

Chapter 1

Introduction

The concept of COBSlicer extends its root to the very first citation of its kind that appeared in 1984, given by Mark Weiser on Program Slicing. Slice, in layman's terms is set of program statements. This set is computed from original program, based on some criteria. Before plunging into technical approaches to Slice, let us get familiarized with applications of slices.

Large number of domains deploy concept of slice to achieve different objectives. To name a few, Program Differencing, Program Integration, Testing, Debugging, Software maintenance, etc. Program differencing is to identify what new functionalities have been added to new version of software as compared to old version, or to identify semantic differences between two programs. Slices of both programs are calculated for each statement, and are compared. If statement in new version has isomorphic slice with some statement in old version, then those statements exhibit same behavior. If no such statement is found in old, then that statement of new is considered as added functionality.

Program integration is a serious issue faced by every industry developing and launching software in versions. Suppose two different versions of same base software have been developed. Later functionalities of those two need to be merged with base version in a way that they do not interfere with each other. In such cases it is necessary to use program differencing between two versions to determine base components and version specific components.

Software maintenance and Testing also make great use of slicing. Versioning and combining different new functionalities with original software over time is an integral part of maintenance. Regression testing is an expensive operation, and many solutions have been suggested over time to make it efficient. Slice technique can be used to determine parts of program that have been added new to original program and this information can then be used to determine only those test cases that affect these newly added parts. Thus,

reducing number of test cases to be executed and hence the cost of testing.

Debugging can be accomplished more effectively using slicing. From point of unexpected outcome we take backward slice, which computes all program statements affecting current outcome. Such outcome reduces the program space, to be searched, while debugging.

Over the time many types of slice have been proposed. Two major categories based on input context are Static Slice and Dynamic Slice. Slice can also be categorized as Executable and Non-Executable Slice. Dynamic Slice considers one particular execution scenario and accordingly produces slice, while static slice considers all possible execution paths. Since slice is set of program statements, some of these set can be executed independently and are known as Executable Slices. Furthermore, Slices could be either Data, Control or Program Slices. These could be computed either in backward direction or in forward direction.

Technically, Slice was described by Mark Weiser in 1984 as decomposition of original program based on Data Flow and Control Flow analysis. He defined slicing criteria as $\langle \text{line_number}, \text{set_of_variables} \rangle$. "line_number" defines line of source program with respect to which slice is to be computed and "set_of_variables" defines set of variables on "line_number" for which slice is to be computed. A Slice w.r.t. slicing criteria is set of program statements above "line_number" where some variables have been defined and their definitions are affecting one particular variable of "set_of_variable". Also slice includes controlling statements(viz., "if-else", "procedure-call") that determine execution of above mentioned definition statements. These slices are now referred as executable backward static slices.

Later Ottenstein noted that backward slices can be more efficiently computed using program dependence graph as intermediate representation as

compared to control flow graph used by Weiser. Horwitz then introduced the notion of forward slice using dependency graph. COBSlicer is based on dependence graph for slice computation.

Program dependence graph (PDG) consists of nodes that represent assignment statements or control predicates within a particular procedure. Also every PDG has one entry node. These nodes are connected via different directed edges. Control edge is between nodes where source is control predicate that controls destination node. Similarly data edge exists when some variable defined at source node is being used at destination node. To address multiple procedure programs, PDG are constructed for each procedure and are connected together via new edges between Call node in some procedure and entry node of corresponding called procedure. Also data flow between procedures is represented using different data edges. This new graph so generated by amalgamating multiple PDG(s) is referred to as System Dependence Graph (SDG).

Slicing criteria is defined as $\langle \text{node}, \text{variable_of_interest} \rangle$ tuple , where “node” represents node of PDG of particular procedure and “variable_of_interest” is variable used at that particular node. Slicing over SDG using slicing criteria is achieved by traversing edges of graph in desired direction. Also to achieve only data or control slices traversal must be done only over data edges or control edges respectively.

Chapter 2

Proposed System

2.1 System Objective :

COBSlicer is a tool to identify IT rules in COBOL Programs using the concept of Static Program Slices. Proposed goal of the system is, given a point in COBOL program, all possible kind of Static Non-Executable Slices, need to be computed based on specified Slice type.

2.2 Acronyms and Abbreviations :

ICFG : Inter-procedural Control Flow Graph.

PDG : Program Dependence Graph.

SDG : System Dependence Graph.

DDG : Data Dependency Graph (PDG with only Data Flow Edges).

CDG : Control Dependence Graph (PDG with only Control Flow Edges).

2.3 System Functionality :

To achieve the goal the system uses PDG and SDG based Slicing approach. Given a slicing criteria, the system provides option to generate forward as well backward slices. The system also allows to choose from data slice or control slice or program slice. Data slice is effectively set of all nodes that either affect the slicing criteria (Backward Data Slice) or that use slicing criteria (Forward Data Slice). Control Slice is set of all nodes that either control execution of slicing criteria (Backward Control Slice) or are controlled by slicing criteria (Forward Control Slice). Program slice is union of Data Slice and Control Slice.

2.4 Tools Used :

COBSlicer is designed and developed in Eclipse using framework named PRISM. PRISM is a program analysis workbench developed by Tata Research

Design and Development Centre, Pune. PRISM uses static program analysis to analyze a program's behavior with respect to given analysis problem. PRISM provides building blocks in terms of low-level analysis information, which can be used to design the solutions for different analysis problems.

2.5 Input :

COBSlicer takes following inputs :

- 1.) COBOL Project.
- 2.) Slicing Criteria.
- 3.) Slice Type.
- 4.) Slice Direction.

2.6 Output :

COBSlicer renders a SDG of the program statements computed to be in Slice.

2.7 Application :

COBSlicer is designed to assist in identifying and apprehending IT rules in COBOL programs. COBSlicer can be used to generate Cut Slices. Given a set of Output points (viz., write to a file, display) in COBOL program, and given a set of Input points (viz., read from file, read from user), all the paths from a output point to all possible input points could be computed by COBSlicer, using Backward Slice. These paths reflect the business logic that governs generation of output from particular input. Other application of COBSlicer is, it can be used to determine impacts of particular COBOL program statement.

2.8 Working :

PRISM takes COBOL program as input and renders internal representation of program constructs as output. These are known as IRObjects in PRISM terminology. It then constructs control flow graph using these IRObjects, known as ICFG. Using this ICFG, it computes reaching definitions and control dependencies in form of Sets. Using these sets, COBSlicer computes PDG for each procedure wherein nodes contain IRObjects of program statements. Once PDG for each procedure has been built, SDG is built by merging all PDGs.

For each COBOL project SDG is built once and then used for slicing for multiple criteria. Given a slicing criteria <node,variable> , COBSlicer computes slice and generates a graph of sliced program statements. Within the sliced SDG, it looks for input points if any and performs a forward traversal from input to output point (i.e. slicing criteria), giving the IT Rule governing the derivation of output from input point.

Chapter 3

Literature and Frameworks

Designing some new system indeed requires an immense survey of existing literature to familiarize oneself with what work has already been done and available and what direction is to be chosen to heed forward.

3.1 WALA Framework :

In initial phase of COBSlicer, to understand concept of Slice, I used WALA framework and explored its functionalities. WALA is T.J.Watson Libraries for Analysis developed by IBM. It is an open-source API to support static program analysis. It takes Java Jar files as input and generates PDG and SDG.

This PDG and SDG helped to understand how they are computed from input. What form nodes and edges need to be maintained. It used Shrike bytecode utility to construct intermediate representation from Java class file. Even though it takes jar file, it actually generates intermediate representation from Java ByteCode.

However, some issues were faced during exploration. Assignment statements were being skipped from SDG, resulting into incomplete SDG. This feature could be turned off by undoing copy propogation using CAsT source front-end. Forward Slice computation was taking too long even for very small program codes. Even though SDG was being computed, there was no direct way available to print it. Backward Slice computation was expensive as Call Graph building and Class hierarchy construction were heavy processes.

3.2 PRISM Framework :

After getting practical exposure to Slice and its computation using WALA, I started with PRISM. PRISM is a framework designed for Program Analysis. Following diagram demonstrates working of PRISM framework :



Figure 3.1 : Work Flow in PRISM.

The Intermediate Representation(IR) generator takes input project and generates several files (viz., .ST, .AST, .ASS , .CDF) which contains IR of input. Using IR, different analysis can be performed based on requirements. Since COBOL was a new language for me, I began exploration with C.

