M.Tech. Thesis Report

Static Analysis of Comments and Dynamic Analysis of Execution Behavior to Build Semantic Graph of Codebases using Machine Learning And Natural Language Processing Tools.

Author:
Shakti S. Papdeja.
16CS60R54 – CSE Dept.
IIT Kharagpur.

Supervisor:
Prof. Partha Pratim Das.
CSE Dept.
IIT Kharagpur.

A thesis submitted in fulfillment of the requirements for the degree of M.Tech
in
Computer Science and Engineering.

Indian Institute of Technology, Kharagpur.
West Bengal, India 721302.
Nov 2017.

Declaration of Authorship

I, Shakti S. Papdeja, declare that this thesis titled, "**Static Analysis of Comments and Dynamic Analysis of Execution Behavior to Build Semantic Graph of Codebases using Machine Learning And Natural Language Processing Tools**" and the work presented in it is my own. I confirm that:

- This work has been done or mainly while in candidature for a research degree at this university.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my team work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what is done by others and the part contributed by me.

Signature : …………………………………… Date: ……………………………………

APPROVAL OF THE VIVA-VOCE BOARD

Date - ………./……………./……….

Certified that the thesis entitled "**Static Analysis of Comments and Dynamic Analysis of Execution Behavior to Build Semantic Graph of Codebases using Machine Learning And Natural Language Processing Tools**", submitted by "***Miss. Shakti S. Papdeja***" to the Indian Institute of Technology, Kharagpur, for the award of degree Master of Technology has been accepted by the examiners and that the student has successfully defended the thesis in the viva-voce examination held today.

Examiner ………………………………… Supervisor ………………………………………..

CERTIFICATE

Date - ………./……………/……….

This is to certify that the thesis entitled "**Static Analysis of Comments and Dynamic Analysis of Execution Behavior to Build Semantic Graph of Codebases using Machine Learning And Natural Language Processing Tools**", submitted by "Shakti S. Papdeja" to the Indian Institute of Technology, Kharagpur, is a record of bonafide research work under my supervision and I consider it worthy of consideration for the award of the degree of Master of Technology of the Institute.

Supervisor ………………………………………….

Acknowledgement

*I would like to thank my professor**, Dr. Partha Pratim Das** for his exceptional guidance and support, without which this thesis would not have been possible. He gave freedom to pursue my ideas and work at my own pace, and is always available to discuss various problems on the way. His guidance during the whole duration of my involvement has been valuable to me. He has always encouraged, supported and corrected me during the project.*

*In conclusion, I recognize that this project would not have been possible without the support of **Phd. Scholar Srijoni Majumdar**, from Computer Science and Engineering Dept, IIT Kharagpur. Many thanks to all who made this project possible.*

Shakti S. Papdeja.

Abstract

Static Analysis of Comments and Dynamic Analysis of Execution Behavior to Build Semantic Graph of Codebases using Machine Learning And Natural Language Processing Tools.

The major portion of the maintenance effort is spent understanding the existing software. We present an integrated code comprehension model and our experimental analysis with the existing projects. In industry the large scale projects are the more prevalent focus of maintenance activities. We have observed that maintenance frequently switches between top-down and bottom-up comprehension. We analyze both the requirement and derive the behavior of code in the form of semantic graphs and information of code along with query processing engine. Further we have reason to believe that cognition during maintenance is the reverse engineering activity that re-builds the existing design from the code.

In this thesis, it is an attempt to statically analyze the code through the comment extractions and finding the correlation of comments with respect to their statements of corresponding lines, storing this information for query processing model, and dynamically analyzing the code through different test cases to build Semantic Graphs using machine learning model. Further to build a query processing tool for predefined and user defined queries.

Contents

Chapter 1.

Challenges in Program Comprehension

Program comprehension is the main activity of the software developers. Although there has been substantial research to support the programmer, the high amount of time developers need to understand source code remained constant over thirty years. Beside more complex software, what might be the reason? In this work, I explore the past of program-comprehension research, discuss the current state, and outline what future research on program comprehension might bring.

Research in program comprehension has evolved considerably over the past decades. However, only little is known about how developers practice program comprehension in their daily work. I review some of the qualitative and quantitative research to comprehend the strategies, tools, and knowledge used for program comprehension. Using these theories I will then explore the tools needed to design to support program comprehension, the work which is lacking and then I will propose the methods to make better use of technologies for program comprehension.

1.1 Introduction

Oftentimes when programmers are given to maintain (fix bugs or improve performance for existing features) or upgrade (extend / add new features) legacy software, they need to comprehend a large codebase in a short span of time. Usually the source code and working software with tested (golden) data sets are available, but proper requirements / design documents, system / code documentation, change traces, organized Knowledge Transfer (KT), and help from earlier developers are inaccessible or scanty. So they need to repeatedly execute the software with golden data and observe output, debug with golden and new test data, and read through the codes to comprehend various aspects of design, models and implementation of the given software. This is monotonous, error-prone, time-intensive, and often unmanageable (especially for the less experienced). The challenges get compounded in case of multi-threaded applications because these need in-depth understanding of various concurrency models and issues.[16]

Program Comprehension

Static analysis, also called static code analysis, is a method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. The process of scrutinizing code by visual inspection alone (by looking at a printout, for example), without the assistance of automated tools, is sometimes called program understanding or program comprehension.

The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Nevertheless, static analysis is only a first step in a comprehensive software quality-control regime. After static analysis has been done, dynamic analysis is often performed in an effort to uncover subtle defects or vulnerabilities. In computer terminology, static means fixed, while dynamic means capable of action and/or change. Dynamic analysis involves the testing and evaluation of a program based on execution. Static and dynamic analysis, considered together, are sometimes referred to as glass-box testing.[14]

1.2 . Challenges in Program Comprehension

In the beginning, most applications look quite manageable. Maybe you have been involved since the very beginning drawing diagrams and writing code. Maybe you have had to take over software and documentation from somebody else. In any case maintaining, changing or extending software can be challenging. Typical problems include

- Keeping Code and Documentation in Sync.
- Maintaining the Architecture when Things are Changing.
- Understanding somebody else's code.
- Search and History
- Loosing context because of interrupts, switching between tasks and window changes

Still you have to fix bugs, add new features or meet changing requirements. It becomes even more difficult if your application includes a lot of code connected to concurrency, synchronization and event handling. Many IoT applications do so and it is hard to get this concurrent behavior right. Two parallel processes are easy to implement. Synchronizing two parallel processes is okay for most programmers as well. With three processes, all gets much more complex and difficult. At some point, any programmer will give up.

- Keeping Code and Documentation in Sync:

As applications are developed by combining building blocks, drawing data flow and coding algorithms, an up to date graphical documentation of the system is always required to be available, but programmers often avoid updating the documentation.

- Maintaining the Architecture when Things are Changing:

Unexpected complexity or changing requirements may lead to the need to update the architecture of the system as well. However, the architecture of an application is easy to understand and easy to communicate due to its graphical representation. Building blocks promote dividing and encapsulation of complexity. From the block interfaces and the specified data flow, the analysis tool is able to find bugs like deadlocks and race conditions. While there is any change in the code which is not updated accordingly to the architecture may mislead the users in future.

- Understanding somebody else's code

When developers implement a solution for a specific problem, they construct a mapping from a problem domain to a programming domain. Between these domains may exist several intermediate domains and all of these domains have to be reconstructed when trying to understand the corresponding piece of code. Often this reconstruction is hypothesis-driven. We observed some participants asking and answering questions leading to informal hypotheses which were then verified by the developers. This problem of discovering individual human-oriented concepts is often called concept assignment problem [17]. During the interview, many of the participants identified "understanding somebody else's code" as one of the major challenges in program comprehension. One participant argued that "others have another style and other way of thinking", which makes it difficult to reconstruct the mappings mentioned above.

- Search and History

Several studies have shown that developers quickly forget details when they move to a different location within the source code or even a different task. During the observation, we noticed some developers that wrote down notes either on a piece of paper or used an editor as temporal memory. Some used these notes as a search history and wrote down which variable or function names they had already searched for. Nevertheless some participants searched for the same function or variables names more than once because they did not remember every detail of the result or to validate their hypothesis about that specific piece of code. During the interview, some participants mentioned that they often have to execute the same search many times and that a tool that keeps track of the searches and problem solving sessions would be extremely helpful. This finding matches with suggestions by Storey [17].

- Loosing context because of interrupts, switching between tasks and window changes

Discussions, meetings and phone calls are a necessary part of software development. Interrupts disturb developers during their activities. Developers being interrupted lose their current focus and context and it takes them extra time to recover from the interrupts and previously gathered knowledge may be lost. Most of the interrupts we observed were initiated by the developers themselves and were related to knowledge or experience exchange. Another problem we observed were the interrupts caused by window changes or switching between tasks. One participant, for example, had to switch between the development environment and the editor in which he wrote the documentation quite often. After every window change, he had to recover the information needed for the current task[17].

Problem Statement –

Current Comment analysis and Metadata analysis work ignores correlating the semantic information of source code with the comments available for corresponding statements. As well as there is no major work on dynamic analysis of code to understand the behavior of code through PIN tool and machine learning tool.

Contribution –

Based on Comment analysis, Metadata analysis and Dynamic analysis of source code, we provide a tool to fill as many gaps of information extraction as possible and provide a query processing engine for developers and users of the source code.

1.3. Related Work –

Comments have been analyzed for different area of interest like feasibility and benefits of automatically analyzing comments to detect software bugs and bad comments for improving software development methodology by addressing bad comments in the code[1] . Action Oriented Graph Representation (AOGR) & NLPA-Natural Language Processing Analysis, correlates verbs in the method name and corresponding action in OOPS and represents it in graph format[2, 4, 5]. iComment combines Natural Language Processing (NLP), Machine Learning, Statistics and Program Analysis techniques to detect inconsistencies between comments and source code, indicating either bugs or bad comments[3]. JavadocMiner is to automatically assess the quality of source code comments and export in-line documentation and attempts to find number of comments per number of lines of code and their quality [6].

