ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 7, July 2014

Design of Software Inspection tool

Rakhee Kundu[1], Umesh Kulkarni[2]

Computer Engineering, ARMIET, Affiliated to Mumbai University, India[1]

Computer Engineering, VIT, Affiliated to Mumbai University, India[2]

Abstract: Although software inspection has led to improvements in software quality, many software systems continue to be deployed with unacceptable numbers of errors, even when software inspection is part of the development process. The difficulty of manually verifying that the software under inspection conforms to the rules is partly to blame. We describe the design and development of a tool designed to help alleviate this problem. The tool provides mechanisms for inspection of software by exposing the results of sophisticated whole-program static analysis to the inspector. The tool computes many static-semantic representations of the program, forward and backward slicing and dependence factors. Whole-program pointer analysis is used to make sure that the representation is precise with respect to aliases induced by pointer usage. Views on the dependency and related representations are supported. Queries on the dependence graph allow an inspector to answer detailed questions about the semantics of the program. Facilities for openness and extensibility permit the tool to be integrated with many software-development processes. The main challenge of the approach is to provide facilities to navigate and manage the enormous complexity of the dependence graph. Which will test the correctness of the program by identifying some of the rules .Whether particular variable in the program is working or malfunctioning, Checking the malfunctioning by the dependency factors by using backward and forward slicing. This will identify the checkpoints and not to identify the errors and which area a particular checkpoint is getting effected will be reflected.

Keywords: Abstract Syntax Tree, Program Dependence Graph (PDG), Predecessor, Slicing, Successor.

I. INTRODUCTION

Software inspection is a technique for detecting problems in software early in the lifecycle. It was introduced by Fagan in 1976 [16] and, since then, it has attracted support as a software engineering best practice. A key phase in the software-inspection process is when the inspectors attempt to find defects by scrutinizing the code in detail. Often, a team will have a checklist of generic and domain-specific rules that must be followed, and the team's task is to find violations of those rules. For example, in a checklist used at NASA for programs written in C [32], one generic rule is "Does code that writes to dynamically allocated memory via a pointer first check for a valid (nonzero) pointer?"Unfortunately, it can be very difficult to manually find violation of this kind of rule. It may be easy to find violations for small programs, but, even for moderately sized programs with multiple pointer indirections, the complexity can quickly thwart manual attempts at under-standing. This paper describes the design and implementation of a tool for helping inspectors navigate this complexity, by providing a means for a user to reason about the deep structure of the code at a high level o detail This tool, named CodeSurfer TM , provides access to and answers queries about—a range of different representations of a program, all created by performing advanced static analysis on the program. These representations go far beyond those providedby traditional program-browsing tools and include an accurate call graph, the results of whole-program pointer analysis, and the program's system dependence graph.

This project is the design and implementation of a C program inspection tool for helping inspectors navigate this complexity, by providing a means for a user to reason about the deep structure of the code at a high level of detail. This tool aims slicing as a main ingredient for software inspection provides access to and answers queries about—a range of different representations of a program, all created by performing advanced static analysis on the program. These representations go far beyond those provided by traditional program-browsing tools and include the program dependence graph. The standard queries on the program dependence graph such as predecessors and successors, slicing backward and forward are of much use in program understanding.

This project describes a language-independent program representation—the *program dependence graph* and discusses how program dependence graphs, together with operations such as program slicing, can form the basis for powerful programming tools that address the problems listed above.

II. QUERIES FOR SOFTWARE INSPECTION

Many of the features of have been designed to aid program understanding and, as such, can be useful for detailed software inspection. This section describes some of the queries available and their application to software inspection.

A. Variable-Usage Information

Each point in the program may access some variables and modify other variables, each possibly through pointers. In order to create the data-dependence graph, the set of variables used and defined at each program point are first computed and associated with the vertex that represents that program point.

This information is easily accessed by the user

ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 7, July 2014

B. *Predecessors/Successors*

It is natural for a user attempting to understand a program to ask "How could variable x have gotten its value here?" or alternatively "Where is the value generated at this point used next?" The predecessors and successors operations provide the answers to these questions.