PRISM represents every IR in form of an object of class **IRObjct**. **IRObjct** is parent class of all other classes in PRISM. Class hierarchy in PRISM for IR Objects is as follows :

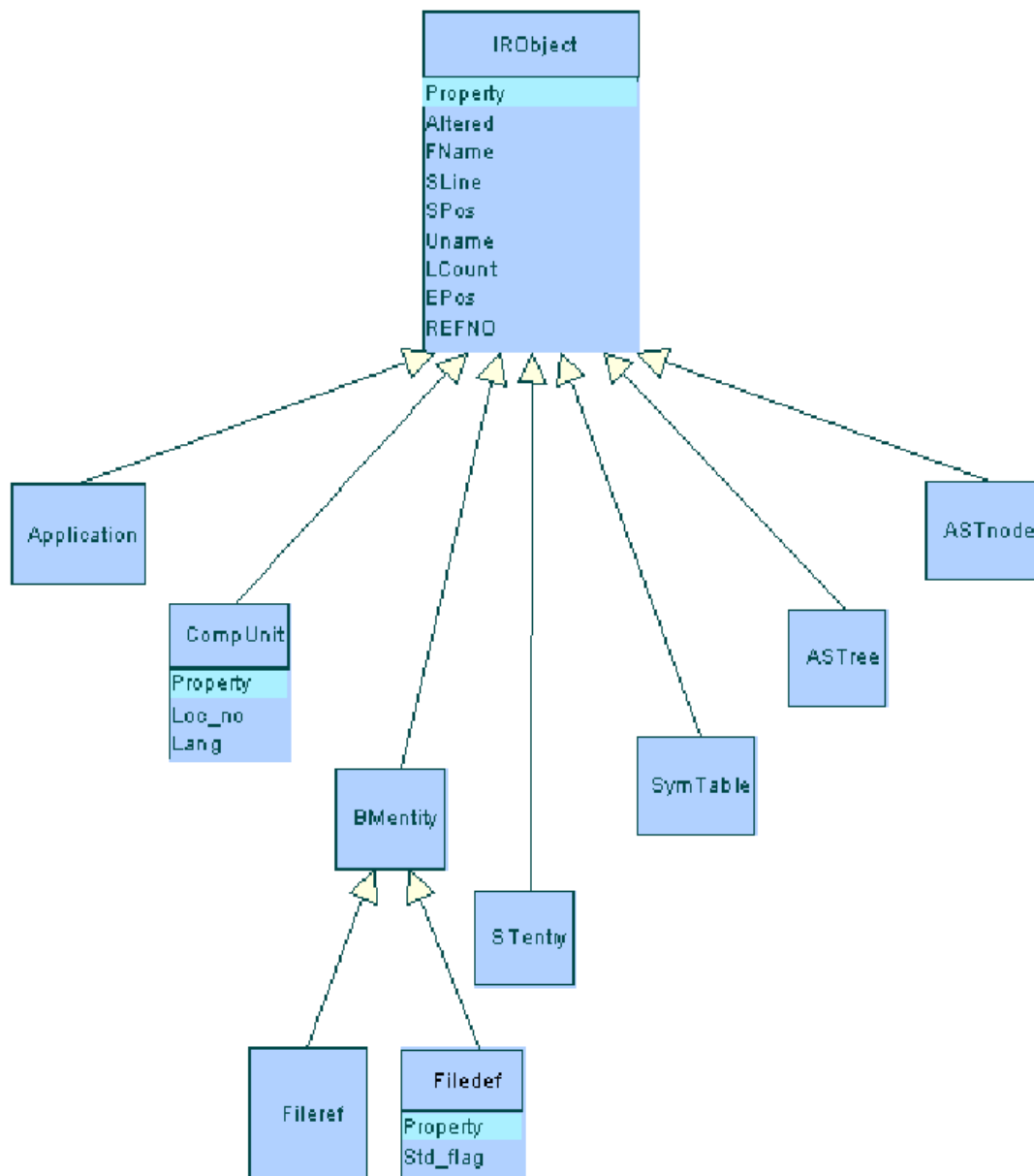


Figure 3.2 : IObject class Hierarchy in PRISM.

3.2.1 Application Meta-Model :

The **Application** object represents an application, which typically comprises of a number of programs. The Application object can have one or more compilation units **CompUnit**. Each **CompUnit** represents a program file that can

contain multiple functions or programs. These functions or programs are represented by the symbol table (**SymTable**) and abstract syntax tree (**ASTree**). The global symbols that an application contains, are represented by **Symbol_ST** objects.

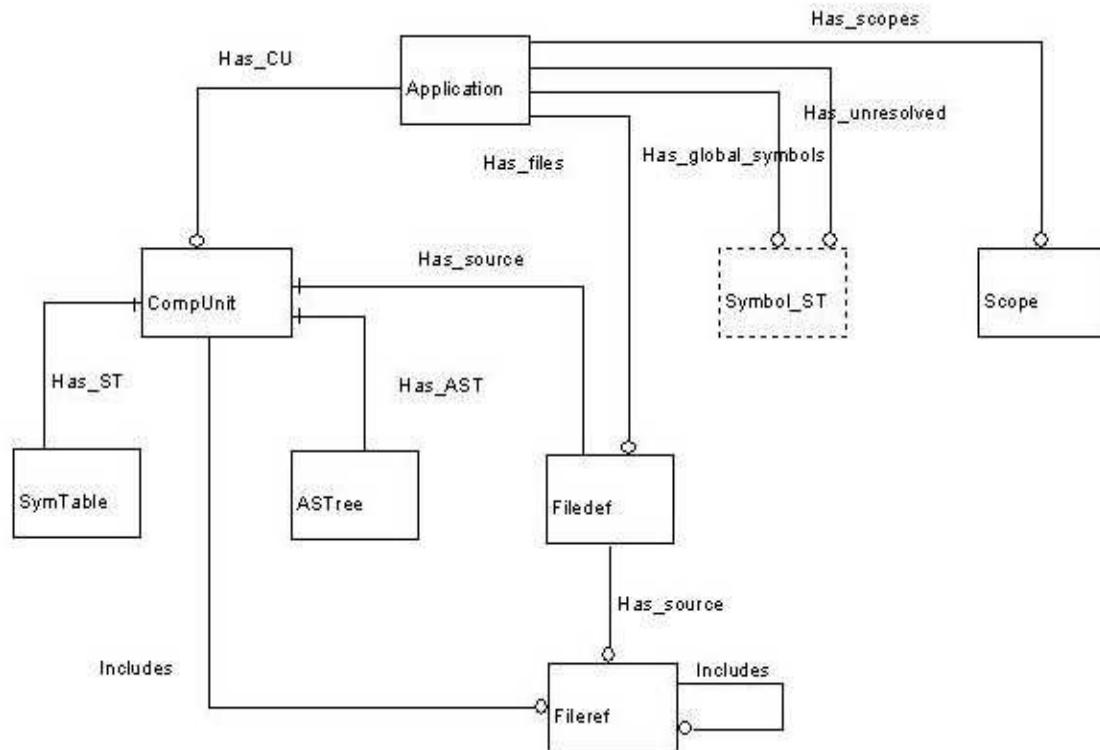


Figure 3.3 : Application IR Model.

3.2.2 Symbol Table Meta-Model :

Symbol (**SymTable**) represents the base class for all kinds of symbols including types, that may occur in a program. There exist four entities that inherit from Symbol, namely,

- 1.) Datatype
- 2.) Ident_ST

4.) Label_ST



Ident_ST represents all the identifiers in the symbol table. The entities inheriting from Ident_ST are **Function_ST**, **Variable** and **Entry_ST**.

1.) Function_ST

Function_ST represents the symbol-table entry for functions defined in the

compilation unit.

2) Variable

Variable represents any variable other than formal parameter. It is concrete for all such cases, while it is abstract when it represents formal parameter.

Thus a variable can be of type:

o **Formalparam**

Formalparam inherits from Variable, and represents formal parameters of Function_ST, representing a concrete entity.

3) Entry_ST

Entry_ST also inherits from Ident_ST and represents the entry-point of a program.

