Software Documentation

C Path Finder – An Automated Test Case Generator for C

Sajed Jalil

12/14/18

Software Project

C Path Finder – An Automated Test Case 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: 14 December 2018

Letter of Transmittal

December 14, 2018

BSSE 4th Year Exam Committee

Institute of Information Technology

University of Dhaka

Dear Sir,

With due respect I would like to state that I have prepared a detailed report on my 8th semester software project named **C Path Finder**. This report includes the details of basic software engineering process.

The primary purpose of this report is to summarize the procedure of the development process and present a full user manual of the developed software.

Sincerely yours,

Sajed Jalil
BSSE 0714

Institute of Information Technology (IIT)
University of Dhaka

Document Authentication

This is to be certify that this project document has been verified and authorized by the following personnel.

Submitted By	Supervised By
Sajed Jalil	**Dr. B M Mainul Hossain**
BSSE 0714	Associate Professor
Institute of Information Technology (IIT)	Institute of Information Technology (IIT)
University of Dhaka	University of Dhaka

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, implementation details, test plans and lastly, User manual.

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. Initial part of this document will act as a guideline for development of C Path Finder. Later part will contain the actual implementation details elaborately. Moreover, test plans and user manuals are provided at the end of the document.

Scope

Requirement and Design section of 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 part will not cover the changed information. Later part will contain the actual implementation details and test plan that were uncovered during actual implementation.

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 **10 seconds** 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 **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:

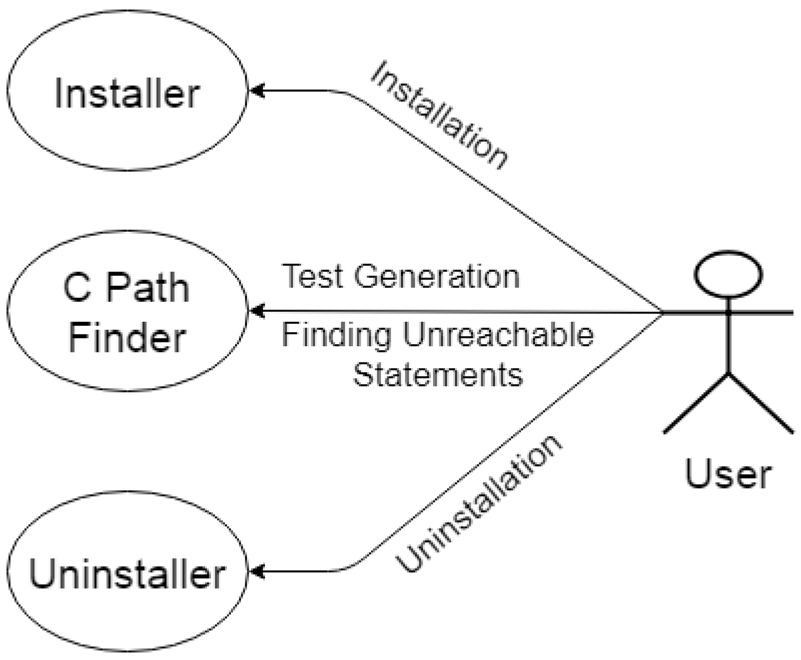


Figure 1: Use Case 0 - Overview

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.

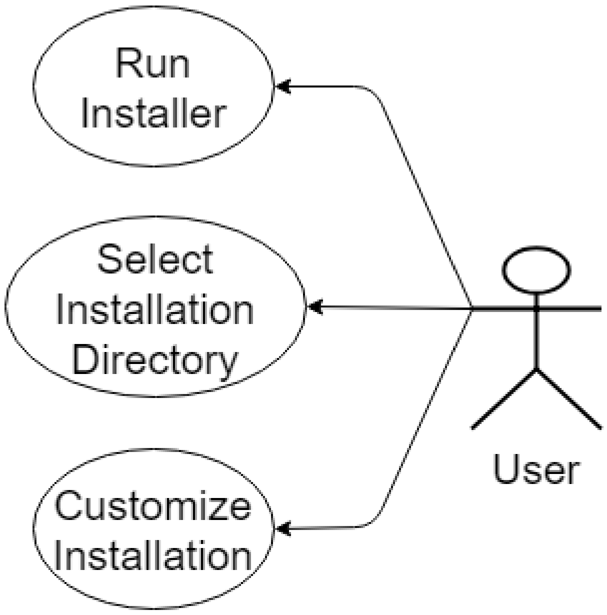


Figure 2: Use case 1.1 - Installation

Level 1.2: C Path Finder

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

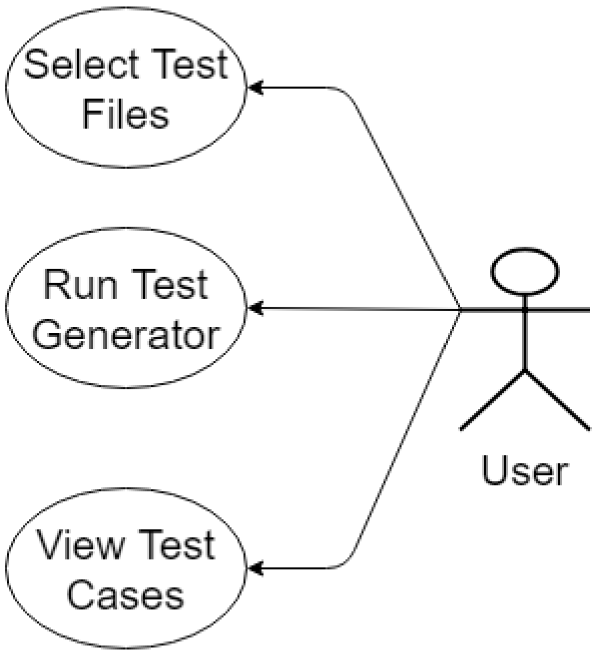


Figure 3: Use Case 1.2 – C Path Finder

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.

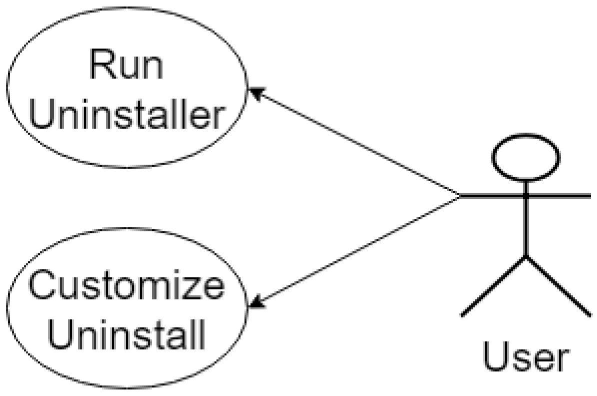


Figure 4: Use Case 1.3 - Uninstaller

Activity Diagram

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

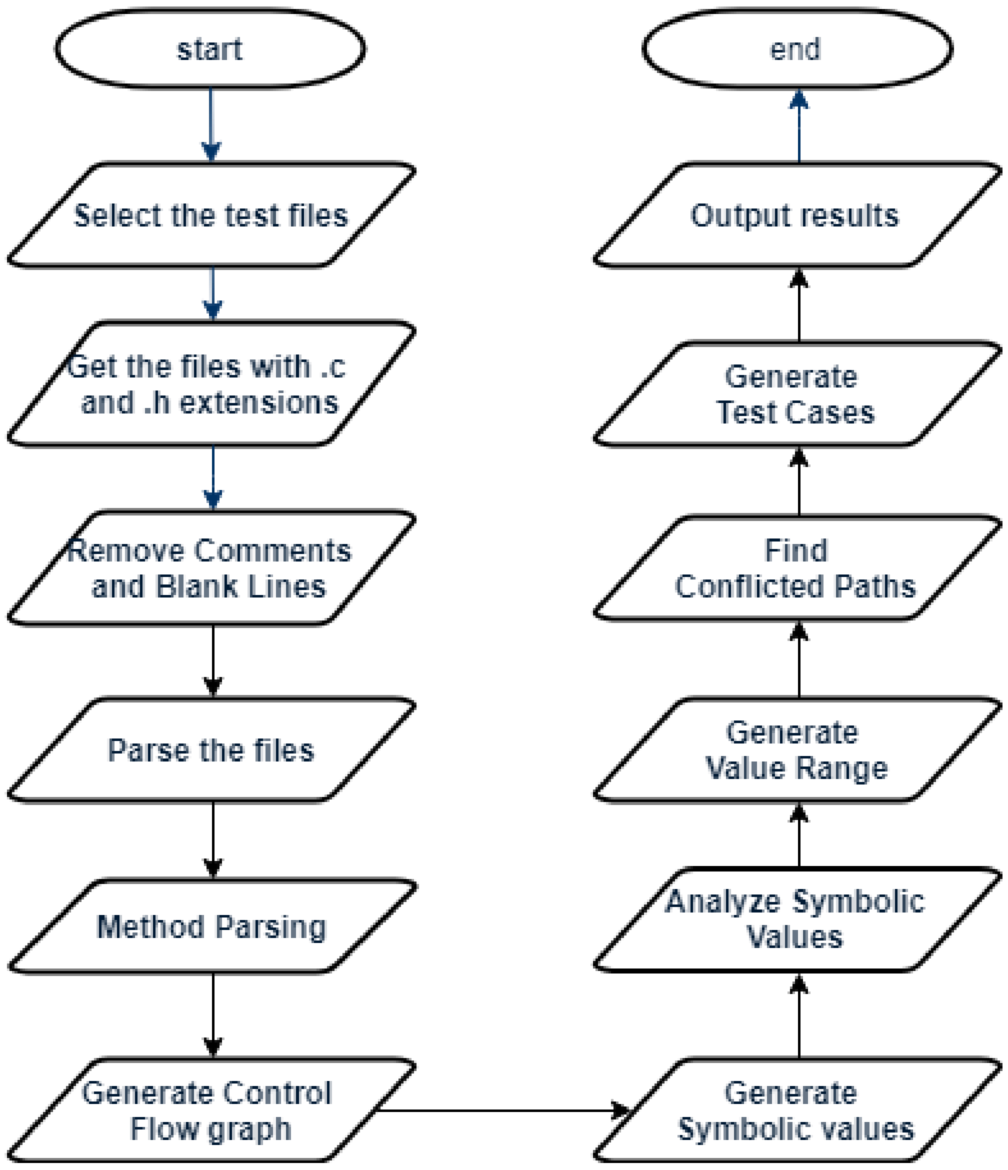


Figure 5: Activity Diagram of Test Generator

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 understood up to now for this software:

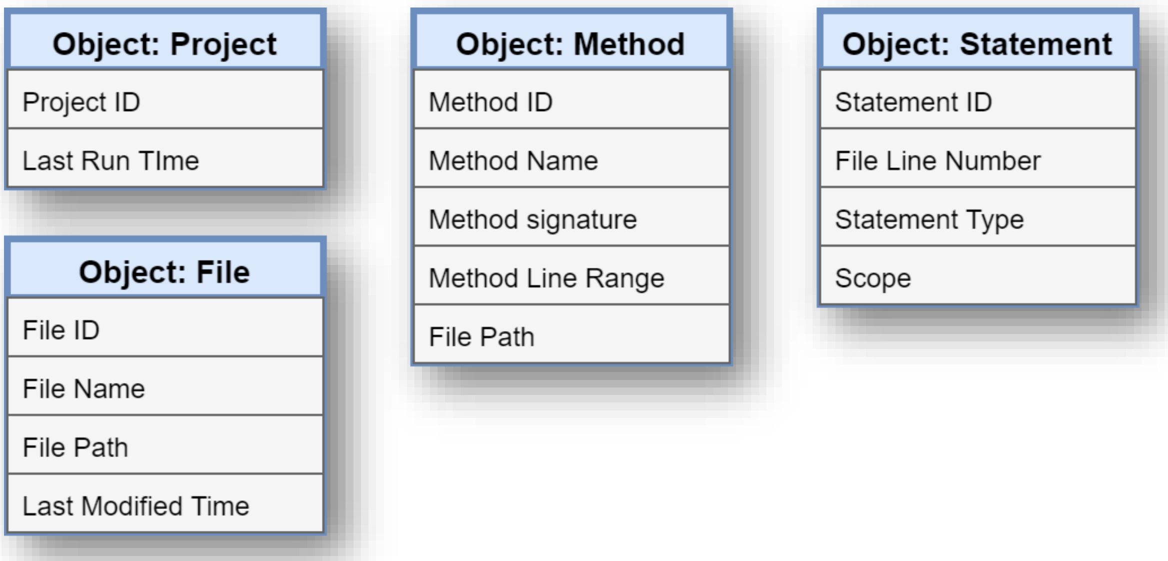


Figure 6: Data Objects

A brief description of these data object is 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.