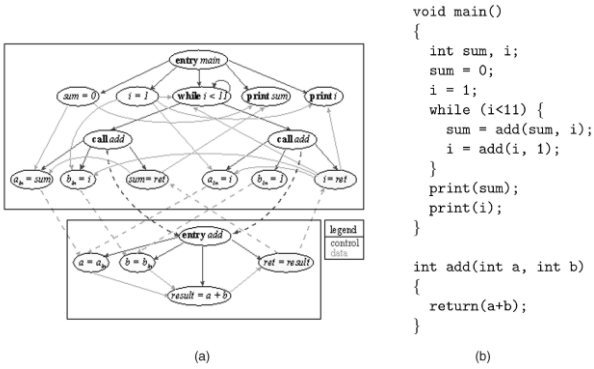
These queries can be posed with respect to the control dependences, the data dependences, or both. A program point's data predecessors are the points where the variables used at that point may have gotten their values. The data successors are the points where the variables that were modified at that point are used.

The fact that the query is performed directly on the dependence graph guarantees that the result will be correct with respect to the dataflow properties of the program.

III. DEPENDENCE GRAPHS

Dependence graphs have applications in a wide range of activities, including parallelization , optimization , reverse engineering, program testing , and software assurance . Fig. 1 shows the dependence-graph representation for a simple program with two procedures. This section briefly describes dependence graphs and how they are built.

A Program Dependence Graph (PDG) is a directed graph for a single procedure of a program. The vertices of the graph represent constructs such as assignment statements, call sites, parameter ,and condition branches.




```
void main()
{
  int sum, i;
  sum = 0;
  i = 1;
  while (i<11) {
    sum = add(sum, i);
    i = add(i, 1);
  }
  print(sum);
  print(i);
}

int add(int a, int b)
{
  return(a+b);
}
```

Figure 1. Program Dependance Graph

An edge between the vertices indicates either a data dependence or a control dependence. The data-dependence edges indicate possible ways in which data values can be transmitted. For example, in Fig. 1, there is a data dependence edge between the vertex for i=1 and the vertex for while (i < 11), which indicates that a value for i may flow between those two vertices.

A control-dependence edge between a source vertex and a destination vertex indicates that the result of executing the source vertex controls whether or not the destination vertex is reached. For example, in Fig. 1, there is a control dependence edge between the vertex for while (i<11) and the vertices for the two call sites on the function add. A System Dependence Graph (SDG) is a directed graph consisting of interconnected PDGs , one per procedure in the program. Inter procedural control-dependence edges connect procedure call sites to the entry points of called procedures. Inter procedural data-dependence edges represent the flow of data between actual parameters and formal parameters (and return values). Nonlocal variables, such as global, file statics, and variables accessed indirectly through pointers, are handled by modeling the program as if those variables are passed in and out as parameters to the program's procedures. Each nonlocal variable used in a function, either directly or indirectly, is treated as a "hidden" input parameter and, thus, gives rise to additional program points. These serve as the function's local working copy of the nonlocal variable. If the variable is modified in the function, then it has an associated output parameter as well.

IV. DEPENDANCE GRAPH QUERIES

A number of types of queries on the dependence graphs can be issued. The backward slice from a program point P returns all points that may influence whether control reaches P and all points that may influence the values of the variables used at P when control reaches P. The forward slice from P includes all program points affected by an assignment or branch performed at P . A program chop between a set of source program points S and a set of target program points T returns the set of program elements that can transmit effects from S to T (and, hence, reveals how S can affect the state of the program at T). These query algorithms can be implemented safely using simple graph reachability. However, they can be greatly improved by filtering out answers that correspond to certain infeasible executions of the program. In particular, a path that enters a procedure through a call site can only exit the procedure by going back to the call site from where it came. We refer to queries on the dependence graph as being inter procedurally precise if they accurately model the call-and-return semantics of procedure calls. A path p between two vertices s and t is only considered to be a valid connection between s and t if the word spelled out by concatenating the labels on the edges is a word in L. It is a simple matter to define a context-free grammar that models the call-and-return semantics of a valid execution path of a program. Let each call site in the program be given a unique index ranging from 1 through N. Let each inter procedural edge leaving from call site i be labeled .i, and each inter procedural edge returning to call site i be labeled .i. Let all other edges be labeled x.