3.2.3 ASTree Meta-Model :

The basic entity (base class) of the AST IR Model is the **ASTnode**. Abstract Syntax Tree (ASTree) consists of multiple AST nodes. Source file information of the symbol, viz., File_name, Line_number and Count (nth occurrence of the symbol on a line) are the meta-properties defined for all AST nodes. Label_AST, Statement and Expression are the entities next in the hierarchy.

The various kinds of statements supported are :

Block : which represents { .. } block in C like languages.

Compstmt : which represents ordered list of statements.

Exprstmt

Ifthen

While : which represents both while() and do..while() loops

For

Loop

Jump : which represents goto.. statement

Mswitch

Lstmt

Break

Continue

Returnstmt

Nullstmt

Casedefault

Directive

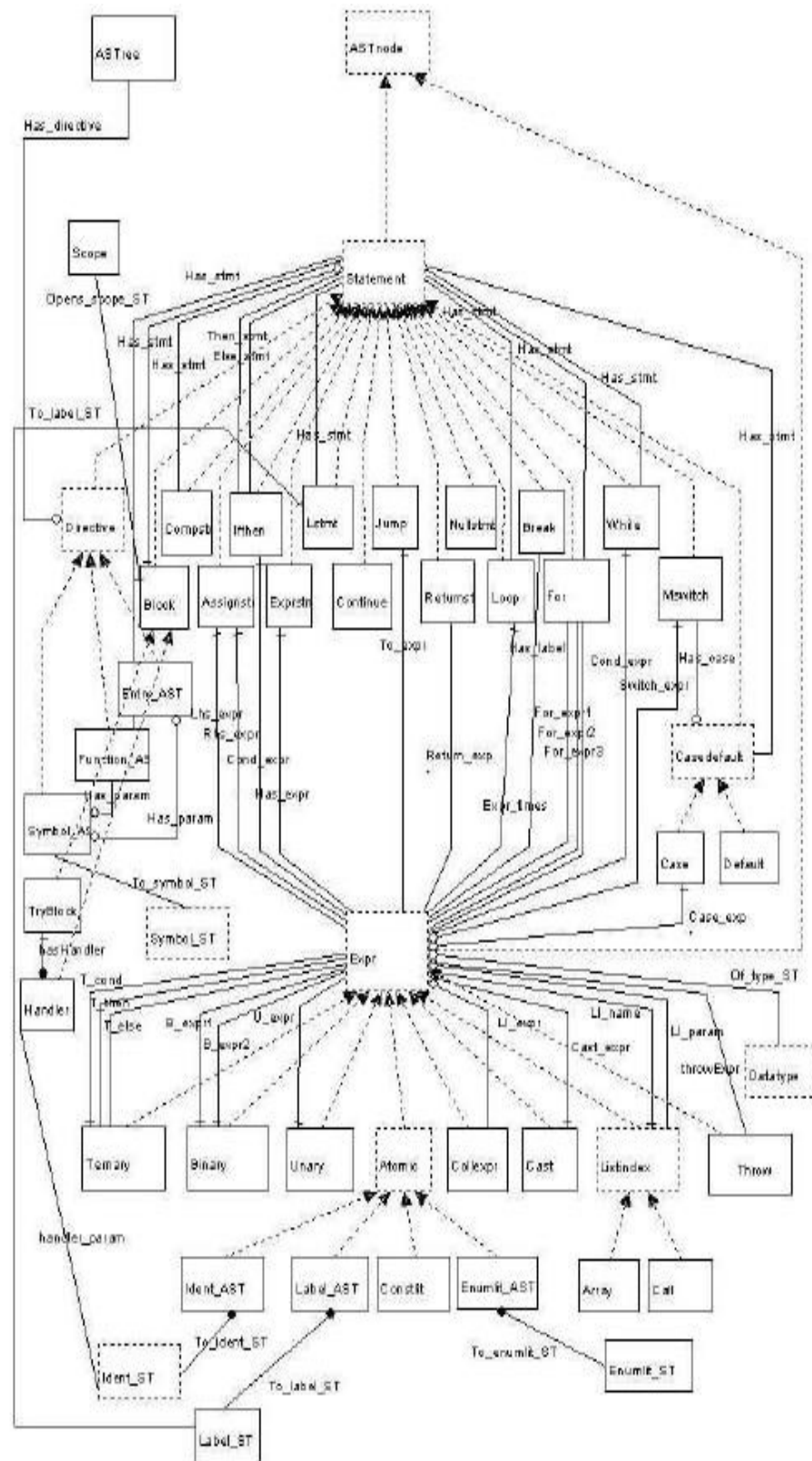


Figure 3.5 : ASTree IR Model.

Directive is an abstract class. Three entities inherit from this class, namely Symbol_AST, Function_AST and Entry_AST. Both the Function_AST and the Entry_AST have a "Has_param" association to the Symbol_AST. Function definition (Function_AST) is a concrete entity representing the function definition. The various kinds of expressions supported are:

Ternary: Ternary expressions are used to model entities like conditional branches.

Binary

Unary

Atomic: Identifier references (Ident_AST), Enumeration Literal references (Enumlit_AST) or Constants (Constlit) may be the Atomic expressions.

Cast

Listindex : Array reference (Array) or function call (Call) are both Listindex – each has one association with expression for array/function name and a one-to-many association with expression for array indexes/function parameters.

Collexpr : It represents a collection of expressions.

3.2.4 NamedEntity :

PRISM uses a special package to deal with all data items contained in source code. The parent interface of all the classes in this package is **NamedEntity**. For each data item used in source code, there is a corresponding NamedEntity object, containing lot of information (viz., name,size,location in source code,parent and offset (if its non-elementary),etc) regarding data item. This package serves various methods that bolster dealing with data items, especially while considering Data edge labeling and Data Slices.

3.2.5 PRISM ICFG :

PRISM takes project files as input, converts source code statements into IRObjets and builds ICFG linking all IRObjets by capturing control flow among them. ICFG consists of nodes and edges. Each logical program statement in the source code is represented as a node in the ICFG (ICFGNode). The individual ICFGs of each function for each Compilation Unit (CU) when combined together forms the complete ICFG for the entire application.

The types of ICFGNodes are :

- 1.) ICFGProgEntry - represents the entry of a program.
- 2.) ICFGProgExit- It represents the exit of a program.
- 3.) ICFGProcEntry- Represents a node which is the entry of a function or procedure.
- 4.) ICFGProcExit - It represents the exit of a procedure or function.
- 5.) ICFGCall- It represents the call of a procedure or function in the program.
- 6.) ICFGRet - This ICFGNode is a place-holder to contain return values from a procedure or function call.
- 7.) ICFGOCall – It represents virtual call to the entry points.
- 8.) ICFGPCall-It represents virtual call to the Compilation Units.
- 9.) ICFGRest - ICFGRest is a "generic" ICFGNode, which contains all other possible kinds of computation nodes, in particular assignments and expressions containing flow of control.

The types of ICFGEDges are :

- 1.) CondEdge - This edge represents *conditional* intra-procedural transfer of control between ICFGNodes. The edge

itself carries a "label" (as an attribute) which identifies on what value of the expression it is taken.

- 2.) UnCondEdge - This edge represents *unconditional* intra-procedural transfer of control between ICFGNodes.
- 3.) CallRetEdge - This represents transfer of control from ICFGCall node to the corresponding ICFGRet node.

After establishing acquaintances with ICFG, I tested for different C programs. Using these ICFG, PDG and SDG were built for C language, and slicing algorithm was implemented for C language constructs. After analyzing all the challenges faced during slicing for C language, base was ready to implement it for COBOL Programs.

3.3 COBOL : A Programming Language

COBOL language was developed as the first high-level programming language suitable for business data processing. In 1965, the first of its kind COBOL compiler was published and since then the journey of COBOL began. Even today a large number of business applications (viz., banking application, security critical applications) use COBOL as back-end programming language. Before exploring slicing on COBOL programs its necessary to apprehend programming constructs of COBOL. A typical COBOL program consists of four divisions :

- 1.) IDENTIFICATION DIVISION : The first division of program. Mainly used to specify Program ID (i.e., Name of Program), name of author, compilation date security, remarks, etc.
- 2.) ENVIRONMENT DIVISION : This division is used to specify

computer and all the peripheral devices required by the program. It consists of two sections :

a.) CONFIGURATION SECTION : Used to specify source computer name, object computer name, and some special names.

b.) INPUT-OUTPUT SECTION : Used to specify information regarding files to be used in the programs. It consists of two paragraphs : FILE-CONTROL and I-O-CONTROL. FILE-CONTROL is used to specify names of files that are used by program.