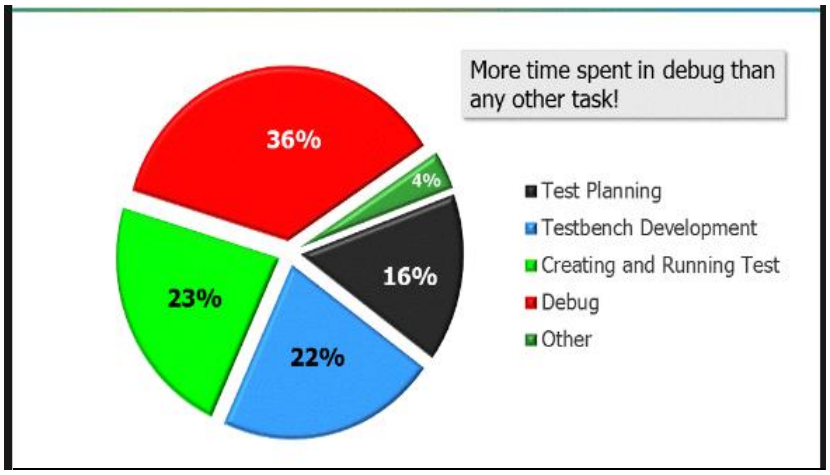
Further Mining of natural language information found within software and its supportive documentation using NLP tools with POS-tagging [8]. There is also the study related to Comment Categorization for Copyright Comment, Header Comment, Member Comment, Inline Comments [9]. There is also proposed work to automatically generate the comments from by analyzing the software repositories [10]. One more research also covers the Comment analysis of Stack Overflow (CrowdSourced Knowledge) to analyze the comments based on popularity relevance, comment rank, word count and sentiment expressed [11].

These all research help to analyze the quality of comment, to correlate the method names with their action, to analyze the source code for bug findings, to construct flow graphs and syntax tree.

1.4. Gap Analysis –

Why Program Comprehension?

Fig. below shows, Mean Time spent by the developers in different Activities for software Development.



Source-
https://www.google.co.in/search?rlz=1C1CHBF_enIN727IN727&biw=1093&bih=510&tbm=isch&sa=1&ei=WXQIWtqHNszvvATl4KOYBA&q=more+time+spent+in+debug+than+any+other+task&oq=more+time+spent+in+debug+than+any+other+task&gs_l=psy-ab.3...65210.88277.0.88546.97.44.0.0.0.0.539.3485.0j3j7j1j1j1.15.0....0...1.1.64.psy-ab..84.6.1336.0..0j0i67k1j0i8i30k1j0i24k1.154.yBbX3TQ-5F8#imgrc=eibdayT4YSDjlM:

The observation from above chart is, the developers spend maximum of their time in debugging the codes than the other activities.

Following points are observed from the previous related work done –

- Initially the legacy code is black box to the programmers, which accepts a set of inputs and provides desired outputs.
- All of the above tools either deals with comment to find quality of comments with respective to code, or statically analyze the code for finding the bugs or flow of the code.
- This project is an attempt to automate the process for creating documentation from metadata extracted from source code and comments of the code.

- This project shall also be producing knowledge graph, which is graphical representation of elements (representing – variables, libraries, methods) with each other, and holding related information in these nodes to explore them.
- This project shall also be taking care of dynamic analysis of code, by executing few test cases and understanding the behavior of code for further analysis of code.

1.5. Suitable Solution for Program Comprehension and Static and Dynamic Analysis of Code –

Program comprehension is one of the first and most frequently performed activities during software maintenance and evolution. In a program, there is not only source code, but also comments. Comments in a program is one of the main sources of information for program comprehension. If a program has good comments, it will be easier for developers to understand the code. Many software systems, often maintain good comments, which can help developers to understand the methods and classes, when they are performing future software maintenance tasks, or for extension of current work. Whenever one wants to make analysis of current code, it majorly seeks following 3 options -

1. Semantic Analysis.
2. Read Documentation.
3. Use Test Cases and try to find out the behavior of code from output.

In this thesis, it is an attempt to reduce the debugging work to analyze the flow, and to automate the process to cover all the above areas and make a tool to facilitate the user with good structured information in the form of graphs, tree based structures and through a query processing engine.

The work done in the project has mainly 3 activities –

1. Information Extraction.
2. Knowledge Graph.
3. Query Processing Engine.

Information Extraction

The chosen strategy is normally that of creating dedicated parsers. Given a grammar and a target meta model, a dedicated parser provides a specific solution which performs both parsing and model-generation tasks. The former is in charge of extracting a syntax tree from the source code and the latter traverses this syntax tree to generate the target model. For example, in refs. [18], dedicated parsers are built to extract models from PL/SQL code.

However, dedicated parser development is a time-consuming and expensive task, because the syntax tree traversals must be hardcoded both to collect scattered information and to resolve references. In addition, mappings are also hardcoded, which hinder maintainability. The effort required is usually alleviated by automatically extracting an AST from the source code. This step is performed using an API, which is intended to make the management of this tree easier. An example of such tools is CLANG and LLVM, which works with C/C++ code. In addition, although these tools tackle AST extraction and management, a mechanism for retrieving scattered information must still be hard-coded, so APIs do not considerably shorten the development time[18].

Knowledge Graph

Software knowledge graph is defined as a graph for representing relevant knowledge in software domains, projects, and systems. In a software knowledge graph, nodes represent software knowledge entities (e.g., classes, issue reports and business concepts), and directed edges represent various relationships between these entities (e.g., method invocation and traceability link). Different nodes and relationships have different properties to describe their inner features.

For example, a method entity's properties include its name, return type, parameter list, access modifier, and description. Knowledge in software knowledge graph is divided into two categories: primitive knowledge and derivative knowledge.

Primitive knowledge is defined as software knowledge added into software knowledge graph through data structure parsing. For example, we may parse the abstract syntax trees (ASTs) of source code files, and then code entities (such as classes, methods and fields) and relationships between them (such as method invocation and inheritance hierarchy) could be added into software knowledge graph.

Query Processing Engine

Once the analysis system is designed, the query evaluation engine supports to find the solution to user's queries from the extracted information and knowledge graph about the source code. There are static and dynamic queries to be processed from the tool. The most generic queries about the source code could be –

1. What are the global variables
2. Number of global variables
3. Number of functions/methods
4. Description about the function names, parameter list and return types.
5. Data structure and algorithms used in the specific block of codes
6. Data structures and algorithms followed throughout the code.

After extracting the information and parsing the comments from the source code, the association between these information can be used to answer these queries.

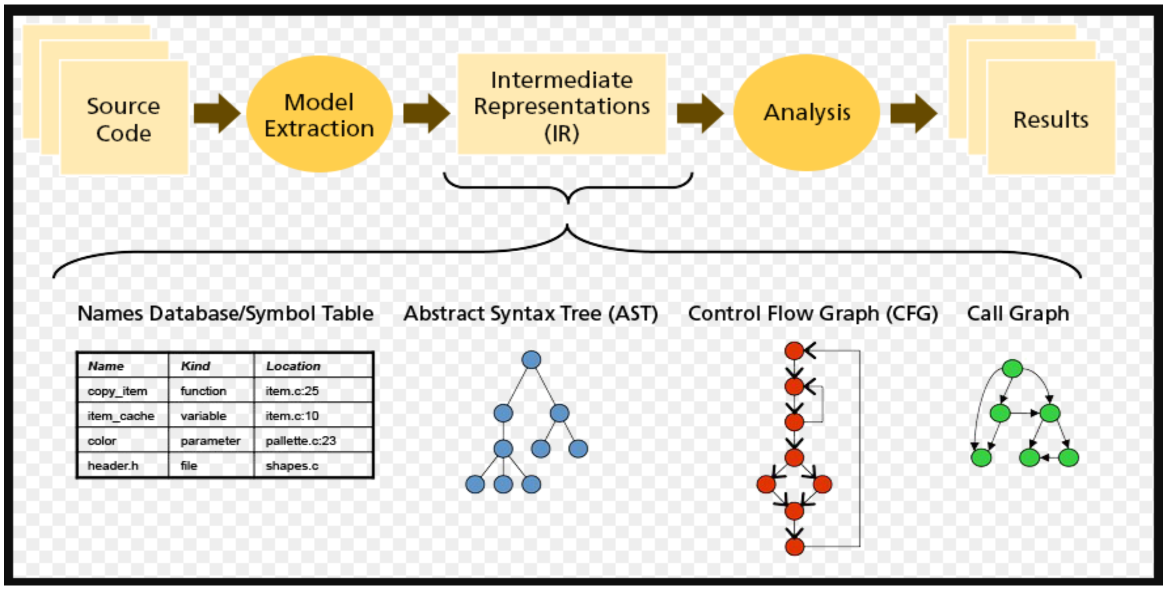
1.6. How does existing Static Analysis model Work?

Source Code Analysis –

Source code analysis is the automated testing of source code for the purpose of debugging a computer program or application before it is distributed or sold. Source code consists of statements created with a text editor or visual programming tool and then saved in a file. The source code is the most permanent form of a program, even though the program may later be modified, improved or upgraded.

Source code analysis can be either static or dynamic. In static analysis, debugging is done by examining the code without actually executing the program. This can reveal errors at an early stage in program development, often eliminating the need for multiple revisions later. After static analysis has been done, dynamic analysis is performed in an effort to uncover more subtle defects or vulnerabilities. Dynamic analysis consists of real-time program testing.