V. REPRESENTATIONAL APPROACH OF THE MODULES

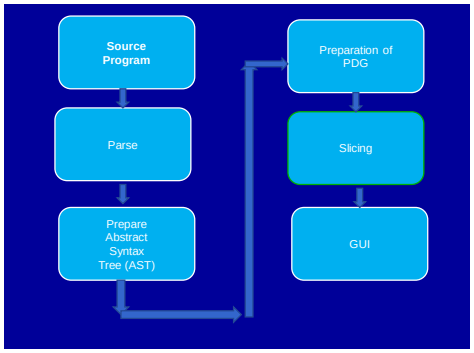


Figure 2. Representational Approach Of Modules

ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 7, July 2014

As shown in the figure 2 above a source program which is to be inspected is given as an input using GUI, the program is parsed and abstract syntax tree is constructed. The PDG is generated to understand the main flow of the program and then by slicing the program using forward, backward , predessor or successor approach the code is segmented for further analysis. Then CFG algorithm is applied on it to obtain dominator tree and post dominator tree . The control dependence graph is constructed to undertstand the dependancing of a particular variable in the entire program and its linkage with other functions and methods. Using this tool it will be easy to find out bugs in the program and their influence on the program control flow will be understood.

VI. SYSTEM ANALYSIS AND DESIGN

The following figure shows the working nature of the system.

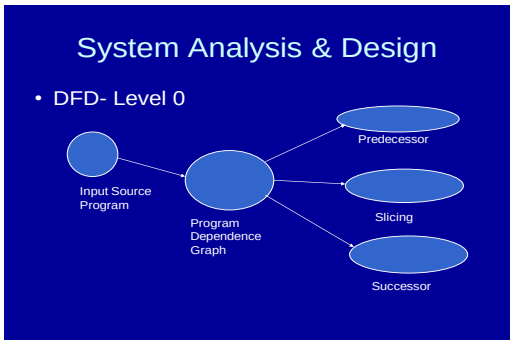


Figure 3. Top Level View Of The System

This figure 3 describes the top level view of the system. That is how the system is going to deal with the source code provided to it. First the input is analyzed to produce the intermediate representation in the form of the graph and than subsequent operations is carried out on this representation to produce the result.

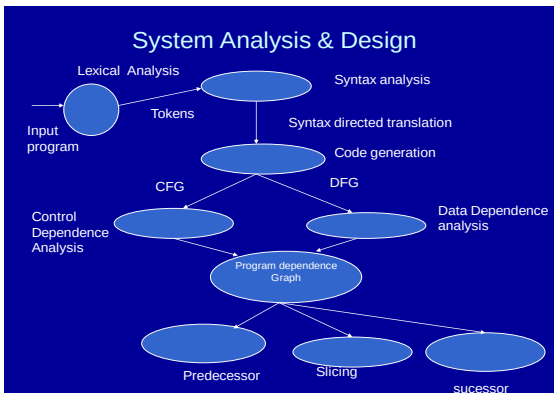


Figure 4. System Analysis and Design

VII. ALGORITHMS AND RELATED THEORY

A. Computation of Basic Blocks

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. We can construct the basic blocks for a program using algorithm GetBasicBlocks, shown in Figure 1. When we analyze a program's intermediate code for the purpose of performing compiler optimizations, a basic block usually consists of a maximal sequence of intermediate code statements. When we analyze source code, a basic block consists of a maximal sequence of source code statements. We often find it more convenient in the latter case, however, to just treat each source code statement as a basic block.

Algorithm GetBasicBlocks
Input. A sequence of program statements.
Output. A list of basic blocks with each statement in exactly one basic block.
Method.

(1) Determine the set of *leaders*: the first statements of basic blocks. We use the following rules.
a) The first statement in the program is a leader.
b) Any statement that is the target of a conditional or an unconditional goto statement is a leader.
c) Any statement that immediately follows a conditional or an unconditional goto statement is a leader.

