DATA-FLOW ANALYSIS FOR PLC PROGRAMS

A THESIS

Submitted by

SREEJA S NAIR (ROLL NO. M120448EE)

In partial fulfillment for the award of the Degree of

MASTER OF TECHNOLOGY
IN
ELECTRICAL ENGINEERING
(Industrial Power and Automation)

Under the guidance of
Dr S. ASHOK

DEPARTMENT OF
ELECTRICAL ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY CALICUT
NIT CAMPUS PO, CALICUT
KERALA, INDIA 673601.

MARCH 2014

ABSTRACT

There are five different programming languages supported by IEC 61131-3 standard for control systems. A tool can be developed for the analysis of these languages for the common run-time errors during the compile time itself, which can reduce the development to commissioning time drastically. The five languages are exported to xml files which can be interpreted by the tool to detect the run-time errors. The run-time errors like array out of bounds, infinite loops, etc, can be defined as rules and the tool can check whether it is being violated. The program code is parsed and converted into Abstract Syntax Tree. Then Control Flow Graph is generated using the AST. This will be used as input to the Data-flow analysis framework. Data-flow analysis algorithms are to be developed for handling the probable error conditions.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

IEC 61131-3 is the third part of the open international standard IEC 61131 for programmable logic controllers, and was first published in December 1993 by the IEC. The current (third) edition was published in February 2013. Part 3 of IEC 61131 deals with programming languages and defines two graphical and two textual PLC programming language standards:

- Ladder diagram (LD), graphical
- Function block diagram (FBD), graphical
- Structured text (ST), textual
- Instruction list (IL), textual
- Sequential function chart (SFC), has elements to organize programs for sequential and parallel control processing.

There are different types of development platforms for IEC 61131-3 based control systems. The development tool provided by ABB is Control Builder. It supports coding in all the five programming languages. The development tools check the syntax and semantics of the languages and throws errors during compilation when the rules of the grammar are violated. Currently these tools are not handling run time errors. They can be detected only in the testing phase. A tool can be developed for the analysis of the common run-time errors during the compile time itself, which can reduce the development to commissioning time drastically. The five languages are exported to xml files which can be interpreted by the tool to detect the run-time errors. The run-time errors like array out of bounds, infinite loops, etc, can be defined as rules and the tool can check whether it is being violated.

1.2 CURRENT SCENARIO

Control Builder, like other development platforms only check the grammar rules of the IEC 61131-3 based control system programs. Currently, the run-time errors are

detected by doing code review or testing. In code review, developers in the same team reviews each others code manually. Testing comprises of many stages which takes a long time schedule.
A very few IDEs which are providing support for static analysis (which are included in the literature review) are closed source. They are developed for the particular Integrated development environments.

1.3 PROBLEM IDENTIFICATION

Both testing and code review, which are performed to detect the run-time errors, are labor-intensive tasks. As the cost of errors in a Systems development life cycle increases by each step, it is better to capture the defects as early as possible in the development stage itself. So we need to automate the process in order to catch the errors during development stage by using data-flow analysis techniques.
A data-flow analysis is to be performed for IEC 61131-3 based control systems which is useful for detecting certain run-time errors during the deveopment time itself. A few probable run-time errors can be sorted out such as the ones listed below:

- Division by zero
- Array access out of bounds
- Use of uninitialized variables
- Unused variables
- Invariant If condition
- Loss of precision
- Using variables other than integer for counting
- Unreachable code
- Infinite loops

The list is not exhaustive. A tool is to be developed such that the tool takes the program as the input and outputs the list of probable errors belonging to the above said category. This tool should be able to perform its duty without actually executing the program. This is called static analysis. This analysis can be performed by

tracking the values of variables at every program point. This is known as data-flow analysis.

1.4 OUTLINE OF THE THESIS

The thesis is organized into four chapters including introduction.

- Chapter 2 gives the literature survey conducted for the particular problem.
- Chapter 3 gives an overview of the problem.
- Chapter 4 explains data-flow analysis developed.

CHAPTER 2

LITERATURE SURVEY

2.1 INTRODUCTION

Literature Survey have been conducted to analyze the need of an analysis tool. Systems Development Life Cycle and cost of errors in SDLC have been analyzed. IEC 61131-3 specifications was studied. Similar tools for data-flow analysis were analyzed.

2.2 IEC 61131-3 STANDARD

IEC 61131-3 is the standard for PLC programming languages. There are five languages defined by the standard. They are

1. Function Block Diagram(FBD), graphical
2. Instruction Set(IL), textual
3. Ladder Diagram(LD), graphical
4. Sequential Function Chart(SFC), has elements to organize programs for sequential and parallel control processing.
5. Structured Text(ST), textual

A syntactic specification for the programming languages of the IEC 611131-3 standard is presented by Flor Narciso, Addison Rios-Bolivar, Francisco Hidrobo and Olga Gonzalez [1]. The specification are modeled into the components of a formal grammar containing a set of terminal symbols, a set of nonterminal symbols, the start symbol and a set of production rules. The formal grammar described corresponds to a context-free grammar. The textual grammar is only specified in the paper. The paper can be summarized as follows.

- Production rules are specified for programming model, characters, digits, identifiers, constants, data types, variables, units of programs organization and configuration elements
- Production rules are presented for Instruction List and Structured Text.

A manual from ABB [2] explains the way of Structured Text programming. It explains using the Control builder[3] development environment.
Motohiko Okabe [4] has explained the development of a processor which is capable of executing the IEC 61131-3 languages directly on the hardware logic.

2.3 SYSTEMS DEVELOPMENT LIFE CYCLE

The systems development life cycle (SDLC) in systems engineering is a process of creating or altering information systems, and the models and methodologies that people use to develop these systems. The SDLC framework provides a sequence of activities for system designers and developers to follow. It consists of a set of steps or phases in which each phase of the SDLC uses the results of the previous one.An SDLC adheres to important phases that are essential for developers, such as planning, analysis, design, and implementation. A number of SDLC models have been created: waterfall, fountain, spiral, build and fix, rapid prototyping, incremental, and synchronize and stabilize. The oldest of these, and the best known, is the waterfall model: a sequence of stages in which the output of each stage becomes the input for the next. These stages can be characterized and divided up in different ways as follows:

- Preliminary Analysis : In this step, you need to find out the organization's objectives and the nature and scope of the problem under study.
- Systems analysis, requirements definition: Defines project goals into defined functions and operation of the intended application. Analyses end-user information needs.
- Systems design: Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudo-code and other documentation.
- Development: The real code is written here.
- Integration and testing: Brings all the pieces together into a special testing environment, then checks for errors, bugs and interoperability.
- Acceptance, installation, deployment: The final stage of initial development, where the software is put into production and runs actual business.

- Operation and Maintenance: During the maintenance stage of the Life-Cycle, the system is assessed to ensure it does not become obsolete. This is also where changes are made to initial software. It involves continuous evaluation of the system.

- Disposal Phase: In this phase, plans are developed for discarding system information, hardware and software in making the transition to a new system. The purpose here is to properly move, archive, discard or destroy information, hardware and software that is being replaced, in a matter that prevents any possibility of unauthorized disclosure of sensitive data. The disposal activities ensure proper migration to a new system.

Rodriguez-Martinez, L.C., Mora M. and Alvarez F. J.[5] presents a comparative study of various process models of Software Development Life Cycles. The proposals can be summarized as follows.

- It proposes that waterfall model of SDLC has been the base for several subsequent models.

- It is possible to elaborate a systematic comparison of SDLC process models by using a meta-model.

- The agile method, which is one among the widely accepted method also considers the classic phases such as analysis, design, coding, test and implementation.

2.4 COST OF ERRORS

It will cost more to fix a requirements error after the product is built than it would if the requirements error was discovered during the requirements phase of a project. According to National Aeronautics and Space Agency (NASA) findings, "problems that are not found until testing are at least 14 times more costly to fix than if the problem was found in the requirements phase"[6]. The error detection follows "the earlier the better" principle. The errors detected in the earlier phases of SDLC are less costly to fix. The following chart gives an idea about the error detection at different stages and the cost involved to correct it. J. Christopher Westland [7] reports the significance of cost of error detection and correction in the software development. The contents of the report can be summarized as follows.