A major advantage of this method is the fact that it does not require developers to make educated guesses at situations likely to produce errors. Other advantages include eliminating unnecessary program components and ensuring that the program under test is compatible with other programs likely to be run concurrently.


Source Code
Model Extraction
Intermediate Representations (IR)
Analysis
Results
Names Database/Symbol Table
Abstract Syntax Tree (AST)
Control Flow Graph (CFG)
Call Graph

Name	Kind	Location
copy_item	function	item.c:25
item_cache	variable	item.c:10
color	parameter	pallette.c:23
header.h	file	shapes.c

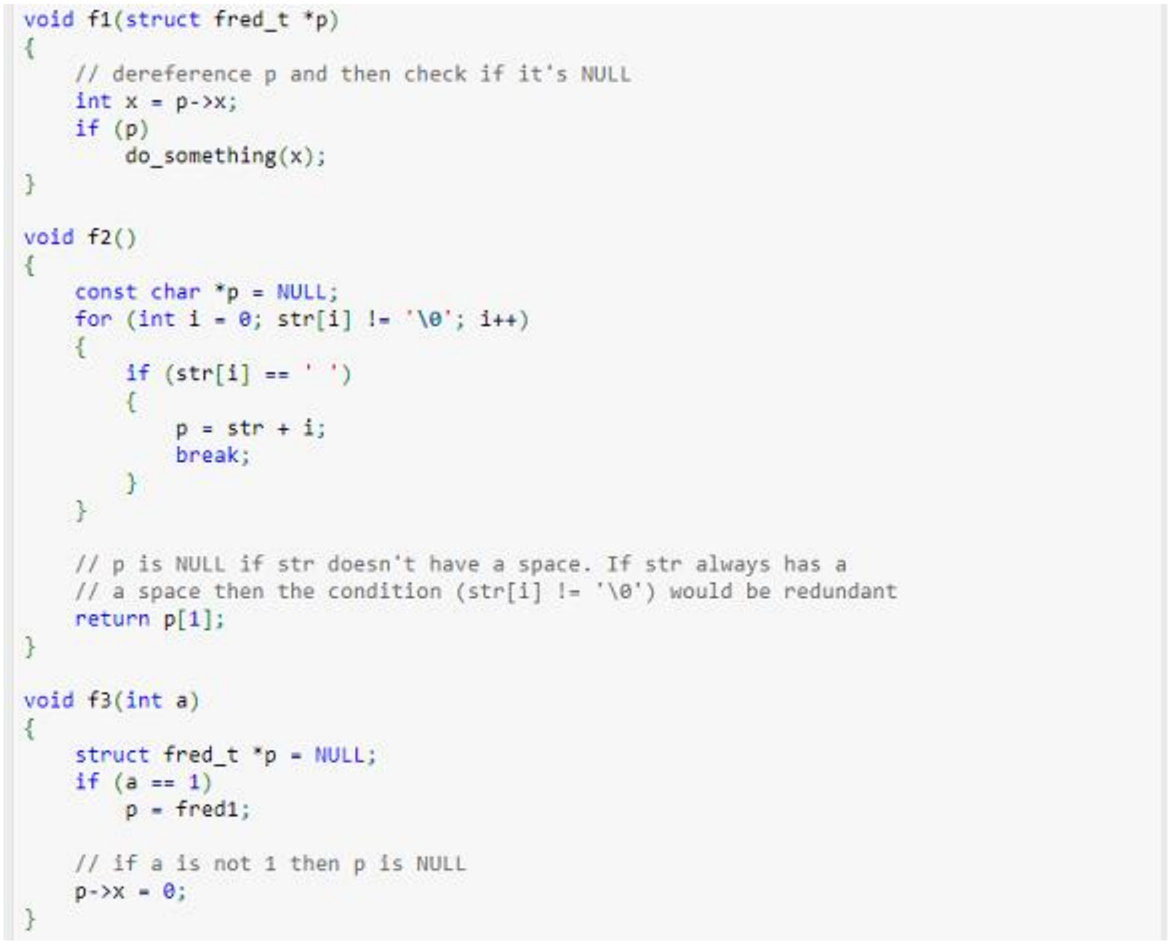
Source - https://www.google.co.in/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwiG-KnFtLnXAhUIQo8KHTiFAwsQjRwIBw&url=http%3A%2F%2Fwww.verifysoft.com%2Fen_grammatech_how_static_analysis_works.html&psig=AOvVaw0mQGMU-F8p1wQumnDbA4nu&ust=1510588932364691

Static analysis has been practical since the 1970s. There are various static analysis tools like – Clang Static Analyzer, CppDepend, CppCheck, SourceMeter, Splint, Doxygen, CodeSonar etc.

By using the existing build environment any of above, it analyzes your code and creates an abstract model of your entire program. In the next step it's symbolic execution engine explores program paths, reasoning about program variables and how they relate. During this process infeasible program paths are pruned from the exploration.

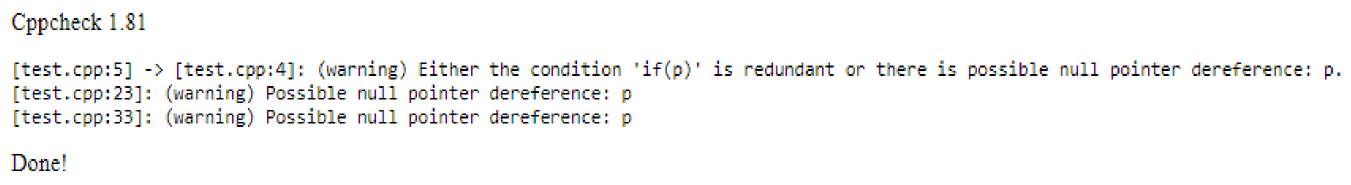
1. AST – **abstract syntax tree** (**AST**), or just **syntax tree**, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.
2. Control Flow Graph – A **control flow graph** (**CFG**) in is a representation, using graph notation, of all paths that might be traversed through a program during its execution.
3. Call Graphs – **Call graphs** are a basic program analysis result that can be used for human understanding of programs, or as a basis for further analyses, such as an analysis that tracks the flow of values between procedures. One simple application of call graphs is finding procedures that are never called.

Example of CppCheck



```
void f1(struct fred_t *p)
{
    // dereference p and then check if it's NULL
    int x = p->x;
    if (p)
        do_something(x);
}

void f2()
{
    const char *p = NULL;
    for (int i = 0; str[i] != '\0'; i++)
    {
        if (str[i] == ' ')
        {
            p = str + i;
            break;
        }
    }

    // p is NULL if str doesn't have a space. If str always has a
    // a space then the condition (str[i] != '\0') would be redundant
    return p[1];
}

void f3(int a)
{
    struct fred_t *p = NULL;
    if (a == 1)
        p = fred1;

    // if a is not 1 then p is NULL
    p->x = 0;
}
```

The output of static analysis tool CppCheck is



```
Cppcheck 1.81

[test.cpp:5] -> [test.cpp:4]: (warning) Either the condition 'if(p)' is redundant or there is possible null pointer dereference: p.
[test.cpp:23]: (warning) Possible null pointer dereference: p
[test.cpp:33]: (warning) Possible null pointer dereference: p

Done!
```

Chapter 2.

Objective of Project

The main objective of project is to extract the information from source code and build the Knowledge graph.

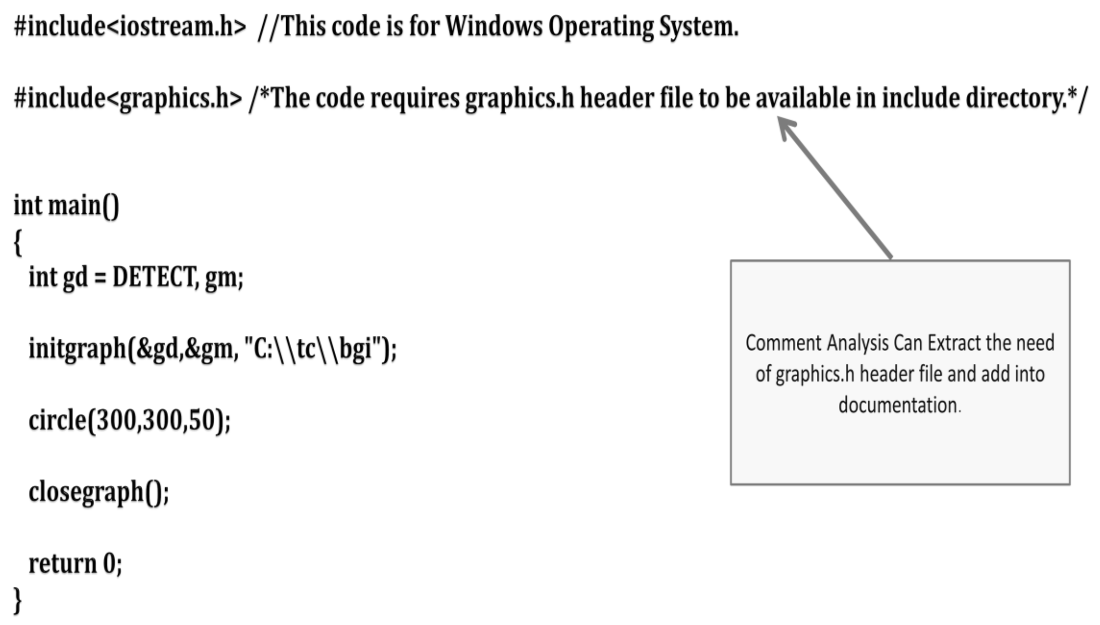
The information extraction is done through

1. Comments of Code
2. AST using LLVM

2.1. Examples for explanation of Project Objective

Following code snippets elaborate the use of comment analysis, which is not taken care by existing semantic analysis tools.