(2) Construct the basic blocks using the leaders. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

B. Computing Control Flow Graph

A *control flow graph* (CFG) is a directed graph in which each node represents a basic block and each edge represents the flow of control between basic blocks. To build a CFG we first build basic blocks, and then we add edges that represent control flow between these basic blocks.

After we have constructed basic blocks, we can construct the CFG for a program using algorithm GetCFG, shown in Figure The algorithm also works for the case where each source statement is treated as a basic block.

To illustrate, consider Figure 3, which gives the code for program Sums on the left and the CFG for Sums on the right. Node numbers in the CFG correspond to statement numbers in Sums: in the graph, we treat each statement as a basic block. Each node that represents a transfer of control (i.e., 4 and 7) has two labeled edges emanating from it; all other edges are unlabeled.

In a CFG, if there is an edge from node *Bi* to node *Bj* , we say that *Bj* is a *successor* of *Bi* and that *Bi* is a *predecessor* of *Bj* . In the example, node 4 has successor nodes 5 and 12, and node 4 has predecessor nodes 3 and 11.
Algorithm GetCFG

Input. A list of basic blocks for a program where the first block (*B*1) contains the first program statement.
Output. A list of CFG nodes and edges.
Method.

1. Create *entry* and *exit* nodes; create edge (entry, *B*1); create edges (*Bk*, exit) for each basic block *Bk* that contains an exit from the program.

ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940
International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 7, July 2014

2. Traverse the list of basic blocks and add a CFG edge from each node *Bi* to each node *Bj* if and only if *Bj* can immediately follow *Bi* in some execution sequence, that is, if:
(a) there is a conditional or unconditional goto statement from the last statement of *Bi* to the first statement of *Bj*, or
(b) *Bj* immediately follows *Bi* in the order of the program, and *Bi* does not end in an unconditional goto statement.

```
int main( ) {
   int sum = 0 ;
   int i = 1 ;
   while ( i < 11 ) {
        sum = sum + i ;
        i = i + 1 ;
   }
   printf("%d\n", sum) ;
   printf("%d\n",i) ;
}
```

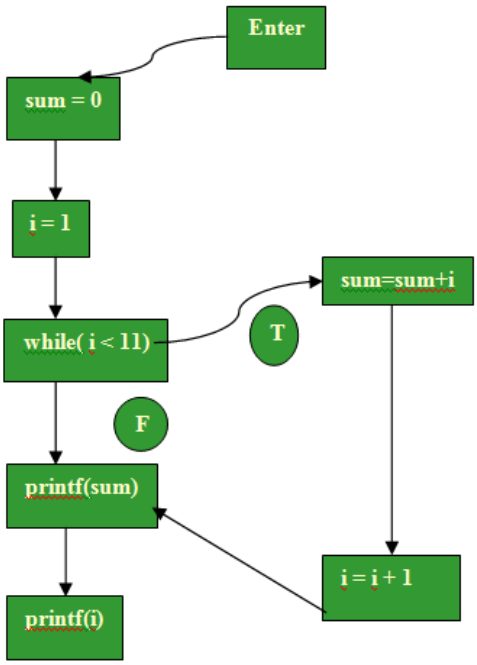


Figure 5. Sample Control Flow Graph

C. Computing Dominator Tree

A node D in CFG G *dominates* a node W in G if and only if every directed path from entry to W (not including W) contains D. A *dominator tree* is a tree in which the initial node is the entry node, and each node dominates only its descendants in the tree.

Figure gives an algorithm, ComputeDom, for computing dominators for a control flow graph *G*. A key to this algorithm is step 3, where, for each node *n* except the entry node, we initialize the set of dominators to *the set of all nodes in G*. We then iterate through the nodes (except the entry node), and for each node *n*, at step 3, we use the intersection operator to *reduce* the set of nodes listed as dominating *n* to those that actually dominate predecessors of *n*. Thus, we start with an overestimate of the dominators and reduce the sets to get the actual set of dominators

Algorithm ComputeDom
Input. A control flow graph *G* with set of nodes *N* and initial node *n*0.
Output. D(*n*), the set of nodes that dominate *n*, for each node *n* in *G*
Method. Use an iterative approach similar to the data flow analysis algorithm ReachingDefs

