Carnegie Mellon
Software Engineering Institute

Program Understanding

Carnegie Mellon University
Pittsburgh, PA 15213-3890

Outline

This lecture will cover:

- **Static Analysis**
- **Program Slicing**
- **Program Plans**
- **[Reverse Engineering]**

Title
Date

Analysis

Extracts and derives information not explicitly available from data gathering

Traditional metrics

Query mechanisms for pattern matching

Presentation

Spatial and visual data

Descriptive and depictive information

Sense integration: look, feel, etc

Some current issues:

- Integration of various visual representations
- High flexibility
- Context based visualization
- Integration of various visualization techniques

Program Understanding – Supporting Techniques

- **Static Analysis**
  - Control Flow
  - Data Flow
- **Program Slicing**
- **Program Plans**

Static Analysis - Introduction

Static analysis of code a program is the analysis of the code without regard to its execution or input.

What analysis is useful for understanding:

- **Control flow analysis; what pieces of the code would be executed and in what sequence**
- **Data flow analysis; how does information flow within a program and across programs**

Control Flow – Introduction

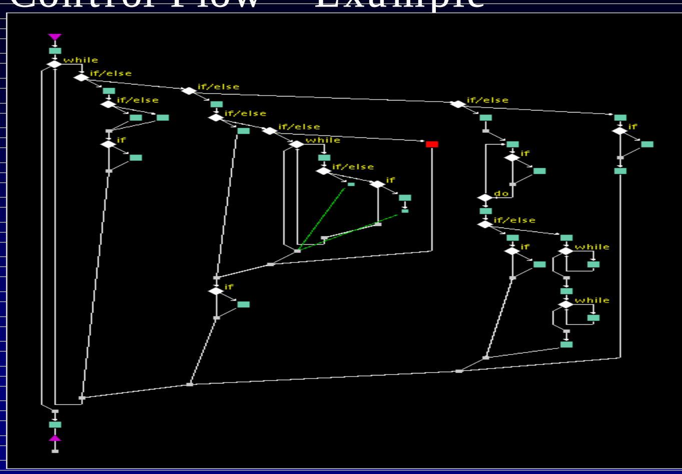
Control Flow

- **Used to identify the possible paths through the program**
- **The flow is represented as a directed graph with *splits* and *joins***
- **Identify *loops***

Control Flow represented as a graph of *Basic Blocks*

- **Sequence of operations with 1-entry and 1-exit (usually a sequence of statements)**
- **Unique *start* point where program begins**
- **Edge between basic blocks shows the flow**

Control Flow – Example



Imagix 4D representation of control flow: http://www.imagix.com

Control Flow – Code View

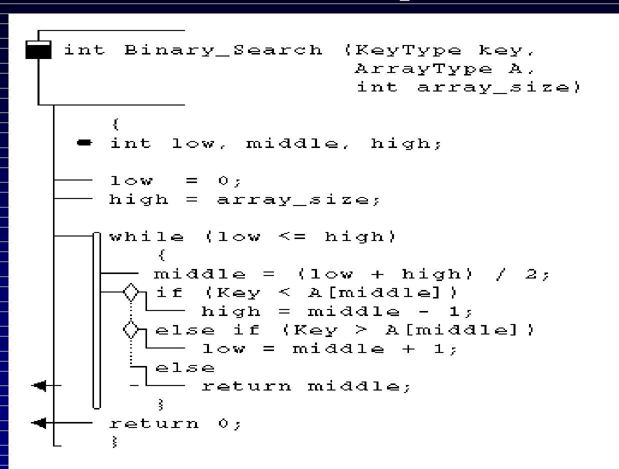
Another example of visualizing the control flow of a program is using a Control Structure Diagram (CSD). CSD is a algorithmic level graphical representation for software.