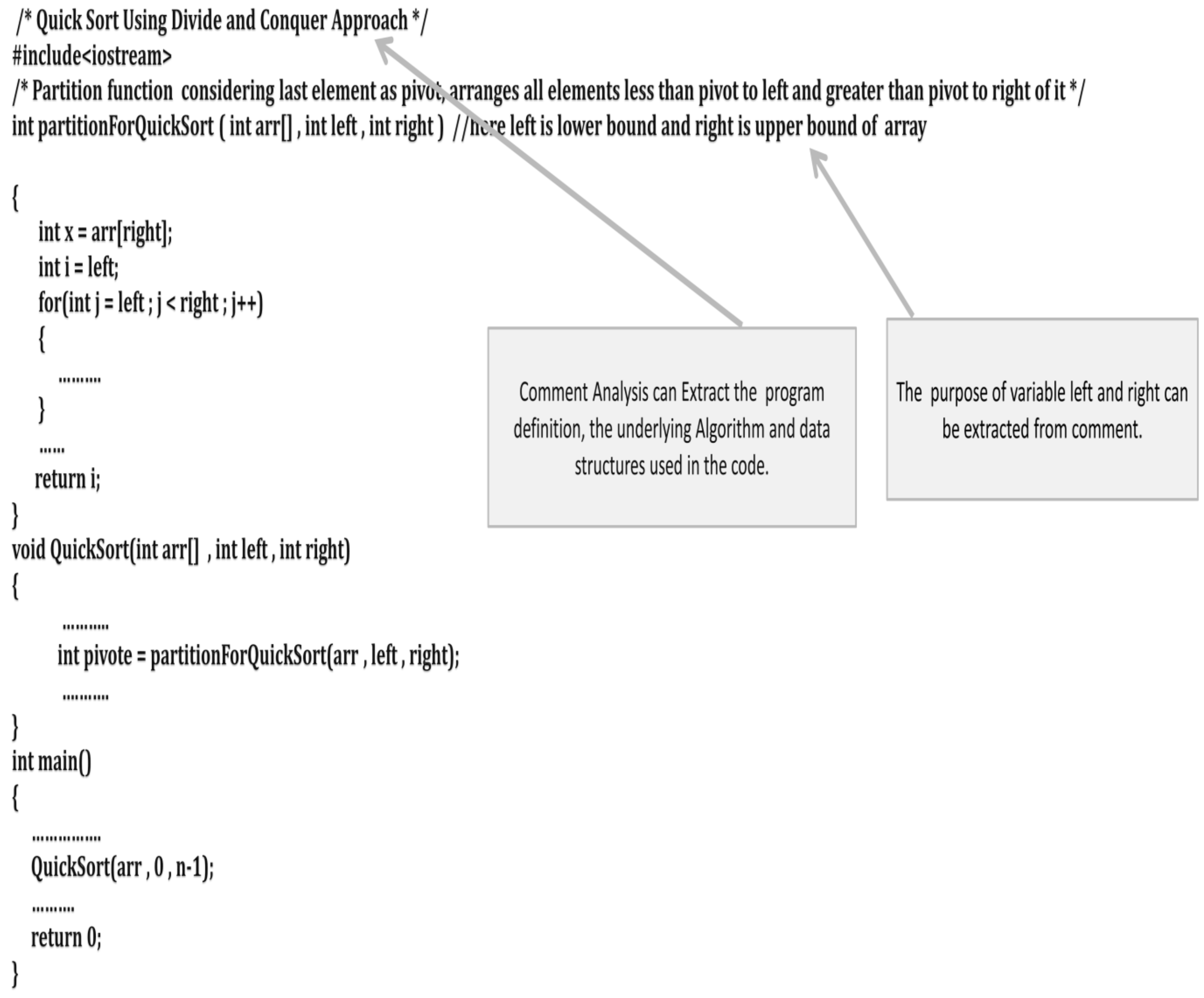
Example 1.



```
#include<iostream.h>  //This code is for Windows Operating System.

#include<graphics.h> /*The code requires graphics.h header file to be available in include directory.*/


int main()
{
  int gd = DETECT, gm;

  initgraph(&gd,&gm, "C:\\tc\\bgi");

  circle(300,300,50);

  closegraph();

  return 0;
}
```

Comment Analysis Can Extract the need of graphics.h header file and add into documentation.

Example 2.

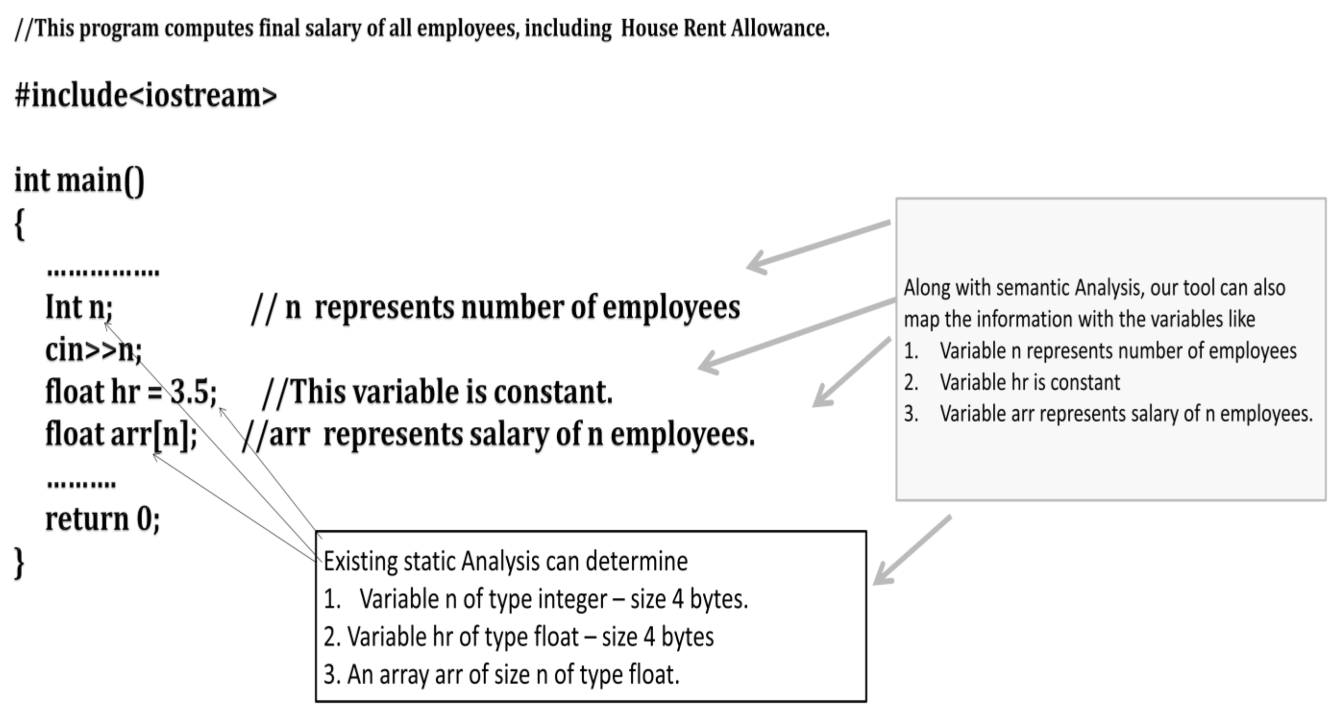


```
/* Quick Sort Using Divide and Conquer Approach */
#include<iostream>
/* Partition function  considering last element as pivot, arranges all elements less than pivot to left and greater than pivot to right of it */
int partitionForQuickSort ( int arr[] , int left , int right )  //here left is lower bound and right is upper bound of  array

{
    int x = arr[right];
    int i = left;
    for(int j = left ; j < right ; j++)
    {
        .........
    }
    ......
    return i;
}
void QuickSort(int arr[]  , int left , int right)
{
        ..........
        int pivote = partitionForQuickSort(arr , left , right);
        ..........
}
int main()
{
    ...............
    QuickSort(arr , 0 , n-1);
    .........
    return 0;
}
```

Comment Analysis can Extract the program definition, the underlying Algorithm and data structures used in the code.

The purpose of variable left and right can be extracted from comment.

Example 3.



```
//This program computes final salary of all employees, including  House Rent Allowance.

#include<iostream>

int main()
{
    ……………
    Int n;                // n  represents number of employees
    cin>>n;
    float hr = 3.5;      //This variable is constant.
    float arr[n];     //arr  represents salary of n employees.
    ………
    return 0;
}
```

Along with semantic Analysis, our tool can also map the information with the variables like

1. Variable n represents number of employees
2. Variable hr is constant
3. Variable arr represents salary of n employees.

Existing static Analysis can determine

1. Variable n of type integer – size 4 bytes.
2. Variable hr of type float – size 4 bytes
3. An array arr of size n of type float.

The observation from above code snippets derives that Static and Dynamic Analysis of Source Code tool, will take the power of existing semantic analysis of codes to the next level for understanding the underlying codes, usage of variables, data structures used, algorithms used, block level analysis of codes, complexities of codes etc.