1. $D(n0) = \{n0\}$
2. for each node *n* in $N - \{n0\}$ do $D(n) = N$
3. while changes to any $D(n)$ occur do
4. for *n* in $N - \{n0\}$ do
5. $D(n) = \{n\} \cup (\cap D(p))$ for all immediate predecessors *p* of *n*
6. endfor
7. endwhile

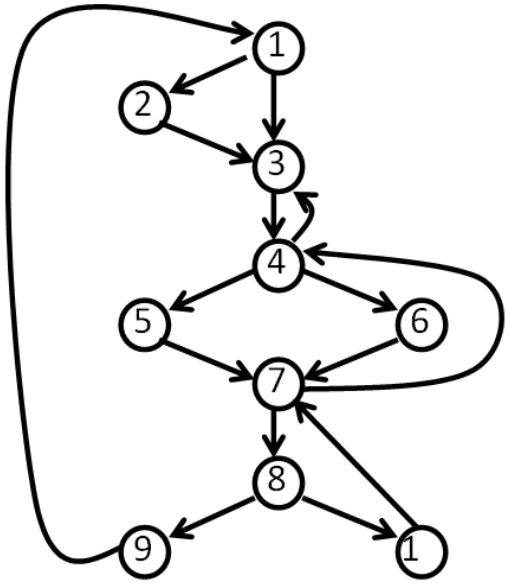


Figure 6. Control Flow Graph

VII. CONTROL FLOW GRAPH (CFG)

A control flow graph describes the sequence in which the different instructions of a program get executed. In other words, a control flow graph describes how the control flows through the program. In order to draw the control flow graph of a program, all the statements of a program must be numbered first. The different numbered statements serve as nodes of the control flow graph . An edge from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. The CFG for any program can be easily drawn by knowing how to represent the sequence, selection, and iteration type of statements in the CFG. After all, a program is made up from these types of statements. Fig.7 summarizes how the CFG for these three types of statements can be drawn. It is important to note that for the iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore the control flow from the last statement of the loop is always to the top of the

ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 7, July 2014

loop. Using these basic ideas, the CFG of Euclid's GCD computation algorithm can be drawn as shown in fig.8 below

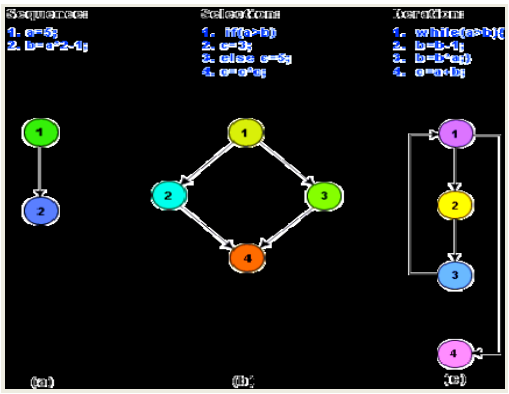


Figure 7 CFG for (a) sequence, (b) selection, and (c) iteration type of constructs

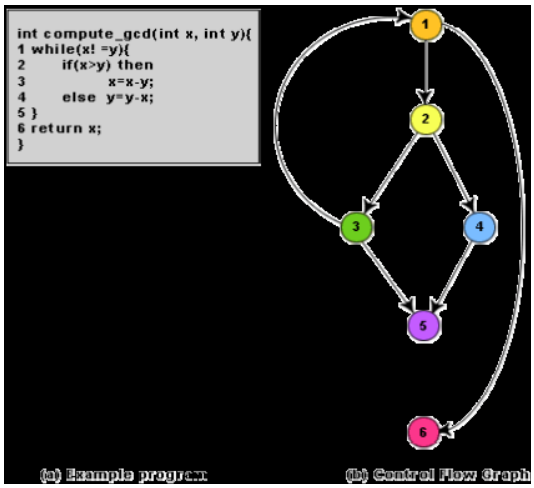


Figure 8. Control flow diagram

A. *Path*

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. There can be more than one terminal node in a program. Writing test cases to cover all the paths of a typical program is impractical. For this reason, the path-coverage testing does not require coverage of all paths but only coverage of linearly independent paths.