The following notations are used:

- **Sequential flow – straight line**
- **If/The/Else/Switch statements – diamonds**
- **For/While – elongated loop**
- **Loop exit – arrow**
- **Function – open-ended box**

The GRASP project at Auburn University
http://www.eng.auburn.edu/department/cse/research/grasp/

Control Flow – Example - CSD



Data Flow - Introduction

Data Flow is used to analyze the flow of data throughout a program and between program

Local Data Flow Analysis
- Analyze the effects of each statement
  - variable(s) defined
  - set of variable(s) referenced
- Compose the effects to derive information from beginning of each basic block to the statement

Data Flow Analysis
- Propagate basic block information over entire Control Flow Graph

Data Flow - Statement

Suppose we have the statement S1:

S1: a = b + c

Then:

- defined(S1) = {a}
- referenced(S1) = {b, c}

Data Flow – Types of Analysis - 1

Gen

- The set of statements where variable definitions are created in the basic block

Kill

- Set of statements that contain variables that are redefined in the basic block.

1: a := b + c	Gen: {1, 2, 3}
2: d := e + f	Kill: {}
3: f := a + b	

Data Flow – Types of Analysis - 1

IN set

↓

Gen = {…}

Kill = {…}

↓

OUT set

OUT = Gen ∪ (IN – Kill)

Data Flow – Types of Analysis - 2

Def-Use (DU) Chain
- **Connects a definition of a variable to all of the possible uses of the variable**

Use-Def (UD) Chain
- **Connects a use of a variable with all possible definitions of the variable**

Data Flow - Interprocedural

Parameter Passing Mechanisms

- **Call-by-value**
- **Call-by-reference (call-by address, call-by-location)**
- **Copy restore (copy-in copy-out, value result)**
- **Call-by-name**

Procedures and Functions as parameters

Identifier scoping problems with nested procedures

Data Flow – Difficulties

Difficulties in understanding data flow:

- **Variable Aliasing (different ways to reference the same variable)**
- **Various parameter passing mechanisms**
- **Pointers**

Program Slicing – Introduction - 1

Program Slice definition:

A slice is taken with respect to a slicing criterion <*s*,*v*>, which specifies a location (statement *s*) and a variable (*v*).

For statement *s* and variable *v*, the slice of program *P* with respect to the slicing criterion <*s*,*v*> includes only those statements of *P* needed to capture the behavior of *v* at *s*.

Program Slicing – Introduction - 2

Applications of program slicing:

- **understanding**
- **debugging**
- **testing**
- **parallelization**
- **integration**
- **software quality**
- **software maintenance**
- **software metrics**

Program Slicing – Introduction - 3

Program Slicing was first introduced by Weiser. He introduced the concept of an *executable backwards static* slice.

- ***executable* - slice is required to be an executable program**
- ***backwards* – because of the direction the edges are traversed when computing the slice using a dependence graph**
- ***static* because they are computed as the solution to a static analysis problem (without considering the program's input)**

Many applications of program slicing (such as debugging) do not require executable slices.

M. Weiser, *Program Slicing*, Proceedings of ICSE 1981, 439-449.

Program Slicing – Introduction - 4

Forward slicing (introduced by Horwitz et al.)

- **"What statements are affected by the value of *v* at statement *s*?".**

S. Horwitz and T. Reps and D. Binkley, *Interprocedural Slicing using dependence graphs*, Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation, 1988.

Dynamic Slicing (introduced by Korel and Laski)

- **A slice is computed for a particular fixed input.**

B. Korel and J. Laski, *Dynamic Program Slicing*, Information Processing Letters, 29(3), Oct 1988, 155-163

Program Slicing – Flow Graphs - 1

Slicing of a flow graph is a two-step process:

1. **Compute the data flow information**
2. **Use this information to extract a slice**

To obtain the data flow information for statement *n* we first obtain:

- ***REF(n)* – the set of variables that are referenced in *n***
- ***DEF(n)* – the set of variables defined (given a value) in *n***

The data flow information is the set of relevant variables at each node *n*.

Program Slicing – Flow Graphs - 2

For the slice with respect to <*s*,*v*> the relevant set for each node contains the variables whose values transitively affect the computation of *v* at *s*.

A statement *n* is in the slice if it assigns a value to a variable relevant at *n* and the slice taken with respect to any predicate node that directly controls *n*'s execution.