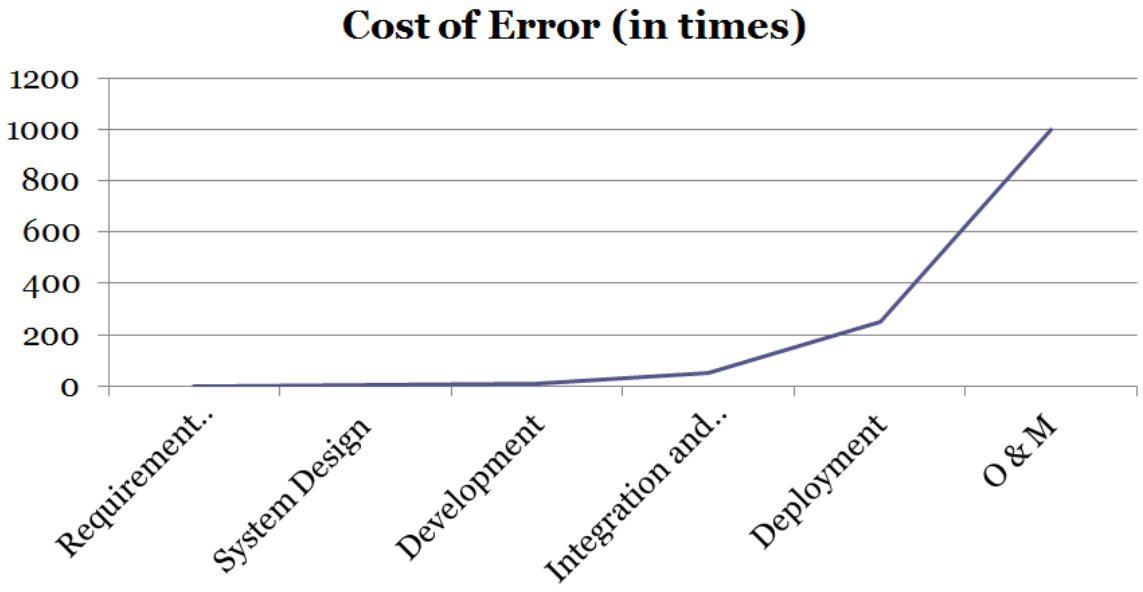


Figure 2.1: Cost of errors (in times)

- Errors generate significant costs if their resolution requires system redesign.
- Uncorrected errors become exponentially more costly with each phase in which they are unresolved.
- Testing must occur over a substantial portion of the useful life of the system in order to detect a substantial portion of the total errors which will ever occur.

2.5 DATA-FLOW ANALYSIS TOOLS

Static analysis of code is an effective method of discovering potential errors. As computers have gained more processing power and memory, static analysis on codes of nontrivial size has become feasible of these systems. Many static analysis tools have been created for general purpose languages.

One of the initial tools to be created to find potential errors was LINT[8] which worked on C and C++. For Java there exists several code checkers like PMD[9] and FindBugs[10].PMD tool applies checks to identify bug patterns on the abstract syntax tree that is built from the source code. Klockwork Insight[11] uses a Klockwork Truepath static analysis engine that performs checks on the source code to find bug patterns. It supports languages like C, C++, Java and C#.

There are some static analysis tools for IEC 61131-3 that target detecting instances of coding violations for IEC 61131-3 languages such as Logi.LINT[12] and Arcade.PLC[13]. Logi.LINT supports only graphical languages FBD, SFC and LD. Arcade.PLC has support for only textual based languages ST and IL. It employs model checking as well as static analysis tools. It is in the development stage only. However, there is no freely available tool for advanced data-flow analysis.
Mario de Sousa [14] has described a tool for data-flow analysis of IEC 61131-3 languages. That paper describes a tool which converts the IEC 61131-3 languages - Structured Text, Sequential Function Chart and Instruction List to ANSI C language. The data-flow analysis is performed on the ANSI C program.

2.6 SUMMARY

Literature survey conducted was summarized in this chapter. Literature survey was conducted on various topics such as IEC 61131-3 standard, SDLC, Cost of errors in SDLC and Data-flow analysis of IEC 61131-3 based programs.

2.7 RESEARCH GAP IDENTIFIED

From the literature survey it is clear that cost of errors contribute a lot to the total cost of system development. So in order to minimize this, an automated method should be adopted. The existing methods are not both free and advanced. The plan of the project is to build an open-source tool which does data-flow analysis in the program and identifies the errors prior to testing.

CHAPTER 3

PROBLEM DEFINITION

3.1 INTRODUCTION

The tool under development will reduce the cost of errors occurring in the systems development life cycle. It uses data-flow analysis method to identify the errors.

3.2 PROBLEM DEFINITION

Currently two main strategies are employed for error detection :

- Testing
- Code Review

In order to assure correctness of the software through testing, all possible combinations of inputs and all possible paths of execution must be exhaustively verified. Unfortunately, it is nearly impossible to exercise all such permutations of events and conditions in a test of reasonable duration. Code reviews, again may not be exhaustive, as they are highly dependent on the skill and patience of the reviewer. Both types of assessments are inordinately labor-intensive.
A tool which can detect some of the possible run-time errors is to be developed. This tool can be used during the development time. This tool can be used to detect some errors and to prove the absence of certain errors.

3.3 OBJECTIVES

A tool is needed which will analyze the control program and detect possible errors in an efficient way. This tool should be able to take care of the five different varieties of IEC 61131-3 programming languages. These five languages can be parsed into a common xml structure and further analysis can be performed using this xml format. The tool should be able to understand the grammar specifications of the IEC 61131-3 based control programs. The first preference is given to develop the grammar and rules for Structured Text, as the other graphical languages can be represented in ST format. Currently the tool to parse the control program to xml is present. Now

the xml has to be parsed to read the structure of the control program, perform the data-flow analysis and highlight the probable errors.
The milestones for the development of the tool are

- Grammar
- Parse Tree
- Abstract Syntax Tree
- Control Flow Graph
- Data-flow analysis framework
- Data-flow algorithms for different error rules
- Error Handler

The work plan for the two semesters can be summarized as follows.

1. S3
    - Develop a grammar for structured text using Irony framework.
    - Develop a robust data-flow analysis algorithm for IEC 61131-3 languages.
    - Identify probable error rules.
2. S4
    - Develop algorithm for error handling.
    - Implement the data-flow analysis algorithm.
    - Implement the error handler.

3.4 SUMMARY

The data-flow analysis for IEC 61131-3 based control systems and the work plan was briefed.

CHAPTER 4

DATA-FLOW ANALYSIS TOOL

4.1 INTRODUCTION

The detailed description of the data-flow analysis tool developed is given in this chapter. Each section includes a brief introduction of the theory, algorithms used and the implementation. The entire description is based on Structured Text, since Control Builder has an inbuilt function to represent Function Block Diagram and Ladder Diagram in Structured Text.

4.2 GRAMMAR

This is the first step implemented for the data-flow analysis tool. In this, the grammar of the language, ie., the syntax is specified.[15]
A grammar naturally describes the hierarchical structure of most programming language constructs. A context-free grammar has four components: [16]

- A set of terminal symbols, sometimes referred to as "tokens". The terminals are the elementary symbols of the language defined by the grammar.
- A set of non-terminals, sometimes called "syntactic variables". Each non-terminal represents a set of strings of terminals, in a manner we shall describe.
- A set of productions, where each production consists of a non-terminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or non-terminals , called the body or right side of the production.
- A designation of one of the non-terminals as the start symbol.

There are different classes of grammar:

- LL(n) - The first "L" in LL(n) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "n" for using n input symbol of look-ahead at each step to make parsing action decisions. Predictive parsers are designed for these types of languages.