B. *Linearly independent path*

A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because, any path having a new node automatically implies that it has a new edge. Thus, a path that is subpath of another path is not considered to be a linearly independent path.

C. *Cyclomatic complexity*

For more complicated programs it is not easy to determine the number of independent paths of the program. McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe's cyclomatic complexity is very simple to compute. Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program. Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.

VIII. BACKWARD SLICING

A backward slice with respect to a set of starting points S answers the question "What points in the program does S depend on?" The control-dependence edges are used to determine how control could have reached S, and the data dependence edges are used to determine how the variables used at S received their values.

```
int main( ) {
   int sum = 0 ;
   int i = 1 ;
   while ( i < 11 ) {
   sum = sum + i ;
    i = i + 1 ;
   }
   printf("%d\n", sum) ;
printf("%d\n", i ) ;
   }
```

Backward slice from: printf("%d\n", i) ; is given by the program subset that may affect variable i in underlined printf();

```
int main( ) {
   int sum = 0 ;
   int i = 1 ;
   while ( i < 11 ) {
        sum = sum + i ;
        i = i + 1 ;
   }
   printf("%d\n", sum) ;
   printf("%d\n",i) ;
}
```

IX. FORWARD SLICING

A forward slice with respect to a set of starting points S answers the question "What points in the program depend on S?" In this also we make use of control and data dependence edges.

```
int main( ) {
   int sum = 0 ;
   int i = 1 ;
   while ( i < 11 ) {
  sum = sum + i ;

i= i+1;
 }
   printf("%d\n", sum) ;
   printf("%d\n",i) ;
}
```

Forward slice from: sum = 0 is given by the program subset that may affected by variable sum in int sum =0 statement

```
int main( ) {
   int sum = 0 ;
```

ISSN (Online) : 2278-1021
ISSN (Print) : 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 3, Issue 7, July 2014

```
   int i = 1 ;
   while ( i < 11 ) {
        sum = sum + i ;
        i = i + 1 ;
   }
   printf("%d\n", sum) ;
   printf("%d\n",i) ;
}
```

A. Predecessors:

It is natural for a user attempting to understand a program to ask "How could variable x have gotten its value here?" This query can be posed with respect to the control dependences, the data dependences, or both. A program point's data predecessors are the points where the variables used at that point may have gotten their values.

```
 1 int main( ) {
 2  int sum = 0 ;
 3  int i = 1 ;
 4 while ( i < 11 ) {
 5        sum = sum + i ;
  6       i = i + 1 ;
  7   }
  8  printf("%d\n", sum) ;
  9  printf("%d\n",i) ;
  10  }
```

The predecessors of variable sum at line no 8 is given by

```
 1  int main( ) {
 2  int sum = 0 ;
 3  int i = 1 ;
 4  while ( i < 11 ) {
 5        sum = sum + i ;
  6       i = i + 1;
  7   }
  8   printf("%d\n", sum) ;
  9   printf("%d\n",i) ;
  10  }
```

B. *Successors*:

It is natural for a user attempting to understand a program to ask "Where is the value generated at this point used next?" This query can be posed with respect to the control dependences, the data dependences, or both. A program point's data successors are the points where the variables that were modified at that point are used.

```
1.          int main( ) {
2  int sum = 0 ;
 3  int i = 1 ;
 4 while ( i < 11 ) {
 5        sum = sum + i ;
  6       i = i + 1 ;
  7   }
  8  printf("%d\n", sum) ;
  9  printf("%d\n",i) ;
  10  }
```

The successors of variable sum at line no 2 is given by

```
 1  int main( ) {
 2  int sum = 0 ;
 3  int i = 1 ;
 4  while ( i < 11 ) {
 5        sum = sum + i ;
  6       i = i + 1;
  7   }
  8   printf("%d\n", sum) ;
  9   printf("%d\n",i) ;
  10  }
```

X. IMPLEMENTATION

The whole system is arranged in the package called **project,** this package contains all the necessary files needed source code and the documentation of the project. The directory contains the two more directories one contains the GUI related code and other contains the back end source code.

