Software Technical Document

C Path Finder – An automated Test Case Generator for C

Sajed Jalil 29 September, 2018 Software Project

C Path Finder – An automated Input generator for C

BSSE 8th Semester Project

Submitted By

……………………………………………………………….

Sajed Jalil

BSSE 0714
Institute of Information Technology (IIT)
University of Dhaka

Supervised By

……………………………………………………………….

Dr. B M Mainul Hossain

Associate Professor
Institute of Information Technology (IIT)
University of Dhaka

Date: 29 September 2018

Table of Contents

Table of Figures

Introduction

In our day to day life, programmers spend time thinking about the correctness of their code. A significant amount of this time is dedicated to the testing of the written source code. Although, this might not include complete testing process, but at least it includes **debugging**. This process is messy when the source code size becomes too large. Programmers have to think to understand which line is executed by which condition. That can make the software development process complicated. Automated Testing tools can be the solution to this problem.

Purpose

Testing is an active field now a days. In our modern software development, Automated tools are being used widely. This **C Path Finder**, is an automated test case generator tool which will ease the development process of software written in C language.

This document aims to specify the requirements and goals of developing C Path Finder. It will explain the features, interfaces, capabilities and constraints for this software. It will also explain the requirements, scenario model, data model and project planning.

The main purpose of this document is to bridge the gap between actual requirements and implemented requirements. Without requirement analysis, some key features and goals will be missed and cannot be addressed properly in time. This document will act as a guideline for development of C Path Finder.

Scope

This document will only address the requirements that are understood at the beginning of the development. If some requirements are changed during the development process this document will not cover the changed information. In that case, a final document with implementation details will be devised later.

This document is solely based on the idea of **research papers**. Therefore, there will be some trial and error techniques. From many ways of implantation of the idea, measures will be taken to select the optimal one. So, it is much likely that only the core structure of the proposed software will be covered here.

Product Overview

C Path Finder is an automated test case generator tool. In this section, detailed descriptions of the product will be given. This includes product perspective, functionalities, constraints, assumptions, internal and external dependencies etc. it will also describe what type of users we are targeting and what are the functionalities available for them.

Product Perspective

The intended system is **Desktop** based. It has only one instance i.e. in the personal workstation. It will be used to generate test cases for C source codes. As the product is desktop based there will be no need for **authentication** as this software cannot be used remotely. The product will be developed in **JAVA SE 10**.

This product works with large volume of data. So, the product is mainly **Data Centric**. For this we need to store data in a specific database that suits the need. The database will also be hosted in the device in which the software exits. There will be no cloud communications.

Symbolic Execution[1] technique will be used as a main idea for this software. This technique is used now a days for static testing. This is a branch of white box testing.

Product Functionalities

The functionalities can be stated to describe the software product more deliberately. In the following the main functionalities of the software are provided with brief description.

1. Automatic Test Case Generation from Source Code

A user can provide C source codes to automatic test case generation. The software will take the source code as input. It will process the source code with Symbolic Execution, SMT solver and will find out the possible test cases for the input source code.

2. Efficient Execution Feature

Generation of test cases from static analysis is a costly technique. Often, we change a part of source code and generate test case again. For this we have to calculated everything from beginning. For this, a technique will be used named memorized execution[2]. In this technique, everything that was calculated previously will be stored. Only the changed part in the source code will be calculated again. This can save time when the test source files are large in size.

3. 100% Test Coverage Through Unit Testing

All possible branch coverage test case generation is infeasible for large source codes[3]. The reason is that the possible paths increase exponentially.

I have tried to tackle the problem through unit testing technique. In this way, all possible paths will be generated based on each method. Calls for other methods will not be handled here. Each Method's runtime will not exceed **1 second** for text case generation. Otherwise, it will be terminated.

Product Constraints, Assumptions and Dependencies

The software is developed to handle C codes only. No other source codes can be given as an input. Besides, the **accuracy** of the test case generation technique is solely dependent upon the **Symbolic Execution** and **SMT solver**. To our knowledge, there is no established way of applying Symbolic execution for **array**, **loops**, **file input-output operations** and **external library calls**.

For this software, we are assuming that the user provides valid source codes as input. Valid means the source code compiles without errors and warnings in the **GCC**. Wrongly provided source code may never bring the desired results.

Quality Function Deployment (QFD)

Specifying QFD is necessary to understand the product goals and commitments. QFD is generally divided into three parts. The requirement specifications are provided in below sections.

Normal Requirements

These are the requirements that are generally expected by the customers / users. They are stated below –

1. Generating test cases from input source files
2. Installer and uninstaller

Expected Requirements

These are expectations from the customers. They are generally secondary focus. They are stated in the following –

1. Finding the spots of code where test cases cannot be generated for ambiguity.
2. Selecting input files interactively
3. Selecting destination folder interactively

Exciting Requirements

Exciting requirements are not stated nor expected by customers. Our exciting features are given below –

1. Implementation of memorized execution technique to bring down execution time
2. Can run in a dual core processor

Scenario Based Model

Scenario based modelling is the first phase where the usage of product can be visualized. This model enables us to get a vivid idea how user will use the product. In the following, we describe how the user story and use case.

User Story

A user can download the installer of **C Path Finder** from internet. Then he/she will install the software in desired location of the computer. After the installation is done, user can open the software and can select the C source files to be tested by the software. In this case, a popup box will appear from where user can select the files from the local hard drives.