Program Slicing – Flow Graphs - 3

Relevant sets for the slice taken with respect to <*n*,*v*> are computed as follows:

1. **Initialize all relevant sets to the empty set.**
2. **Insert *v* into *relevant(n).***
3. **For *m*, *n*'s immediate predecessor, assign *relevant(m)* the value**
   $(relevant(n) - DEF(m)) \cup$
   $(REF(m)$ **if** $relevant(n) \cap DEF(m) \neq \{\})$
4. **Working backwards, repeat step 3 for *m*'s predecessors until $n_{initial}$ is reached**

Program Slicing – Flow Graph - 4

n	statement	*refs(n)*	*defs(n)*	*relevant(n)*
1	b = 1		b	
2	c = 2		c	b
3	d = 3		d	b,c
4	a = d	d	a	b,c
5	d = b + d	b,d	d	b,c
6	b = b + 1	b	b	b,c
7	a = b + c	b,c	a	b,c
8	print a	a		a

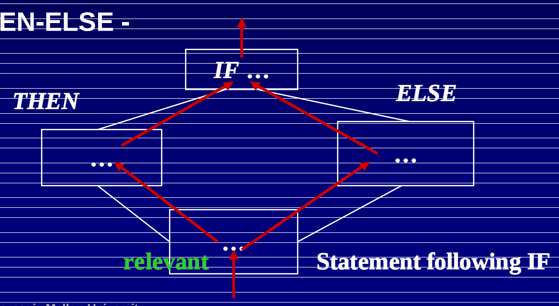
Slice on <8,a>: {7, 6, 2, 1}

Program Slicing – Flow Graph - 5

Slicing with control statements such as:

- **IF-THEN-ELSE**
- **Loop Statements**

IF-THEN-ELSE -



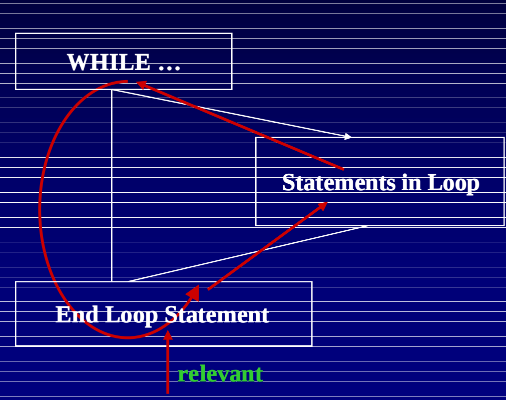
Program Slicing – Flow Graph - 6

n	statement	*refs(n)*	*defs(n)*	*control(n)*	*relevant(n)*
1	b = 1		b		
2	c = 2		c		b
3	d = 3		d		b,c
4	a = d	d	a		b,c,d
5	if (a) then	a			b,c,d
6	d = b + d	b,d	d	5	b,d
7	c = b + d	b,d	c	5	b,d
8	else			5	b,c
9	b = b + 1	b	b	8	b,c
10	d = b + 1	b	d	8	b,c
11	endif				b,c
12	a = b + c	b,c	a		b,c
13	print a	a			a

Slice on <13,a>: {12, 11, 9, 8, 7, 6, 5, 4, 3, 2, 1}

Program Slicing – Flow Graph - 7

Loop Statements:



Program Slicing – Flow Graph - 8

Loop Example:

n	statement	*refs(n)*	*defs(n)*	*control(n)*	*relevant(n) Iter 1*	*relevant(n) Iter 2*
1	b = 1		b			
2	c = 2		c			b
3	d = 5		d			b,c
4	a = 3		a			b,c
5	While (a < 10)	a			a,b,c	a,b,c
6	b = b + c	b,c	b	5	b,c	b,c
7	c = c + 1	c	c	5	b	b,c
8	a = b	b	a	5	b	b,c
9	EndWhile			5	a	
10	print a	a			a	