3.) DATA DIVISION : Used to specify every data item processed by COBOL program. It consists of multiple sections but most commonly used are :

a.) WORKING-STORAGE SECTION : The data items which are developed internally as intermediate results as well as constants are described in this section.

b.) FILE SECTION : It describes all the data items that should be read from or written onto some external file.

c.) LINKAGE SECTION : It appears in Called programs (subroutines) and describes records or non-contiguous data items in respect of which connection should be established with Calling program.

4.) PROCEDURE DIVISION : Is the division where algorithm is actually implemented. It contains statements which specify the operation to be performed by the computer.

Some Semantic Rules :

- Every statement begins with the VERB, specifying the action that the statement does.
- Identifiers may be of maximum length 30. Names must begin with letter or must contain atleast one letter and can use only letters,numbers and '-'.
- Every data item could be either elementary or it could be record. A record is collection of multiple elementary data items or other records.
- Every data item begins with a number specifying its level. All the elements of a record must have higher level number compared to parent record level.
- To share memory area among two records, REDEFINES keyword is used.
- To rename a memory area, RENAME keyword is used.
- PERFORM statement is similar to JUMP statement of Assembly language, only difference being it is treated as a call by PRISM.
- Database queries (viz., INSERT,UPDATE,SELECT,etc) can be used in PROCEDURE DIVISION.
- To specify size of data item PIC (or PICTURE) Clause is used.
- In PIC clause, X represents any character from COBOL's allowed character set. 9 represents numeral, A represents either letter or space , V indicates position of assumed decimal point, S indicates data item is signed.
- PIC Clause can be used only for elementary data items. Record size need not be specified. Its size is sum of all its member data items' size.

Chapter 4

Algorithm and its working

PDG based Slicing was first suggested by Horwitz. He demonstrated how efficiently slices could be computed as compared to Weiser's approach that used Control flow graph. The approach used by COBSlicer is very much similar to the one suggested by Horwitz, with slight modifications in PDG and SDG building approach. The algorithm used, has different activities performed in sequence, and these activities are explained below.

IR Generation :

The input COBOL project received by COBSlicer is in COBOL language. It is converted into Intermediate representation which uses IR meta models of PRISM. Every COBOL program statement is now represented by unique object of Statement. This object in turn could be collection of multiple objects of other sub-classes of IRObjct, since every construct of some program statement will belong to different sub-class of IRObjct.

ICFG Construction :

From above generated IR, ICFG is built that captures control flow within each file of COBOL project as well as captures inter-files control flow (i.e., control flow across files).

Data Dependency Computation :

On ICFG constructed for COBOL project, UseDef query is executed which captures data flow across and within files. The result of such query is set of tuples. Each tuple consists of two entries. First entry contains name of data item that is used at some program point. Second entry is a set containing all those program points where this data item is defined and that definition has not been killed by some other definition.

Control Dependency Computation :

On ICFG constructed for COBOL project, Conds query is executed which captures control dependencies across and within files. The result of such query is set of tuples. Each tuple consists of two entries. First entry contains program point which is control dependent on some other program point(s). Second entry is a set of all the program points which control program point of first entry of tuple.

Here the control dependency doesn't mean transitive execution dependency as in Control flow graph of some procedure. In PDG for program P , node v is said to be control dependent on node u iff one of the following holds :

- 1.) u is the entry vertex, and v represents a component of P that is not nested within any loop or conditional statement.
- 2.) u represents a control predicate, and v represents a component of P immediately nested within the loop or conditional statement whose predicate is represented by u .

PDG Construction :

Using ICFG, Data Dependency, and Control Dependency, PDG for each procedure of COBOL project is constructed. The nodes of PDG are obtained by iterating over set of nodes of ICFG and creating PDG node containing same IRObj as represented by ICFGNode, but excluding ICFGProcExit Nodes. Also the line number of corresponding statement in program is maintained within PDG node.

```
1 read(n) ;
2 i:=1;
3 while (i<=n) do
4 begin
5     if (i mod 2 = 0) then
6         x:=17;
7     else
8         x:=18;
9     z:=x;
10    i:=i+1;
11 end
12 write(z)
```

Code Snippet 4.1

Once PDG nodes have been generated, Control Dependency set is used and iterated over each tuple of set. Node in PDG corresponding to IRObjct in first entry of tuple is matched and marked as destination of Control Dependency Edge, while nodes corresponding to IRObjcts in set in second entry of tuple are marked as source node and Control Dependency edges are thus added from each source to destination.

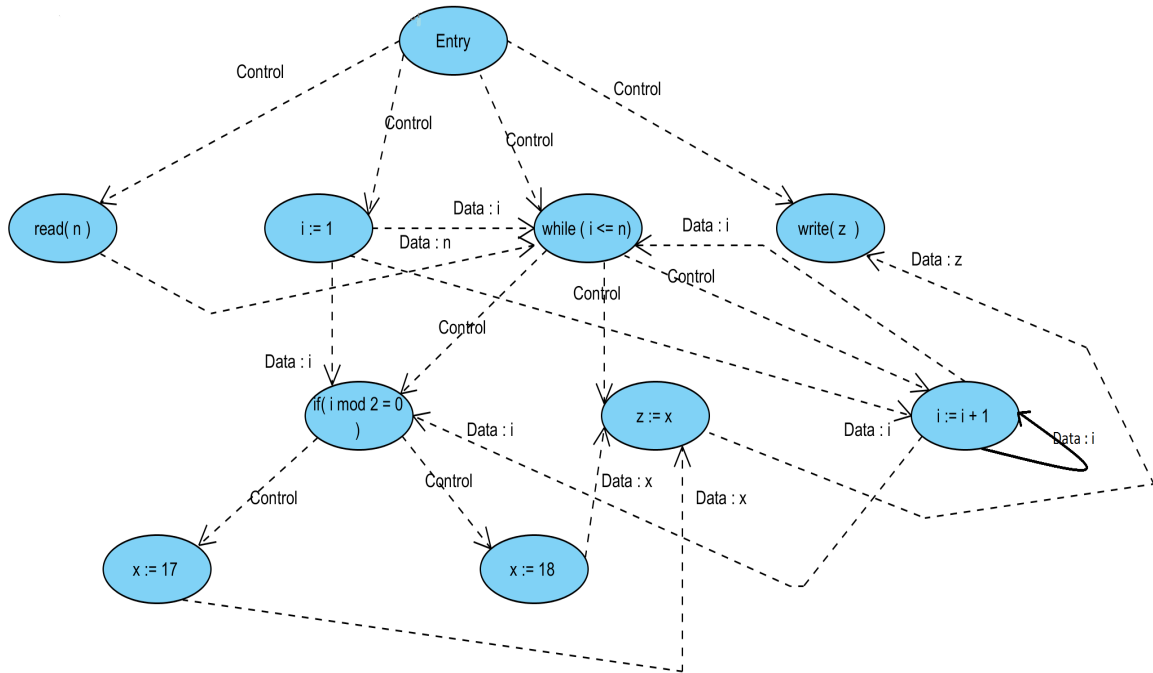


Figure 4.1 PDG of Code Snippet 4.1

Similarly, Data Dependency set is iterated over each tuple. Node corresponding to IObject in first entry of tuple is determined from PDG and marked as destination of Data Dependence edge, while nodes corresponding to IObjects in set in second entry of tuple are marked as source nodes for data edges, and the data edges are constructed from each source to destination. Each edge is labelled to contain name of data item being defined at source node.

SDG Construction :

For every COBOL program of input project, a PDG is constructed. During PDG construction all Call sites along with entry point of Called procedures are preserved in separate structure. This structure is now iterated for each call site, and Call edges are inserted between these nodes from Call site node to entry

point node of Called procedure. Also for each parameter passed, a data edge is added from Call node to entry node of Called procedure. Each of data edge is labeled to contain both, actual parameter used at Call site and formal parameter used in Called procedure.

```
1 main()  
2 begin  
3     a := 17;  
4     b := 18;  
5     Call P(a,b)  
6     write(b)  
7 end
```

Code Snippet 4.2

```
1 P( c , d)  
2 begin  
3     if(c >= d)  
4         d := d+1;  
5     else  
6         c := c+1;  
7 end
```

Code Snippet 4.3

If Called procedure modifies some incoming parameter, then data edge is added, from program point in Called procedure, where parameter is being redefined, to Call site, with edge labeled with parameter getting redefined. These edges are referred as Return Data edges. These edges help in improving precision of Data Slices.