2.2. Three Phases of Project –

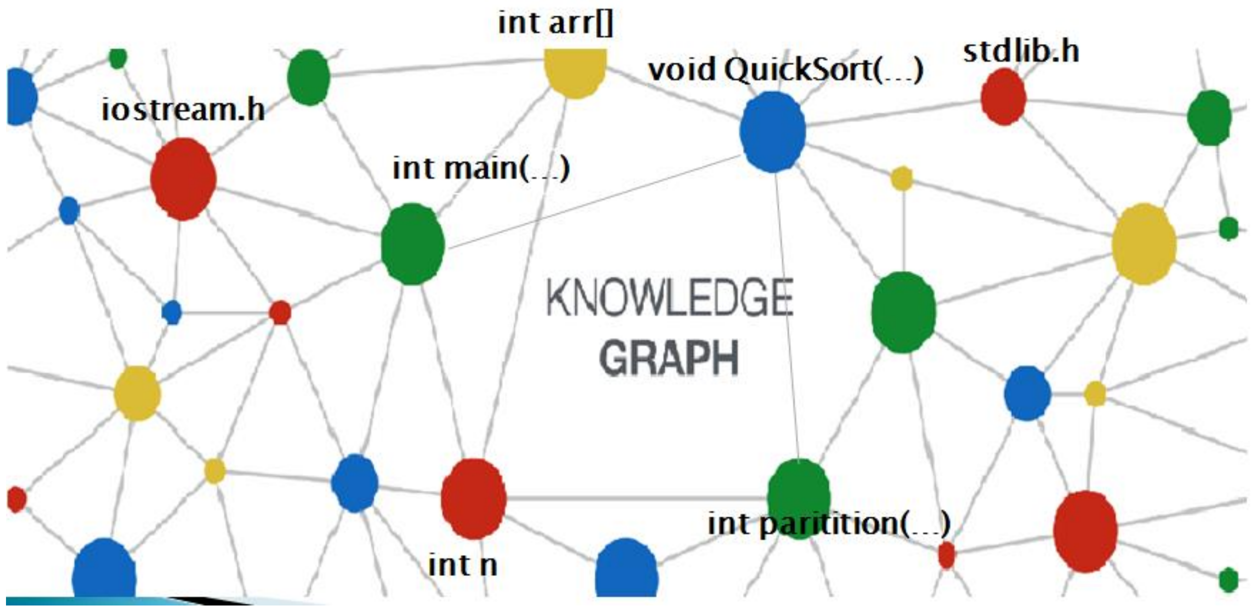
1. **Extracting Information from source Code –**

   The information from source code is extracted using

   1. Extraction of comments using python script, tokenizing them and categorizing the tokens using handcrafted ontology set. The comment extraction is based on regular expression matching process which searches for C/C++ type of comments like
      ```
      //Single line Comments
      /* Multi Line Comments
        …………………….
      ……………………..
      */
      ```
   2. The extracted information is stored into database along with line numbers.
   3. These comments are further tokenized and there vocabulary type information is also stored into database.
   4. All these tokens are further processed to store their Part Of Speech tags and Stemmed words with their POS tags for future purpose.
   5. These all processing are done using NLP tools as discussed in chapter 3.

The second phase of information extraction is using CLANG and LLVM tools for retrieving information from AST and building hierarchical tables from them. The details of this method are also discussed in chapter 3.

2. **Knowledge Graph –**



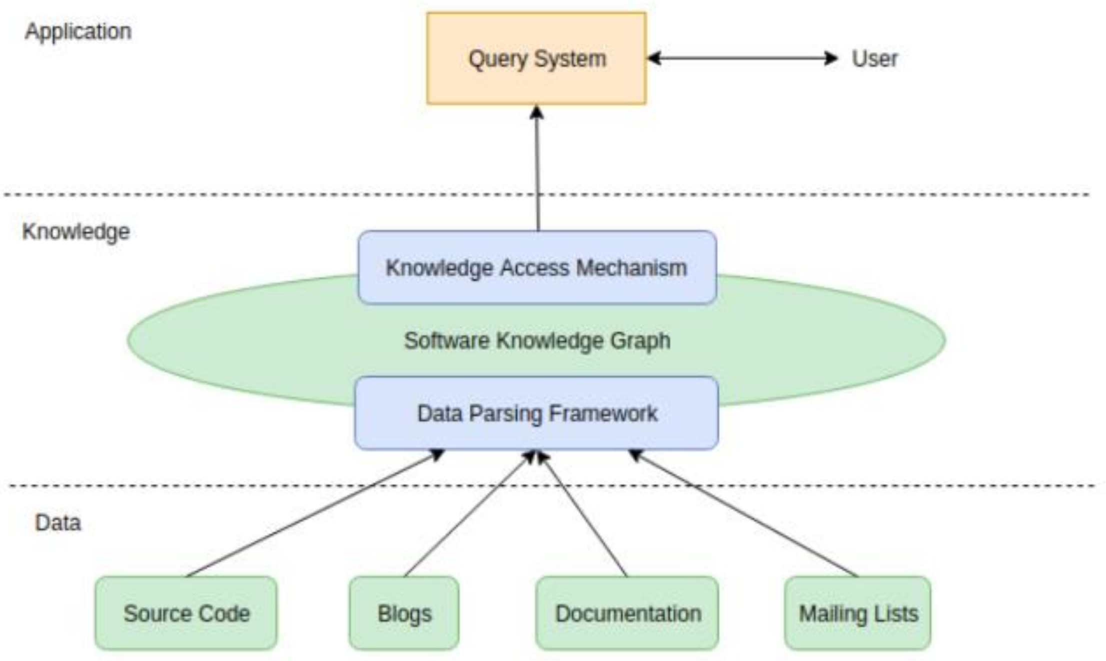
From the analysis of source code using comments and AST using LLVM, the tool retrieves a graph representation of correlation of entities of code, this graph is termed as Knowledge Graph.

Software knowledge graph is defined as a graph for representing relevant knowledge in software domains, projects, and systems. In a software knowledge graph, nodes represent software knowledge entities (e.g., classes, issue reports and business concepts), and directed edges represent various relationships between these entities (e.g., method invocation and traceability link). Different nodes and relationships have different properties to describe their inner features.

3. Query processing Engine –

This is the graphical user interface to the user of the tool to get the information about source code from user defined queries and few pre-defined queries like –

1. What are the global variables
2. Number of global variables
3. Number of functions/methods
4. Description about the function names, parameter list and return types.
5. Data structure and algorithms used in the specific block of codes
6. Data structures and algorithms followed throughout the code.



Chapter 3.

Methodology

As discussed in the previous chapters, this project is completely about extracting information from source code, analyzing codes and comments, finding correlation between them. As well as building a knowledge graph for the code bases. And once we have that we can build a query system that can answer various queries of the programmer about the source code. In this chapter we will focus on the complete methodology followed by this project.

3.1. The complete architecture of the project is as shown below –

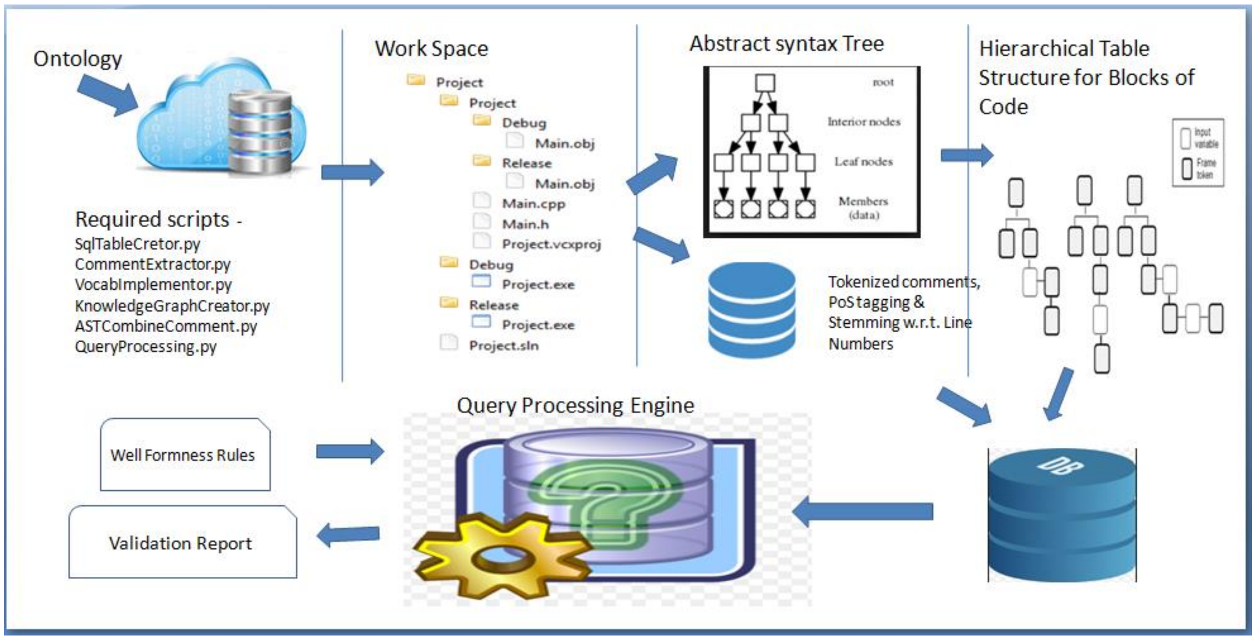


Fig 3.1. Architecture Design of Static Analysis of Source Code.

3.2. Natural Language Processing (NLP) and Ontology

Since the comments in the code is structured or unstructured information, which requires preprocessing for extracting meaning from the sentences. Natural Language Processing tool has many features to help in doing the extraction of information from the human sentences and to find correlation between them to meet the requirement as per need. The tools and their usage is as follows:

STEP 1:

The input to natural language processing will be a simple stream of Unicode characters (typically UTF-8). Basic processing will be required to convert this character stream into a sequence of lexical items (words, phrases, and syntactic markers) which can then be used to better understand the content.

The basics include:

Structure extraction – identifying fields and blocks of content based on tagging

- **Identify and mark sentence, phrase, and paragraph boundaries** – these markers are important when doing entity extraction and NLP since they serve as useful breaks within which analysis occurs.
- **Tokenization** – to divide up character streams into tokens which can be used for further processing and understanding. Tokens can be words, numbers, identifiers or punctuation (depending on the use case)

- Open source tokenizers include the Lucene analyzers and the Open NLP Tokenizer.