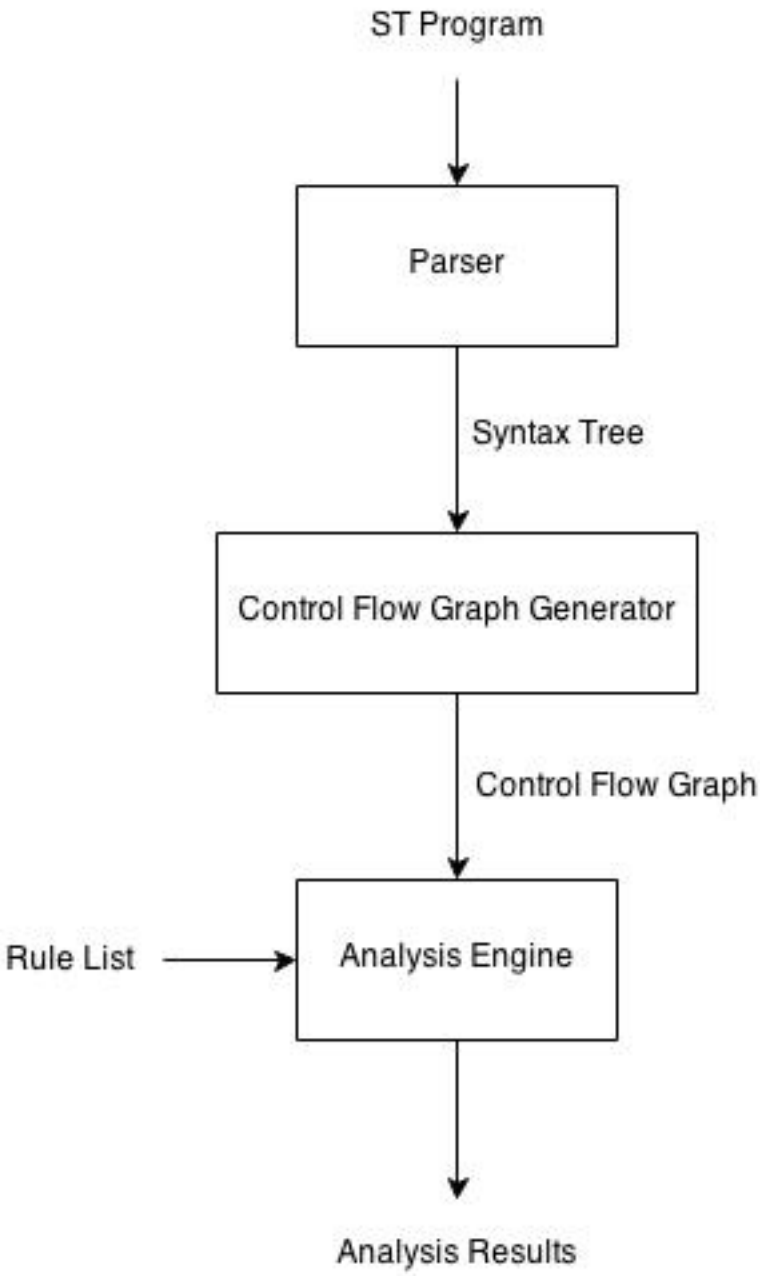


Figure 4.1: Overview of the Tool

- LR(k) - the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of look-ahead that are used in making parsing decisions. It is also know as shift-reduce parsing method.
  - canonical-LR - makes full use of the look-ahead symbol(s)
  - LALR - based on the LR(0) sets of items, and has many fewer states than typical parsers based on the LR(l) items.

LALR parsers are desirable because they are very fast and small in comparison to other types of parsers. An LALR parser generator accepts an LALR grammar as input and generates a parser that uses an LALR parsing algorithm (which is driven by LALR parser tables). Frank DeRemer invented LALR parsers with his PhD dissertation, called "Practical LR(k) Translators", in 1969, at MIT.

4.2.1 Implementation

The context-free grammar for structured text was obtained from the paper of Flor Narciso, Addison Rios-Bolivar, Francisco Hidrobo, Olga Gonzalez[1]. The paper presents the syntax specification of the Structured Text language in EBNF form.
The grammar was implemented with the help of Irony framework [17]. Irony provides a rich interface for implementing terminals, non-terminals, production rules and start symbol. Irony is a LALR parser generator.
An LALR parser generator is a software tool that reads a BNF grammar and creates an LALR parser which is capable of parsing files written in the computer language defined by the BNF grammar. Extended BackusNaur Form (EBNF) is a family of metasyntax notations, any of which can be used to express a context-free grammar. EBNF is used to make a formal description of a formal language which can be a computer programming language. They are extensions of the basic BackusNaur Form (BNF) metasyntax notation.

4.3 PARSE TREE

A concrete syntax tree or parse tree or parsing tree is an ordered, rooted tree that represents the syntactic structure of a string according to some formal grammar.
A parse tree pictorially shows how the start symbol of a grammar derives a string

in the language. Their structure and elements more concretely reflect the syntax of the input language. Parse trees may be generated for sentences in natural languages as well as during processing of computer languages, such as programming languages. Parse Tree is the output of a parser.

4.3.1 Algorithm

Formally, given a context-free grammar, a parse tree according to the grammar is a tree with the following properties:

- The root is labeled by the start symbol.
- Each leaf is labeled by a terminal or by ϵ.
- Each interior node is labeled by a non-terminal.
- If A is the non-terminal labeling some interior node and $X_1, X_2, ..., X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1X_2...X_n$. Here, $X_1, X_2, ..., X_n$ each stand for a symbol that is either a terminal or a non-terminal . As a special case, if $A \rightarrow \epsilon$ is a production, then a node labeled A may have a single child labeled ϵ.

4.3.2 Implementation

The Irony framework which was used to implement the grammar produces the Parse Tree.

4.4 ABSTRACT SYNTAX TREE

An 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. The syntax is "abstract" in not representing every detail appearing in the real syntax. Abstract syntax trees are used in program analysis and program transformation systems.
Abstract syntax trees are a data structure widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as a higher level intermediate representation of the program through several stages that the compiler

requires, and has a strong impact on the final output of the compiler. Being the product of the syntax analysis phase of a compiler, the AST has a few properties that are invaluable to the further steps of the compilation process. When compared to the source code, an AST does not include certain elements, such as inessential punctuation and delimiters. A more important difference is that the AST can be edited and enhanced with properties and annotations for every element it contains. Such editing and annotation is impossible with the source code of a program, since it would imply changing it. At the same time, an AST usually contains extra information about the program, due to the consecutive stages of analysis by the compiler. A simple example of the additional information present in an AST is the position of an element in the source code. This information is used in case of an error in the code, to notify the user of the location of the error.
The AST is used intensively during semantic analysis, where the compiler checks for correct usage of the elements of the program and the language. Also, during semantic analysis the compiler generates the symbol tables based on the AST. A complete traversal of the tree allows to verify the correctness of the program. After verifying the correctness, the AST serves as the base for the code generation step. It is often the case that the AST is used to generate the 'intermediate representation' '(IR)' for the code generation.

4.4.1 Implementation

The Abstract Syntax Tree is also provided by the Irony framework. The nodes and the properties have to be specified. The nodes have to be derived from AstNode base class.

4.5 CONTROL FLOW GRAPH

A control flow graph (CFG) is an intermediate representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow

leaves.[16]

For constructing a control flow graph, the basic blocks should be constructed first. Then the edges, i.e. the connection between the basic blocks are constructed. The edges may be labeled (true or false) or unlabeled. Then the dominators of each block is identified. A block B is said to dominate a block C if and only if all the paths that reach C from the entry node passes through block B. A block dominates itself.