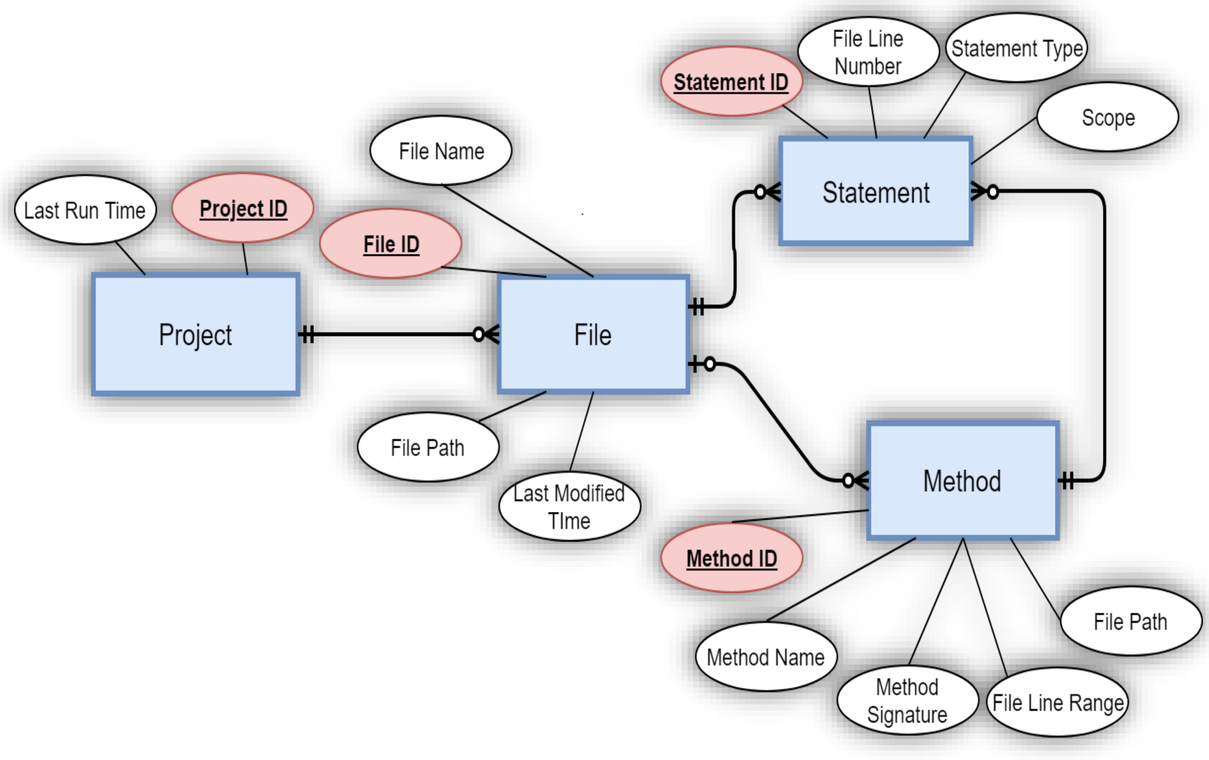


Figure 7: ER Diagram

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, z3 Solver.**

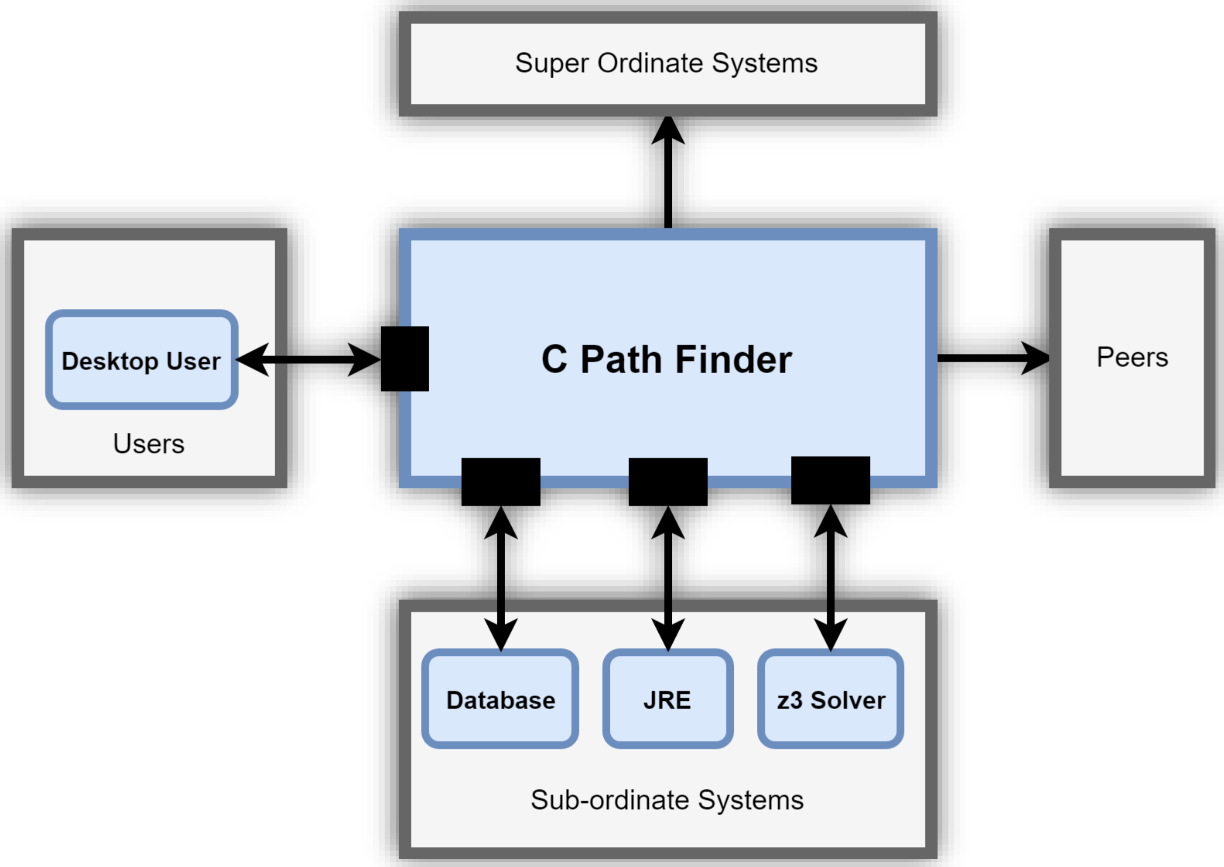


Figure 8: Representing C Path Finder in context

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

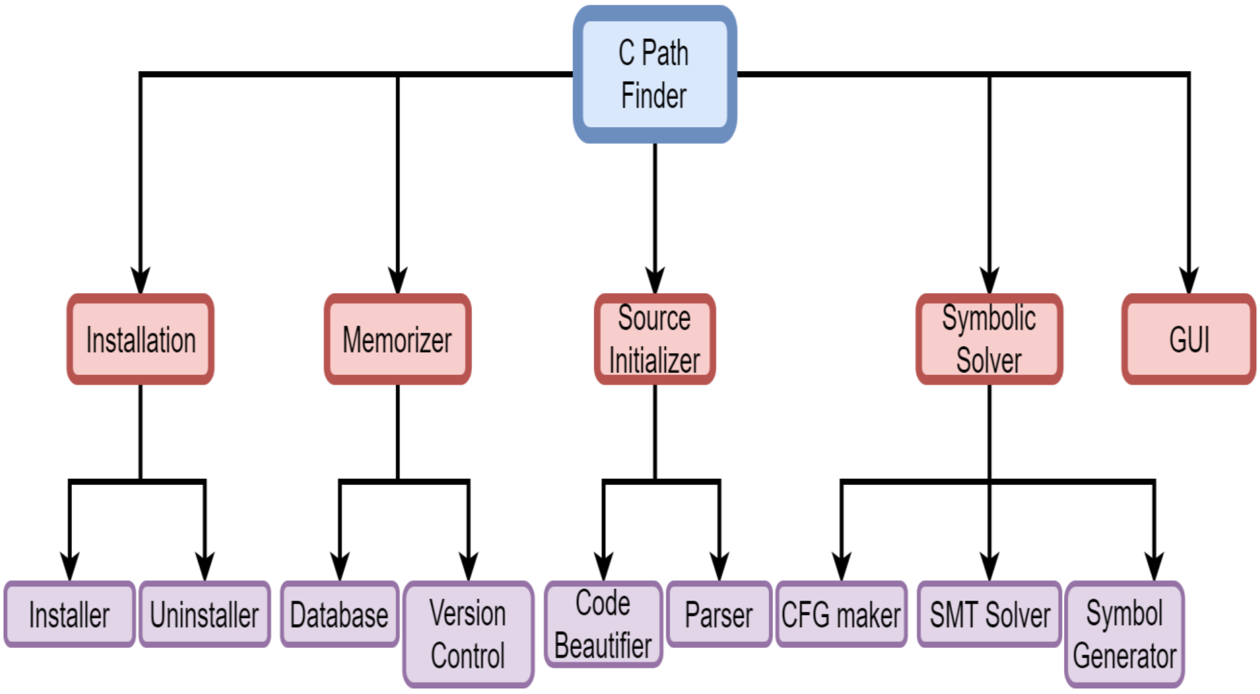


Figure 9: High Level Components

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 –

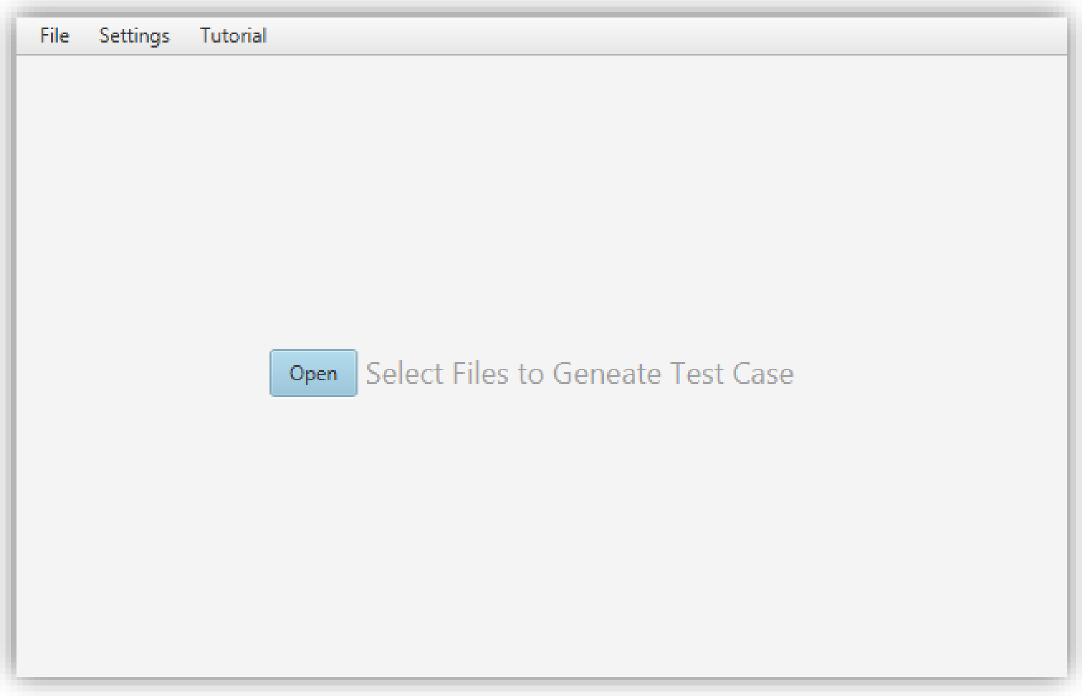


Figure 10: Opening a Test Folder

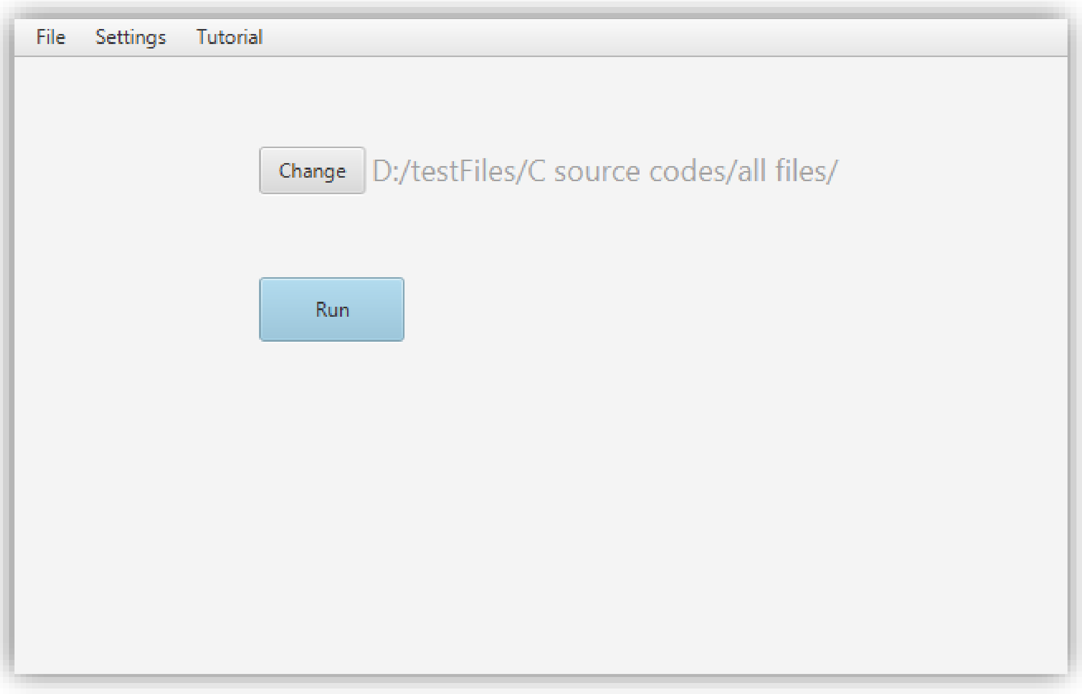
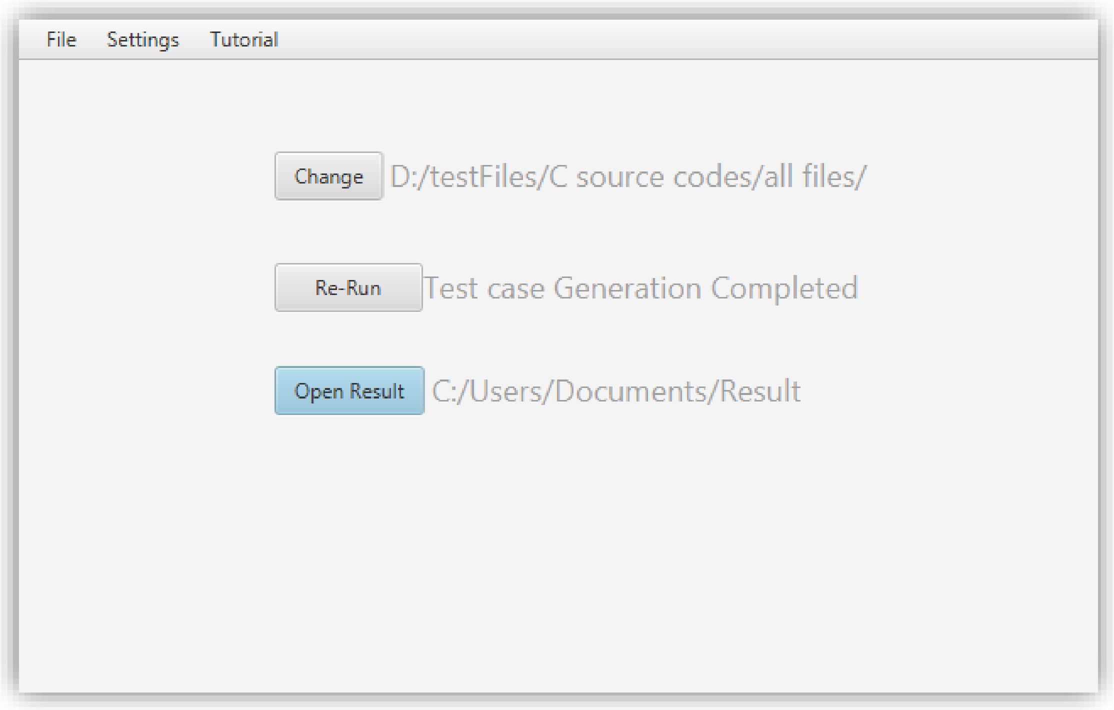


Figure 11: Running Selected Files



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

Implementation Detail

The core goal of this project was to build a software that is usable by the targeted group of users. The SRS and design document done before is a base guideline for the Implementation of this project. In the following sections, implementation details will be presented along with the challenges and its overcoming strategies.

Implementation Plan

Software implementation is a tedious task. Whenever the project is new to the developer, the task becomes more complicated. Implementation of testing tool with Symbolic Solver was completely new to me. Before this project, I did not have sufficient idea about the process of its actual implementation. To my knowledge, there is no Symbolic solver tool that implements memorized execution technique. Therefore, a structured plan was necessary to successfully complete this project.

Implementation plan includes many aspects of development. It mostly depends on time and resource constraints. The following aspects were most important in the development perspective:

1) Selection of Technology
2) Selection of Supporting Tools
3) Development Environment
4) Time management
5) Task Management
6) Bug Fixing

In the following sections, these aspects will be discussed briefly.

Selection of Technologies

On the perspective of software development, selection of technology in development is a key factor for the success considering various constraints. Some technology provides greater efficiency, others provide more security whereas some provide fast development time. It is necessary to consider the requirements and constraints before selection of development technology.

For the development purpose, I had decided to use **JAVA SE** programming language. During the start of my development process **JAVA 10** was the latest version. So, I had decided to go with it. For Compiling my source code, I have use **JDK 10**.

As the software implements memorized execution technique it is by nature datacentric and database is one of the core concerns here. Embedded database was necessary as the software is desktop based. I have used **SQLite JDBC 3.23.1** as my embedded database.

For developing User Interface (UI), **JavaFX** was used. JavaFX is far responsive and stable than Java Swing. Moreover, some recent IDE has made development of UI with JavaFX much interactive. The required libraries are by default integrated in JAVA JRE and JDK.

One of the requirements of this software was developing interactive installer and uninstaller. This was done using **Inno Setup Compiler**. It provides many suitable options for interactive installation and uninstallation.

Selection of Supporting Tools

Every software is not developed from the scratch. In most of the cases, we do not reinvent. It is redundant to remake what already exists. Therefore, to use already developed tool is good for fast and efficient development.

Like mentioned in the requirement and design document, a SMT solver was necessary to get a feasible solution within the constraints. SMT solver is very much mathematics intensive. I have decided to use **z3 Solver** developed by **Microsoft**. Upon determining the constraint, it can provide the satisfiable values successfully. This tool is capable of solving problems with real numbers.

Development Environment

Development environment is another core aspect of implementation. These details can help a lot whenever we want to do testing. In the following a brief description of the developing environment is given –

Category	Name
Processor	Intel Core i5 M540 (2.53 GHz × 2 cores)
RAM	4 GB DDR3
Java IDE	Eclipse Neon
GUI IDE	Scene Builder
Operating System	Windows 10 Pro
Installed Libraries	JDK 10, SQLite, z3 Solver, JavaFX