Slice on <10,a>: {9, 8, 7, 6, 5, 4, 2, 1}

Program Slicing – Flow Graph - 9

0	While (a)
1	$x_n = x_{n-1}$
2	$x_{n-1} = x_{n-2}$
	…
n	$x_1 = x_0$
n + 1	EndWhile

If x_n in slicing criteria - need n passes through loop

Program Slicing - Dynamic

- Dynamic Program Slicing
  - Only the dependencies that occur in a specific execution of the program are taken into account.
  - A *dynamic slicing criterion* specifies the input, occurrence of a statement, and a variable
  - dynamic slicing assumes a fixed input for a program whereas a static slice does not make assumptions about the input.
- Hybrid approaches that use both static and dynamic slicing also exist.

Program Slicing – Dynamic e.g. -1

n	statement
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	i := i + 1
10	end;
11	write(x);

What is dynamic slice with criterion (n = 2, 11, x)?

Program Slicing – Dynamic e.g. -2

n	statement
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	
9	i := i + 1
10	end;
11	write(x);

Dynamic slice with criterion (n=2, 11, x) is entire program without line 8.

Static slice (11, x) is the entire program.

Interlude

See John Field’s slides.

Program Plans – Introduction

The goal is to recognize clichés using plans.

A cliché is a pattern that appears frequently in programs (e.g., algorithms, data structures, domain-specific patterns).

A plan is an abstract representation of a cliché.

Representation is at the semantic level rather than at the syntactic level.

Clichés - Examples

Data structure clichés: lists, trees, tables, vectors, matrices

Algorithmic clichés: list, tree, graph traversals; iterators, applicators, manipulators; linear, binary, hash searches; event handler; exception handler

ADT clichés: dictionary, priority queue, heaps

Plan Recognition

Approaches:
- Top-down: Start with set of goals to be achieved; determine what plans can achieve these goals; connect these plans to source code patterns.
  - Problem: Requires detailed advance knowledge otherwise connection to code is unrealistic
- Bottom-up: Start with source code; identify plans that match source code; infer higher-level goals from these plans.
  - Problem: Combinatorial explosion of alternatives
- Hybrid: top-down and bottom-up

Plan Recognition – Method

Typical method of plan recognition
- An effective (language-independent) program representation
- A translator to transform source text into this program representation
- A library of programming plans representing clichés at various levels of abstraction
- A plan recognizer which parses the program to recognize plans stored in the library
- The result is a tree or lattice with program components at the leaves, programming plans, and the goals of the program at the root
- Bottom-up program understanding

Plan Recognition - Issues

Syntactic variations: Recognizer works on the basis of structure information only; syntactic variations lead to the same paraphrase, modulo identifiers
Non-contiguousness: Recognizer works with graph structures, can accommodate equivalent sequences of statements
Implementation variations: Similar programs are matched against the same plans, lead to the same paraphrases
Recognition algorithm depends polynomially on size of the program and plan library; graph grammars and graph recognition algorithms deployed

Reverse Engineering Activities

The three main Reverse Engineering activities:

- Data Gathering
- Knowledge Organization
- Information Exploration

Data Gathering

Raw data is used to identify a system's artifacts and relationships

Techniques include:

- Static source code analysis (parsing)
- Dynamic Analysis (profiling)
- Informal extraction (interviewing)

Knowledge Organization

Goals of Knowledge Organization are:
- Efficient storage of knowledge
- Permit automated analysis
- Reflect user's perspective

Classical data models
- Hierarchical
- Network
- Relational

Abstraction Mechanisms

Abstraction: Selective emphasis on detail

Common Mechanisms:
- **Classification**
- **Aggregation**
- **Generalization**

Conceptual Modeling

Information Exploration

Probably the most important activity:
- **Data gathering: necessary to begin**
- **Knowledge organization: structure model**
- **Information Exploration: understanding**

Composite Activities:
- **Navigation**
- **Analysis**
- **Presentation**

Navigation

Traverse non-linear information structures

Link relationships:

- **Component hierarchies**
- **Inheritances**
- **Control and data flow**

Hypotheses postulation ⇒ exploration