Chapter - 8

Implementation of Our Slicing Algorithm

8.1 Introduction

For implementation of our slicing algorithm, we first devise a method for efficient the construction and storage of OPDG. This involves storage of class information along with the members. Additionally, the storage of inheritance edges, and control-flow and data-flow edges have to be considered.

In the following data structure used for storing the sliced vertices and edges is described later the modules which use this data structure to compute the slice are shown along with their significance. Display of the slice of the program using GUI is an important step and it is implemented using X-Motif.

8.2 Data Structures

The data structure used for storage of Class Hierarchy Graph is as given below:

```c
struct ClassHierarchy {
    char name[20];
    int npar;  /* no of parents of the class i.e. base classes */
    int parent[5]; /* inheritance edge */
    int par_type[5]; /* 0 for private 1 for public */
    int nprv;    /* no of private member children */
    int npub;    /* no of public member children */
    int npro;    /* no of protected member children */
    struct met prvchild[5]; /* private membership edge */
};
```
\begin{verbatim}
struct met pubchild[5]; /* public membership edge */
struct met prochild[2]; /* protected membership edge */

};

For each class the member function structure is as follows:

struct method
{
    char type[10];
    char name[20];
    int narg;
    char argtype[4][10];
    char varname[4][10]; /* formal parameters in the program */
    int par_cl; /* The class to which it belongs */
    int scope; /* 0 for private 1-public 2-protected */
    struct a *entry; /* starting (entry) node of PDG */
    node * ret_n;
}friend[5];

Structure for the storage of PDG for a procedure:

struct node
{
    char typ[4]; /* line no of the vertex */
    char text[40]; /* text of the node */
    int tag; /* for depth first traversal */
    int nchl; /* no of children */
    int np; /* no of parents */
    struct cal *callv; /* call edge */
    char group; /* Type of vertex (statement, predicate, region) */
    struct a *child[SIZE]; /* control flow */
    struct a *parent[SIZE]; /* back control flow edge */
}

/** call edges from the call vertex to the class whose instance was sent a message */
\end{verbatim}
struct calledge
{
    char clname[20];
    char f_name[20];
    int narl;
    char argtype[4][10];
    char var[4][10];
};

We use pointers to store the edges in the graph. We have pointers in both directions in order to facilitate the traversal through the graphs in either directions. For ease of implementation, the number of friend functions, classes and members in each class are constrained to some predefined maximum value.

8.3 Steps in Slicing

The implementation of the algorithm presented earlier involves the following basic steps:

1. Read the C++ routine.
2. Read the Class Hierarchy Graph.
3. Read the PDG for the individual members functions.
4. Read the slicing criterion and the type of slice i.e. forward/backward/both
5. Use the Routine_Slice to compute the slice.
6. Display the slice using GUI(Graphical User Interface).
7. Display of the edges in the logfile.(optional)
8.4 Assumptions

Many assumptions were made to simplify our implementation for static slicing. They are:

- The class declaration was done in the beginning of the program before the function declarations
- The member declaration is outside the class-definition
- No arrays and pointers are allowed in the routine
- No calls by reference
- Calls only by the instances of the class
- The variables used by the functions are assumed to be defined in the function itself.
- If any, only one return statement in a member function is assumed.

8.5 Modules Used

Following are the important modules used in slicing C++ routines:

Reading the files

name : read_prog
Purpose : Read the C++ routine

name : read_class
Purpose : Read the Class description of the C++ routine

name : read_input
Purpose : Read the PDG of the individual member functions
name : read_call

Purpose : Read the called vertices and edges in OPDG
name : read_mem

Purpose : Read the individual member functions of the class
name : read_node

Purpose : Read the vertex with respect to which slice is to be taken

Generating Outputs

name : prn_clh
Purpose : Print the CHS in a file

name : status
Purpose : Give the description of the nodes in the OPDG along with the
no. of children and parent connected to it.

Name : display
Purpose : Give the description of the edges in the OPDG

name : write_output
Purpose : output the sliced vertices in a file
Construction of OPDG from CHS and SDG

name : add_pdg
Purpose : adds the individual PDGs to the CHS to get the OPDG

Computation of Slice

name : connect
Purpose : Joins the two nodes by an edge

name : create_node
Purpose : Creates a new vertex for a statement

name : Search_node
Purpose : Searches for the slicing vertex in the PDG

name : parse_function
Purpose : Identifies the details of the member function including its type, the class to which it belongs, the parameters passed to the function and their types.

name : back_slice
Purpose : Uses reverse data and control flow edges to travel backwards in the procedure from the point ()vertex specified.

name : search_class
Purpose : Searches for the member function in the class for which call was made using call edge
name: travel_class
Purpose: Used the depth first through the inheritance edges to travel through the classes and then use search_class to fine the concerned function.

name: get_callback
Purpose: Gets the return statements from the PDG in case of backward slice

name: forward_slice
Purpose: Perform forward slice by moving down a vertex using depth first search

8.6 The Graphical User Interface (GUI)
The GUI was used for the following purposes:

1. Searching a file to be sliced using selection box
2. Viewing the selected files on the screen
3. Specifying the slicing vertex and the type of slice needed.
4. For viewing the slice of the program GUIs are preferred because of their
   • Attractiveness
   • Ease of Learning
   • Speed in use
• Reduction of error possibilities

8.7 Implementation

The GUI was implemented in Motif 1.2 under X Windows, Version 1.1 Release 5. The implemented menu-driven GUI is shown in Fig 8.1

Menus:

• **File** to open the close files. It used Selection Box for selecting the file to open and loaded on to the screen.