After the file selection, user can use three options –

1. **Get Unreachable Statements**
2. **Test Generator** – Generating test cases for the given source code

After the result is calculated, they will be output to a file. The destination output file can be changed by the user. But by default, it will be in a predefined folder by the software. After that, user can test source code again or close the application.

The user can even uninstall the software through clicking the uninstall icon.

Use Case Diagrams

In the following, a use case model has been developed to understand the scenario more clearly.

Level 0: System View

Here, an overall system view is given:

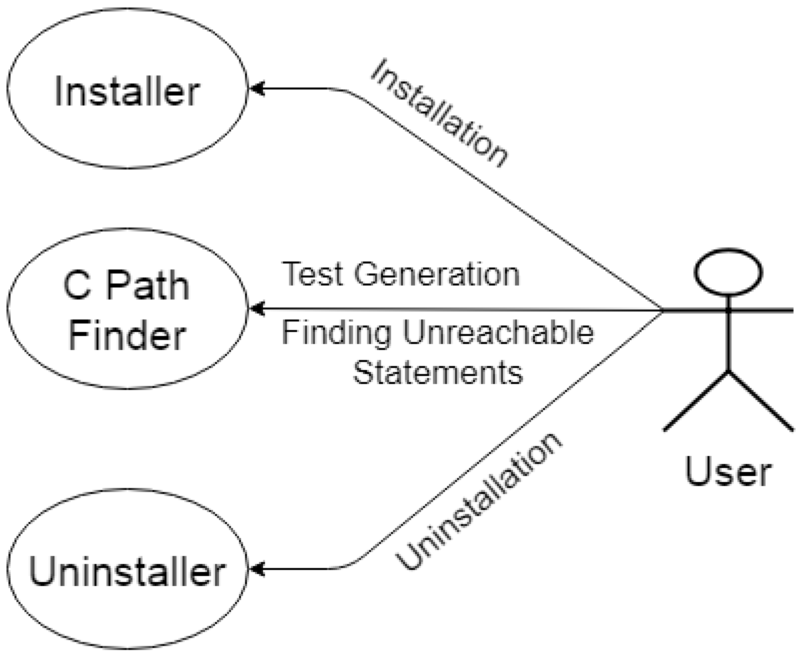


Figure 1: Use Case 0 - Overview

Actor: User

Functionalities:

1. Installation
2. Test Case generation
3. Finding unreachable code statements
4. Uninstallation

Level 1.1: Installation

The following describes the installation module.

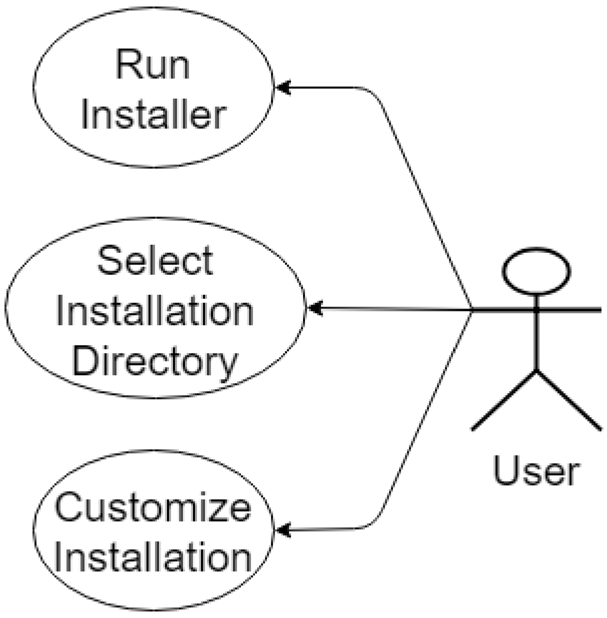


Figure 2: Use case 1.1 - Installation

Level 1.2: C Path Finder

The following contains the basic part of the test generator module. The detail will be given in the activity diagram in the later part.

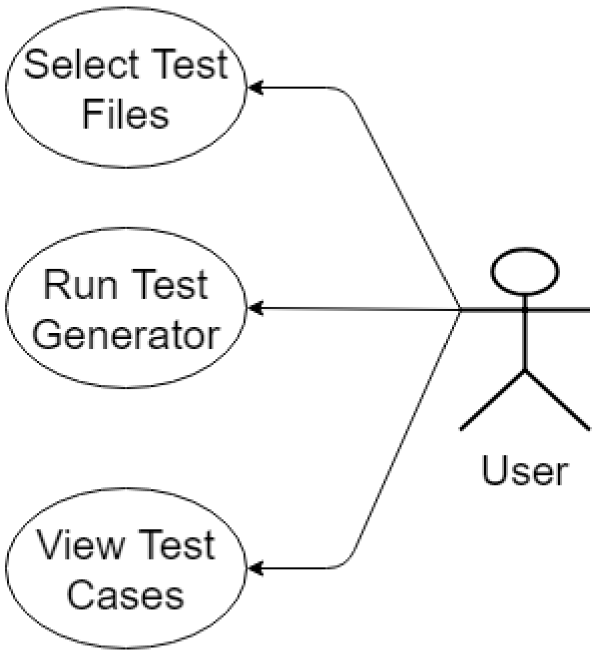


Figure 3: Use Case 1.2 – C Path Finder

Level 1.3: Uninstaller

This part contains the uninstallation process. A user can use uninstaller executable for uninstalling **C PATH FINDER**. There will be options to keep the current database that was used for processing.

