>(1)
"\t"(0)
<identifier>(1)
tfinish(0)
if(2)
or(2)
<(2)
<identifier>(1)
tstart.getDate(0)
rhscall(1)
<identifier>(1)
tfinish.getDate(0)
pstart.getDate(0)
>(2)
rhscall(1)
<identifier>(1)
tfinish.getDate(0)
rhscall(1)
<identifier>(1)
pfinish.getDate(0)
block(1)
emptyprintf(1)
<string>(1)
"The task start and finish dates have to be within the project start and finish dates"(0)
call(1)
<identifier>(1)
rset.close(0)
call(1)
stmt.close(0)
call(1)
stmt.close(0)
call(1)
conn.close(0)
catch(4)
<identifier>(1)
Exception(0)
<identifier>(1)
e(0)
printf(2)
<string>(1)
"ERROR : ", e(0)
<identifier>(1)
e(0)
call(2)
<identifier>(1)
e.printStackTrace(0)
<identifier>(1)
System.out(0)
function(8)
<identifier>(1)
float(0)
<identifier>(1)
checkDuration(0)
params(1)
dcln(6)
<identifier>(1)
Date(0)
<identifier>(1)
s1(0)
<identifier>(1)
Date(0)
<identifier>(1)
t1(0)
<identifier>(1)
float(0)
<identifier>(1)
f1(0)
dclns(1)
dcln(2)
<identifier>(1)
float(0)
<identifier>(1)
revisedcost(0)
if(3)
<(2)
rhsCall(1)
<identifier>(1)
s1.getDate(0)
rhsCall(1)
<identifier>(1)
t1.getDate(0)
<integer>(1)
10(0)
block(2)
assign(2)

<identifier>(1)

revisedcost(0)

+(2)

<identifier>(1)

f1(0)

*(2)

<identifier>(1)

f1(0)

/(2)

<identifier>(1)

.20(0)

<identifier>(1)

.100(0)

printf(2)

<string>(1)

"Estimated New Task Unit Cost : "(0)

<identifier>(1)

revisedcost(0)

block(1)

assign(2)

<identifier>(1)

revisedcost(0)

<identifier>(1)

f1(0)

return(0)

<identifier>(1)

revisedcost(0)

<null>(0)

function(8)

<identifier>(1)

void(0)

<identifier>(1)

checkifValidDate(0)

dclns(1)

<identifier>(1)

Date(0)

<identifier>(1)

i1(0)

dclns(1)

dcln(2)

<identifier>(1)

Date(0)

<identifier>(1)

i1(0)

call(2)

<identifier>(1)

d.setYear(0)

<identifier>(1)

1970(0)

call(2)

<identifier>(1)

d.setMonth(0)

<identifier>(1)

1(0)
.. call(2)
..  <identifier>(1)
..   . d.setDate(0)
..   <integer>(1)
..     1(0)
.. if(2)
..  > (2)
..   rhscall(1)
..    . <identifier>(1)
..     . i1.getDate(0)
..   rhscall(1)
..    . <identifier>(1)
..     . d.getDate(0)
.. block(1)
.. emptyprintf(1)
..   <string>(1)
..    "Invalid Date !"(0)
---------------------------
APPENDIX F
SEMANTIC ANALYSIS RESULTS OUTPUT

Variable Name   : pfinish
Alias            :
Table Name      : MSP_Projects
Column Name     : Project_Finish_Date
Data Type       : Date
Meaning         : Project Finish Date for Avalon
Business Rules  :
if ((b.getDate() < Project Start Date .getDate()) || (a.getDate() > Project Finish Date for Avalon .getDate()))
{
    printf("The task start and finish dates have to be within
the project start and finish dates");
}

Variable Name : revisedcost
Alias            :
Table Name : MSP_Tasks
Column Name : Task_Start_Date
Data Type : float
Meaning : Estimated New Task Unit Cost :
Business Rules : Estimated New Task Unit Cost :  = c + c * 20/100;
Estimated New Task Unit Cost :  = c;

Variable Name : tfinish
Alias : t1
Table Name : MSP_Projects
Column Name : Task_Finish_Date
Data Type : Date
Meaning : Task Ending Date
Business Rules :
c = checkDuration(b, a, c);
if ((b.getDate() < Project Start Date .getDate()) || (a.getDate() > Project Finish Date for Avalon .getDate()))
{
    printf("The task start and finish dates have to be within
the project start and finish dates");
}
if (b.getDate() - a.getDate() < 10)
{
    // 20 % raise in cost for rush orders
    Estimated New Task Unit Cost :  = c + c * 20/100;
printf("Estimated New Task Unit Cost : "+ Estimated New Task Unit Cost :);
}
else
{
    Estimated New Task Unit Cost : = c;
}

Variable Name : tstart
Alias : i1
: s1
Table Name : MSP_Tasks
Column Name : Task_Start_Date
Data Type : Date
Meaning : Task Beginning Date
Business Rules :
c = checkDuration(b, a, c);

if ((b.getDate() < Project Start Date .getDate()) || (a.getDate() > Project Finish Date for Avalon .getDate()))
{
    printf("The task start and finish dates have to be within the project start and finish dates");
}

if (b.getDate() > d.getDate())
{
    printf("Invalid Date !");
}

if (b.getDate() - a.getDate() < 10)
{
    // 20 % raise in cost for rush orders
    Estimated New Task Unit Cost : = c + c * 20/100;
    printf("Estimated New Task Unit Cost : "+ Estimated New Task Unit Cost :);
}
else
{
    Estimated New Task Unit Cost : = c;
}

Variable Name : tcost
Alias : f1
Table Name : MSP_Tasks
Column Name : Task_UnitCost
Data Type : float
Meaning : Task Cost
Business Rules :
c = checkDuration(b, a, c);
Estimated New Task Unit Cost : = c + c * 20/100;
Estimated New Task Unit Cost : = c;

Variable Name : pstart
Alias : MSP_Projects
Table Name : MSP_Projects
Column Name : Proj_Start_Date
Data Type : Date
Meaning : Project Start Date
Business Rules :
   displayString = "Project Start Date " + Project Start Date ;

   if ((b.getDate() < Project Start_Date.getDate()) || (a.getDate() > Project Finish_Date_for_Avalon.getDate()) )
   {
       printf("The task start and finish dates have to be within the project start and finish dates");
   }

Variable Name : pcost
Alias :
Table Name : MSP_Projects
Column Name : Project_Cost
Data Type : float
Meaning :
Business Rules :
   if (checkCost(pcost) > 1000000) |
   {                                  
       //Give 10% discount for big budget projects
       pcost = pcost - pcost * 10/100;
   }
LIST OF REFERENCES


BIOGRAPHICAL SKETCH

Sangeetha Shekar was born on June 20th, 1977, in Madras, India. She received her Bachelor of Engineering degree in chemical engineering from the Birla Institute of Technology and Sciences (BITS), Pilani, India, in June, 1998.

She joined the department of Computer and Information Science and Engineering at the University of Florida in fall 2000. She worked as a research assistant under Dr. Joachim Hammer and was a member of the Database Systems Research and Development Center. She received a Certificate of Achievement for Academic Excellence from the University of Florida. She completed her Master of Science degree in computer engineering at the University of Florida, Gainesville, in May 2003.

Her research interests include database systems, Internet technologies and programming languages.