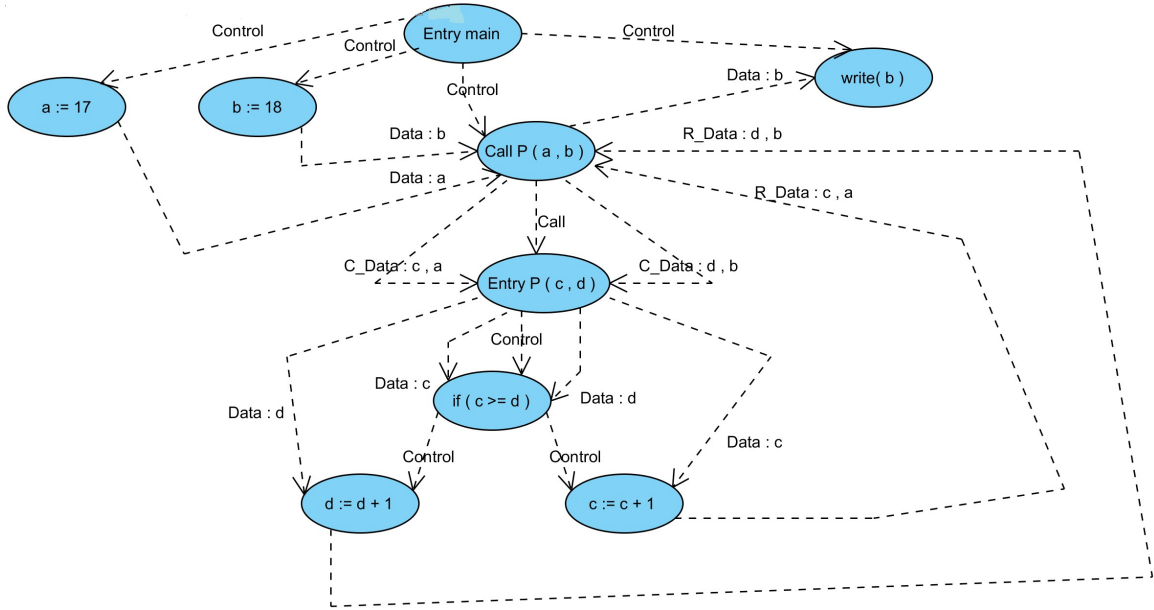


Figure 4.2 : SDG of Code Snippets 4.2 and 4.3

Slicing Criteria Specification :

User needs to specify slicing criteria so that COBSlicer can generate slice accordingly. Slicing criteria is of following form :

<Procedure_Name,Line_No,Slice_Direction,Slice_Type,Variable_Of_Interest>

where,

Procedure_Name : represents name of COBOL file containing slicing point

Line_No : represents line number in COBOL file where slicing point exists.

Slice_Direction : represents either Forward Slice or Backward Slice.

Slice_Type : represents type of slice which could be either Data or Control

or Program.

Variable_Of_Interest : represents name of data item based on which slicing needs to be done. Used only in case of Data Slice.

Slicer :

Based on SDG generated for COBOL project and slicing criteria specified by user, slicer begins by identifying node in SDG corresponding to slicing criteria. That node is referred as seed point. From this seed node, SDG is traversed either in forward or backward direction based on user's specification. Also edges considered during traversal are dependent on slicing criteria. For Data slice only data edges would be traversed. For Control slice only control edges would be traversed while for Program slice all the edges would be traversed. During traversal all edges traversed and all the nodes reached via those edges will be inserted into new graph, called Sliced SDG. This Sliced SDG represents the slice for given slicing criteria, in graphical form.

Input-Output Point Generator :

In COBOL language, clauses such as READ, SELECT, ACCEPT etc are considered as *input points* since they allow accepting input from file or database. Similarly clauses such as DISPLAY, INSERT, UPDATE, WRITE are considered as *output points* since they allow writing data to terminal or file or database. Using PRISM and external code, COBSlicer computes input and output points located in source code and stores them in a structure.

4.1 Algorithm for Computing Slices :

1. IR Generator is invoked with path of COBOL project as input.
2. ICFG Constructor is invoked.
3. Data Dependence Set is generated by using ICFG as input.
4. Control Dependence Set is generated by using ICFG as input.
5. Using ICFG, Control Dependence Set and Data Dependence Set PDG is constructed for each procedure in COBOL Project.
6. SDG is built using all the PDG(s) generated.
7. Slicing criteria is resolved to unique node of SDG and marked as seed node.
8. Slicer is now invoked over seed point to compute slice as per specified context and direction.

4.2 Algorithm for generating IT Rules :

1. Input-Output points are determined and nodes corresponding to output points are resolved and marked as seed nodes in SDG.
2. Slicer is now invoked over each seed point to compute corresponding backward slice.
3. In each of these slices, resolve input point node if any, and make a forward traversal keeping track of traversal in form of graph.
4. Such graphs are termed as IT Rules which are recorded in data-structure.

Chapter 5

Implementation

COBSlicer consists of four main packages which are associated with different functionalities. These packages are :

1. `com.tcs.irgenerator`

This package contains classes that are entitled with task of generating IR from COBOL source code and also to generate ICFG.

2. `com.tcs.iranalyzer`

This package contains classes that perform the task of computing Control Dependence Set and Data Dependence Set.

3. `com.tcs.graphbuilder`

This package contains classes that perform the task of building PDG(s) and SDG.

4. `com.tcs.slicer`

This package contains classes that define slicing criteria, PDG nodes and PDG edges as well as classes responsible for performing slicing.

5.1 Data Structure Description :

The following table indicates the data structures used to describe nodes and edges used in PDG and SDG as well as also describes what information is maintained by slicing criteria.

Table 5.1 : Data Structures

<u>Package</u>	<u>Class</u>	<u>Fields</u>	<u>Description</u>
com.tcs.slicer	PDGNode	iobj	iobj is object of IRObj, corresponding to program statement represented by <i>this</i> node.
		sourceLine	Represents line number of program statement represented by <i>this</i> node in COBOL file.
com.tcs.slicer	PDGEdge	source	Object of PDGNode class represents source node of edge.
		destination	Object of PDGNode class represents destination node of edge.
		type	String object representing type of edge and can have values from set : (Data,Control,Call, C_Data,R_Data).
		ne	Object of NamedEntity Class represents data item associated with <i>this</i> Data Edge.
		formalParam	Object of

			NamedEntity Class represents formal parameter associated with <i>this</i> C_Data or R_Data Edge.
		actualParam	Object of NamedEntity Class represents actual parameter associated with <i>this</i> C_Data or R_Data Edge.
com.tcs.slicer	SlicingCriteria	node	Array of PDGNode representing all nodes of PDG, corresponding to program point specified as slice start point.
		sliceVariable	Object of NamedEntity class representing data item specified as slice variable.
		fnAST	Object of Function_AST represents procedure in which slicing seed point belongs.

5.2 Function Description :

Table 5.2 : Functional Description

<u>Package</u>	<u>Class</u>	<u>Method</u>	<u>Description</u>
com.tcs.iranalyzer	ICFGBuilder	public static FL_ICFG_Constructor buildICFG(Application app)	This method generates ICFG for Application object app, which represents IR of COBOL project. This ICFG is stored in file and method returns its handler.
com.tcs.iranalyzer	ControlDependencyGenerator	public static Set computeControlDependency(Application app)	This method generates control dependencies by performing Conds query on app and returns a Set of tuples containing Dependencies.
com.tcs.iranalyzer	UseDefGenerator	public static Set computeUseDef(Application app)	This method generates data dependencies by performing Cusedef query on app and returns a Set of tuples containing Dependencies.
com.tcs.graphbuilder	ICFGHandler	synchronized void displayICFG(ICFGNode node,ICFG entityICFG, DirectedSparseM	This method displays ICFG generated for given COBOL project, where node is entry node of ICFG,

		ultigraph<PDGNode, PDGEdge> entityPdg)	entityICFG is ICFG of particular Function_Ast or CompUnit, entityPdg is PDG in which PDGNode corresponding to node is added.
com.tcs.graphbuilder	ICFGHandler	public SlicingCriteria getSlicingCriteria() ()	This method return object of SlicingCriteria that represents all slicing information provided by user.
com.tcs.graphbuilder	ICFGHandler	public HashMap<Call, Function_AST> getCallMap()	This method returns the object of HashMap class, which was populated with all Call site nodes and entry point of corresponding Called procedure during PDG construction.
com.tcs.graphbuilder	PDGHandler	public void pdgBuild(Application app)	This method takes app object as input, uses pre-computed PDG containing only nodes, uses Data Dependence and Control Dependence Set and builds edges among PDG nodes.
com.tcs.graphbuilder	SDGHandler	public void	This method

der		sdgBuild()	builds SDG from pre-computed PDG(s), by incorporating all nodes and all edges of each PDG(s),and later build inter-procedural Data and Control edges using Call Map data structure computed by ICFGHandler.
com.tcs.slicer	Client	public void runClient(Application app)	This method actually initiates slicing by feeding generated SDG with user specified Slicing criteria.
com.tcs.slicer	Slicer	public void getBackWardSlice(char sliceType)	This method generates backward slice of the type specified by sliceType.
com.tcs.slicer	Slicer	public void getForwardSlice(char sliceType)	This method generates forward slice of the type specified by sliceType