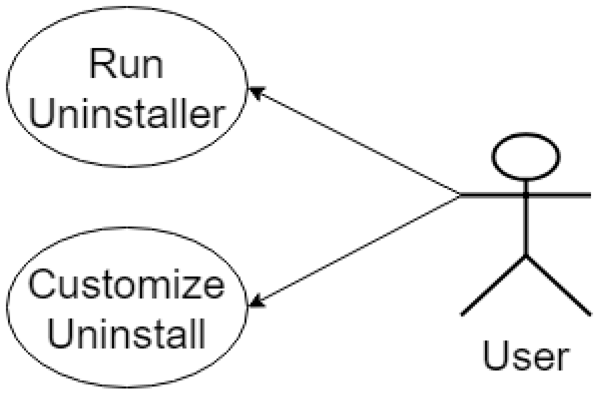


Figure 4: Use Case 1.3 - Uninstaller

Activity Diagram

Activity diagram shows the control flow of execution. In the below working steps of test generator are shown:

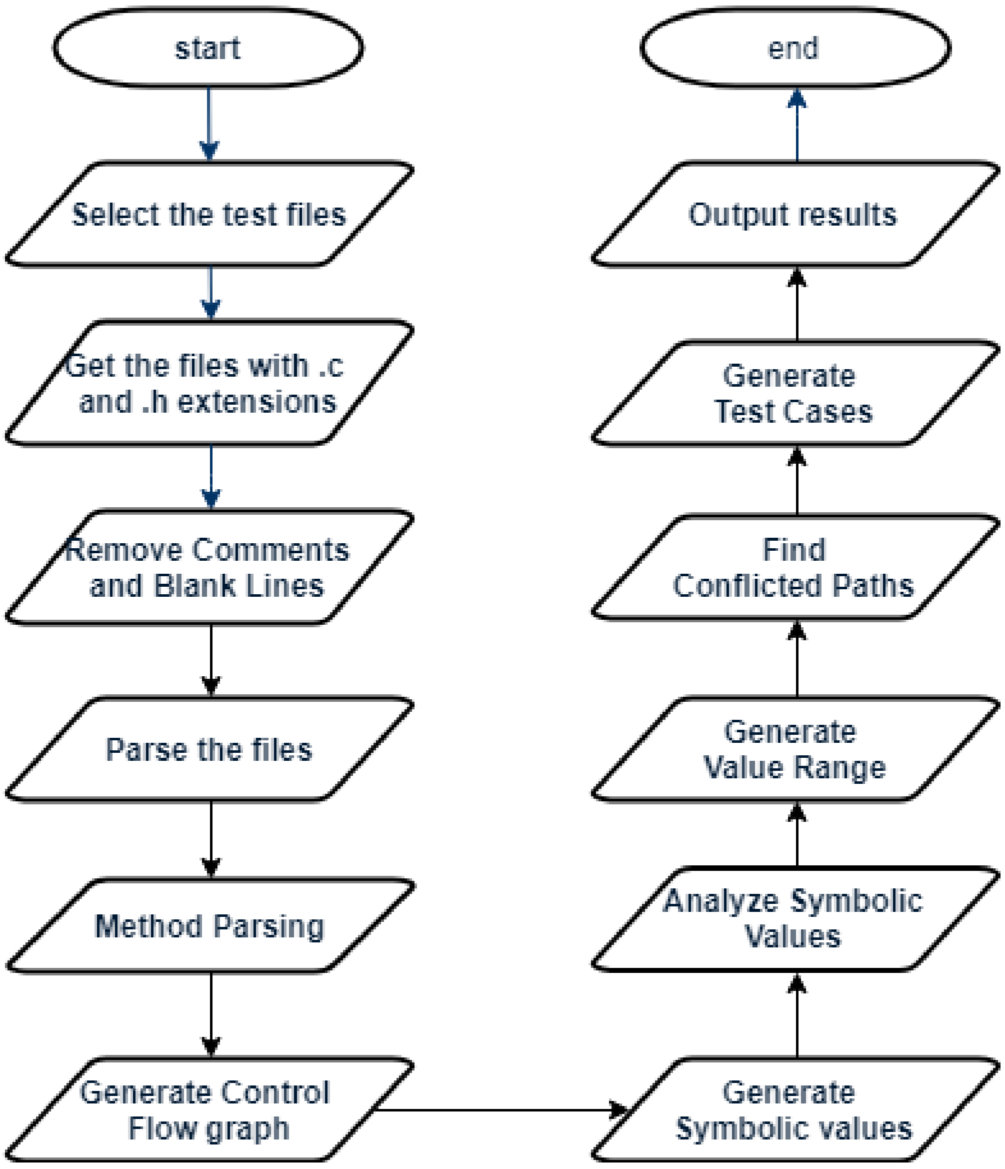


Figure 5: Activity Diagram of Test Generator

Data Based Model

Data is the key element in a software. The main task of a software is to take data as input and produce information. In the following sections, discussion will be made regarding data objects and their relationships.

Data Objects

C Path Finder takes C source code files as input. The calculations are done on per method based. Calculated results are to be stored for **memorized execution**. Besides, a record has to be preserved about which source file or method was changed after the execution. Based on these assumptions, we define the following data objects.