A. The Application Package

This package contains the source code and the necessary make file to compile the source code to produce executable of the GUI program. The code is produced with the help of the QT Designer in C++. The application is the gui which offers all the features of the general purpose text editors. It contains the menu bar which has the File, Help and Tools as the main menus. The New, Open, Save, Save As are the common drop down menus, and in Tools the Slice, Predecessors, Successors, Formatted C code are the drop down menus. The menus are implemented as the components provided by QT designer. The dialog boxes for taking the input from the user are provided using the components provided by the QT designer and C++

The input information collected from the user is outputted in the file called "input.txt" which analyses the C program. Depending on the choice of the user selected the GUI program and the back end which analyses the C program. Depending on the choice of the user selected the GUI program invokes the back end program with appropriate arguments. The back and then performs the appropriate operations depending on the arguments supplied to it and writes the result back to the "output.txt" which is then read and displayed by the GUI program.

XI. CONCLUSION

We have described a tool for inspecting and manipulating the Control flow graph representation of a program for the purposes of program understanding and discussed how it can be used for software inspections. We have described the means by which the system answers queries about the dataflow properties of the program using context-free language graph reachability.

We have described using a model checker to answer questions about possible paths through the program. There are two main thrusts in its development. The first is we have improved the scalability of the system. This is achieved partly by using demand-driven techniques to reduce the up-front cost of building the dependence graph. The other thrust is we have extended the domain of applications for the system.

We can make any source program efficient by minimizing the dependency graph. The Future scope of this project is one can apply the technology to software assurance, and to program-testing problems.

XII. REFERENCES

[1] L.O. Andersen, "Program Analysis and Specialization for the C Programming Language," PhD thesis, DIKU, Univ. of Copenhagen,May 1994.
[2] T. Ball and S.K. Rajamani, "Bebop: A Symbolic Model Checker for Boolean Programs," Proc. SPIN Workshop, pp. 113-130, 2000.
[3] S. Bates and S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," Proc. Symp. Princples of Programming Languages, pp. 384-396, 1993.
[4] P. Bishop, R. Bloomfield, S. Guerra, and T. Clement, "Software Criticality Analysis of COTS/SOUP," Proc. Safecomp 2002, Sept. 2002.
[5] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization," Proc. SIGPLAN '86 Symp. Compiler Construction,pp. 162-175, 1986.
[6] Bell Canada, http://www.iro.umontreal.ca/labs/gelo/datrix, 2001.
[7] E.M. Clarke, M. Fujita, P.S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, "Program Slicing of Hardware Description Languages,Proc. Conf. Correct Hardware Design and Verification Methods (CHARME '99), Sept. 1999.
[8] [8] E.M. Clarke, O. Grumberg, and D.A. Peled, Model Checking. MIT Press, 1999.
[9] K.D. Cooper and K. Kennedy, "Interprocedural Side-Effect Analysis in Linear Time," Proc. ACM SIGPLAN 88 Conf. Programming Language Design and Implementation, pp. 57-66, June 1988.
[10] J.R. Cordy, C.D. Halpern, and E. Promislow, "TXL: A Rapid Prototyping System for Programming Language Dialects," Computer Languages, vol. 16, no. 1, pp. 97-107, Jan. 1991.
[11] D.E. Denning and P.J. Denning, "Certification of Programs for Secure Information Flow," Comm. ACM, vol. 20, no. 7, pp. 504-513, July 1977
[12] J. Drake, V. Mashayekhi, J. Riedl, and W. Tsai, "A Distributed Collaborative Software Inspection Tool: Design, Prototype, and Early Trial," Technical Report TR-91-30, Univ. of Minnesota, Aug. 1991.
[13] A. Dunsmore, "Comprehension and Visualisation of Object-Oriented Code for Inspections," Technical Report EFoCS-33-98, Computer Science Dept., Univ. of Strathclyde, 1998.
[14] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," Proc. Fourth Symp. Operating Systems Design and Implementation, pp. 1-16, Oct. 2000.
[15] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient Algorithms for Model Checking Pushdown Systems," Computer Aided Verification, pp. 232-247, 2000.
[16] M. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," IBM Systems J., vol. 15, no. 3, pp. 182-211, 1976