Chapter 6

Test Cases and Screen Shots

Test Cases help ascertain the validity of developed system and also help to determine flaws if any. Following are few test cases that were tested with COBSlicer and the results generated by COBSlicer have also been adjoined.

6.1 Sample Codes :

```

1  000100 IDENTIFICATION DIVISION.                                00010000
2  000200 PROGRAM-ID. BKOACTUP.                                    00020000
3  001100 DATA DIVISION.                                          00110000
4  001200 WORKING-STORAGE SECTION.                                00120000
5      001600 COPY ACCTCONS.                                       00160000
6      001700 01 ACCOUNT.                                          00170000
7      001800 COPY ACCCPY.                                         00180000
8  001900 LINKAGE SECTION.                                         00190000
9      002000 01 L-ACCOUNT.                                         00200000
10     002100 COPY LACNT.                                           00210000
11     002200 01 ACCT-RETURN-CODE PIC 99.                          00220000
12  002300 PROCEDURE DIVISION USING L-ACCOUNT, ACCT-RETURN-CODE.  00230000
13     002400 MOVE-LINKAGE-DATA.                                     00240000
14     002500     MOVE L-ACCOUNT-ID TO ACCOUNT-ID.                 00250000
15     002600 CHECK-ACCOUNT.                                         00260000
16     002700*    CALL 'BK-PER-ACCOUNT-GET' USING ACCOUNT ,        00270000
17     002800     CALL 'BKPACTGA' USING ACCOUNT ,                  00280000
18     002900                                     ACCT-RETURN-CODE.  00290000
19     003000     IF ACCT-RETURN-CODE = ACCOUNT-OPERFAIL           00300000
20     003100         GOBACK.                                       00310000
21     003200     IF ACCT-RETURN-CODE = NEW-ACCOUNT                00320000
22     003300         GOBACK.                                       00330000
23     003400     PERFORM UPDATE-ACCOUNT.                           00340000
24     003500     GOBACK.                                           00350000
25     003600 UPDATE-ACCOUNT.                                         00360000
26     003700     IF L-ACCOUNT-LASTTRDATE = ' '                    00370000
27     003800         CALL 'BKPACTUB' USING L-ACCOUNT ,            00380000
28     003900                                     ACCT-RETURN-CODE  00390000
29     004000     ELSE                                              00400000
30     004100         CALL 'BKPACTUD' USING L-ACCOUNT ,            00410000
31     004200                                     ACCT-RETURN-CODE  00420000
32     004300     END-IF.                                           00430000
33     000000 CALL 'BKPACTUD' USING L-ACCOUNT , ACCT-RETURN-CODE.  00000000

```

Code Snippet 6.1

1	000001 IDENTIFICATION DIVISION.	00000100
2	000002 PROGRAM-ID. BKPACTUB.	00000200
3	000010 DATA DIVISION.	00001000
4	000011 WORKING-STORAGE SECTION.	00001100
5	000015 EXEC SQL	00001500
6	000016 INCLUDE SQLCA	00001600
7	000017 END-EXEC.	00001700
8	000018 EXEC SQL	00001800
9	000019 INCLUDE ACCOUNT	00001900
10	000020 END-EXEC.	00002000
11	000027 COPY ACCTCONS .	00002700
12	000028 LINKAGE SECTION.	00002800
13	000029 01 L-ACCOUNT.	00002900
14	000030 COPY LACNT.	00003000
15	000040 01 ACCT-RETURN-CODE PIC 99.	00004000
16	000050 PROCEDURE DIVISION USING L-ACCOUNT, ACCT-RETURN-CODE.	00005000
17	000060 MOVE-LINKAGE-DATA.	00006000
18	000070 MOVE L-ACCOUNT-ID TO ACCOUNT-ID.	00007000
19	000080 MOVE L-ACCOUNT-BALANCE TO ACCOUNT-BALANCE.	00008000
20	000090 UPDATE-ACCOUNT-DETAILS.	00009000
21	000100 EXEC SQL	00010000
22	000200 UPDATE	00020000
23	000300 ACCOUNT	00030000
24	000400 SET	00040000
25	000500 BALANCE = :ACCOUNT-BALANCE	00050000
26	000600 WHERE	00060000
27	000700 ACID = :ACCOUNT-ID	00070000
28	000800 END-EXEC.	00080000
29	000900 IF SQLCODE NOT = 0	00090000
30	001000 MOVE ACCOUNT-OPERFAIL TO ACCT-RETURN-CODE	00100000
31	001100 DISPLAY ACCOUNT-ID	00110000
32	001100 ELSE	00110000
33	001200 MOVE ACCOUNT-OPERSUCC TO ACCT-RETURN-CODE	00120000
34	001300 END-IF.	00130000

Code Snippet 6.2

1	000001 IDENTIFICATION DIVISION.	00000100
2	000002 PROGRAM-ID. BKPACTUD.	00000200
3	000010 DATA DIVISION.	00001000
4	000011 WORKING-STORAGE SECTION.	00001100
5	000015 EXEC SQL	00001500
6	000016 INCLUDE SQLCA	00001600
7	000017 END-EXEC.	00001700
8	000018 EXEC SQL	00001800
9	000019 INCLUDE ACCOUNT	00001900
10	000020 END-EXEC.	00002000
11	000027 COPY ACCTCONS .	00002700
12	000028 LINKAGE SECTION.	00002800
13	000029 01 L-ACCOUNT.	00002900
14	000030 COPY LACNT.	00003000
15	000040 01 ACCT-RETURN-CODE PIC 99.	00004000
16	000050 PROCEDURE DIVISION USING L-ACCOUNT, ACCT-RETURN-CODE.	00005000
17	000060 MOVE-LINKAGE-DATA.	00006000
18	000070 MOVE L-ACCOUNT-ID TO ACCOUNT-ID.	00007000
19	000080 MOVE L-ACCOUNT-BALANCE TO ACCOUNT-BALANCE.	00008000
20	000090 MOVE L-ACCOUNT-LASTTRDATE TO ACCOUNT-LASTTRDATE.	00009000
21	000000 MOVE ACCT-RETURN-CODE TO ACCT-RETURN-CODE.	
22	000100 UPDATE-ACCOUNT-DETAILS.	00010000
23	000200 EXEC SQL	00020000
24	000300 UPDATE	00030000
25	000400 ACCOUNT	00040000
26	000500 SET	00050000
27	000600 BALANCE = :ACCOUNT-BALANCE ,	00060000
28	000700 LASTTRDATE = :ACCOUNT-LASTTRDATE	00070000
29	000800 WHERE	00080000
30	000900 ACID = :ACCOUNT-ID	00090000
31	001000 END-EXEC.	00100000
32	001100 IF SQLCODE NOT = 0	00110000
33	001200 MOVE ACCOUNT-OPERFAIL TO ACCT-RETURN-CODE	00120000
34	001300 ELSE	00130000
35	001400 MOVE ACCOUNT-OPERSUCC TO ACCT-RETURN-CODE	00140000
36	001500 END-IF.	00150000

Code Snippet 6.3

```

1  000001 IDENTIFICATION DIVISION.                                00000100
2  000002 PROGRAM-ID. UNION2.                                     00000200
3  000012 DATA DIVISION.
4  WORKING-STORAGE SECTION.
5  000070 01 BOOK.
6      10 B-CODE          PIC X(4) .                                00000000
7      10 B-PUBLICATION   PIC X(12) .                               00000000
8      01 HOTEL.
9      10 H-NAME          PIC X(8) .
10     10 H-ADDRESS       PIC X(8) .
11     66 HOTEL-DETAILS RENAME H-NAME THRU H-ADDRESS.
12 PROCEDURE DIVISION.                                           00007000
13 000000 UPDATE-ACCOUNT-ID.                                       00020000
14     IF B-CODE = "xxx"
15         MOVE "YYY" TO B-PUBLICATION
16         MOVE "ZZZ" TO H-NAME
17         MOVE "ZZZ" TO H-ADDRESS
18     ELSE
19         MOVE "NNN" TO B-PUBLICATION
20         MOVE "ZZZ" TO HOTEL-DETAILS
21     END-IF.
22 000000 CALL 'CHECK' USING BOOK.
23 000000 CALL 'CHECK' USING HOTEL.

