**MCFromC Project Overview**

# Glossary

* AST – Abstract Syntax Tree – a tree data structure that contains the given source code as a syntax tree.
* CFG – Control Flow Graph
* MC – Monotonicity Constraints
* ACFG – Annotated Control Flow Graph is CFG that contains MCs on its edges.

# General Architecture Overview

The project architecture is based on stages where each stage takes an input, processes it, and outputs it as input to the next stage. **Figure 1** below shows a high level overview of the data flow between the system stages.

Figure 1 - System Stages

Output

CFG

CFG

AST

C Source

Syntax Parser

CFG Creation

CFG Simplifier

MC Creation

## Syntax Parser

This stage is responsible for converting a text file containing a subset of the  
**C programming language**[[1]](#footnote-1) into an AST that which will be processed into a CFG in the next stage. This stage is implemented using the following 3rd party tools:

* **Flex** – This tool uses **lex** file and produces source file that holds function that takes input stream and returns tokens according to the **lex** file.
* **Bison** – This tool is implementation for **YACC**. The tool receives a **YACC** input file (language grammar in a format similar to BNF, where each rule contains C code snippet) and generates a C source file which is the code of a parser for the given language.

Since the above tools produce a source files in the C language (and not C++), while the system main algorithms are coded in C++, this stage is implemented as a separate DLL with an interface which takes a file name and returns a pointer to the root node of the AST.

## CFG Creation

This stage is responsible for generating a CFG based on the AST input. The output is a CFG and all the program state variable names. The graph data structure implementation is based on a 3rd party library named **boost graph library** (**BGL**). The library is a C++ library, implemented in templates and contains only header files. This stage is implemented in a separate, C++ written, DLL named CFGGen. The algorithm for CFG generation based on the AST of the input program will be covered in a later section.

## CFG Simplifier

The CFG simplifier is a small stage, implemented in the same DLL as the previous stage (CFG Creation) and is responsible to removal of redundant flow points from the CFG. One example for such redundancy is appearance of consecutive expression flow points (basic block). Those flow points can be reduced into a single flow point since there are no branches within the block. The current stage aggregates those flow points into a single flow point that which represents and holds the information of all other expression flow points.

## MC Creation

This MC Creation layer is responsible for trying to evaluate changes in the values of the program state variables in each abstract transition of the program (represented as an edge in the CFG). The input for this stage is a CFG (simplified from the previous stage) and the names of all state variables (found by traversing the AST, during the CFG generation). Abstract Interpretation is used in order to track changes in the values of the program state variables. This stage will be covered in detail in **MC generation** section.

The following chart describes the normal flow of the application with regard to code projects that are involved.

Figure - Code Project Flow

# User Manual

## Installation

The installation of the project is pretty simple all you need to do is to perform double click on MCFromCSetup and proceed with the installation, Note that you have to be administrator in order to perform installation.

### Inspecting the Install Directory

Under the installation directory you will find the following items:

* Main project's binary outputs that includes the following: SyntaxParserDLL.dll,CFGGen.dll,CFGViewer.exe
* Zip file containing the project's source along with documentation.

## Usage

When launching the application you will see the following screen ()

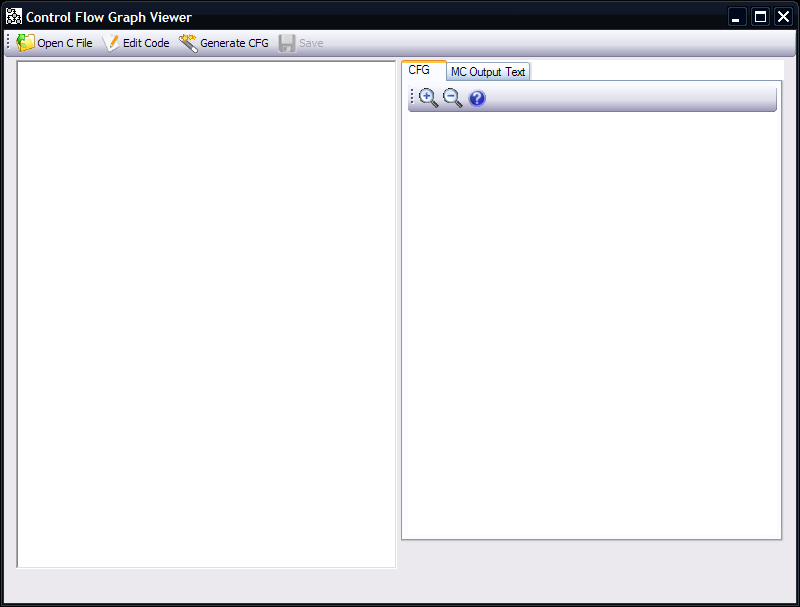


Figure - Main Application Window

### Generating CFG from C code

In order to generate a CFG from a "C" source you will need to provide the "C" source by either loading a source file:

1. Click on the "Open C File" that is located on the toolbar.
2. Select the "C" source file.

Or by editing the code manually:

1. Click on the "Edit Code" button that is located on the toolbar.

Then just click on the "Generate CFG" button that is located on the toolbar.