These are the data objects that are understood up to now for this software:

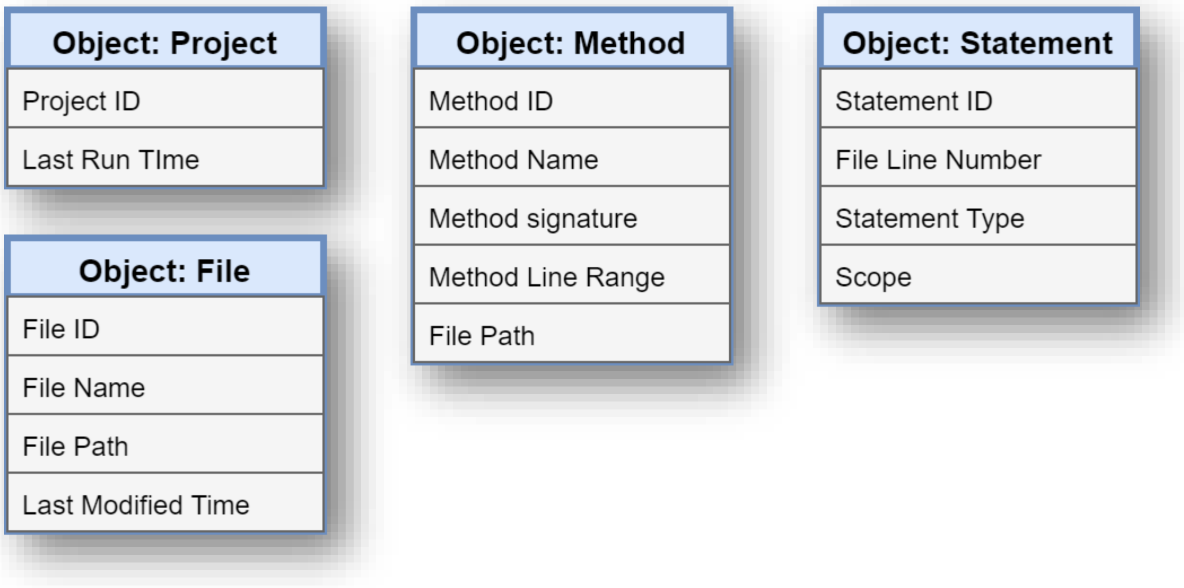


Figure 6: Data Objects

A brief description of these data object are provided in the next page.

Object 1 – Project

We will assign a **unique ID** to a project. This is for distinguishing one project from another throughout the database. ID will be **six digits** long containing **numerals [0-9]** only. Moreover, **last run time** of the project through the software will be stored.

Object 2 – File

A test source will consist of one or more files. Each file will have a **unique ID** across a its project. The ID will consist of **eight-digit numerals** [0-9]. We will also store **file name**, **file path**. Last modified time will be used in **memorized execution**. If the last modified time of a file is after the last run time of project then we can conclude that the file has been modified and it needs to be recalculated.

Object 3 – Method

Method ID will be used to uniquely identify a method in a specific file. The ID will consist of **six-digit numerals** [0-9]. Other attributes include – **method name**, **method signature**, **file path**, **method line range** (the starting and ending line number of the method in that file).

Object 4 – Statement

Statement will have an ID containing **six-digit** numerals [0-9]. Statement may occur outside the method also. Like as **global declarations** which are not part of any specific methods. This will be defined by **scope** (global/method). So, our considerations for statements includes – **file line number**, **statement type** (condition / declaration / assignment).

ER Diagram

In the following, the Entity Relationship Diagram is given below. They are constructed on the basis of data objects and their relationships.

