Chapter 6
Static Analysis

J. C. Huang
Department of Computer Science
University of Houston
Static Analysis

*Static analysis* is a process in which we attempt to find faults in a program by examining the source code systematically without test-executing it.
Capability

It can be used to detect

- errors in event sequencing,
- structural flaws in a program module, and
- flaws in module interface.
Types of structural flaw detectable

- some syntactically correct but logically suspicious constructs.
- Improper loop nesting.
- Unreferenced labels.
- Unreachable statements.
- Transfer of control into a loop.
Example of syntactically correct but semantically suspicious construct

For example, in C++, a beginner may write

```c
char* p;
strcpy( p, "Houston" );
```

which is syntactically correct but semantically wrong. It should be written like

```c
char* p;
p = buffer;       //p points to buffer
strcpy( p, "Houston" );  //place a copy of "Houston" in buffer
```
Types of interface flaw detectable

- Inconsistencies in the declaration of data structures.
- Improper linkage among modules (e.g., discrepancy in the number and types of parameters).
- Flaws in other inter-program communication mechanism such as common blocks.
Detectable event-sequencing errors

- Priority interrupt handling conflict
- Error in file handling
- Data-flow anomaly
- Anomaly in concurrent programs
Data-flow Anomaly

When a program is being executed, it may act on a variable (datum) in three different ways, namely, define, reference, and undefine.
Data-flow Anomaly (continued)

The dataflow with respect to a variable is said to be anomalous if the variable is either undefined and referenced, defined and then undefined, or defined and defined again.
Data-flow Anomaly (continued)

The presence of a data-flow anomaly in the program is only a symptom of possible programming error. The program may or may not be in error.
Data-Flow Anomaly Detection in Concurrent Programs

Possible events that may occur:

- define
- reference
- undefine
- schedule
- unschedule (not scheduled)
- wait
Possible types of anomaly:

- a dead definition of a variable
- waiting for a process not scheduled
- scheduling a process in parallel with itself
- waiting for a process guaranteed to have terminated previously
- referencing an uninitialized variable
- referencing a variable which is being defined by a parallel process
- referencing a variable whose value is indeterminate
Example program

(see 6static port)
The process-augmented flow-graph

(1) Main: program
(15) schedule T1
(16) schedule T2
(17) def flag
(18) if flag
  then def x
(19) ref x
(20) def y
(21) wait for T2
(22) if flag
  then def y
(23) ref y
(24) wait for T2
(25) schedule T1
(26) close Main

(4) T1: task
  (5) ref x
  (6) wait for T3
  (7) close T1

(8) T2: task
  (9) def x
  (10) def y
  (11) close T2

(12) T3: task
  (13) def x
  (14) close T3

(36x36 to 46x46)
Possible anomalies

- An uninitialized variable (x) may be referenced at line 5, as task T1 may execute to completion before T2 begins.

- The definitions of y as found in task T2 (line 10) and the main program (line 20) may be useless since y may be redefined at line 22 before y is ever referenced.
Possible anomalies (continued)

- y is defined by two processes that may be executed concurrently, and thus the reference at line 23 may be to an indeterminate value.

- Variable x is assigned a value by task T2 (line 9) while simultaneously being referenced by the main program at line 19.
Possible anomalies (continued)

- There is a possibility that task T1 will be scheduled in parallel with itself at line 25 since there is no guarantee that T1 terminates after its initial scheduling.
- The wait at line 24 is unnecessary, as T2 was guaranteed to have terminated at line 21, and it has not been scheduled subsequently.
- The wait at line 6 will never be satisfied as T3 was never scheduled.
Symbolic Evaluation (Execution)

The basic idea is to execute the program with symbolic inputs and produce symbolic formulae as output.
Example:

```plaintext
read(x, y);
z := x + y;
x := x - y;
z := x * z;
write(z);
```
Ordinary execution with $x = 2$ and $y = 4$.

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(x, y);</td>
<td>2</td>
<td>4</td>
<td>undefined</td>
</tr>
<tr>
<td>z := x + y;</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>x := x - y</td>
<td>-2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>z := x * z;</td>
<td>-2</td>
<td>4</td>
<td>-12</td>
</tr>
<tr>
<td>write(z);</td>
<td>-2</td>
<td>4</td>
<td>-12</td>
</tr>
</tbody>
</table>
Symbolic execution with $x = a$ and $y = b$

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(x,y);</td>
<td>a</td>
<td>b</td>
<td>undefined</td>
</tr>
<tr>
<td>z:=x+y;</td>
<td>a</td>
<td>b</td>
<td>a+b</td>
</tr>
<tr>
<td>x:=x-y</td>
<td>a-b</td>
<td>b</td>
<td>a+b</td>
</tr>
<tr>
<td>z:=x*z;</td>
<td>a-b</td>
<td>b</td>
<td>a<em>a-b</em>b</td>
</tr>
<tr>
<td>write(z);</td>
<td>a-b</td>
<td>b</td>
<td>a<em>a-b</em>b</td>
</tr>
</tbody>
</table>
Path condition

If the program consists of more than one execution path, it is necessary to choose a path through the program to be followed, and the result of execution should include path condition, or $pc$ for short, which is a Boolean expression over the symbolic values.
Execution tree

The execution paths followed can be conveniently represented by an execution tree.