Algorithm 1 Algorithm for generating Control Flow Graph

Require: Syntax Tree from syntax analysis phase
Ensure: Control Flow Graph
1: Construct Basic Blocks using algorithm 2
2: Construct the list of edges using algorithm 3

Algorithm 2 Algorithm for generating Basic Blocks

Require: Syntax Tree from syntax analysis phase
Ensure: List of Basic Blocks
1: First statement of program, first child statement after conditional or unconditional jumps and the statement pointed by the jumps are identified as starters
2: Start with a block name when the first starter is encountered
3: **repeat**
4: Till the next starter is seen, include every statements in the same block
5: Start a new block name and start with the new starter
6: **until** End of the syntax tree

4.5.1 Implementation

The control flow graph generator is implemented in a class. It takes the abstract syntax tree as the input and generates the control flow graph. Classes are made for basic blocks and connections. The control flow graph is drawn using Microsoft GLEE. [18]

4.6 DATA-FLOW ANALYSIS FRAMEWORK

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph is used to determine those parts of a program to which a particular value assigned to a variable might propagate. A simple way to perform data-flow analysis

Algorithm 3 Algorithm for generating Edges

Require: List of Basic Blocks
Ensure: List of edges
1: Make a new entry block and exit block
2: Make an edge to the first block from the entry block
3: **repeat**
4: Make an edge from the current block to the next block depending upon the last statement in the block
5: **if** the last statements are IF or ELSIF **then**
6: Make two branches for true and false
7: **else if** the last statements are FOR or WHILE **then**
8: Make an exit edge to the next block
9: Make a back edge from the last block of the loop body
10: **else if** the last statement is REPEAT **then**
11: Make a back edge from the last block of the loop body
12: Make an exit edge from last block of loop body
13: **else if** the last statement is CASE statement **then**
14: Make multiple edges for each case line
15: **else if** the last statement is EXIT **then**
16: Make an edge to next block after the loop body
17: **else if** the last statement is RETURN **then**
18: Make an edge to the exit block of the program
19: **end if**
20: **until** End of the Basic Block List
21: The last block is connected to the exit block

of programs is to set up data-flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint. Data-flow analysis attempts to obtain particular information at each point in a procedure. Usually, it is enough to obtain this information at the boundaries of basic blocks, since from that it is easy to compute the information at points in the basic block. [16]
In forward flow analysis, the exit state of a block is a function of the block's entry state. This function is the composition of the effects of the statements in the block. The entry state of a block is a function of the exit states of its predecessors. Some data-flow problems require backward flow analysis. This follows the same plan, except that the transfer function is applied to the exit state yielding the entry state, and the join operation works on the entry states of the successors to yield the exit state. The most common way of solving the data-flow equations is by using an iterative algorithm. It starts with an approximation of the in-state of each block. The out-states are then computed by applying the transfer functions on the in-states. From these, the in-states are updated by applying the join operations. The latter two steps are repeated until we reach the so-called fixpoint: the situation in which the in-states (and the out-states in consequence) do not change. The kind of data-flow analysis (forward or backward) should be decided according to the nature of the error to be detected. The equations for forward data-flow and backward data-flow can be seen in Equation 4.1 and Equation 4.2 respectively.

$$OUT[B] = f_B(IN[B])IN[B] = \bigcap_{P \epsilon predecessor(B)} OUT[P] \tag{4.1}$$

$$IN[B] = f_B^{'}(OUT[B])OUT[B] = \bigcap_{S \epsilon successor(B)} IN[S] \tag{4.2}$$

The data-flow analysis engine works on the control flow graph. The engine performs customized analysis according to the rules specified. Data-flow analysis consists of algorithms to gather information about a program. An abstraction of a set of all possible program states are associated with each program point (or basic block). Each data-flow analysis algorithm(D, V, $\wedge$, F) have to specify mainly the following properties:

- A direction of data-flow D, which is either FORWARDS or BACKWARDS
- A semi-lattice (V, $\wedge$) which includes

  - A domain V
  - A binary meet operator $\wedge$

- A family of transfer functions F, which contains the boundary conditions as well

4.6.1 Literature

Hyunha Kim, Tae-Hyoung Choi, Seung-Cheol Jung, Hyoung-Cheol Kim, Oukseh Lee, Kyung-Goo Doh [19] have proposed a software vulnerability checker. That tool takes rules describing vulnerability patterns and a source program as the input and detects locations and paths of the patterns in the program. This is simply based on pattern matching. The pattern matcher takes a program and a pattern, and gives the set of program points that match the pattern.
Alfred V.Aho, Ravi Sethi, Jeffrey D.Ullman, Monica S. Lam in their book [16] have explained the concept of data-flow analysis in detail. They have outlined several classic data-flow analysis algorithms such as reaching definitions, live variable analysis etc.
Patrick Cousot and Radhia Cousot [20] in their paper have developed an algorithm for static analysis. It analyses using the principle of interval abstraction. This paper has outlined the importance of Abstract Values and abstract contexts. This paper presents an algorithm for analysis of flowcharts.
Hanne Riis Nielson, Flemming Nielson, and Chris Hankin in their book [21] have described the four methods for program analysis. They have chapters dedicated to cover the concept of each type of analysis.
Florian Martin in his PhD thesis[22] has introduced a method to generate program analyzers. The work deals with interprocedural analysis and intraprocedural analysis as well.

4.6.2 Implementation

A generalized data-flow analysis framework is designed, which can be used for developing the data-flow analysis algorithms for different error rules. The data-flow analysis framework consists of direction, a semilattice, a transfer function set, initial and boundary conditions. These are the basic requirements of a data-flow analysis

algorithm. Each data-flow analysis algorithm can inherit this class and override the methods implemented for each of these.

Direction of data-flow

The direction of data-flow can be either forwards or backwards.

Semilattice

A semilattice is a partially ordered set which has the following properties.

- Reflexive, $x \leq x$
- Antisymmetric, if $x \leq y$ and $y \leq x$, then $x = y$
- Transitive, if $x \leq y$ and $y \leq z$, then $x \leq z$

A semilattice will have a greatest lower bound or a least upper bound. A semilattice can be defined as a meet semilattice and a join semilattice. A meet semilattice can be described as $< S, \bigwedge >$ where S denotes the set and $\bigwedge$ denotes the binary operation having the following properties:

- Idempotent - $x \bigwedge x = x$
- Commutative - $x \bigwedge y = y \bigwedge x$
- Associative - $x \bigwedge (y \bigwedge z) = (x \bigwedge y) \bigwedge z$
- For all x in S, $\top \bigwedge x = x$
- For all x in S, $\bot \bigwedge x = \bot$

A meet semilattice will contain a top element $\top$ and an optional bottom element $\bot$.

Transfer Function

Transfer function gives the relation between IN[B] and OUT[B] for a basic block. The transfer function $F : V \longrightarrow V$ has the fllowing properties :

- F has an identity function I such that I(x) = x for all x in V
- F is closed under composition : for any f, g in F, h(x) = f(g(x)) is in F

4.7 DATA-FLOW ALGORITHM FOR DIFFERENT ERROR RULES

4.7.1 Theory

The error rules which are identified for handling are :

- Division by zero
- Array access out of bounds
- Use of uninitialized variables
- Initialization of variables not used in the program
- Invariant if condition
- Unreachable code
- Infinite loops

Data-flow analysis algorithm are developed for each kind of errors to be handled. Division by zero, array access out of bounds and similar kind of error rules can be detected by using interval analysis. Use of uninitialized variables and unused variables can be handled using def-use chains.

4.7.2 Intraprocedural Analysis

There are different methods to handle function calls in data-flow analysis. Some of them are: [22]

1. The function call control flow graph is embedded in the calling program and the analysis is performed on the overall control flow graph. This is not practical for large programs as the graph may grow exponentially.
2. The return variables of the function call can be assumed to contain the worst case.
3. The argument values are passed into the function call control flow graph and the return values are returned to the calling program.

This prototype uses a combination of two techniques for intraprocedural analysis. The second method is used when the code of the function call is not available due to protection and the third method is used where the code inside the function block is available for analysis.

4.7.3 Algorithm

The algorithm for FORWARDS and BACKWARDS analysis are shown in algorithms 4 and 5 respectively. Some necessary information which are needed for performing

Algorithm 4 Algorithm for forward data-flow analysis

Require: Control flow graph
Ensure: IN and OUT of each basic block
1: OUT[ENTRY] = ϕ
2: **for all** basic block B **do**
3: OUT[B] = ϕ
4: **end for**
5: **while** any OUT changes **do**
6: **for all** basic block B other than ENTRY **do**
7: IN[B] = $\bigcap_{P\ a\ predecessor\ of\ B}$ OUT[P]
8: OUT[B] = f_B(IN[B])
9: **end for**
10: **end while**

Algorithm 5 Algorithm for backward data-flow analysis

Require: Control flow graph
Ensure: IN and OUT of each basic block
1: IN[EXIT] = ϕ
2: **for all** basic block B **do**
3: IN[B] = ϕ
4: **end for**
5: **while** any IN changes **do**
6: **for all** basic block B other than EXIT **do**
7: OUT[B] = $\bigcap_{S\ a\ successor\ of\ B}$ IN[S]
8: IN[B] = f_B(OUT[B])
9: **end for**
10: **end while**

data-flow analysis are as follows :

- use_B
- def_B
- $kill_B$
- gen_B

The algorithms for gathering each information is given in algorithm 6, 7, 9, 8. Examples for data-flow analysis algorithms can be seen in the following sections.