Installer Builder	Inno Setup Compiler
Code Quality Test Tool	Java Metrica

Table 1: Development Environment

Time management

Time management was an important issue in my project. I have tried to tackle the challenge by initial development plans understood from the SRS and design document. Beside this project, I had four other courses in this semester. I have tried to put small efforts regularly on this project.

I was ahead of my schedule during the development of SRS and design. But due to some academic pressures from other courses, I fall behind the schedule in the middle. But in the end, I have successfully completed the project on time.

Task Management

For managing different tasks, there are many tools and frameworks. I have hosted my project in the GitHub. In GitHub, there is a task management tool which is very much interactive and easy to use.

I have utilized the tool regularly as a guide for completing my project. I have committed my work to GitHub on a regular basis. When there was a bug or an issue, I utilized the issue feature on GitHub to track down the progress regarding it.

Bug Fixing

No software is completely free of bugs. During development a lot of bug arises in the software system. An efficient strategy is required for tacking them. To discover a bug, I have used unit testing method in my program. I have tested every code after I wrote them. Further, integration and system testing were done on the program.

C Path Finder is a testing tool. Therefore, it requires parsing its input. Developing a parser by own is a very much critical task. A huge amount of bug was discovered after I first developed my program. Step by step, all known bugs were engaged and fixed.

Implementation Challenges

With every complicated task, there comes a greater challenge. During the development of C Path Finder, I was faced with many known and unknown challenges. A brief description of them a provided below:

Implementation of C Parser on own

Although there are many existing C parser, none of them provided output according to my need. Therefore, to start my work I had to implement a C Parser on my own. The task of this parser was to separate every different component like – variable name, type, assigned value, types of loop, assignment conditions etc.

Moreover, I had to filter unnecessary parts like – comments, custom directives, preprocessing directives before separating various components from C code. This was an enormous, frustrating and undoubtedly complicated task.

Making Control Flow Graph

To generate all possible test cases of a program, a control flow graph is necessary. All possible path means the different directions that a program can execute. This was again another insanely difficult task. Making simple control flow graph is easy but when there are nested conditional statements the control flow graph becomes pretty complex.

I tried building control flow graph in my SPL 1 where I failed. But this time, after 3 weeks of intensive effort I could successfully implemented a program that can build control flow graph with unlimited depth of nested conditions.

Integrating z3 Solver

Z3 solver is a constraint solver tool developed by Microsoft. The tool is written mostly in C and python. To use it, I had to integrate in java code. To Run .dll files from java is a complicated task. The problem multiplies whenever we package the whole program in an installer. Configuring automatically during installation required huge efforts.

Implementing Top Down Recursion in using Z3

After fetching conditions from the C code, I had to analyze every single variable into its type and value. This required a recursion with top down approach. To find out the type of the variable, I had to use my self-written parser during the recursion. Recursion can overflow memory if not handled properly. Moreover, use of parser inside it made it more complicated to implement in the end.

Standalone Installer

To build standalone installer, I have used Inno Setup Compiler. Software works fine in the developing environment. But many problems occur when the environment is changed. For example, Software after installing in any folder other than C:/programfiles works fine. But it fails in that place. After thorough investigation it was discovered that there were a administrator permission issue refraining it from performing as desired.