```

Code Snippet 6.4

```

1  000001 IDENTIFICATION DIVISION.                                00000100
2  000002 PROGRAM-ID. CHECK.                                       00000200
3  000012 DATA DIVISION.
4  000000 WORKING-STORAGE SECTION.
5      01 COMPANY          PIC X(8) .                                00000000
6  000000 LINKAGE SECTION.                                       00000000
7  000070 01 PERSON.
8      10 P-NAME          PIC X(8) .                                00000000
9      10 P-ADDRESS       PIC X(8) .                                00000000
10 PROCEDURE DIVISION USING PERSON.                               00007000
11 000000 UPDATE-ACCOUNT-ID.                                       00020000
12     IF P-NAME = "xxx"
13         MOVE "MMM" TO COMPANY
14         MOVE P-ADDRESS TO P-NAME
15         MOVE "YYY" TO P-ADDRESS
16     ELSE
17         MOVE "ABC" TO COMPANY
18         MOVE "NNN" TO P-ADDRESS
19     END-IF.

```

Code Snippet 6.5

6.2 Test Cases :

```
"BKOACTUP" : : 4294967306  
"BKPACTUB" : : 8589934602  
"BKPACTUD" : : 12884901898
```

Figure 6.1 : File-Number Corresponding to File Names.

```
"CHECK" : : 4294967306  
"UNION2" : : 12884901898
```

Figure 6.2 : File-Number Corresponding to File Names.

Each test case describes one slicing criteria in form $\langle name_of_program, Line_num, direction, context \rangle$ where *direction* is from (B-Backward, F-Forward) and *context* is from (D-Data, C-Control, P-Program). Output describes types of edges where *C-Data* represents data passed from Call Site to Called procedure, and *R-Data* represents modified data item passed from called procedure to call site.

Backward Slices :

Test Case 1 : BKPACTUD 43 B D (DCLACCOUNT.ACCOUNT-BALANCE)

```
----- SDG of Backward Slice-----  
-----Displaying Sliced SDG Edges-----  
FromFile->12884901898 -> SrLNo ->43 to File->12884901898 -> SrLNo ->35 of type Data  
FromFile->12884901898 -> SrLNo ->35 to File->12884901898 -> SrLNo ->2 of type Data  
FromFile->12884901898 -> SrLNo ->2 to File->4294967306 -> SrLNo ->44 of type C_Data  
FromFile->12884901898 -> SrLNo ->2 to File->4294967306 -> SrLNo ->41 of type C_Data  
FromFile->4294967306 -> SrLNo ->44 to File->4294967306 -> SrLNo ->2 of type Data  
FromFile->4294967306 -> SrLNo ->41 to File->4294967306 -> SrLNo ->2 of type Data
```

Test Case 2 : CHECK 14 B C (PERSON.P-ADDRESS)

```
----- SDG of Backward Slice-----  
-----Displaying Sliced SDG Edges-----  
FromFile->4294967306 -> SrLNo ->14 to File->4294967306 -> SrLNo ->12 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->2 of type Control  
FromFile->4294967306 -> SrLNo ->2 to File->12884901898 -> SrLNo ->22 of type Call  
FromFile->4294967306 -> SrLNo ->2 to File->12884901898 -> SrLNo ->23 of type Call  
FromFile->12884901898 -> SrLNo ->22 to File->12884901898 -> SrLNo ->2 of type Control  
FromFile->12884901898 -> SrLNo ->23 to File->12884901898 -> SrLNo ->2 of type Control
```

Test Case 3 : CHECK 14 B P (PERSON.P-ADDRESS)

```
----- SDG of Backward Slice-----  
-----Displaying Sliced SDG Edges-----  
FromFile->4294967306 -> SrLNo ->14 to File->4294967306 -> SrLNo ->12 of type Control  
FromFile->4294967306 -> SrLNo ->14 to File->4294967306 -> SrLNo ->2 of type Data  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->2 of type Data  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->2 of type Control  
FromFile->4294967306 -> SrLNo ->2 to File->12884901898 -> SrLNo ->22 of type C_Data  
FromFile->4294967306 -> SrLNo ->2 to File->12884901898 -> SrLNo ->23 of type C_Data  
FromFile->4294967306 -> SrLNo ->2 to File->12884901898 -> SrLNo ->22 of type Call  
FromFile->4294967306 -> SrLNo ->2 to File->12884901898 -> SrLNo ->23 of type Call  
FromFile->4294967306 -> SrLNo ->18 to File->4294967306 -> SrLNo ->12 of type Control  
FromFile->12884901898 -> SrLNo ->23 to File->4294967306 -> SrLNo ->14 of type R_Data  
FromFile->12884901898 -> SrLNo ->23 to File->12884901898 -> SrLNo ->20 of type Data  
FromFile->12884901898 -> SrLNo ->23 to File->12884901898 -> SrLNo ->16 of type Data  
FromFile->12884901898 -> SrLNo ->23 to File->12884901898 -> SrLNo ->2 of type Control  
FromFile->12884901898 -> SrLNo ->23 to File->4294967306 -> SrLNo ->18 of type R_Data  
FromFile->12884901898 -> SrLNo ->23 to File->12884901898 -> SrLNo ->17 of type Data  
FromFile->12884901898 -> SrLNo ->23 to File->4294967306 -> SrLNo ->15 of type R_Data  
FromFile->12884901898 -> SrLNo ->16 to File->12884901898 -> SrLNo ->14 of type Control  
FromFile->12884901898 -> SrLNo ->14 to File->12884901898 -> SrLNo ->2 of type Control  
FromFile->12884901898 -> SrLNo ->20 to File->12884901898 -> SrLNo ->14 of type Control  
FromFile->12884901898 -> SrLNo ->17 to File->12884901898 -> SrLNo ->14 of type Control  
FromFile->4294967306 -> SrLNo ->15 to File->4294967306 -> SrLNo ->12 of type Control  
FromFile->12884901898 -> SrLNo ->22 to File->12884901898 -> SrLNo ->19 of type Data  
FromFile->12884901898 -> SrLNo ->22 to File->4294967306 -> SrLNo ->14 of type R_Data  
FromFile->12884901898 -> SrLNo ->22 to File->4294967306 -> SrLNo ->15 of type R_Data  
FromFile->12884901898 -> SrLNo ->22 to File->4294967306 -> SrLNo ->18 of type R_Data  
FromFile->12884901898 -> SrLNo ->22 to File->12884901898 -> SrLNo ->15 of type Data  
FromFile->12884901898 -> SrLNo ->22 to File->12884901898 -> SrLNo ->2 of type Control  
FromFile->12884901898 -> SrLNo ->19 to File->12884901898 -> SrLNo ->14 of type Control  
FromFile->12884901898 -> SrLNo ->15 to File->12884901898 -> SrLNo ->14 of type Control
```

Forward Slices :

Test Case 4 : BKPACTUB 34 F D (DCLACCOUNT.ACCOUNT-ID)

```
----- SDG of Forward Slice-----  
-----Displaying Sliced SDG Edges-----  
FromFile->8589934602 -> SrLNo ->34 to File->8589934602 -> SrLNo ->38 of type Data  
FromFile->8589934602 -> SrLNo ->34 to File->8589934602 -> SrLNo ->47 of type Data
```