• **Weight** Changes the weight of the text displayed.

• **Font** Changes the font of the text displayed.

• **Size** Changes the size of the text

• **Compile** Reads the CHS, PDG of the file opened.

• **Slice** Asks for the type of slice and the slicing criterion

• **Logfile** Displays the logifle containing the edges in the slice

• **Exit** Exit for GUI
8.8 Implementation of Multi-threading Aspects

The interesting feature of Java is that multi-threading is supported with a high degree of encapsulation of each of the threads. A thread is said to be derived from the parent class Thread.
class A extend Thread
{
    int a,b:
    run()
    {
        code for the thread
    }
}

Once a variable of the type of class A is defined, the constructor is called. That is followed by the execution of the run() method. This run method is executed concurrently by each thread while the variables a, b are common to all the threads. This implies the following:

- We cannot have a thread hierarchy. It is a flat level.
- The threads are encapsulated.
- The parent/main thread has little control over the execution of the thread. Thus we can have very little dependencies between the main thread and the child threads. The dependencies arise from the shared memory dependencies of the threads belonging to the same class.

The schematic Diagram of the Java Slicer is shown in Figure 8.2. In this section, we discuss, in brief, the details of implementation.

8.9 Platform and Tools

Java is being used as the development platform. The system runs on Sun Ultra SPARC workstations running Sun’s JDK 1.1.3 and Intel Pentium II.
based workstations running GNU Kaffe version 1.0vb3 on a Red Hat Linux 5.2 platform. This allows us the develop the tool independent platform. We are using Jlex lexical analyzer (see[35]) and the CUP parser generator (see[36]).

The reason why chose the above tools on a Java platform are :

- Java allows us to build platform independent applications
- CUP and Jlex allows us to build Object Oriented parsers.
- CUP enforces a cleaner design compared to yacc
- CUP specifications are more readable, and, hence, are easily maintainable.
- GUIs are easier to build.

8.10 Abstract Syntax Tree

We have first transformed our input program into an abstract syntax tree. This leads to a two layered analysis tool. We have done this because :

- It converts the concrete syntax into a clean grammar independent abstract syntax tree.
- It frees the parser quickly i.e. immediately after the abstract syntax tree is created, the control returns to the next module.
- It leads to a modular coding.
- It leads to a Object Oriented Syntax Tree is for every production in the grammar, we define a class which stored the necessary information. In other words, the AST is a collection of these classes.
Figure 8.2: Flow chart of our slicing tool
Figure 8.3: Part of our Abstract Syntax Tree.
We have defined our own Abstract Syntax Tree for this purpose (see figure 8.3). The parser, while reducing the source program, executes the actions from the CUP specifications. This contains code for the instantiation of the objects of the AST. The parser is encapsulated into a package.

Along with the creation of the AST, we also fill up two lists per statement:

- **defs list** - this is the set of variables defined
- **refs list** - this is the referenced set

### 8.10.1 System Dependency Graph

Once the abstract syntax tree is created, we can proceed to create the system dependency graph. The control flow edges are created as a part of the parse step as we traverse the tree. This can be described briefly as:

- For each class, we write methods to construct the control flow edges, the defined and the referenced lists.
- As we traverse the AST, we use the instanceof operator (in Java) to determine the class and we generate the above information by a call to a polymorphic method. Thus a tree is created with all the control dependencies and the information for the data flow calculation.
- This half constructed tree is then taken and the data flow edges inserted. This is the most tricky task. It is non trivial for Object Oriented Programs with or without multi-threading.

The system dependency graph is difficult to create in the case of multi-threaded object oriented programs because:
• We have a hierarchical model. so we need to insert edges at the different levels of the tree.

For multi-threading, we add shared memory dependencies. These involve back-tracking along the already constructed tree several times to add the shared edges.

8.10.2 Efficient Symbol Tables

For the construction of the SDG, we need a symbol table supporting efficient access. We have designed a symbol table using hashing that can handle scoping. We have implemented scoping by putting a special marker into the hash table. When the lookup is being done (say for the symbol ‘a’). Suppose ‘a’ is defined at two nested levels. The ‘get’ operation will get the symbol from the present scope since the marker will tell us that we are in a particular scope. To begin a scope, we use the method beginScope(). After the scope is over, we do a endScope(). This pops symbols from each bucket of the hashtable till the first scope marker is found.

8.10.3 Level 0 SDG

Our algorithm to calculate the Level 0 SDG is as follows:

for each class in AST do
    create the class node and insert into PDG
    set currentNode, currentClass
    put all localvars in the environment symbol table
    for all methods in currentClass {
        create the method node
set currentmethod to this
for all statements in this method {
    beginScope();
    make a node - full up info
    check the refs /defs
    if the refs appear in the symbol table {
        link this node to the newly constructed node
    }
    else {
        create a new var node in the symbol table
        and set the defs, refs of the symbol table entry
    }
    endScope();
}

8.11 Summary

In this chapter we discussed the implementation issue of the algorithms we discussed earlier. We considered the implementation of a slicer for Java to illustrate the handling of multithreading aspects. We discussed the implementation environment and the steps followed to implement the slicer. We discussed the datastructures used and the graphical user interface design issues. The tools have been tested with a large number of test programs and the tools have been found to work satisfactorily. This proves the validity and effectiveness of our algorithms.