Overview of Source Code Structure

Source code structure is necessary for providing an overview of the project. As C Path Finder is comparatively large project, an overview is necessary. The whole C Path Finder project is divided into multiple java packages. They are shown below with brief examples –

Package Name	Responsibility	No of Classes
Main	Responsible for the start execution of the program and contains the code for GUI	3
Database	This is a database accessing layer for SQLite. Controls memorized execution.	2
IO	Does all input output file operations in the file systems	4
Code Beautifier	Removes unnecessary parts like – comments, blank lines, directives	6
Parser	Separates and parses - methods, parameters, structs, condition statements, return types etc.	4
Parser Components	Helping package for Parser. Contains model of various objects for parsing.	7
Control Flow Graph Builder	This package builds control flow graph from the parsed components. Stores them into a tree of nodes with component information	3
Result	This package is responsible for building result into a suitable format for output	2
Symbolic Solver	Implements z3 solver with top down recursive approach. Also responsible for building path conditions.	3

Table 2: Project Source Code Structure

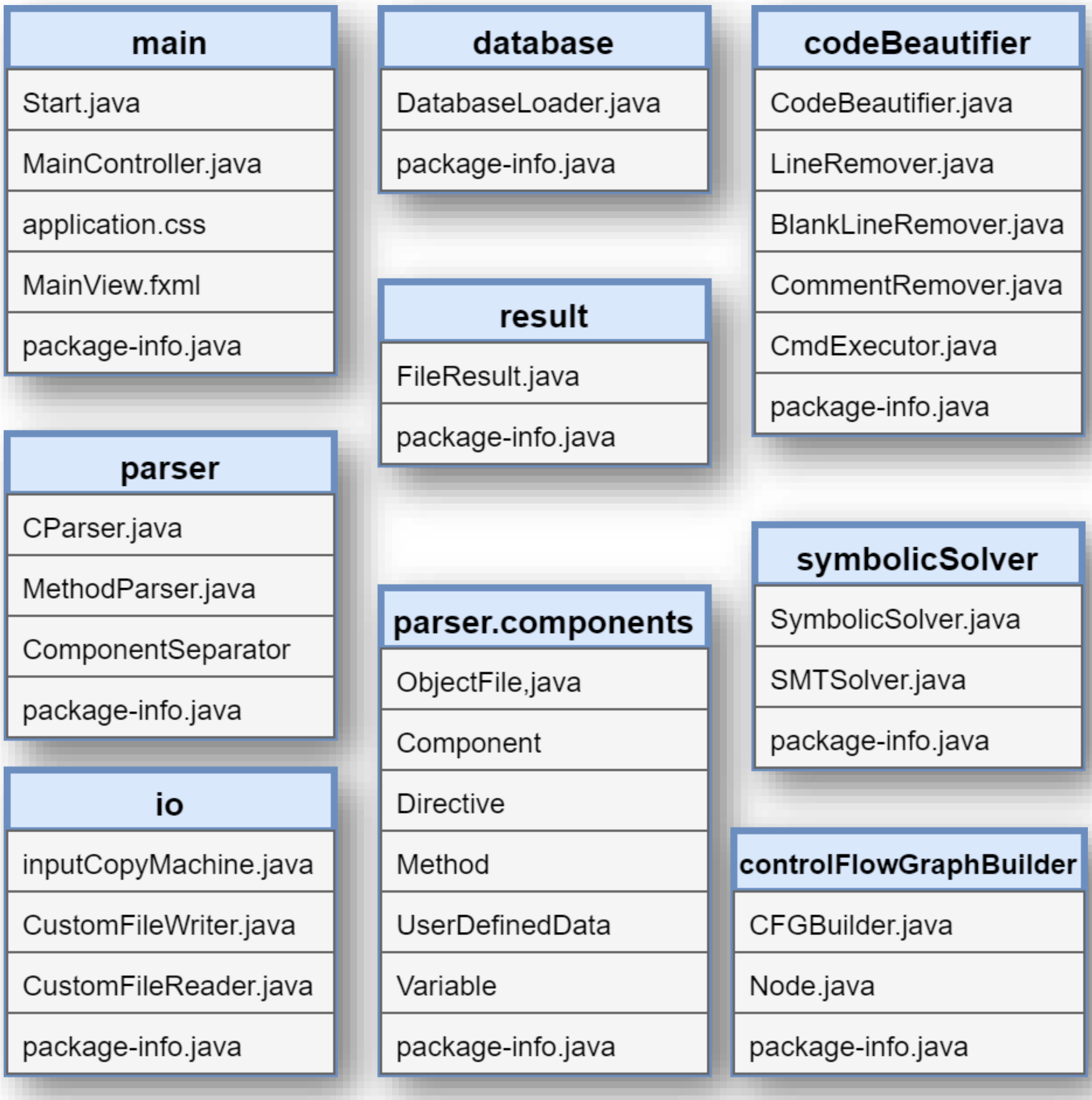


Figure 13: Source Code Package Details

Details of Functionalities

C Path Finder can be viewed as a tool of three functionalities. It can provide the followings –

1) Generate Test Cases with Symbolic execution
2) Get unreachable paths with Symbolic Execution
3) Memorized Symbolic Execution

Let us now describe each of the functionality's one by one and its implementation details.

Generation of test Cases

The program execution starts with **Start.java**. The class is extended by a JavaFX class known as Application.java. From the GUI, it collects necessary information like file paths and destination paths. Everything in GUI are controlled by class **MainController**.

Upon getting this information, when user hits the run button the main processing part starts. Class **DatabaseLoader** is invoked. It initializes the database with default tables. Then junk files are removed from the database. Then **Codebeautifier** is invoked with to remove the comment, blank lines and seven preprocessor directives in C. This provides a fresh code for parsing.

After that we invoke **CParser** with the necessary files to be analyzed. From there we create an **ObjectFile** for each of the files to be analyzed. The main task of objectFile is to do initial component separation like – method, user defined data types, and preprocessing directives.

MethodParser is invoked from to parse methods. It separates the parameters, method body and return types. Further, it beautifies the method body for further processing. When method components are separated, **CFGBuilder** is invoked to build a control flow graph for the method body. It breaks every statement into **Node** components. Then we use **backtracking** to find all the possible path from the control flow graph.

Then, **SymbolicSolver** is invoked implementing **z3**. We place the parsed variable name, type and conditions into z3 for each path. Conditions are provided to z3 through **top-down recursive** technique. z3 solves the constraint and generates **test cases**. **FileResult** is used for storing all the test cases for each C files.

Getting Unreachable Paths

Unreachable paths are a byproduct of the test case generation technique. They occur where test cases cannot be generated for ambiguity. For the test cases whose output is “**Unsatisfiable”** are regarded as unreachable paths. The process is similar as mentioned above.

Memorized Symbolic Execution

Memorized symbolic execution is executed before the invocation of **CodeBeautifier**. It checks the last modified time of every file with the last execution time of the file. If the file is not present in database or the modification time is later than the time of execution then the file is run again, otherwise that file is skipped. Results of the skipped file is fetched from the database.

DatabaseLoader manages the responsibility of memorized execution technique. This technique reduces the processing power required for test case generation for consecutive runs.

Testing

Testing is a required part of Software Development Life Cycle. Testing includes many activities like – Generation of test plan, analyzing test case coverage, finding out the maximum test coverage, testcase design etc.

Three types of testing were done during the test phase. They are:

1) Unit Testing
2) Integration Testing
3) System testing
4) **Code Quality** Testing

In the following we are presenting a brief description of the testcases that were tested against the software system.

Unit Testing

Unit Testing means testing each method of separately. The goal of this type of testing is to discover bug in a method level. Unit test was done for all methods side by side with the development. **As it is not possible to include all the unit tests some of them are included here**. The Unit Tests of C Path Finder are given below:

Test ID	Method Name	Input	Expected	Obtained	Status
1	makeIntConstant	“23”	Z3 IntExpr with value 23	Z3 IntExpr with value 23	Ok
2	makeIntVariable	“a”	Z3 IntExpr with variable name a	Z3 IntExpr with variable name a	OK
3	makeFloatConstant	“2.02”	Z3 FPExpr with value 2.02	Z3 FPExpr with value 2.02	OK
4	makeFloatVariable	“b”	Z3 FPExpr with variable name b	Z3 FPExpr with variable name b	OK

5	loadParameters	Method with parameters	Load parameters along with datatypes	Load parameters along with datatypes	OK
6	loadParameters	Method without parameters	Loads no value	Loads no value	OK
7	getUpdatedValue	Given that a = 5 and b = 7, input provides a = a + b	a = 5 + 7 b = 7	a = 5 + 7 b = 7	OK
8	removeComments	Printf("//abc");	Printf("//abc")	Printf("//abc")	OK
9	removeComments	Printf("/**abc* /");	Printf("/**abc*/");	Printf("/**abc*/");	OK
10	removeComments	Printf("ab"); //bm	Printf("ab");	Printf("ab");	OK
11	removeComments	Printf("ab"); /* Test comment */	Printf("ab");	Printf("ab");	OK
12	parsebeforeParameter	Int main (void){}	Return type = int, method name = main	Return type = int, method name = main	OK
13	parsebeforeParameter	main (void){}	Return type = "", method name = main	Return type = "", method name = main	OK

Table 3: Unit Tests

Integration Testing

Integration testing is important because a method may separately act as expected but differently when the modules are integrated. In the following some key integration testing are provided:

Test ID	Test Description	Input	Expected	Obtained	Status
1	Database should be created if not exits and then its containing tables should be created after that	Program runs for the first time	Database and tables are created	Database and tables are created	Ok
2	Codebeautifier should combinedly remove single line comments, multiple line comments and blank lines	C source file containing comments and blank lines	Removes all comments and blank lines	Removes all comments and blank lines	OK
3	CodeBeautifier should not remove any execuable statements as comments	C source file containing comments, blank lines and executable statements	No executable statements are removed	No executable statements are removed	OK
4	Symbolic solver should be able to generate test cases for nested if statements	If(a>= b) { If(a==b) { } }	Any input that satisfies	a = 0 , b = 0	OK
5	Symbolic solver should be able to generate test cases for combined conditions	If(a> b && c > a)	Any input that satisfies	a = 1, b = 0, c = 2	OK
6	Symbolic solver should inform about unreachable path	If(a>b && b>a) where a,b is integer type	“unsatisfiable”	“unsatisfiable”	OK

Table 4: Integration Testing

System Testing

System Testing is done as a whole. It identifies if there are any environment related problems like – OS specific bugs, OS computability, Hardware performance etc. Some of the key system testing done is provided below:

Test ID	Test Description	Input	Expected	Obtained	Status
1	Running Installer	Following the steps of installer	Program installs	Program installs	Ok
2	Running Uninstaller	Execute the uninstaller	Program uninstalls	Program uninstalls	OK
3	Program should work in windows 10 pro	Run the program in windows 10 pro	Program works normally	Program works normally	OK
4	Program should run in systems having 2 or more GB RAM	Run the program	Program can generate testcases	Program can generate testcases	OK
5	Program should run in systems having 2 or more Processor Cores	Run the program	Program can generate testcases	Program can generate testcases	OK

Table 5: System Tests

Coding Quality Testing

In the era of Software Engineering, code quality is very much important. An automated Tool named "**Java Metrica**" was used to analyzed the various metrics of the source code. Metrics are very useful in determining various software problems and future problem prediction. This tool we have used measure 9 metrics regarding java source code.

Class Name	LOC	Comment	CC	WMC	Coupling	Lack of Coh	RFC	NoC	DOI
CFGBuilder	313	4	38	3	1	1	91	0	0
Node	58	2	9	4	0	7	14	0	0
DatabaseLoader	320	6	14	0	0	2	137	0	0
BlankLineRemover	14	0	3	3	0	1	4	0	0
CmdExecutor	46	0	3	3	0	1	13	0	0
CodeBeautifier	44	6	4	2	0	1	12	0	0
CommentRemover	66	2	13	13	0	1	22	0	0
LineRemover	67	0	8	2	0	2	33	0	0
CustomFileReader	50	0	4	4	0	1	12	0	0
CustomFileWriter	39	0	4	4	0	1	7	0	0
InputFileCopyMachine	65	0	6	3	0	2	28	0	0
MainController	76	0	3	0	0	2	26	0	0
Start	99	14	1	0	4	4	38	0	0
Component	7	1	1	1	0	0	0	3	0
Directive	11	0	1	1	0	2	0	0	1
Method	73	2	7	3	0	8	8	0	1
ObjectFile	58	0	2	0	5	4	13	0	0
UserDefinedData	10	0	1	1	0	2	0	0	1
Variable	61	4	4	0	0	2	20	0	0
ComponentSeparator	236	2	34	3	3	1	65	0	0
CParser	99	3	4	1	1	2	24	0	0
MethodParser	229	22	27	3	1	2	68	0	0
FileResult	15	0	1	1	0	4	2	0	0
SMTSolver	477	33	83	3	0	2	233	0	0
SymbolicSolver	153	20	16	2	2	1	72	0	0

Table 6: Result from Java Metrica

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.

User Manual

C Path Finder is a unit test case generator intended for C programmers. Therefore, it is assumed that the user has experience of running general IDE.