Algorithm 6 Algorithm for generating Use set for all basic blocks

Require: Control Flow Graph
Ensure: Use set for each Basic Block

```
1: for all basic block in basic block list do
2:    for all node in nodes in basic block do
3:       if node.type = expression then
4:          Calculate use set for each child node
5:       else if node.type = assignment then
6:          Calculate use set for the right-hand side of the assignment statement
7:       else if node.type = IF or ELSIF or WHILE or REPEAT or CASE then
8:          Calculate use set for the condition expression
9:       else if node.type = Function call then
10:         Calculate use sets for the argument nodes except the return nodes
11:      end if
12:   end for
13: end for
```

Algorithm 7 Algorithm for generating Def set for all basic blocks

Require: Control Flow Graph
Ensure: Def set for each Basic Block

```
1: for all basic block in basic block list do
2:    for all node in nodes in basic block do
3:       if node.type = assignment then
4:          Add the left-hand side variable into the def set
5:       end if
6:       if node.type = function return then
7:          Add the function return variable into the def set
8:       end if
9:    end for
10: end for
```

Dominators

The dominators for each basic block in control flow graph can be calculated using the data-flow analysis having the parameters as shown in algorithm 10

Algorithm 8 Algorithm for generating Gen set for all basic blocks

Require: Control Flow Graph
Ensure: Gen set for each Basic Block

```
1: for all basic block in basic block list do
2:    for all node in nodes in basic block do
3:       if node.type = assignment then
4:          Add the left-hand side variable into the gen set
5:       end if
6:       if node.type = function return then
7:          Add the function return variable into the gen set
8:       end if
9:    end for
10: end for
```

Algorithm 9 Algorithm for generating Kill set for all basic blocks

Require: Control Flow Graph
Ensure: Kill set for each Basic Block

```
1: for all basic block in basic block list do
2:    for all node in nodes in basic block do
3:       if node.type = assignment then
4:          Add the left-hand side variable into the kill set
5:       end if
6:       if node.type = function return then
7:          Add the function return variable into the kill set
8:       end if
9:    end for
10: end for
```

Algorithm 10 Algorithm for generating Dominators

Require: Control Flow Graph
Ensure: Dominators for each Basic Block

```
1: D - FORWARDS
2: V - Power set of basic blocks, N
3: ∧ - ∩
4: F - x ∩ B
5: Boundary condition - OUT[ENTRY] = ENTRY
6: Initial condition - OUT[B] = N
```

Reaching Definitions

Reaching definitions calculates the set of definitions which can reach a program point. The properties for reaching definitions are shown in algorithm 11

Algorithm 11 Algorithm for calculating Reaching Definitions

Require: Control Flow Graph
Ensure: Set of definitions coming to and going from each Basic Block
1: D - FORWARDS
2: V - Sets of definitions
3: $\wedge$ - $\cup$
4: F - $gen_B \cup (x - kill_B)$
5: Boundary condition - OUT[ENTRY] =
6: Initial condition - OUT[B] =

Live Variable Analysis

Live variable analysis indicates which all variables are in use after the particular program point. The variables which have no use in the future are considered to be dead. The properties for live variable analysis are as given by algorithm 12.

Algorithm 12 Algorithm for Live Variable Analysis

Require: Control Flow Graph
Ensure: Set of variables alive before and after each Basic Block
1: D - BACKWARDS
2: V - Sets of variables
3: $\wedge$ - $\cup$
4: F - $use_B \cup (x - def_B)$
5: Boundary condition - IN[EXIT] =
6: Initial condition - IN[B] =

Interval Analysis

Interval Analysis captures the range of values each variable can take at any given program point. The properties for interval analysis are as given by algorithm 13

4.7.4 Implementation

A general data-flow framework is specified in a class and each analysis algorithms are developed by inheriting the framework. The user is provided an option to select

Algorithm 13 Algorithm for Interval Analysis

Require: Control Flow Graph
Ensure: Range of values each variable can take before and after each Basic Block
1: D - FORWARDS
2: V - Sets of intervals
3: $\wedge$ - widening operator, $\bigtriangledown$
4: F - evaluate each basic block
5: Boundary condition - OUT[ENTRY] = maximum possible interval for each variable, $\perp$
6: Initial condition - OUT[B] = $\perp$

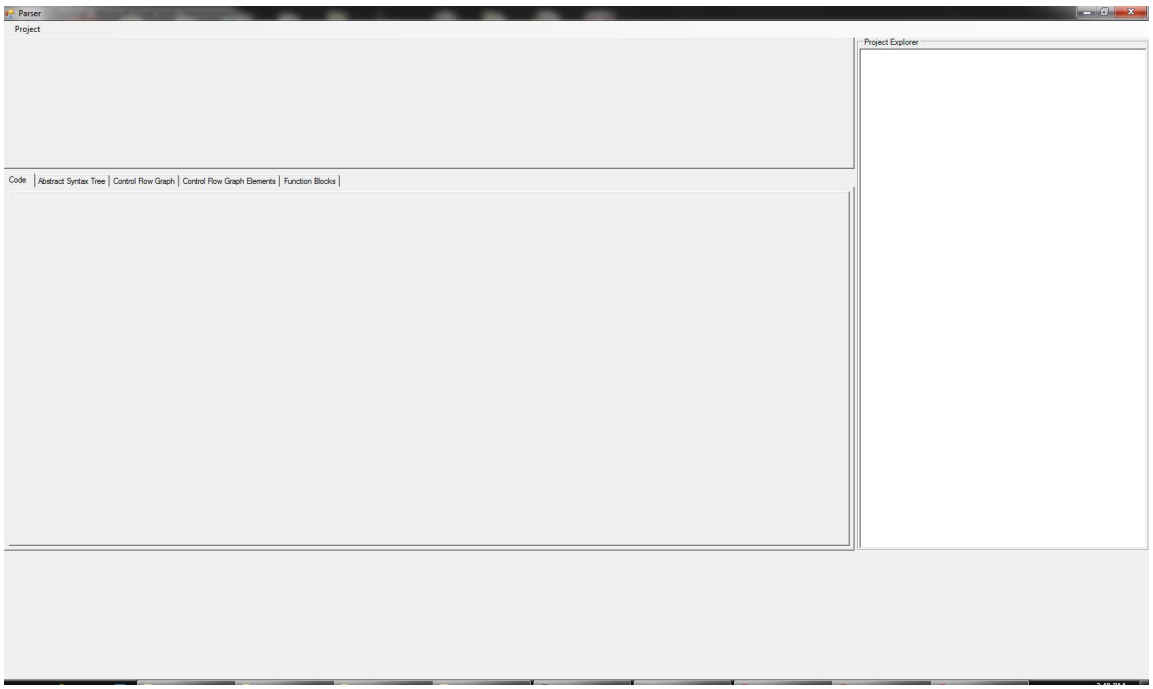
the rules which are to be analysed. Depending upon the user selection, the analyses are customised to give the necessary information of the errors.