- Basis Technology offers a fully featured language identification and text analytics package which is often a good first step to any language processing software. It contains language identification, tokenization, sentence detection, lemmatization, decompounding, and noun phrase extraction.

- Search Technologies has many of these tools available, for English and some other languages, as part of our Natural Language Processing toolkit. NLP tools include tokenization, acronym normalization, lemmatization (English), sentence and phrase boundaries, entity extraction.

- **Lemmatization / Stemming** – reduces word variations to simpler forms that may help increase the coverage of NLP utilities.

- Lemmatization uses a language dictionary to perform an accurate reduction to root words. Lemmatization is strongly preferred to stemming if available. Search Technologies has lemmatization for English and our partner, Basis Technologies, has lemmatization for 60 languages.

- Stemming uses simple pattern matching to simply strip suffixes of tokens (e.g. remove "s", remove "ing", etc.). The Porter Stemmer library of NLTK provides stemming for English language.

STEP 2:

In corpus linguistics, part-of-speech tagging (POS tagging or PoS tagging or POST), also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition and its context—i.e., its relationship with adjacent and related words in a phrase, sentence, or paragraph. A simplified form of this is in the identification of words as nouns, verbs, adjectives, adverbs, etc.

Like a comment:

int Element; //Element represents Variation of the apartments

It will get following POS tagging, which is stored in the database to correlate the entities in future work.

```
[('Element','NN'),('represents','VB') , ('Variation', 'JJ'), ('of', 'IN'), ('the', 'AT'),
('apartments', 'NNS')]
```

3.3. Ontology -

Ontology is set of vocabulary for the categorization of all tokens of the comment. Since the category can then be used for analysis of the code, it is very important to have set of concrete categories. A word or set of words may require to categorize hence the vocabulary set is stored into **Dictionary Data structure of Python,** and after tokenization process, it implements **Trigram / Bigram / Unigram** match, for storing appropriate match of word(s) into database.

Upon observation of existing codes and their performance, the vocabulary set is hand crafted and stored in the database to categorize the underlying code into different element sets. Like following category and their respective vocabulary set is:

1. Algorithm - Dynamic Programming, Divide and Conquer, Branch and Bound, Searching, Sorting, Quick Sort , Merge Sort etc.
2. Data Structure – Array, Linked List, Tree, Trie, Hash Tables, Map, Set, list, vector, red black tree etc.
3. Visibility Scopes – global, local, external, auto, static, register, private, public, protected etc.
4. Concurrency – process, thread, parallel, data race, deadlock, livelock, synchronization, safety, message.
5. Memory – new, space, object, free, pointer, destroy, bytes, contiguous, leak, garbage, allocator, makes, assign, memory, contiguous, allocated, storage, allocation, initialize, declarations, usage, store.
6. Class – constructor, destructor, copy constructor, virtual, inheritance, polymorphism, instance, new, mutable, resolution, visible.

The tool contains facility to add more ontology while coming across the new keywords into required category, so as to have incremental ontology upon observation and testing phase of the project.

3.4. The LLVM and CLANG for AST

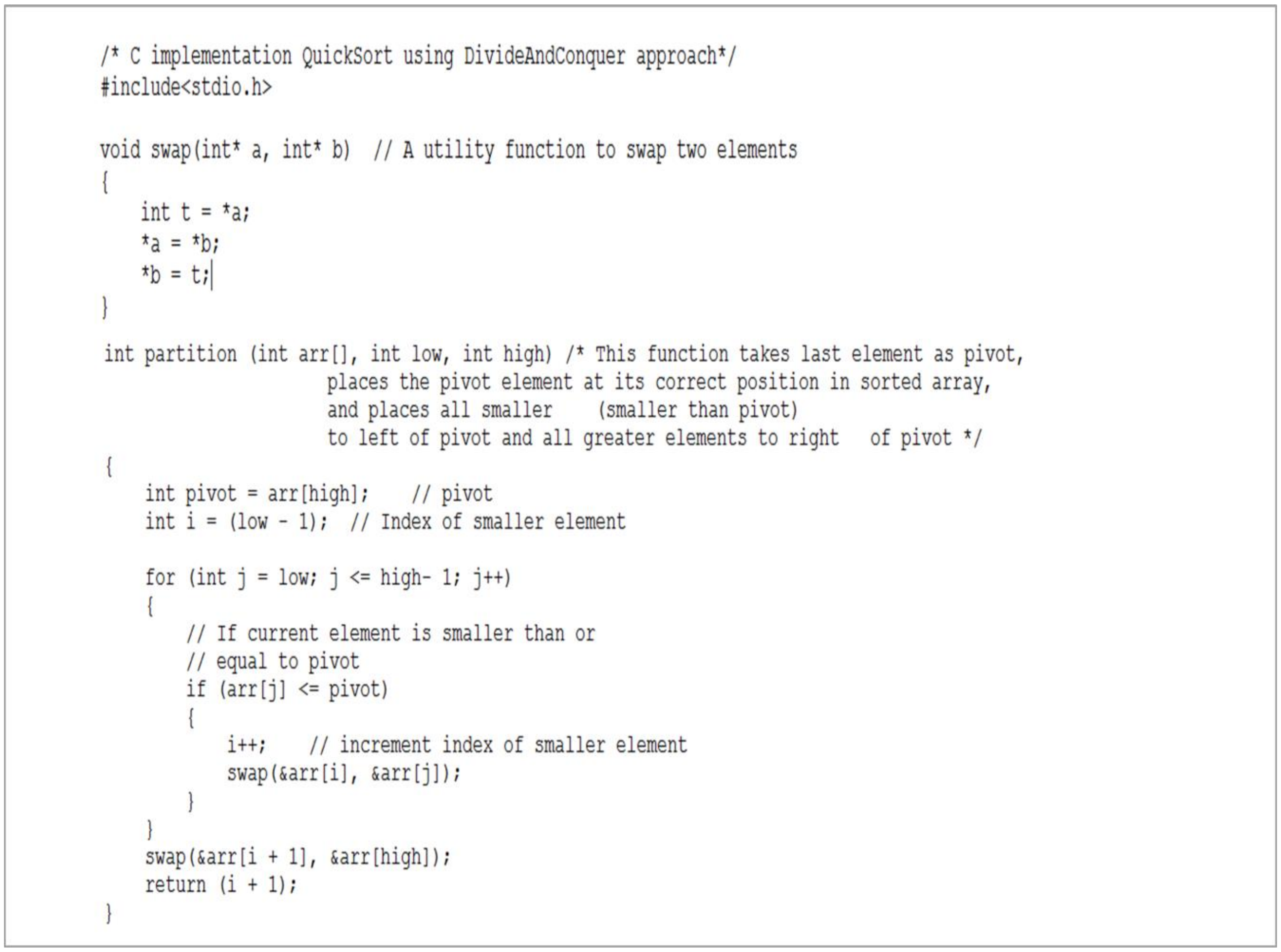
As discussed earlier, it is required to parse the abstract syntax tree in order to build the hierarchical table structure and syntax tree. The LLVM Project is collection of modular and reusable compiler and tool chain technology. The Clang compiler is an open-source compiler for the C family of programming, aiming to be the best in class implementation of these languages. Clang builds on the LLVM optimizer and code generator, allowing it to provide high-quality optimization and code generation support for many targets. We use Clang in order to parse the abstract syntax tree and convert it into annotated syntax tree.

Chapter 4.

Experimental Results.

Considering the following example for Experiment and running this through the model. The Results and Observations are as follows:

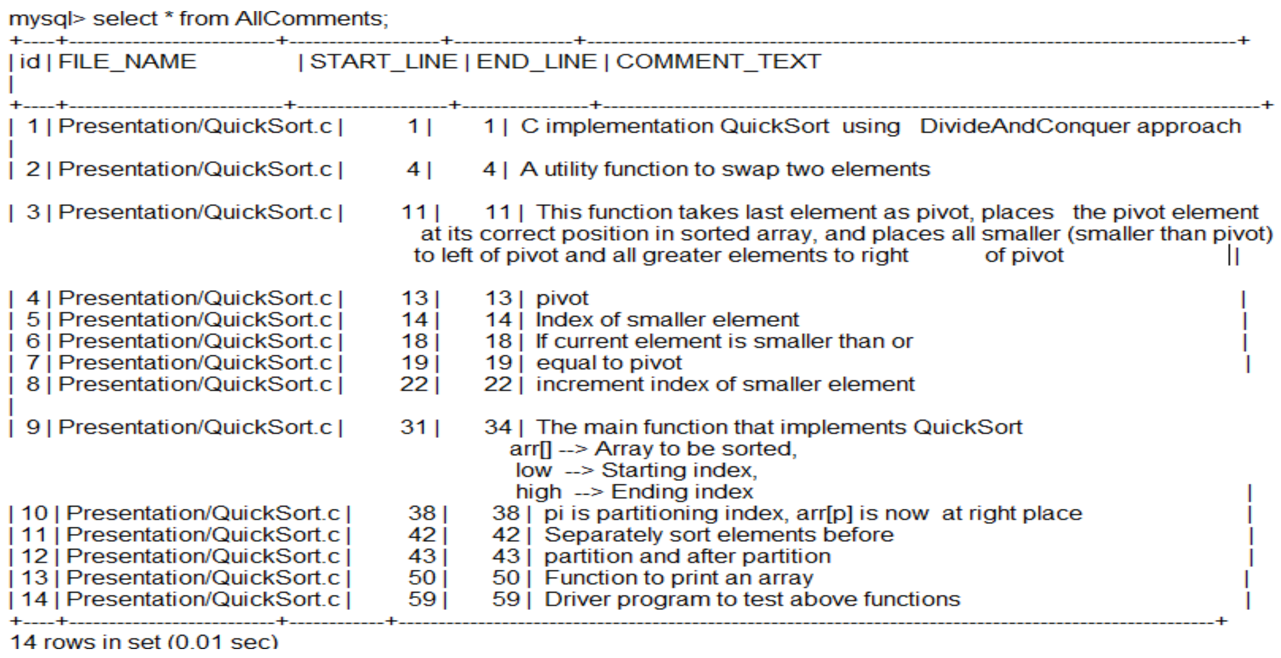
4.1. The C Code for QuickSort



```
/* C implementation QuickSort using DivideAndConquer approach*/
#include<stdio.h>

void swap(int* a, int* b)  // A utility function to swap two elements
{
    int t = *a;
    *a = *b;
    *b = t;
}

int partition (int arr[], int low, int high) /* This function takes last element as pivot,
                      places the pivot element at its correct position in sorted array,
                      and places all smaller    (smaller than pivot)
                      to left of pivot and all greater elements to right   of pivot */
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

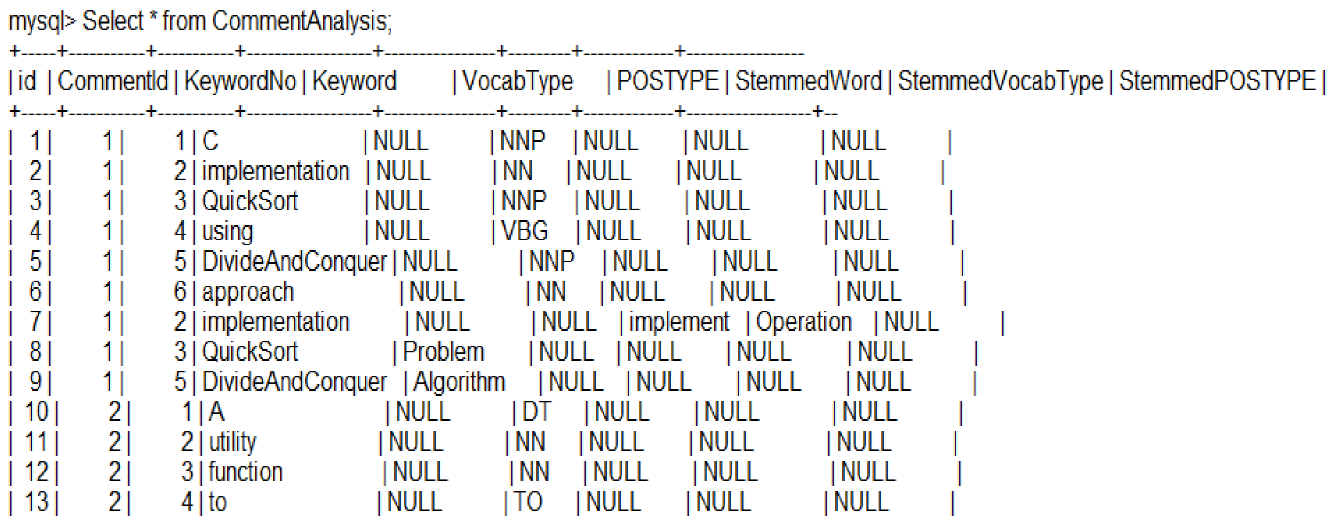
```
void quickSort(int arr[], int low, int high) /* The main function that implements QuickSort
                                                arr[] --> Array to be sorted,
                           low  --> Starting index,
                           high  --> Ending index */
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

  void printArray(int arr[], int size)   /* Function to print an array */
  {
      int i;
      for (i=0; i < size; i++)
         printf("%d ", arr[i]);
      printf("n");
  }


  int main()    // Driver program to test above functions
  {
      int arr[] = {10, 7, 8, 9, 1, 5};
      int n = sizeof(arr)/sizeof(arr[0]);
      quickSort(arr, 0, n-1);
      printf("Sorted array: n");
      printArray(arr, n);
      return 0;
  }
```

<u>The comment extraction script stores the information in the database as follows</u>



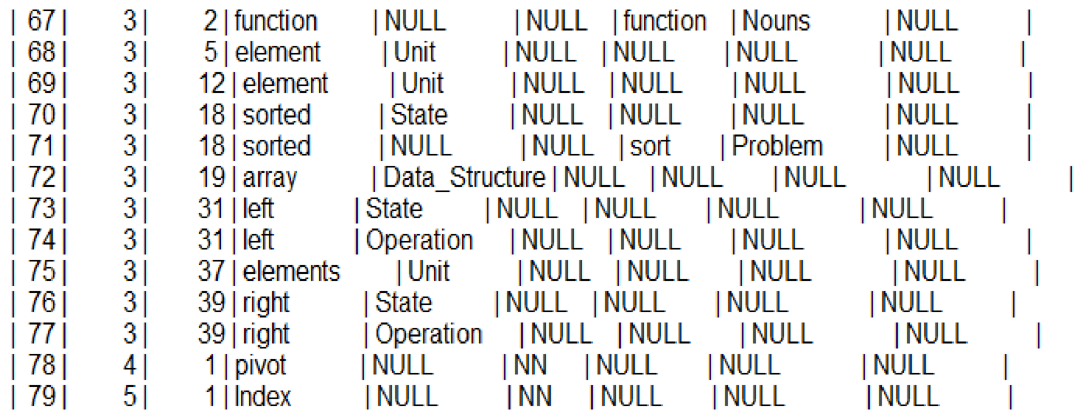
```
mysql> select * from AllComments;
+----+-------------------------+-----------------+---------------+-------------------------------------------------------------------------------------------+
| id | FILE_NAME               | START_LINE | END_LINE | COMMENT_TEXT
|
+----+-------------------------+-----------------+---------------+-------------------------------------------------------------------------------------------+
|  1 | Presentation/QuickSort.c |       1 |       1 |  C implementation QuickSort  using   DivideAndConquer approach
|
|  2 | Presentation/QuickSort.c |       4 |       4 |  A utility function to swap two elements

|  3 | Presentation/QuickSort.c |      11 |      11 |  This function takes last element as pivot, places   the pivot element
                                  at its correct position in sorted array, and places all smaller (smaller than pivot)
                                 to left of pivot and all greater elements to right           of pivot                  ||

|  4 | Presentation/QuickSort.c |      13 |      13 |  pivot                                                              |
|  5 | Presentation/QuickSort.c |      14 |      14 |  Index of smaller element                                           |
|  6 | Presentation/QuickSort.c |      18 |      18 |  If current element is smaller than or                              |
|  7 | Presentation/QuickSort.c |      19 |      19 |  equal to pivot                                                     |
|  8 | Presentation/QuickSort.c |      22 |      22 |  increment index of smaller element
|
|  9 | Presentation/QuickSort.c |      31 |      34 |  The main function that implements QuickSort
                                          arr[] --> Array to be sorted,
                                           low  --> Starting index,
                                           high  --> Ending index                                     |
| 10 | Presentation/QuickSort.c |      38 |      38 |  pi is partitioning index, arr[p] is now  at right place      |
| 11 | Presentation/QuickSort.c |      42 |      42 |  Separately sort elements before                               |
| 12 | Presentation/QuickSort.c |      43 |      43 |  partition and after partition                                |
| 13 | Presentation/QuickSort.c |      50 |      50 |  Function to print an array                                   |
| 14 | Presentation/QuickSort.c |      59 |      59 |  Driver program to test above functions                       |
+----+-------------------------+-----------+-----------------------------------------------------------------------------------------+
14 rows in set (0.01 sec)
```

<u>A portion of Database showing Tokens, their corresponding VocabType, PosType, StemmedWord and Its VocabType.</u>



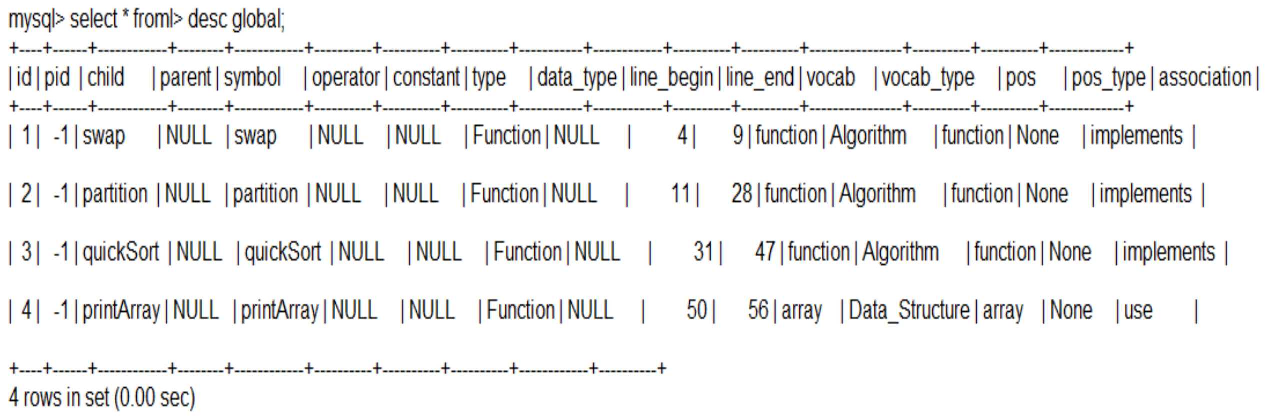
```
mysql> Select * from CommentAnalysis;
+-----+-----------+-----------+-----------------+---------------+---------+------------+----------------
| id  | CommentId | KeywordNo | Keyword         | VocabType     | POSTYPE | StemmedWord | StemmedVocabType | StemmedPOSTYPE |
+-----+-----------+-----------+-----------------+---------------+---------+------------+-----------------+--
|  1 |       1 |        1 | C               | NULL       | NNP   | NULL       | NULL         | NULL        |
|  2 |       1 |        2 | implementation  | NULL       | NN    | NULL       | NULL         | NULL        |
|  3 |       1 |        3 | QuickSort       | NULL       | NNP   | NULL       | NULL         | NULL        |
|  4 |       1 |        4 | using           | NULL       | VBG   | NULL       | NULL         | NULL        |
|  5 |       1 |        5 | DivideAndConquer | NULL      | NNP   | NULL       | NULL         | NULL        |
|  6 |       1 |        6 | approach        | NULL       | NN    | NULL       | NULL         | NULL        |
|  7 |       1 |        2 | implementation  | NULL       | NULL  | implement  | Operation    | NULL        |
|  8 |       1 |        3 | QuickSort       | Problem    | NULL  | NULL       | NULL         | NULL        |
|  9 |       1 |        5 | DivideAndConquer | Algorithm | NULL  | NULL       | NULL         | NULL        |
| 10 |       2 |        1 | A               | NULL       | DT    | NULL       | NULL         | NULL        |
| 11 |       2 |        2 | utility         | NULL       | NN    | NULL       | NULL         | NULL        |
| 12 |       2 |        3 | function        | NULL       | NN    | NULL       | NULL         | NULL        |
| 13 |       2 |        4 | to              | NULL       | TO    | NULL       | NULL         | NULL        |
```

Continued…….