Test Case 5 : UNION2 22 F C (PERSON.P-NAME)

```
----- SDG of Forward Slice-----  
-----Displaying Sliced SDG Edges-----  
FromFile->12884901898 -> SrLNo ->22 to File->4294967306 -> SrLNo ->2 of type Call  
FromFile->4294967306 -> SrLNo ->2 to File->4294967306 -> SrLNo ->12 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->13 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->17 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->15 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->18 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->14 of type Control
```

Test Case 6 : UNION2 22 F P (PERSON.P-NAME)

```
----- SDG of Forward Slice-----  
-----Displaying Sliced SDG Edges-----  
FromFile->12884901898 -> SrLNo ->22 to File->4294967306 -> SrLNo ->2 of type Call  
FromFile->12884901898 -> SrLNo ->22 to File->4294967306 -> SrLNo ->2 of type C_Data  
FromFile->4294967306 -> SrLNo ->2 to File->4294967306 -> SrLNo ->12 of type Data  
FromFile->4294967306 -> SrLNo ->2 to File->4294967306 -> SrLNo ->12 of type Control  
FromFile->4294967306 -> SrLNo ->2 to File->4294967306 -> SrLNo ->14 of type Data  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->17 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->15 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->13 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->18 of type Control  
FromFile->4294967306 -> SrLNo ->12 to File->4294967306 -> SrLNo ->14 of type Control  
FromFile->4294967306 -> SrLNo ->15 to File->12884901898 -> SrLNo ->22 of type R_Data  
FromFile->4294967306 -> SrLNo ->15 to File->12884901898 -> SrLNo ->23 of type R_Data  
FromFile->4294967306 -> SrLNo ->18 to File->12884901898 -> SrLNo ->22 of type R_Data  
FromFile->4294967306 -> SrLNo ->18 to File->12884901898 -> SrLNo ->23 of type R_Data  
FromFile->4294967306 -> SrLNo ->14 to File->12884901898 -> SrLNo ->23 of type R_Data  
FromFile->4294967306 -> SrLNo ->14 to File->12884901898 -> SrLNo ->22 of type R_Data  
FromFile->12884901898 -> SrLNo ->23 to File->4294967306 -> SrLNo ->2 of type Call  
FromFile->12884901898 -> SrLNo ->23 to File->4294967306 -> SrLNo ->2 of type C_Data
```


Test Case 7 : Data Cut Slice

Input : COBOL Project containing one COBOL file and multiple data files as follows :




























Name ^	Date modified	Type	Size
 ACCCPY	5/24/2012 4:08 PM	File	1 KB
 ACCOUNT	5/24/2012 4:08 PM	File	2 KB
 ACCRTNCD	5/24/2012 4:08 PM	File	1 KB
 ACCTCONS	5/24/2012 4:08 PM	File	1 KB
 ACHSCPY	5/24/2012 4:08 PM	File	1 KB
 ACHSKEEP	5/24/2012 4:08 PM	File	1 KB
 BKPACTGA	4/1/2014 6:40 PM	COB File	6 KB
 CONSDATA	5/24/2012 4:08 PM	File	1 KB
 CUSRTNCD	5/24/2012 4:08 PM	File	1 KB
 CUSTCONS	5/24/2012 4:08 PM	File	1 KB
 CUSTCPY	5/24/2012 4:08 PM	File	1 KB
 CUSTOMER	5/24/2012 4:08 PM	File	2 KB
 DEPOCPY	5/24/2012 4:08 PM	File	1 KB
 DEPOSIT	5/24/2012 4:08 PM	File	2 KB
 LACNT	5/24/2012 4:08 PM	File	1 KB
 LAKEEP	5/24/2012 4:08 PM	File	1 KB
 LCUST	5/24/2012 4:08 PM	File	1 KB
 LDPSIT	5/24/2012 4:08 PM	File	1 KB
 LTRACT	5/24/2012 4:08 PM	File	1 KB
 LXTION	5/24/2012 4:08 PM	File	1 KB
 Sqlca	5/24/2012 4:08 PM	File	1 KB
 SQLCAOLD	5/24/2012 4:08 PM	File	1 KB
 TIONCPY	5/24/2012 4:08 PM	File	1 KB
 TIONFCPY	5/24/2012 4:08 PM	File	1 KB
 TRANSACT	5/24/2012 4:08 PM	File	2 KB
 TRSTTFR	5/24/2012 4:08 PM	File	1 KB
 XACTCONS	5/24/2012 4:08 PM	File	1 KB

Figure 6.3 Input COBOL Project for Data Cut Slice

Output :

```
7 : Slice Variables Extracted ...
5 : Unique Input Slice Variables
(DCLACCOUNT.ACCOUNT-TYPE)::38->BKPACTGA.cob
(DCLACCOUNT.CUSTOMER-ID)::38->BKPACTGA.cob
(DCLACCOUNT.ACCOUNT-BALANCE)::38->BKPACTGA.cob
(DCLACCOUNT.ACCOUNT-CRDATE)::38->BKPACTGA.cob
(DCLACCOUNT.ACCOUNT-LASTTRDATE)::38->BKPACTGA.cob
2 : Unique Output Slice Variables
ACCT-RETURN-CODE::67->BKPACTGA.cob
(L-ACCOUNT.L-ACCOUNT-BALANCE)::68->BKPACTGA.cob

"BKPACTGA"::4294967306

Slicing for :BKPACTGA::>ACCT-RETURN-CODE
----- SDG of Backward Slice-----
-----Displaying Sliced SDG Edges-----
FromFile->4294967306 -> SrLNo ->67 to File->4294967306 -> SrLNo ->58 of type Data
FromFile->4294967306 -> SrLNo ->67 to File->4294967306 -> SrLNo ->56 of type Data
FromFile->4294967306 -> SrLNo ->67 to File->4294967306 -> SrLNo ->65 of type Data
FromFile->4294967306 -> SrLNo ->58 to File->4294967309 -> SrLNo ->3 of type Data
FromFile->4294967306 -> SrLNo ->56 to File->4294967309 -> SrLNo ->5 of type Data
FromFile->4294967306 -> SrLNo ->65 to File->4294967309 -> SrLNo ->7 of type Data

Slicing for :BKPACTGA::>(L-ACCOUNT.L-ACCOUNT-BALANCE)
----- SDG of Backward Slice-----
-----Displaying Sliced SDG Edges-----
FromFile->4294967306 -> SrLNo ->68 to File->4294967306 -> SrLNo ->62 of type Data
FromFile->4294967306 -> SrLNo ->68 to File->4294967306 -> SrLNo ->2 of type Data
FromFile->4294967306 -> SrLNo ->62 to File->4294967306 -> SrLNo ->38 of type Data
-----Displaying Cut Slice -----
From File->4294967306 -> SrLNo ->38 to File->4294967306 -> SrLNo ->62
From File->4294967306 -> SrLNo ->62 to File->4294967306 -> SrLNo ->68
```

Here “4294967309” is file number of data file of input project.

Chapter 7

Conclusion and Future Extensions

7.1 Conclusion

As demonstrated by above test cases and screen shots, the precision of slices computed by COBSlicer is fairly well. Also it is capable of generating slice between two program points. The system developed, provides visualization and navigation in a very efficient manner. It generates the output in a manner that aids user in understanding IT Rules more clearly and quickly.

7.2 Future Extensions

1. The efficiency of Data slices computed by COBSlicer can be improved if Data Dependence set is computed with more precision. We look forward to discovering new heuristics to compute data dependence set.
2. We also look forward to include context sensitivity while slice computation to improve precision of slices and reduce false positives.
3. To develop an Eclipse plug-in to improve visualization and user interfacing.
4. Currently Data dependence is being computed using PRISM's inbuilt functionality. There is yet another framework called HEROS which allows Data dependence computation. We plan to experiment data dependence computation using HEROS framework.

BIBLIOGRAPY

- [1]. M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10:352-357, July 1984.
- [2]. D. W. Binkley and K. B. Gallagher. Program Slicing. *Advances in Computers*, 43:1-50, 1996.
- [3]. K. B. Gallagher and J. R. Lyle. Using Program Slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751-761, August 1991.
- [4]. K. Pingali and G. Bilardi. Optimal Control Dependence Computation and the Roman Chariots Problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462- 491, May 1997.
- [5]. T. Langauer and R. E. Trajan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121-141, July 1979.
- [6]. Java Collections Framework (www.tutorialspoint.com)