4.8 ERROR HANDLER

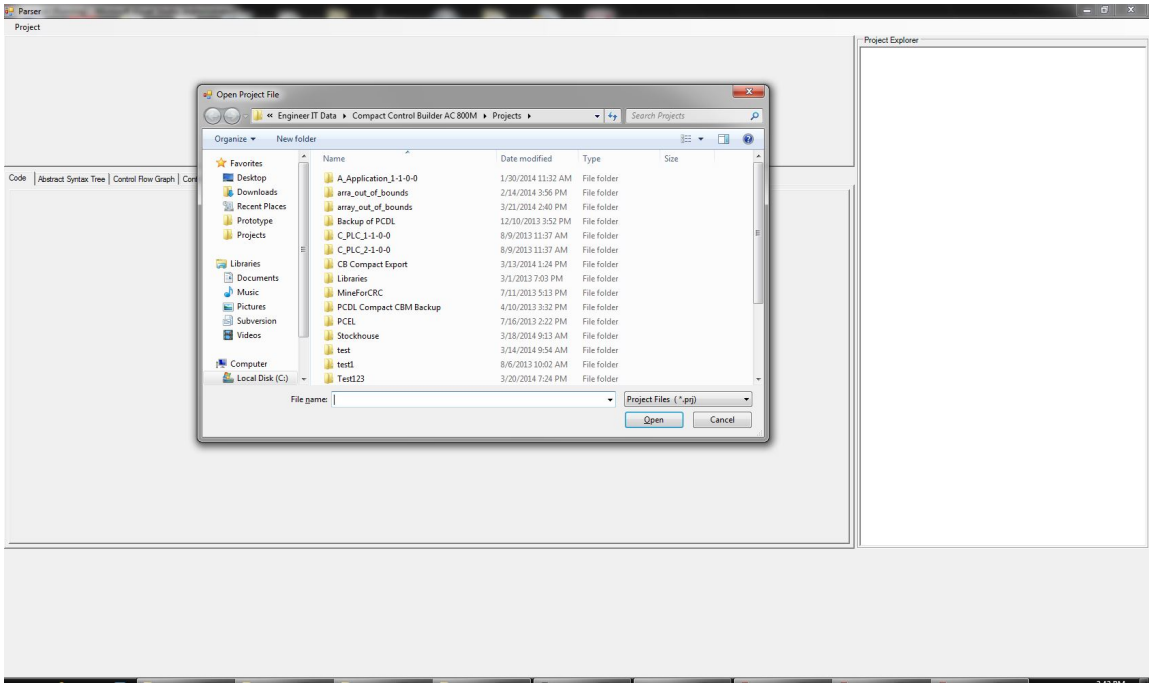
The error handler is implemented which takes care of the action to do after the error has been detected. It gives the details of the error and the source of it.

4.9 SCREEN-SHOT

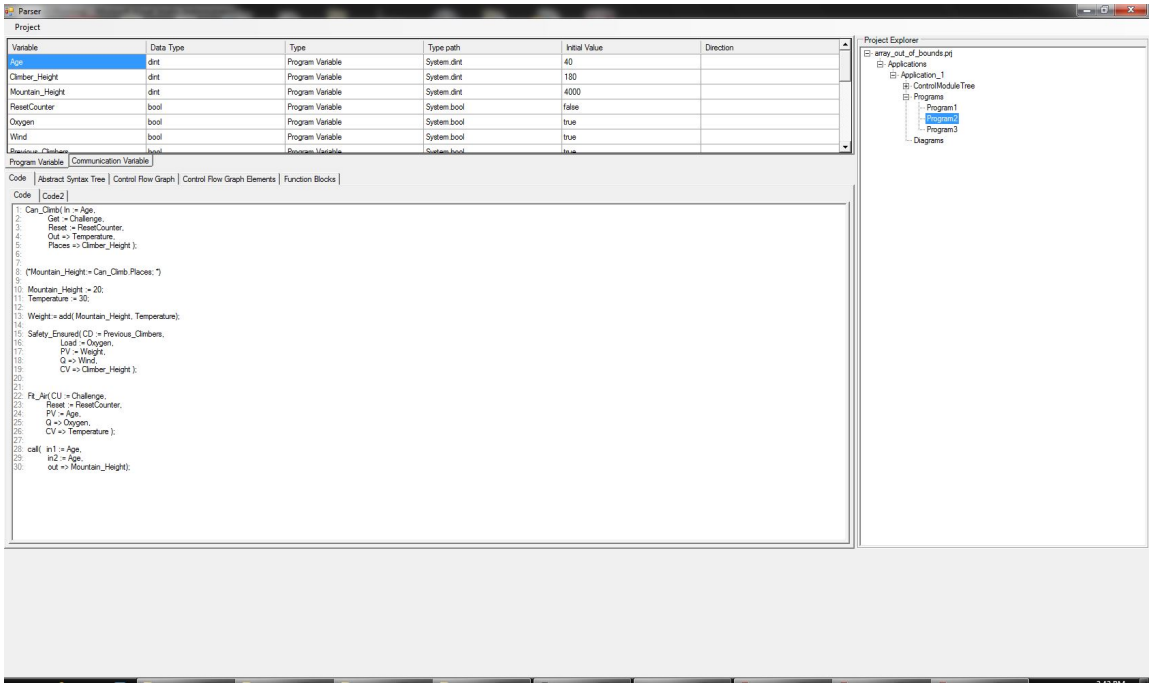
The prototype's user interface is designed using Windows forms UI and the Engine is written in C#. The first page of the application can be seen in Fig. 4.2a. When the user clicks on Project - Load menu item, a File Browser Window comes, from which the user can select the project which is to be loaded. Then the project is loaded in the prototype in the "Project Explorer" pane as in Fig. 4.2c. When the user clicks on the control modules or programs, the corresponding code will be shown on the code pane as in Fig. 4.2c and the details of the function blocks which are called from the code can be seen as in Fig. 4.2d. The corresponding AST for each code block can be seen in the AST tab as in Fig. 4.2e. The Control Flow Graph of the code can be viewed as in Fig. 4.2f and the details of the elements used to build the Control Flow Graph can be seen in Fig. 4.2g. When the user clicks on Project -¿ Analyze, a new window containing the rules appear as in Fig. 4.2h. The user can select the rule and when Analyze button is clicked, The analysis results can be seen as in Fig. 4.2i.