### Inspecting Results

After the Generating CFG you will provided with screen like the following screen

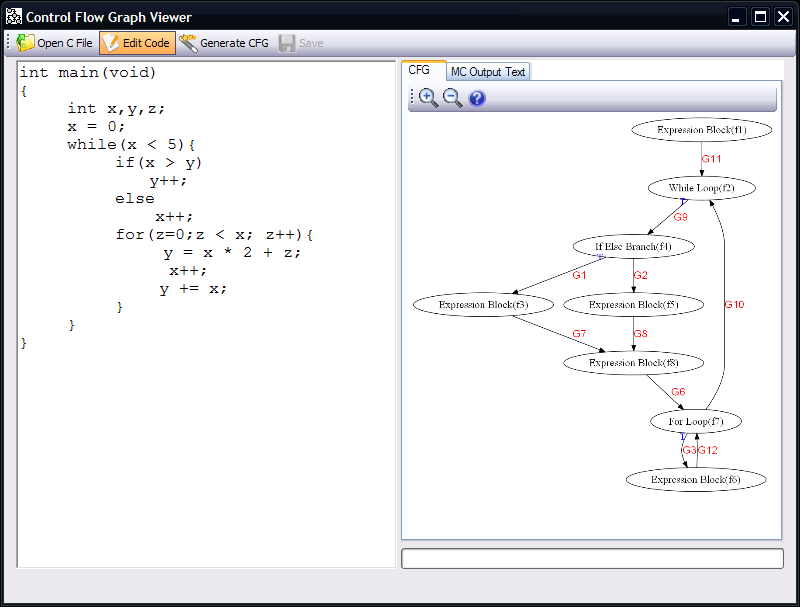


Figure - Result Window

As you can see the code input "C" source is on the left and the result C.F.G is on the right when you hover the mouse on a flowpoint (vertex in the CFG) you will see the code that caused this flowpoint to be created in CFG. In order to inspect a transition you can double click on the graph name that is located on the edge the represents the requested transition. This action will pop up a window similar to the following:

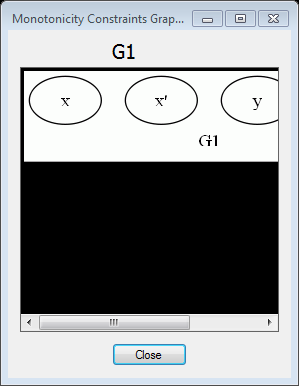


Figure - MC Inspection window

Now you can inspect how specific transition affects the program state variables. In case you would like you can inspect the result in "C. S. Lee MCS Format" (you can find the format in Appendix A: C. S. Lee MCS Format) by clicking the "MC Output Text" button that is located at the main application window's toolbar, you can also save this format by clicking on the "Save" button that is located at the main application window's toolbar.

# Algorithms

## Converting AST to CFG

This page describes the algorithm used for converting the Abstract Syntax Tree(AST), as received by the Syntax Parser, into a Control Flow Graph(CFG).

### Algorithm Details

*Input*: **AST** - an Abstract Syntax Tree.

*Output*: **CFG** - a Control Flow Graph, containing Flow Points and directed edges between them.

1. For each handled node in the AST (mainly expressions and control flow constructs), create a Flow Point in the CFG.
2. Merge consecutive commands Flow Points into a compound block - a block of commands which are consecutive in the program code. This block does not appear in the CFG.
3. Merge consecutive expressions Flow Points into a single Flow Point in the CFG(representing an expression block)
4. Recursively traverse the AST:
   1. Connect each node Flow Point to it's sub-nodes Flow Points.
   2. If current node is a loop node, connect all end-nodes Flow Points to it's Flow Point.
   3. If current node's Flow Point is part of a compound block, connect it's end points (might be the node's Flow Point itself) to the next Flow Point inside the compound block.
   4. Return current end points to the calling level.
5. If there are consecutive expression blocks, merge them into a single block.

### Remarks

A For loop of the following form:

  for ({Initialization}; {Condition}; {Increment})  
  {  
    {Body}  
  }

is interpreted into the following graph:

* Initialization --> Condition
* Condition --> Body
* Body --> Increment
* Increment --> Condition

## MC generation

### Description of how expressions are evaluated

### Algorithm

# Supported grammar (in BNF format)

# Appendix A: C. S. Lee MCS Format

**Syntax**

*MCS* ::= *MC* { *MC* } every rule on separate line

*MC* ::= *head* **':-'** *body* **';'** *tail*

*head, tail* ::= *FPid* **'('** *Vid* { , *Vid* } **')'**

*body* ::= **'['** *inequality* { , *inequality* } **']'**

*inequality* ::= *Vid* [ **'>'** | **'>='** | **'='** | **'<'** | **'<='** ] *Vid*

**Lexical Tokens:**

*FPid* flow-point identifier

*Vid* variable identifier

**Rules:**

Every *Vid* that appears in a rule's body has to appear either among the variables of the head or those of the tail.

# Benchmark programs and result graphs and MCs.

1. See **Supported grammar (in BNF format)** section for the exact language definition [↑](#footnote-ref-1)