Minimum System Requirements

Processor - Dual core 2.0 GHz Processor

Ram - 2 GB DDR3

JDK/JRE – 10 or later versions

Hard Disk – Minimum 50 MB available Hard Drive for Installation

Recommended System Requirements

Processor – Intel Core i7 (6th Generation or upwards)

Ram - 8 GB DDR3

JDK/JRE – 10 or later versions

Hard Disk – 100 MB available Hard Drive for Installation

Downloading the installer

Interactive windows Installer can be downloaded from https://github.com/Sajed49/C-Path-Finder/releases/tag/1.0.1.

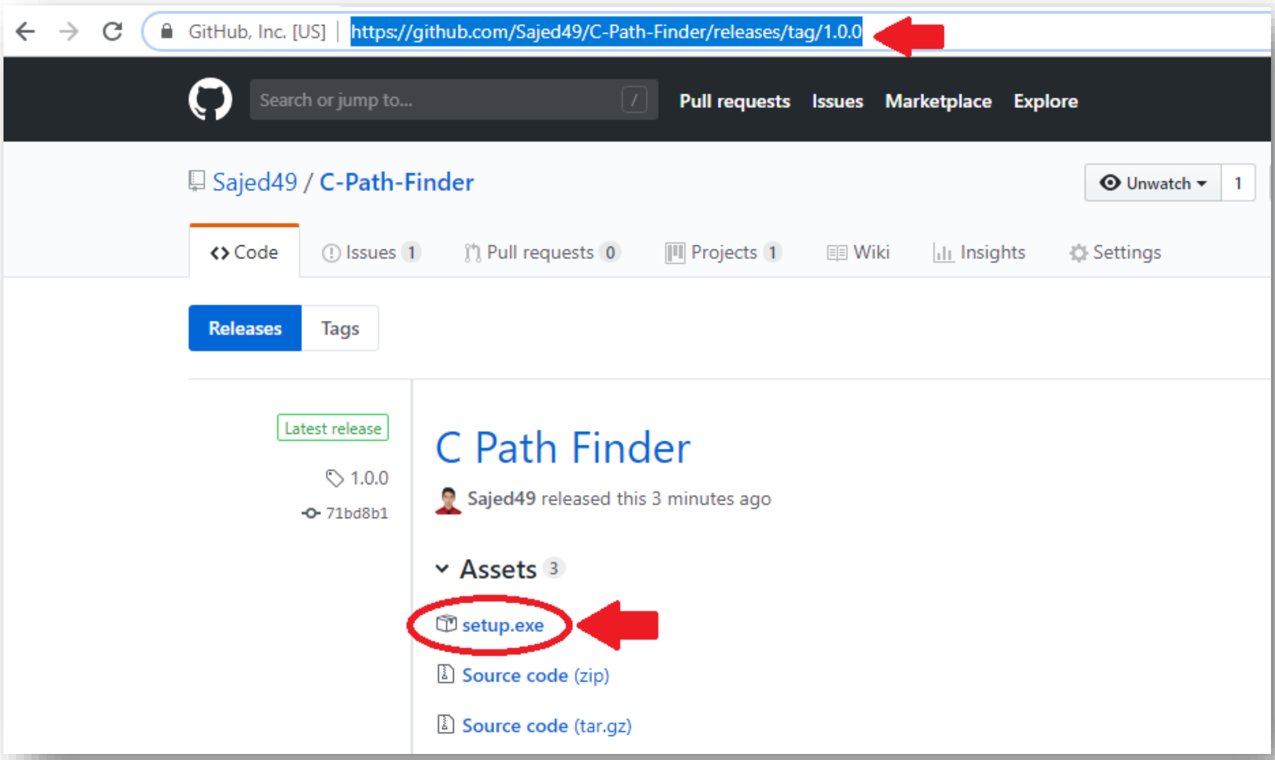
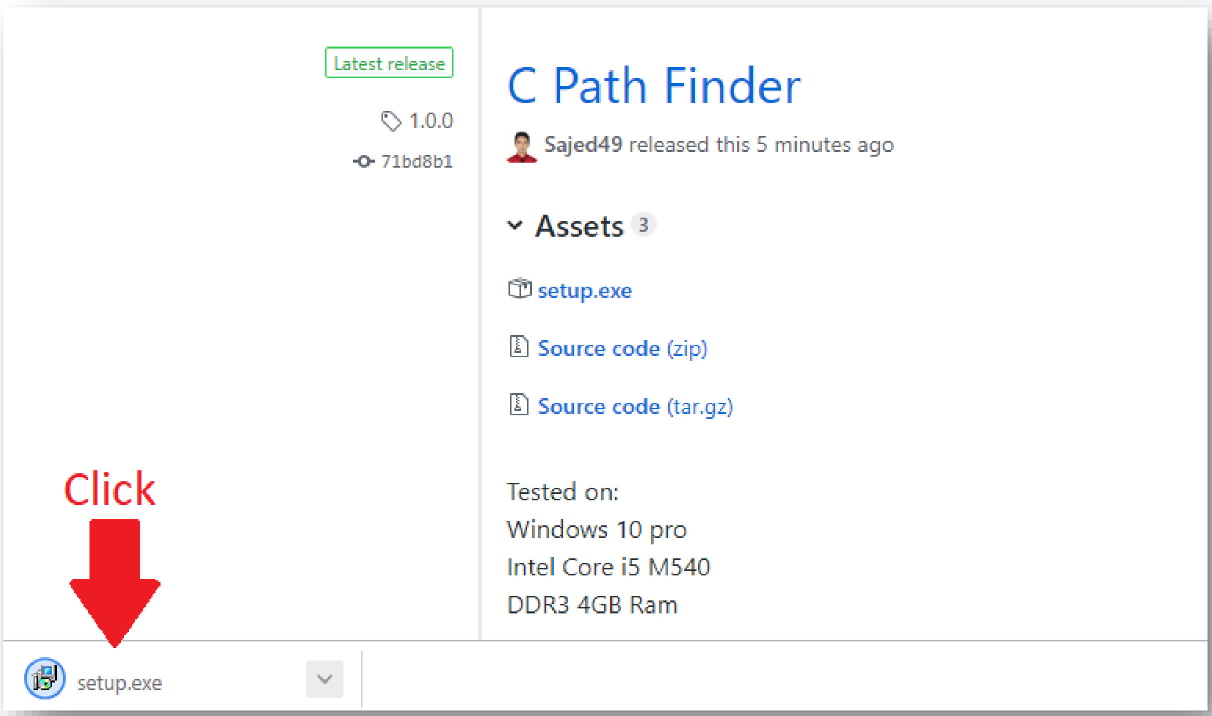


Figure 14: Download from website

Running the Installer

The downloaded installer can be run as windows executable.



C Path Finder

Sajed49 released this 5 minutes ago

Assets 3

setup.exe

Source code (zip)

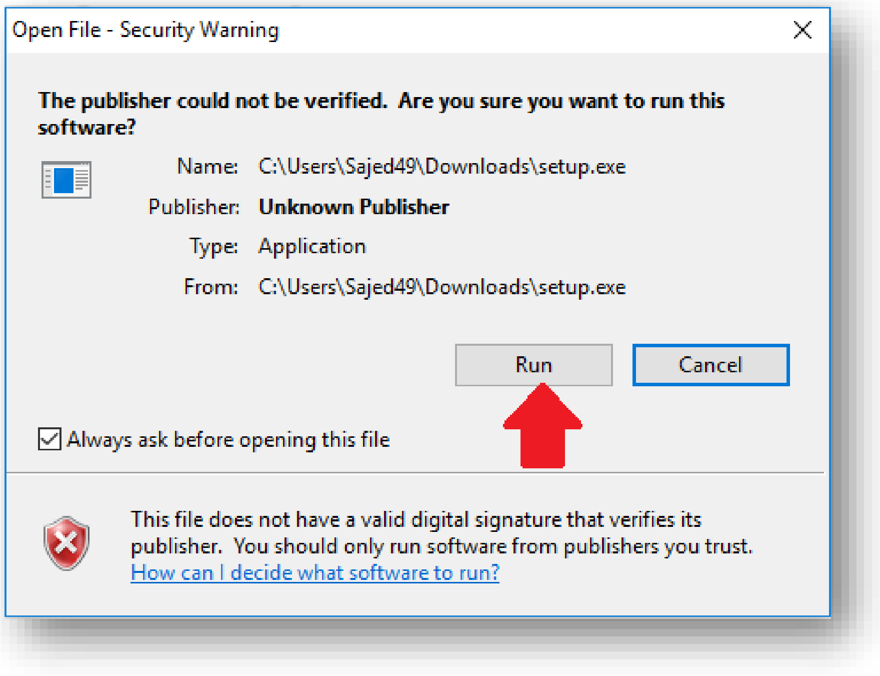
Source code (tar.gz)

Tested on:
Windows 10 pro
Intel Core i5 M540
DDR3 4GB Ram

Figure 15: Executable File

Installing the application

In the following, step by step installation process are shown with figure. The user can customize the installation path as provided in our requirement. Moreover, User can choose whether to create a desktop shortcut icon.


Open File - Security Warning

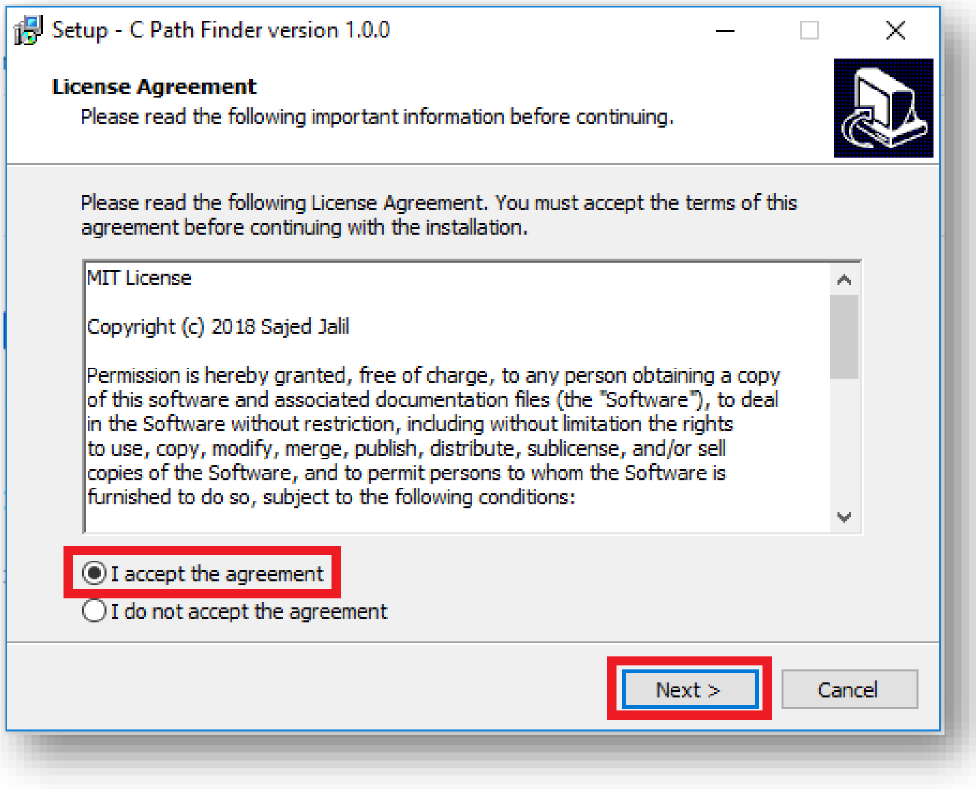
The publisher could not be verified. Are you sure you want to run this software?

Name: C:\Users\Sajed49\Downloads\setup.exe
Publisher: **Unknown Publisher**
Type: Application
From: C:\Users\Sajed49\Downloads\setup.exe

[✓] Always ask before opening this file

This file does not have a valid digital signature that verifies its publisher. You should only run software from publishers you trust.
How can I decide what software to run?