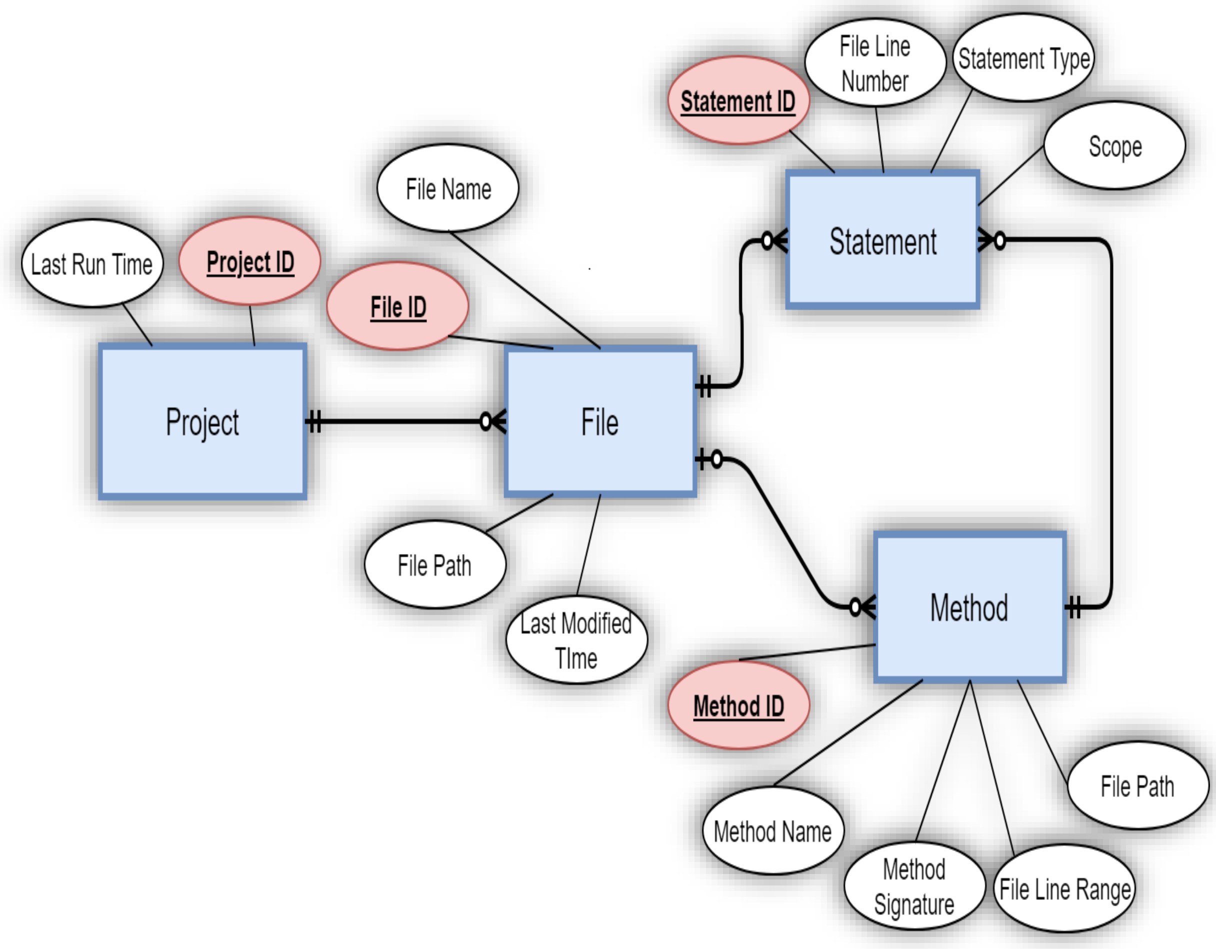


Figure 7: ER Diagram

Here, a useful insight is that a statement can be a part of both method and file. Global declarations are part of a file and local declarations are part of a specific method.

Class Based Model

Class based modelling are necessary part of requirement analysis. It provides us a brief idea about the classes required in object-oriented paradigm. We have to consider **Coad** and **Yourdon's** following **six** selection criteria choosing our classes –

1. Retained Information
2. Needed Services
3. Multiple Attributes
4. Common Attributes
5. Common Operations
6. Essential Requirements

Selected Packages

After considering the above-mentioned criteria we conclude to introduce the following packages in our system –

1. **Installer** – Classes related installation of the system
2. **Startup** – Responsible for starting and controlling the execution of the program
3. **Memorizer** – Will be responsible for memorized execution
4. **Database** – All database operations will be handled with this package
5. **File Loader** – Responsible for all types of input output operations
6. **Input Code Beautifier** – Beautifies the input source code for easier parsing
7. **Parser** – Responsible for parsing the input files
8. **Control Flow Graph maker** – This package will build the control flow graph for each method
9. **Symbol Assigner** – Will assign symbols required for Symbolic execution
10. **Symbolic Solver** – Core part of the project. Generates test result
11. **Uninstaller** – Responsible for Uninstalling the project

Details of Package Classes

As the project is research-paper based, **the detail class diagram will be provided later**. To our knowledge, there is no existing open implementation details of this type of software yet.

There will be trial and error during implementation time. That means, implementation structure is flexible and also decision have been taken to follow incremental process model. So unlike waterfall model, deep insight about class structure will not be given in this document. All the class packages have been identified. Their implementation details are **black boxed** now.

Architectural Design

Every software has a definite architecture. System Architecture represents the skeleton of the software. A well-developed architectural design helps us to refine and define the overall system with ease.

Representing System in Context

The foremost step of architectural design is to represent the system in context. This shows the interactions with the external elements. The external elements are –

1. **Super Ordinate Systems** – Systems that use C Path Finder – **None**
2. **Peers** – Systems that interact with C Path Finder on a peer-to-peer basis - **None**
3. **Users** – Entities that produces or consumes C Path Finder – **Desktop User**
4. **Sub-ordinate Systems** – Systems that C path Finder use to function and produce data – **Database, JRE**