```
| 67 |     3 |      2 | function      | NULL           | NULL   | function    | Nouns        | NULL      |
| 68 |     3 |      5 | element       | Unit           | NULL   | NULL        | NULL         | NULL      |
| 69 |     3 |     12 | element       | Unit           | NULL   | NULL        | NULL         | NULL      |
| 70 |     3 |     18 | sorted        | State          | NULL   | NULL        | NULL         | NULL      |
| 71 |     3 |     18 | sorted        | NULL           | NULL   | sort        | Problem      | NULL      |
| 72 |     3 |     19 | array         | Data_Structure | NULL   | NULL        | NULL         | NULL      |
| 73 |     3 |     31 | left          | State          | NULL   | NULL        | NULL         | NULL      |
| 74 |     3 |     31 | left          | Operation      | NULL   | NULL        | NULL         | NULL      |
| 75 |     3 |     37 | elements      | Unit           | NULL   | NULL        | NULL         | NULL      |
| 76 |     3 |     39 | right         | State          | NULL   | NULL        | NULL         | NULL      |
| 77 |     3 |     39 | right         | Operation      | NULL   | NULL        | NULL         | NULL      |
| 78 |     4 |      1 | pivot         | NULL           | NN     | NULL        | NULL         | NULL      |
| 79 |     5 |      1 | Index         | NULL           | NN     | NULL        | NULL         | NULL      |
```

The association mapping of information extracted from Comment Analysis and AST



```
mysql> select * froml> desc global;
+----+------+-----------+--------+-----------+---------+---------+---------+----------+-----------+---------+---------+---------------+---------+---------+------------+
| id | pid | child     | parent | symbol    | operator | constant | type    | data_type | line_begin | line_end | vocab    | vocab_type     | pos      | pos_type | association |
+----+------+-----------+--------+-----------+---------+---------+---------+----------+-----------+---------+---------+---------------+---------+---------+------------+
| 1 |  -1 | swap      | NULL  | swap      | NULL  | NULL  | Function | NULL  |     4 |     9 | function | Algorithm      | function | None  | implements |

| 2 |  -1 | partition | NULL  | partition | NULL  | NULL  | Function | NULL  |    11 |    28 | function | Algorithm      | function | None  | implements |

| 3 |  -1 | quickSort | NULL  | quickSort | NULL  | NULL  | Function | NULL  |    31 |    47 | function | Algorithm      | function | None  | implements |

| 4 |  -1 | printArray | NULL | printArray | NULL | NULL  | Function | NULL  |    50 |    56 | array    | Data_Structure | array    | None  | use        |

+----+------+-----------+--------+-----------+---------+---------+---------+-----------+---------+
4 rows in set (0.00 sec)
```

Following Testing Queries have been implemented and their respective output on QuickSort.C code is as follows:

Few Predefined Queries executed on the information extracted from source Code of QuickSort.C

```
$ python QueryProssingFromInformation.py

What is the Problem Statement ?

-> C implementation QuickSort using DivideAndConquer approach


Which Data Structure is identified?
 -> array

Which Access Scopes are identified ?
-> local

Which Algorithms are identified to be followed by the code?
-> DivideAndConquer, swap

Which Associations are mapped using information from Comment and AST?
-> implements and use

MappedWithFunction     |     Keyword          |     Identified Type of Keyword |     Association
swap                         function               Algorithm                          implements
partition                    function               Algorithm                          implements
quickSort                    function               Algorithm                          implements
printArray                    array                 Data_Structure                      use



 What are the variable association found?
        Variable                                     Comment Association
 ->       low                                      low   --> Starting index,
         high                                       high   --> Ending index
```

Chapter 5.

Conclusion and Future Work

In the current semester, the static analysis and comment analysis based on C programmed language has been implemented, which determines the Vocabulary Type, identifies data structures, algorithms, scopes, operations performed. Also it finds association of AST with the comments extracted and annotated, so as find more meaningful information from every block of code. Since comments are in natural language, the current implementation also does Part Of Speech(POS) tagging to extract and correlate the information for documentation of code. The knowledge graph that has been built would have annotated syntax tree, also the comment analysis of code and the query system. In the next semester this work shall be extended to C++ programming language where new graph has to be implemented to take care of all classes, member functions, data members, and all others OOPS features.

Till now only the static analysis of code has been done, this work will be extended for dynamic analysis of code with following goals –

- Using Dynamic Analysis for extracting Flow of the code execution using PIN – tool.
- Using predefined codes, to extract the features and generating model from machine learning algorithms (Neural Network).
- Executing few test cases on the given project, and extracting its features, which will be supplied to prepared model to understand its behaviour.
- Implementing Query Processing Engine to answer some static and user defined queries about the source code.

Bibliography

1. HotComments: How to Make Program Comments More Useful?
   Lin Tan, Ding Yuan and Yuanyuan Zhou, Department of Computer Science, University of Illinois at Urbana-Champaign, {lintan2, dyuan3, yyzhou@cs.uiuc.edu

2. Towards Supporting On Demand Virtual Remodularization Using Program Graphs, David Shepherd, Lori Pollock, and K. Vijay Shanker Computer and Information Sciences University of Delaware Newark, Delaware 19816 shepherd, pollock, vijay@cis.udel.edu

3. /* iComment: Bugs or Bad Comments? */Lin Tan†, Ding Yuan†, Gopal Krishna†, and Yuanyuan Zhou†‡ †University of Illinois at Urbana-Champaign, Urbana, Illinois, USA ‡CleanMake Co., Urbana, Illinois, USA {lintan2, dyuan3, gkrishn2, yzhou}@cs.uiuc.edu

4. Introducing Natural Language Program Analysis Lori Pollock, K. Vijay-Shanker, David Shepherd, Emily Hill, Zachary P. Fry, Kishen Maloor Computer and Information Sciences University of Delaware Newark, Delaware 19716 {pollock, vijay, shepherd, hill, fryz, maloor}@cis.udel.edu

5. NATURAL LANGUAGE IN SOFTWARE ENGINEERING, Analysing source code: looking for useful verb–direct object pairs in all the right places Z.P. Fry, D. Shepherd, E. Hill, L. Pollock and K. Vijay-Shanker

6. Automatic Quality Assessment of Source Code Comments: The JavadocMiner Ninus Khamis, Ren´e Witte, and Juergen Rilling Department of Computer Science and Software Engineering Concordia University, Montr´eal, Canada.

7. 18th IEEE International Conference on Program Comprehension, Natural Language Parsing of Program Element Names for Concept Extraction, Surafel Lemma Abebe and Paolo Tonella, Software Engineering research unit, Fondazione Bruno Kessler, Trento, Italy, fsurafel,tonellag@fbk.eu.

8. Improving Identifier Informativeness Using Part of Speech Information, Dave Binkley Loyola University Maryland, Matthew Hearn Loyola University Maryland, Dawn Lawrie, Loyola University Maryland.

9. Quality Analysis of Source Code Comments, Daniela Steidl Benjamin Hummel Elmar Juergens, CQSE GmbH, Garching b. M¨unchen, Germany, fsteidl, hummel, juergensg@cqse.eu.

10. CloCom: Mining Existing Source Code for, Automatic Comment Generation, Edmund Wong, Taiyue Liu, and Lin Tan, Department of Electrical and Computer Engineering University of Waterloo, Waterloo, Ontario, Canada{e32wong, t67liu, lintan} @ uwaterloo.ca

11. Recommending Insightful Comments for Source, Code using Crowdsourced Knowledge, Mohammad Masudur Rahman, Chanchal K. Roy, Iman Keivanloo.

12. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt - Everton da S. Maldonado, Emad Shihab, Member, IEEE, and Nikolaos Tsantalis, Member, IEEE.

13. Natural language is a programming language-Applying natural language processing to software development- University of Washington Computer Science & Engineering, Seattle, WA, USA - mernst@cs.washington.edu.

14. http://searchwindevelopment.techtarget.com/definition/static-analysis

15. http://www.infosun.fim.uni-passau.de/publications/docs/FoSE16.pdf

16. http://ieeexplore.ieee.org/document/7589781/authors

17. http://pi.informatik.uni-siegen.de/stt/32_2/01_Fachgruppenberichte/SRE_TAV/02-tiarks.pdf

18. https://link.springer.com/article/10.1007/s10270-012-0270-z