Figure 16: Installation permission 1


Setup - C Path Finder version 1.0.0

License Agreement
Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

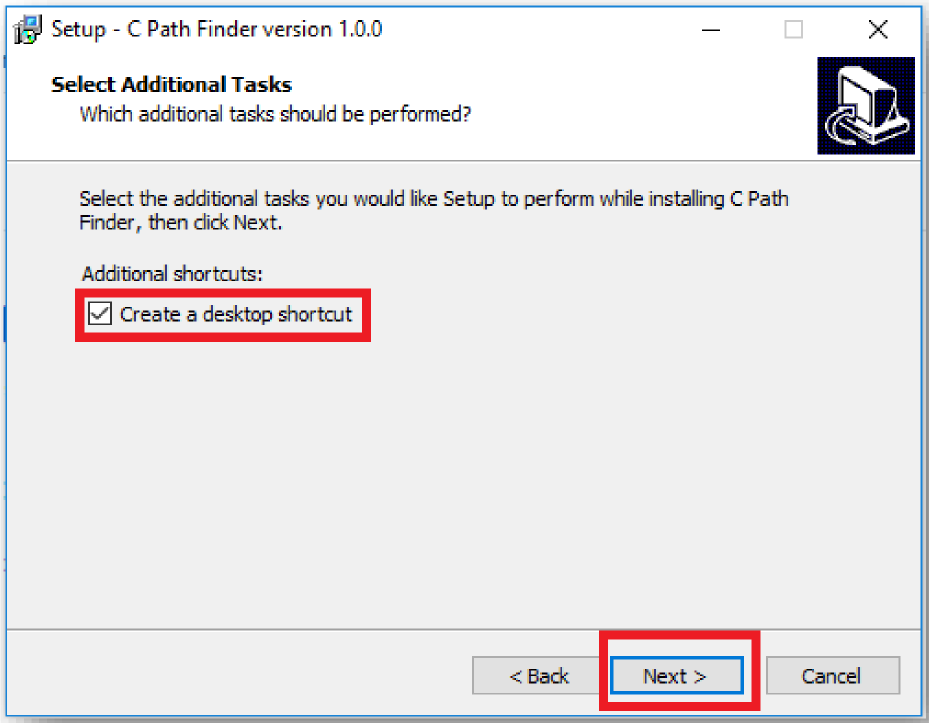
MIT License

Copyright (c) 2018 Sajed Jalil

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

● I accept the agreement
○ I do not accept the agreement

Figure 17: License Agreement

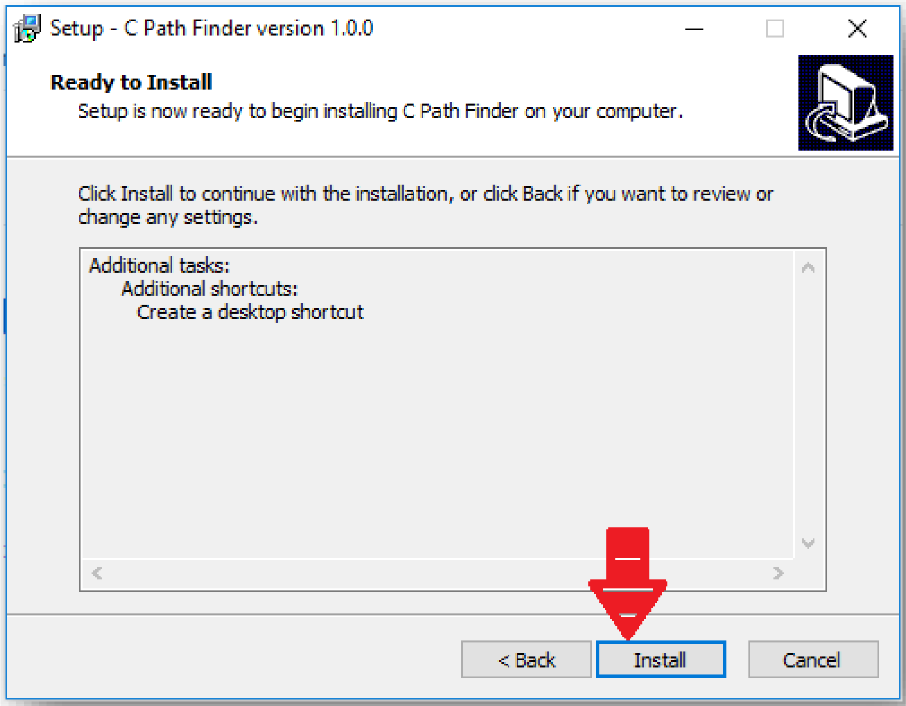

Setup - C Path Finder version 1.0.0

Select Additional Tasks
Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing C Path Finder, then click Next.

Additional shortcuts:
☑ Create a desktop shortcut

Figure 18: Desktop Shortcut

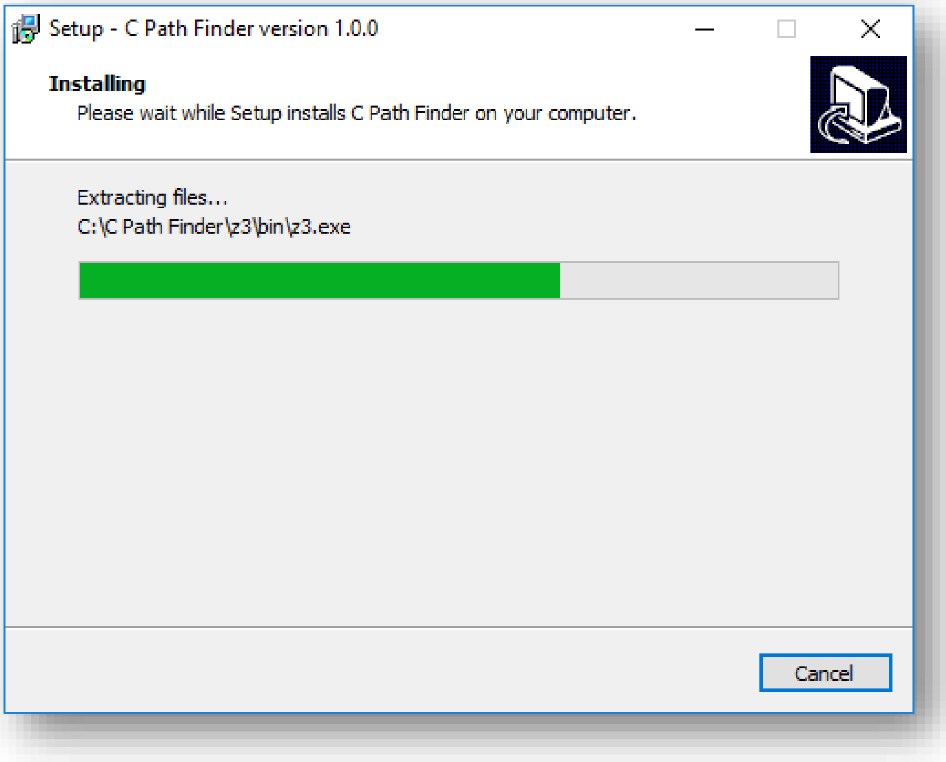

Setup - C Path Finder version 1.0.0

Ready to Install
Setup is now ready to begin installing C Path Finder on your computer.

Click Install to continue with the installation, or click Back if you want to review or change any settings.

Additional tasks:
Additional shortcuts:
Create a desktop shortcut

Figure 19: Install button


Setup - C Path Finder version 1.0.0

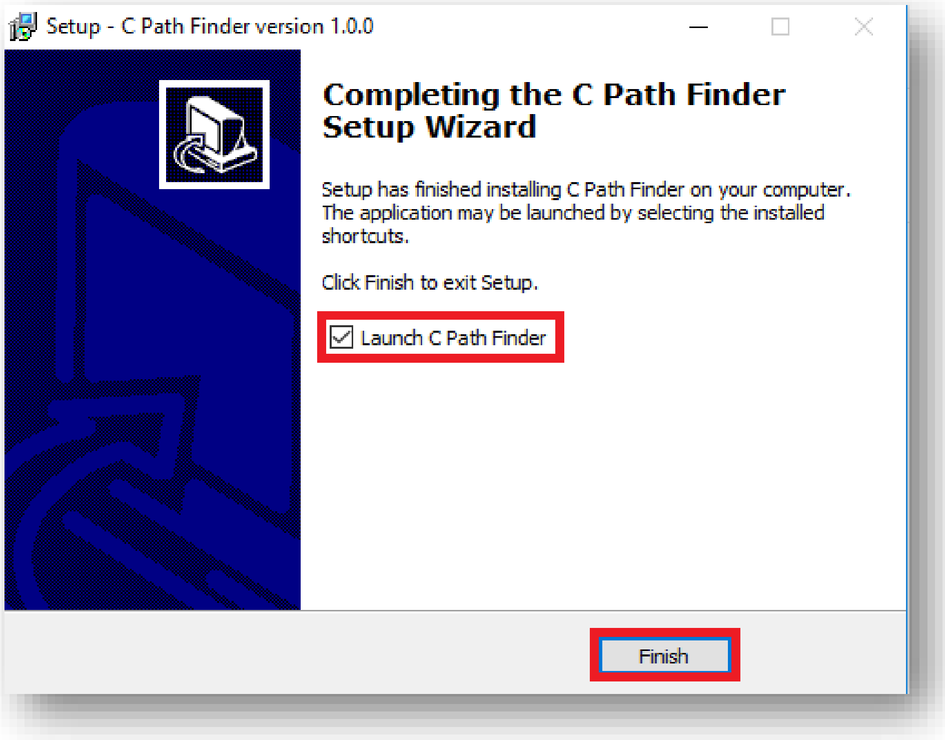
Installing

Please wait while Setup installs C Path Finder on your computer.

Extracting files...

C:\C Path Finder\z3\bin\z3.exe

Figure 20: Installation on progress


Setup - C Path Finder version 1.0.0

Completing the C Path Finder Setup Wizard

Setup has finished installing C Path Finder on your computer. The application may be launched by selecting the installed shortcuts.

Click Finish to exit Setup.

[✓] Launch C Path Finder

Figure 21: Installation completed

Running the Application

The application can be launched from desktop shortcut or the installation directory.

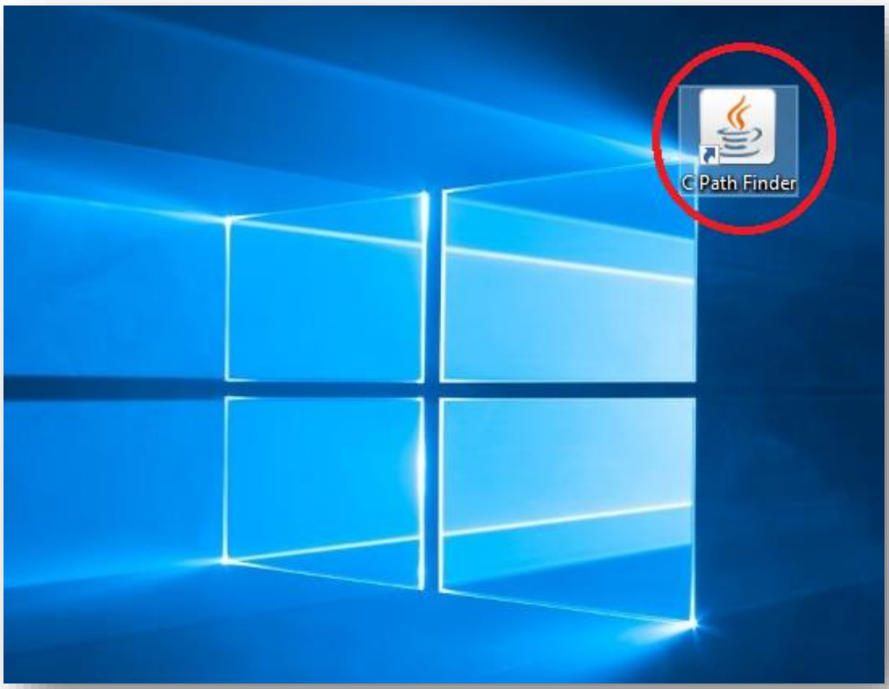


Figure 22: Launching from shortcut icon

From the **File** option we can open system explorer. From the **Settings** we can change the result generation directory. **Tutorial** menu opens the user manual for the application. To import the test files, we simple have to click the **select** button.

After selection a system explorer view will pop up. Form there we have to select the input directory where C source codes are located. In our example, the folder is located in C drive named as **test**.