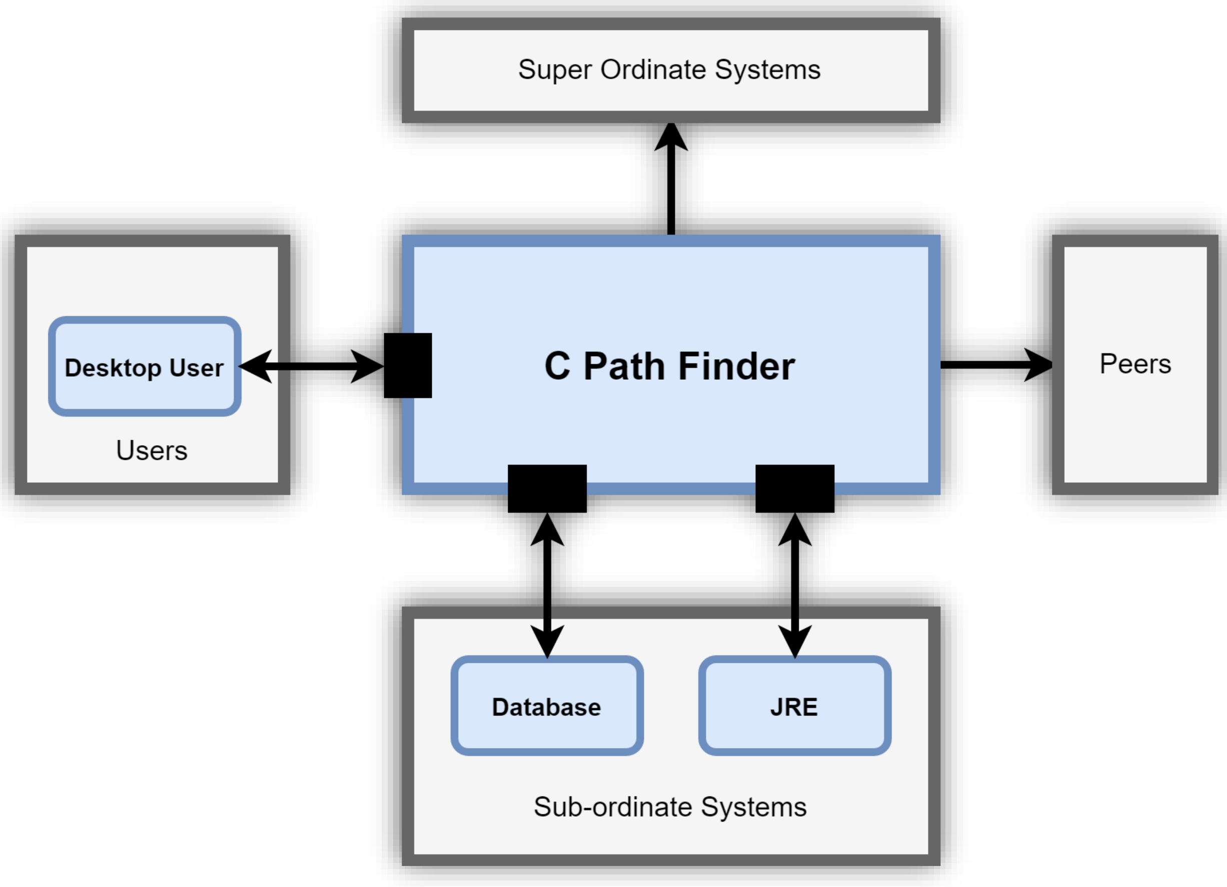


Figure 8: Representing C Path Finder in context

Defining Archetypes

Archetype is a class or pattern that helps to understand the core structure of the software. This portion is critical to design. From the class-based modelling, we have defined the following archetypes for C Path Finder –

1. **Memorization Controller** – Responsible for memorization techniques
2. **Source Code Initializer** – Responsible for filtering and parsing C codes
3. **Symbolic Solver** – Executes Symbolic execution technique

Refining Architecture into Components

We need to modularize the architecture into components to better understand the system. The high-level components derived are given below –

1. Installation
2. Memorizer
3. Source Initializer
4. Symbolic Solver
5. Graphical User Interface

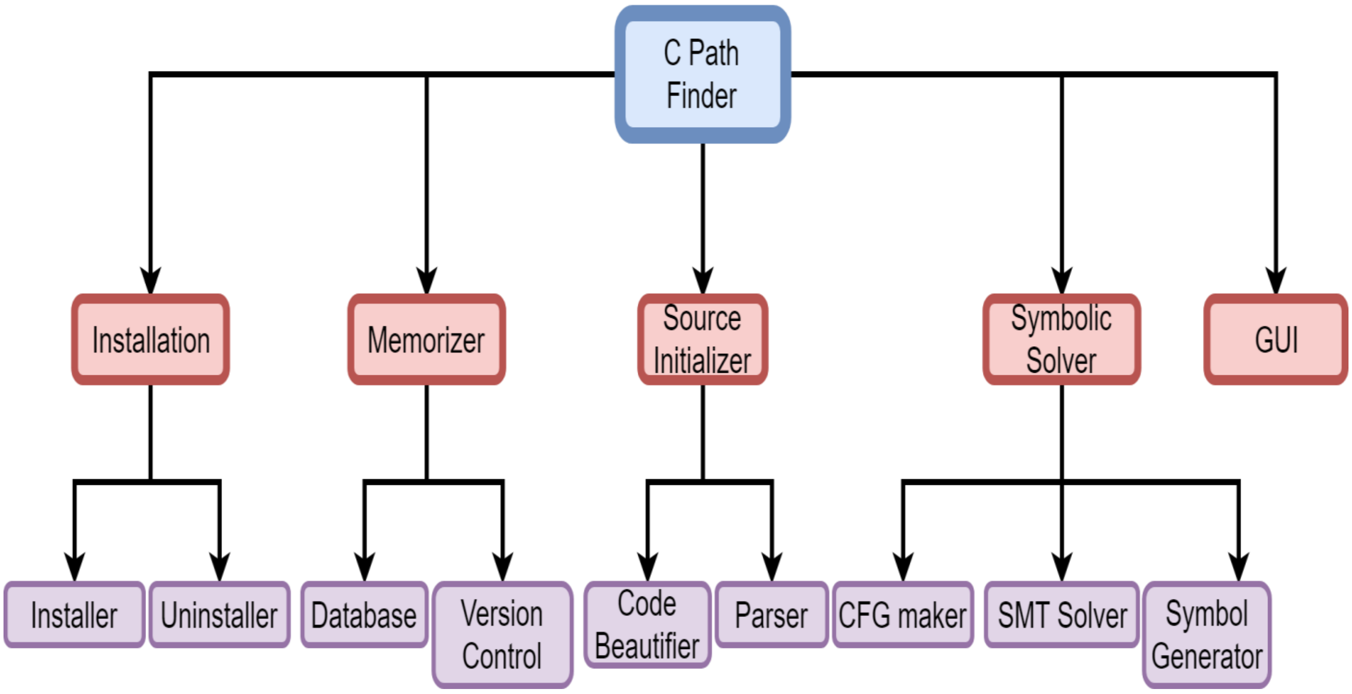


Figure 9: High Level Components

User Interface Design

User Interface is an important part of C Path Finder. The aim of the GUI is to make the testing process easy. A brief overview is provided below –