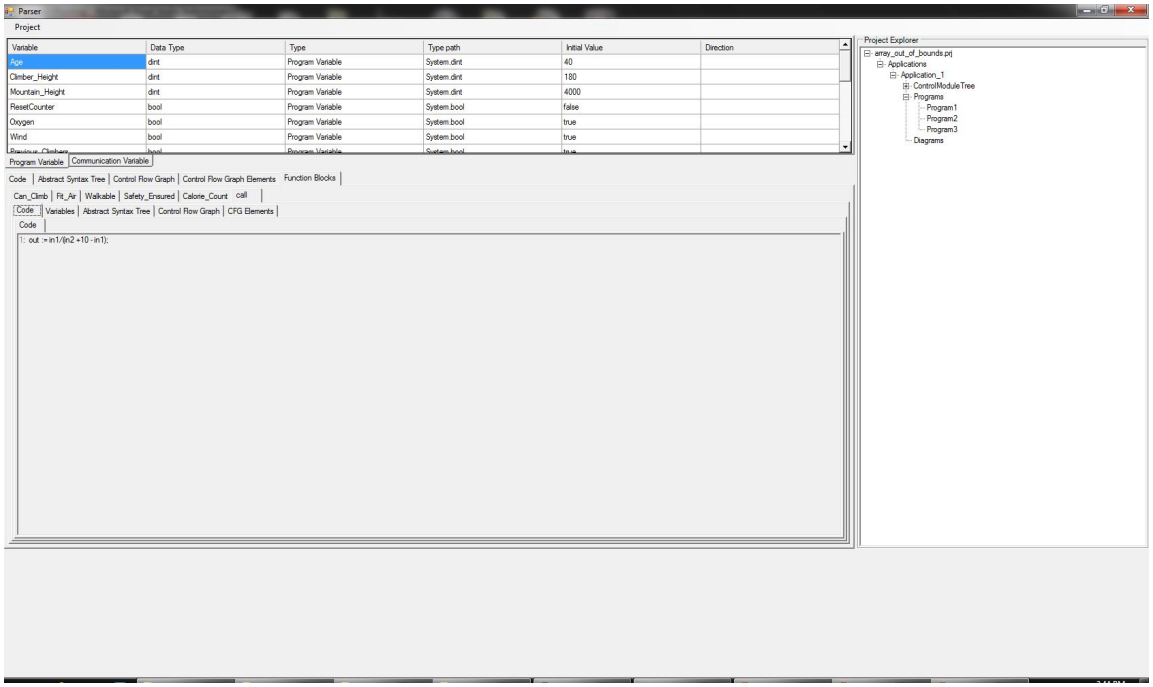
(a) The starting screen



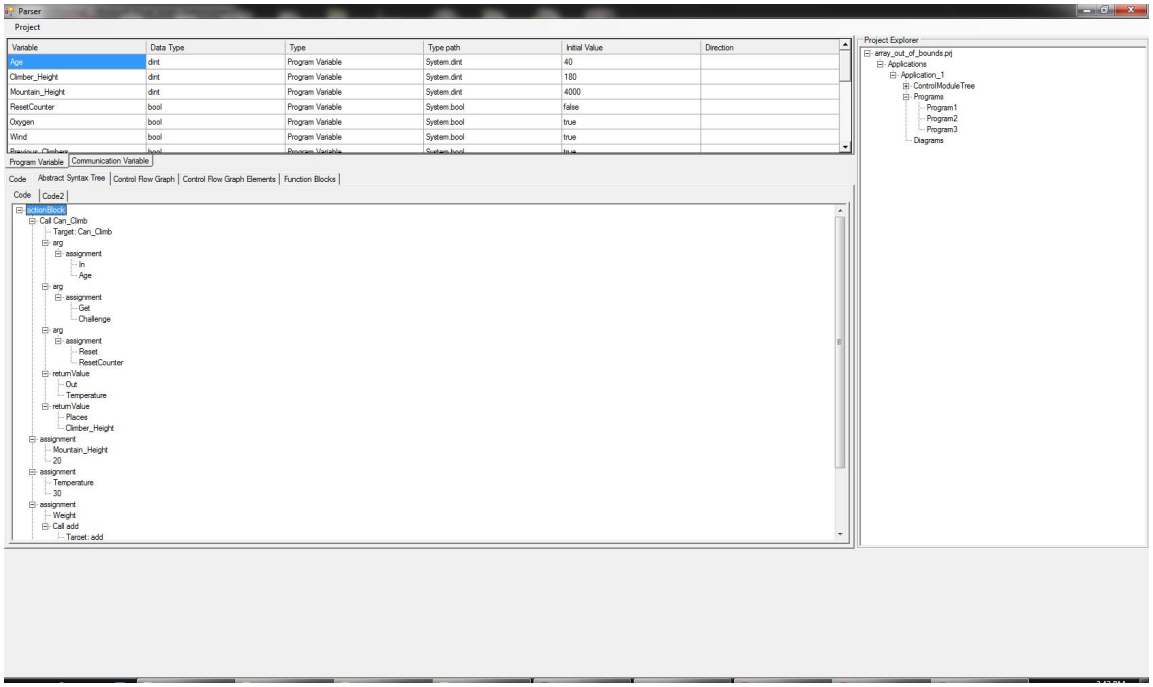
(b) The project loading



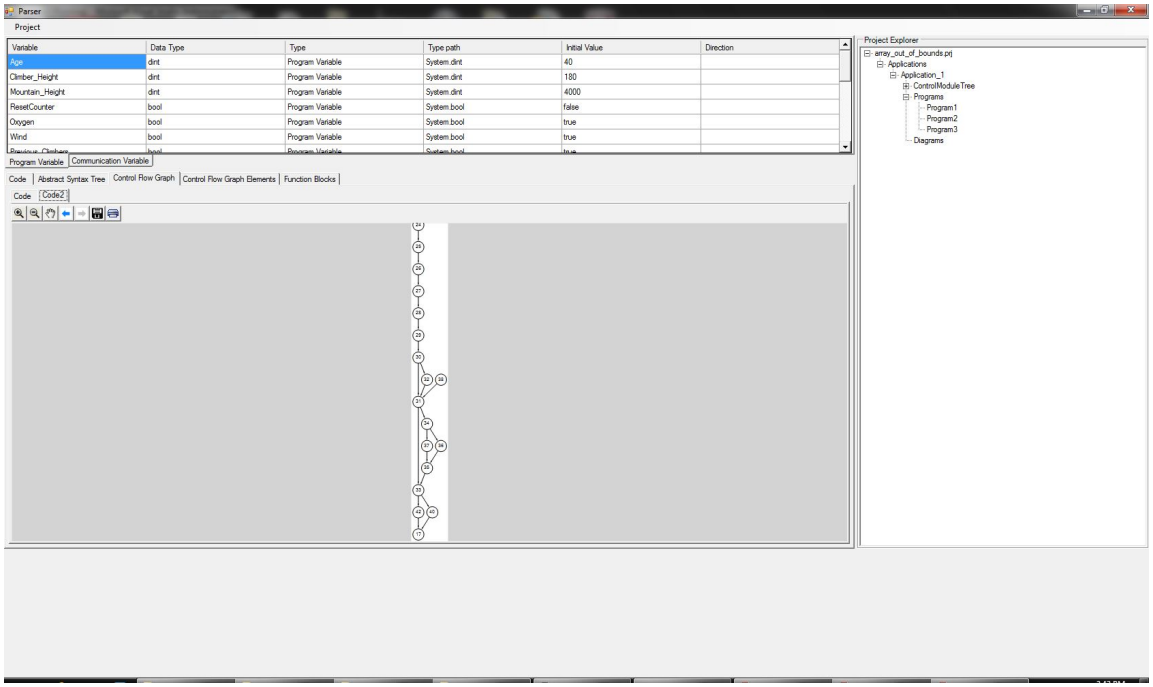
(c) The project structure and code



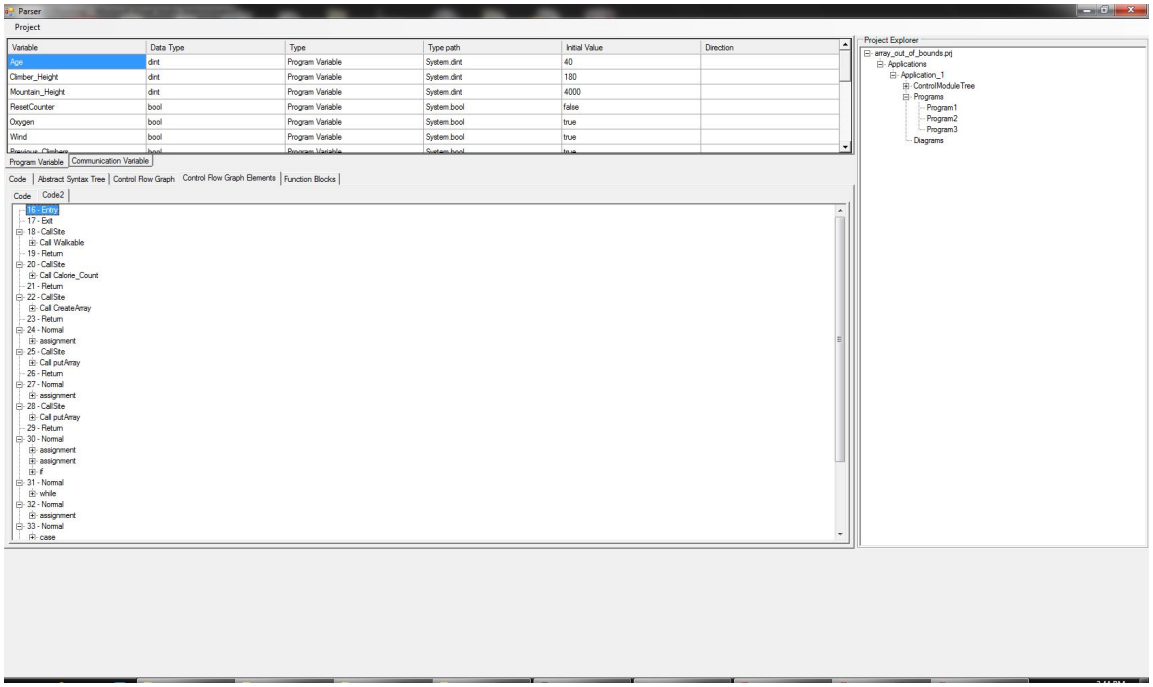
(d) The function block details called from the code