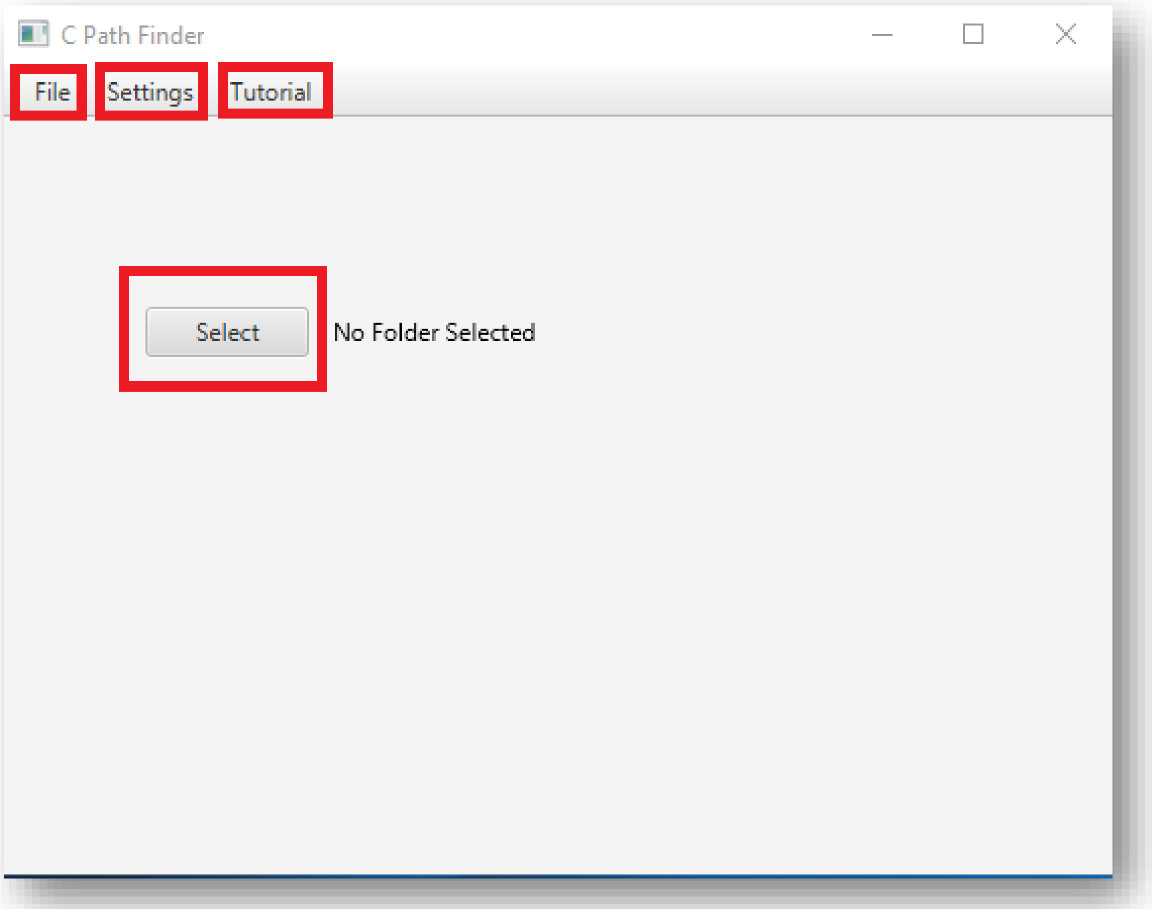
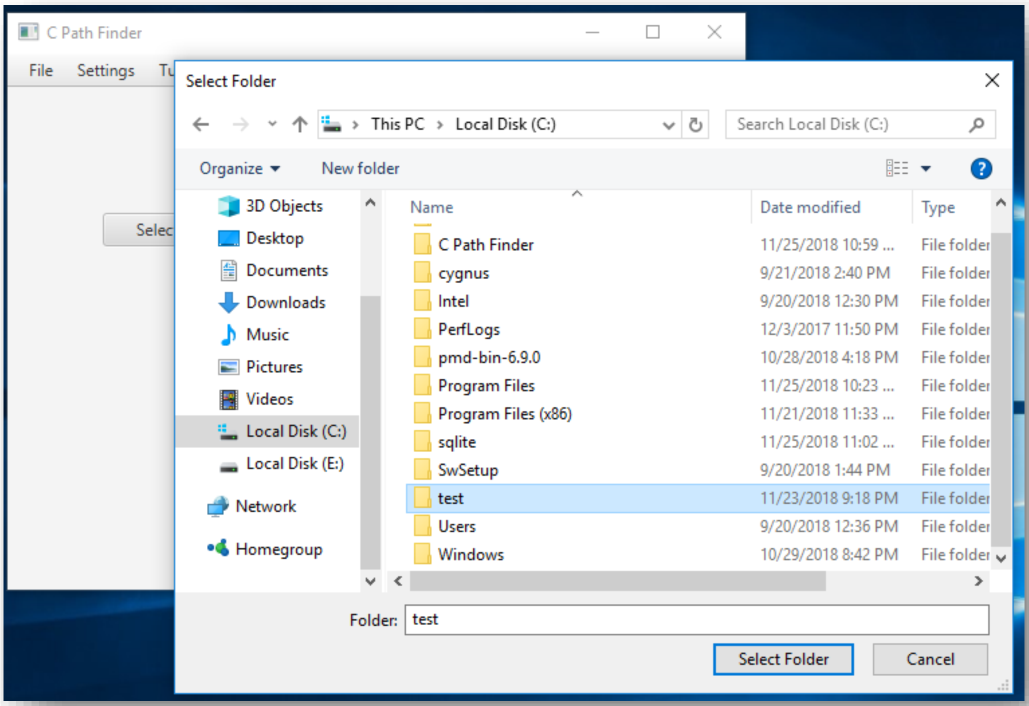


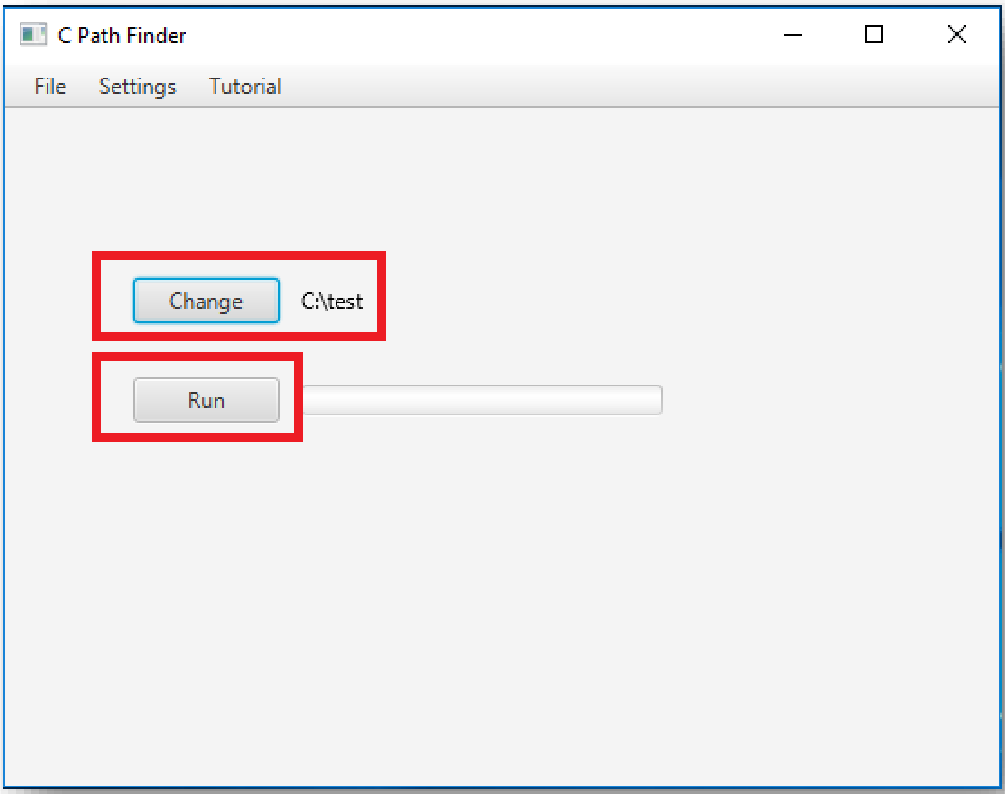
Figure 23: Application basics



Name	Date modified	Type
C Path Finder	11/25/2018 10:59 ...	File folder
cygnus	9/21/2018 2:40 PM	File folder
Intel	9/20/2018 12:30 PM	File folder
PerfLogs	12/3/2017 11:50 PM	File folder
pmd-bin-6.9.0	10/28/2018 4:18 PM	File folder
Program Files	11/25/2018 10:23 ...	File folder
Program Files (x86)	11/21/2018 11:33 ...	File folder
sqlite	11/25/2018 11:02 ...	File folder
SwSetup	9/20/2018 1:44 PM	File folder
test	11/23/2018 9:18 PM	File folder
Users	9/20/2018 12:36 PM	File folder
Windows	10/29/2018 8:42 PM	File folder

Folder: test

Figure 24: Source File Selection



C:\test

Figure 25: Running Source Files

After selection, we can change the test files or run the test files. When the processing is completed, the progress bar will be filled and a new button named open result will be shown. WE can re-run the source files. In that case, run will use **memorized execution** technique. **Open Result** button will pop up the result directory containing testcases in txt file format.