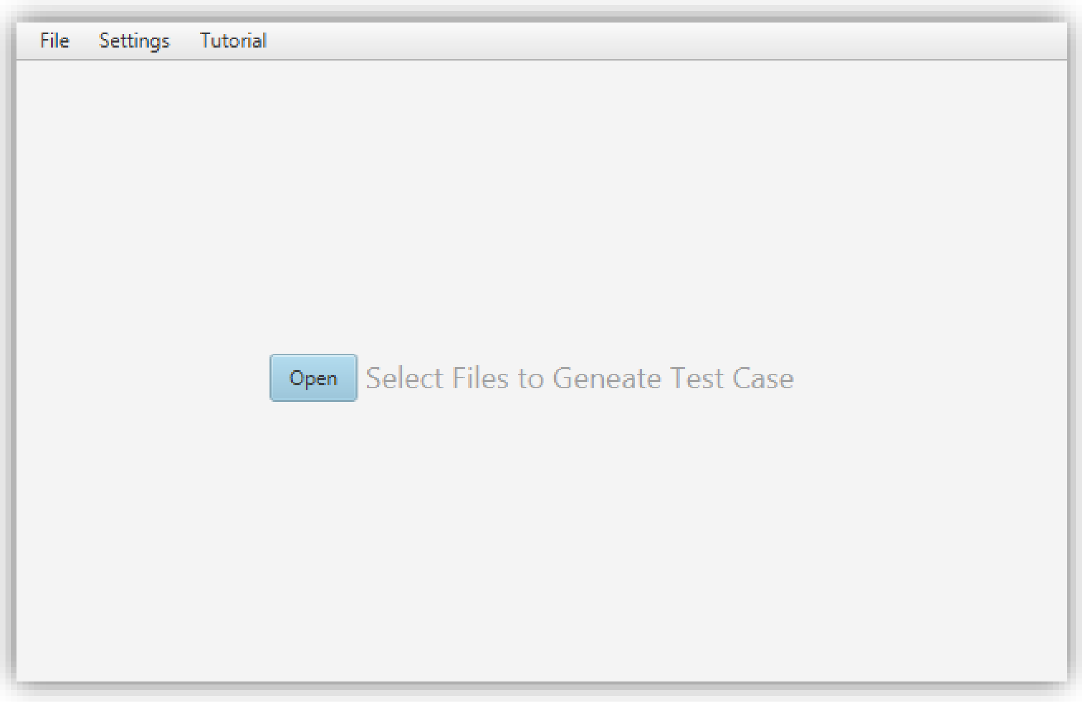


Figure 10: Opening a Test Folder

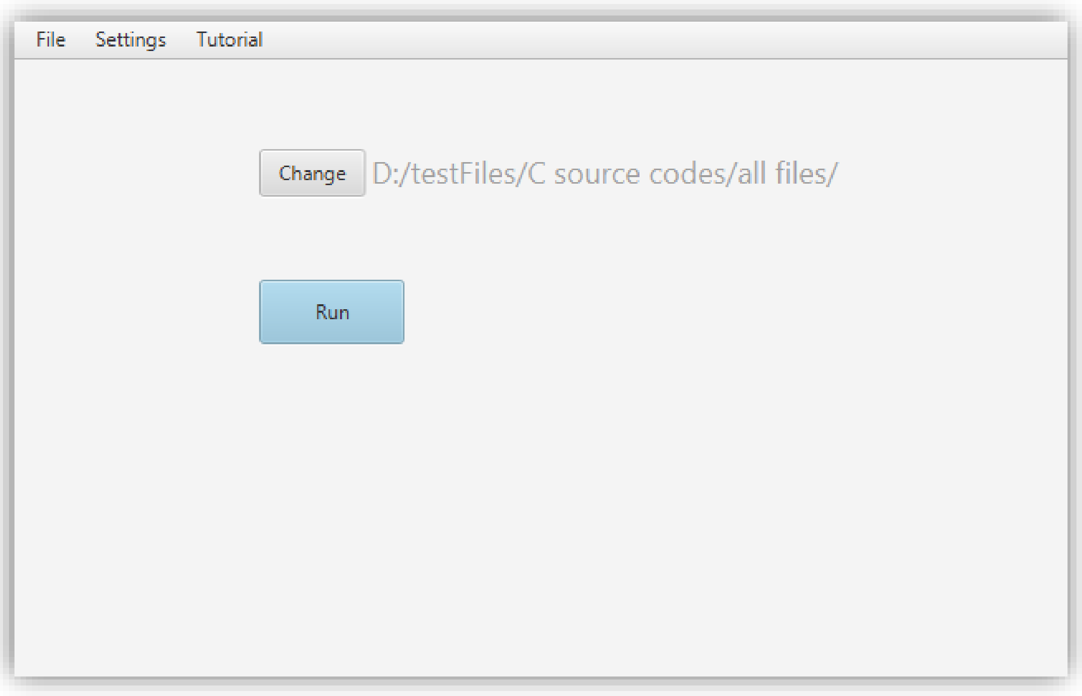
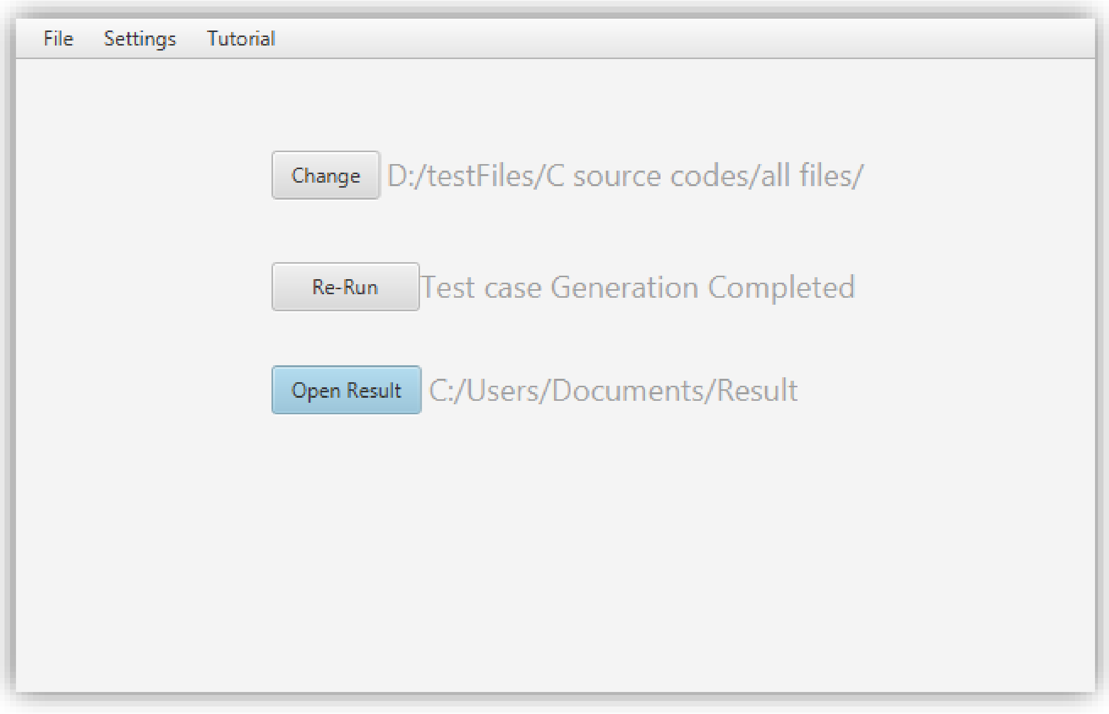


Figure 11: Running Selected Files



D:/testFiles/C source codes/all files/

Test case Generation Completed

C:/Users/Documents/Result

Figure 12: Opening Generated Results

In the above figures, the overall process for using the software is defined with steps. At first, the user needs to choose the test file. After choosing, user can change the files or run test generator. After test generation, user can re-run the generator or view the generated test cases.

Besides, there are other options in the menu bar for additional settings like – changing result directory, viewing tutorial on how to use.

Planning

Planning is an important aspect of software engineering. It is necessary to undertake a well-developed plan for a successful project. This include project model for software and also the timespan for the various activities. This section will describe the detail planning of the project.

Process model

This software project will be developed using **Incremental Process model.** This project is mainly based on some research paper techniques. There are many sets of features that will be delivered in each increment. There will be total four increments in this project. Their deliverables are described below:

1. **Increment 1 –** Test case generator, unreachable code detector
2. **Increment 2 –** Memorized test case generation
3. **Increment 3 –** GUI, Probable bug fixes
4. **Increment 4 –** GUI improvements, Installer, Uninstaller

Projected Timeline

In the following table, a timeline is shown about the distribution of the project work.