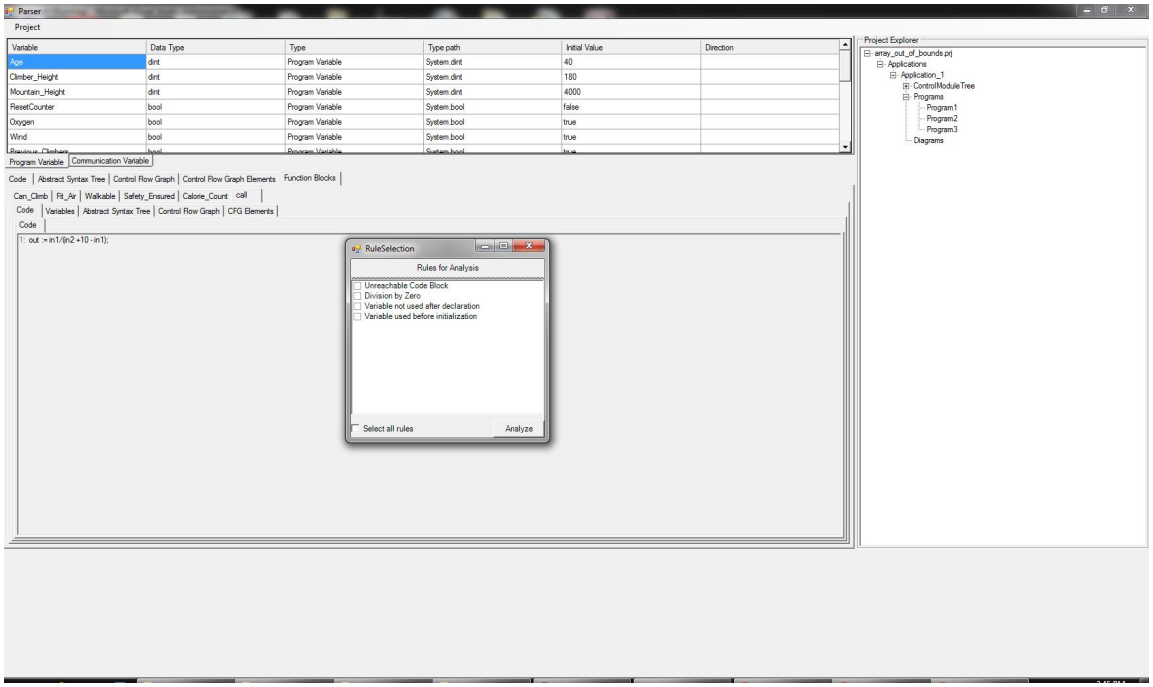
(e) The Abstract Syntax Tree



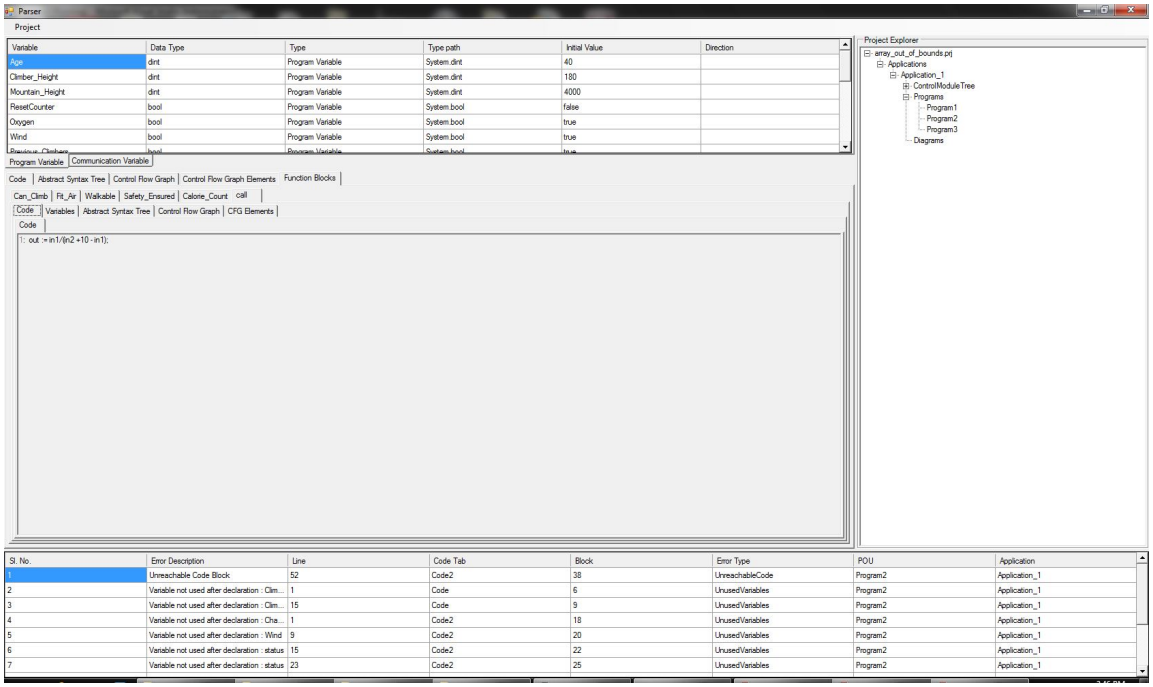
(f) The Control Flow Graph



(g) The elements of Control Flow Graph



(h) The Rule Selection Window



(i) The results of the analysis

Figure 4.2: The screenshots of the prototype developed

4.10 SUMMARY

The different stages of the data-flow analysis algorithm was explained in this chapter.

CONCLUSION

A prototype has been generated for Structured text programming language. A data-flow analysis is performed on the program to detect the probable run-time errors. The efficiency of the analyzer is seen in the results.

The prototype can be extended to handle the other four programming languages defined by the IEC 61131-3 standard. More error rules can be defined and the tool can be extended to handle them as well. A semantic analysis phase can be included after syntax analysis phase and the result can be used to generate syntax tree needed for control flow graph generation. Static analysis can be performed on the program using different program analysis methods other than data-flow analysis and the results can be compared.

4.11 RESULTS

The prototype has been tested for different programs of varying Lines of Code (LOC). The results are tabulated in table 4.1.

Sl. No.	LOC	Errors detected	Time taken(in seconds)
1	24	12	8.703
2	82	172	121.298
3	123	252	115.970

TABLE 4.1: Results

REFERENCES

[1] Francisco Hidrobo Olga Gonzalez Flor Narciso, Addison Rios-Bolivar, "A syntactic specification for the programming languages of the iec 61131-3 standard," in *Advances in Computational Intelligence, Man-Machine Systems and Cybernetics*.

[2] "Structured text manual from abb," August 2013.

[3] "Compact control builder ac 800m 5.1," September 2013.

[4] Motohiko Okabe, "Development of processor directly executing iec61131-3 language," in *SICE Annual Conference*, 2008, pp. 2215–2218.

[5] Mora M. Alvarez F. J. Rodriguez-Martinez, L.C., "A descriptive/comparitive study of process models of software development life cycles (pm-sdlcs)," *Mexican International Conference on Computer Science*, pp. 298–303, 2009.

[6] Hyatt L. Hammer T. Huffman L. Rosenberg, L. and W. Wilson, "Testing metrics for requirement quality," in *Eleventh International Software Quality Week, San Francisco, CA*, 1998.

[7] J. Christopher Westland, "The cost of errors in software development: evidence from industry," *Journal of Systems and Software*, vol. 62, no. 1, pp. 1–9, 2002.

[8] S. C. Johnson, "Lint, a c program checker," *Bell Laboratories*, pp. 78–1273, 1978.

[9] "Pmd for java," August 2013.

[10] William Pugh David Hovemeyer, "Finding bugs is easy," *ACM SIGPLAN Notices*, pp. 132–136, 2004.

[11] "Klockwork insight," August 2013.

[12] "Logi.lint," August 2013.

[13] "Arcade.plc," August 2013.

[14] Mario de Sousa, "Data-type checking of iec61131-3 st and il applications," *2012 IEEE 17th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1–8, 2012.

[15] Torben Aegidius Mogensen, *Basics of Compiler Design*, DIKU, University of Copenhagen.

[16] Jeffrey D.Ullman Monica S. Lam Alfred V.Aho, Ravi Sethi, *Compilers: Principles Techniques & Tools, second edition*, Addison Wesley, 2006.

[17] "Irony, https://irony.codeplex.com/," August 2013.

[18] "Microsoft glee, http://research.microsoft.com/en-us/downloads/f1303e46-965f-401a-87c3-34e1331d32c5/," August 2013.

[19] Seung-Cheol Jung Hyoung-Cheol Kim Oukseh Lee Kyung-Goo Doh Hyunha Kim, Tae-Hyoung Choi, "Applying dataflow analysis to detecting software vulnerability," *10th International Conference on Advanced Communication Technology*, pp. 255–258, 2008.

[20] Radhia Cousot Patrick Cousot, "Static determination of dynamic properties of programs," *Second International Symposium on Programming*, pp. 106–126, 1976.

[21] Flemming Nielson Hanne Riis Nielson and Chris Hankin, *Principles of Program Analysis*, Springer, 1999.

[22] Florian Martin, *Generating Program Analyzers*, Saarland University, June 1999.