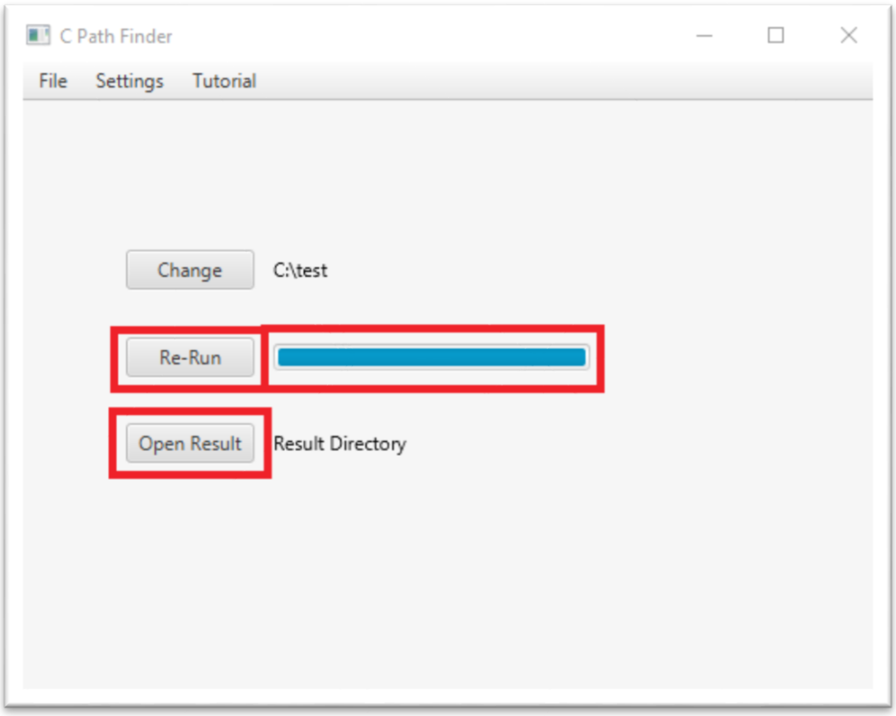


Figure 26: Progress Bar and Result

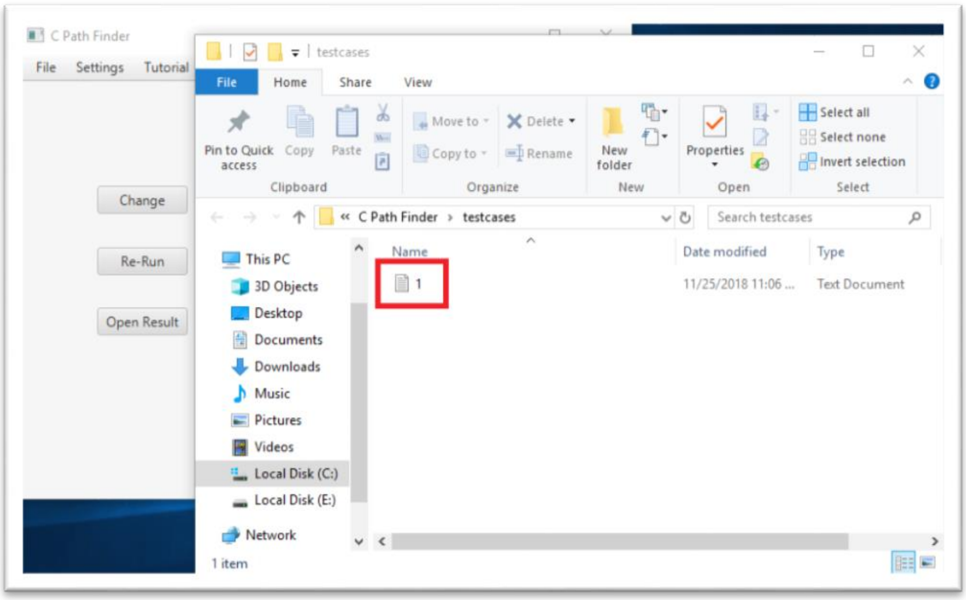
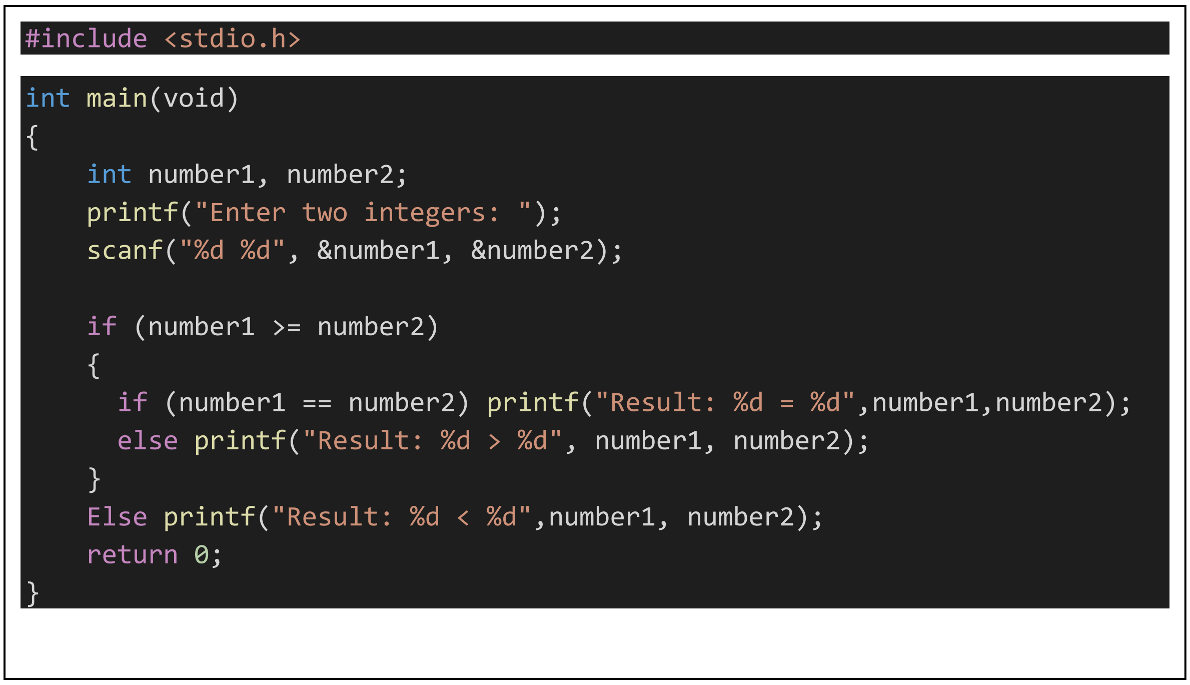


Figure 27: Result Output Directory

This figure below shows the sample test code for the file name 1.c . The source code had three possible paths. WE can see in the next result figure that our program could completely detect three paths and successfully generated test case that traverse these three paths.

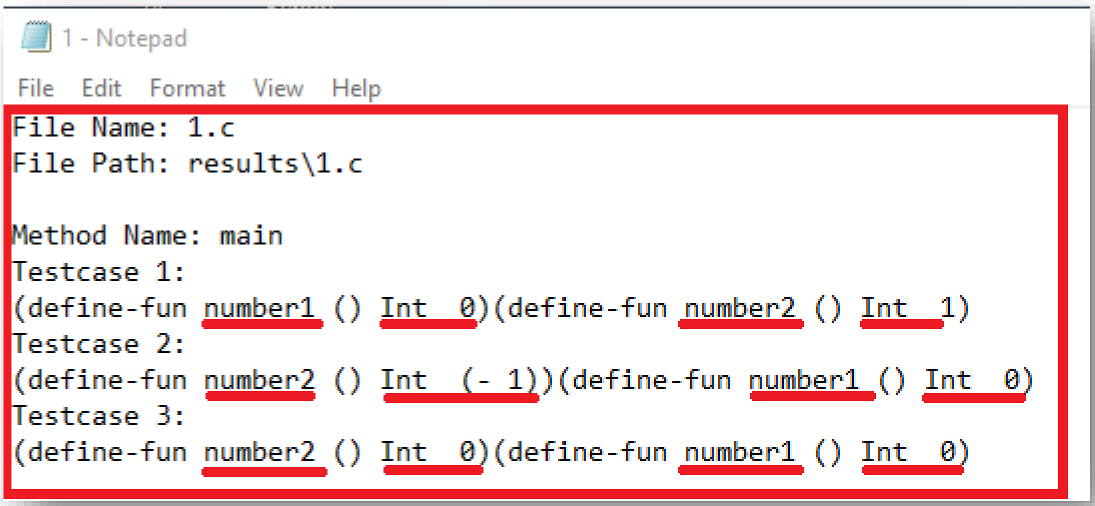


```c
#include <stdio.h>

int main(void)
{
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    if (number1 >= number2)
    {
      if (number1 == number2) printf("Result: %d = %d",number1,number2);
      else printf("Result: %d > %d", number1, number2);
    }
    Else printf("Result: %d < %d",number1, number2);
    return 0;
}
```

Figure 28: Input Test Code



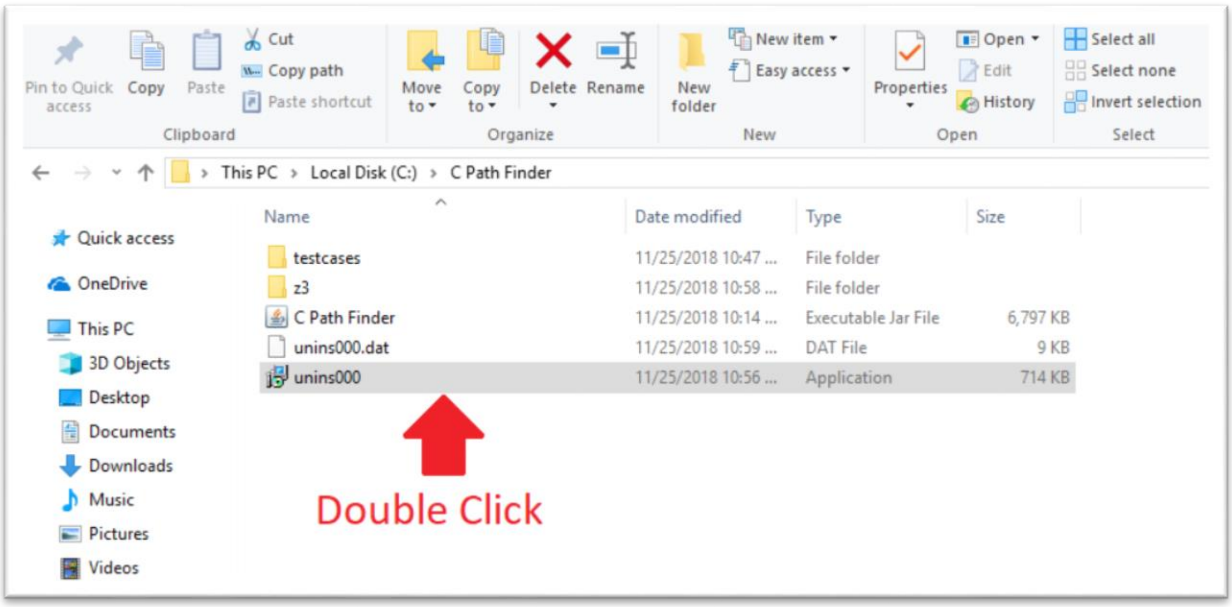
```
File Name: 1.c
File Path: results\1.c

Method Name: main
Testcase 1:
(define-fun number1 () Int  0)(define-fun number2 () Int  1)
Testcase 2:
(define-fun number2 () Int  (- 1))(define-fun number1 () Int  0)
Testcase 3:
(define-fun number2 () Int  0)(define-fun number1 () Int  0)
```

Figure 29: Generated Test Cases

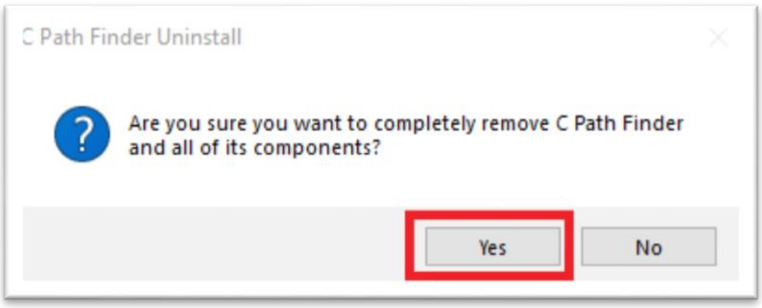
Uninstalling the Application

Besides interactive installation feature, we have also added uninstallation feature in our application. The following shows the steps of uninstallation process.



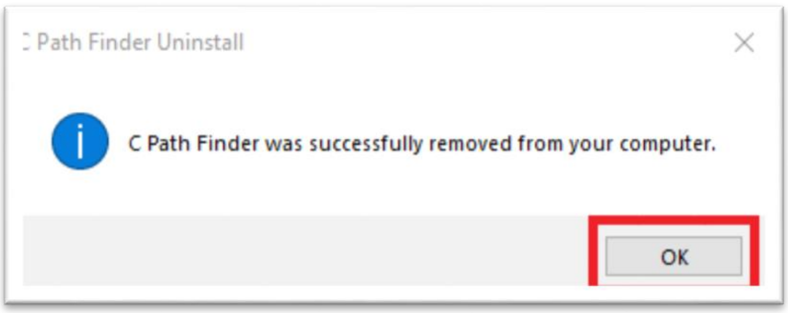
Name	Date modified	Type	Size
testcases	11/25/2018 10:47 ...	File folder	
z3	11/25/2018 10:58 ...	File folder	
C Path Finder	11/25/2018 10:14 ...	Executable Jar File	6,797 KB
unins000.dat	11/25/2018 10:59 ...	DAT File	9 KB
unins000	11/25/2018 10:56 ...	Application	714 KB

Figure 30: Uninstallation Steps



Are you sure you want to completely remove C Path Finder and all of its components?

Figure 31: Confirm Uninstallation


C Path Finder Uninstall

C Path Finder was successfully removed from your computer.

Figure 32: Uninstallation Completed

Troubleshoot

1. For any test generation related problems, please uninstall the application and install again.
2. Logs can be viewed for deeper insight.
3. For further problems, please raise an issue in GitHub repository or mail us at bsse0714@iit.du.ac.bd

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.

[4] King, J.C., 1976. Symbolic execution and program testing. *Communications of the ACM*, *19*(7), pp.385-394.

[5] Khurshid, S., Păsăreanu, C.S. and Visser, W., 2003, April. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 553-568). Springer, Berlin, Heidelberg.

Appendix

CFG – Control Flow Graph

GCC – GNU Compiler Collection

GUI – Graphical User Interface

Java SE 10 – JAVA Standard Edition 10

JRE – Java Runtime Environment

JVM – Java Virtual Machine

QFD – Quality Function Deployment

SMT – Satisfiability Modulo Theory

SRS – Software Requirements Specification