Increment 1 (July 20 – September 30)

- SRS
- Design
- Implementation – test case generator, unreachable code detector
- Testing

Increment 2 (October 1 – October 20)

- Modified SRS (If necessary)
- Modified Design (If necessary)
- Implementation – Memorized Test Generator, Previous Bug Fixes
- Testing – Integration Testing

Increment 3 (October 21 – November 7)

- Implementation – GUI, Previous Bug Fixes
- Testing – Integration Testing

Increment 4 (November 8 – November 25)

- Implementation – GUI improvements, Installer and Uninstaller
- Testing – Integration Testing, System Testing

Conclusion

Developers will face problems during development. There is no end of it. But the amount of difficulties can be brought down to a comfortable level through automated development. Testing tool is an automation of such kind. The popularity of testing tools in software development is increasing fast.

C Path Finder is a testing tool based on a novel Memorized Symbolic Execution technique. With this software, testing for C will be easier than ever before.

References

[1] Cadar, Cristian, et al. "Symbolic execution for software testing in practice: preliminary assessment." Software Engineering (ICSE), 2011 33rd International Conference on. IEEE, 2011.

[2] Yang, Guowei, Corina S. Păsăreanu, and Sarfraz Khurshid. "Memoized symbolic execution." Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM, 2012.

[3] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." OSDI. Vol. 8. 2008.

Appendix

CFG	Control Flow Graph
GCC	GNU Compiler Collection
GUI	Graphical User Interface
Java SE 10	JAVA Standard Edition 10
JRE	Java Runtime Environment
QFD	Quality Function Deployment
SMT	Satisfiability Modulo Theory
SRS	Software Requirements Specification