To construct such a tree, associate a node with each statement executed by labeling it with the statement number, and with each possible control transfer with a directed edge connecting the associated nodes.
Example:

```c
main()
{
    int x, y, z;
    scanf("%d %d", &x, &y);
    z = 1;
    while (y != 0) {
        if ((y / 2) * 2 != y)
            z = z * x;
        y = y / 2;
        x = x * x;
    }
    printf("%10d\n", z);
}
```
Example (continued):

The execution tree of this program is shown below.

Note that the execution tree of a program with loop constructs is infinite.
Execution tree of the example
Comment

Generally speaking, the usefulness of symbolic execution is limited to numerical programs designed to compute a function describable by a closed formula.
Example

For example, the technique is useful to the following Fortran program designed to solve quadratic equations by using the formula:

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]
The example program

See the slide in the other file.
A trace subprogram

READ (5, 11) A, B, C
\ NOT. (A .EQ. 0.0 .AND. B .EQ. 0.0 .AND. C .EQ. 0.0)
\ (A .NE. 0.0 .OR. B .NE. 0.0)
\ (A .NE. 0.0)
\ (C .NE. 0.0)
RREAL = -B/(2.0*A)
DISC = B**2 - 4.0*A*C
RIMAG = SQRT(ABS(DISC))/(2.0*A)
\ NOT. (DISC .LT. 0.0)
R1 = RREAL + RIMAG
R2 = RREAL - RIMAG
WRITE (6, 31) R1, R2
We can rewrite it into the canonical form first,

```fortran
READ (5, 11) A, B, C
\( (A \neq 0.0 \lor B \neq 0.0 \lor C \neq 0.0) \)
\( (A \neq 0.0 \lor B \neq 0.0) \)
\( (A \neq 0.0) \)
\( (C \neq 0.0) \)
\( (B^2 - 4.0*A*C \geq 0.0) \)
RREAL = -B/(2.0*A)
DISC = B**2 - 4.0*A*C
RIMAG = SQRT(ABS(DISC))/(2.0*A)
R1 = RREAL + RIMAG
R2 = RREAL - RIMAG
WRITE (6, 31) R1, R2
```
and then symbolically execute it to yield

\[
R_1 = -\frac{B}{2.0A} + \frac{\sqrt{\text{ABS}(B^2 - 4.0AC)}}{2.0A}
\]

\[
R_2 = -\frac{B}{2.0A} - \frac{\sqrt{\text{ABS}(B^2 - 4.0AC)}}{2.0A}
\]

\[
\text{pc: } A \neq 0.0 \text{ AND } C \neq 0.0 \text{ AND } B^2 - 4.0AC \geq 0.0
\]

This demonstrate the usefulness of a symbolic execution because it clearly indicates what the program will do for the cases where the path condition \(\text{pc}\) is satisfied.
Another possible application

Symbolic execution can also be used to guide simplification of source code. For example, consider the following segment of code:

```c
r = a % b;
if (b == 0) { exit(-1); }
a = b;
b = r;
r = a % b;
```

Symbolic execution with $a=A$ and $b=B$

<table>
<thead>
<tr>
<th>after execution of statement</th>
<th>the symbolic values becomes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r=a % b$</td>
<td>$a=A % B$</td>
</tr>
<tr>
<td>$a=b$</td>
<td>$b=B$</td>
</tr>
<tr>
<td>$b=r$</td>
<td>$a=A % B$</td>
</tr>
<tr>
<td>$r=a % B$</td>
<td>$b=A % B$</td>
</tr>
<tr>
<td>$a=b$</td>
<td>$r=B % (A % B)$</td>
</tr>
<tr>
<td>$b=r$</td>
<td>$a=A % B$</td>
</tr>
<tr>
<td></td>
<td>$b=B % (A % B)$</td>
</tr>
</tbody>
</table>
Suggested simplification

The result of symbolic execution strongly suggests that the code can be simplified to:

\[
\begin{align*}
  a &= a \% b; \\
  r &= b \% a; \\
  b &= r;
\end{align*}
\]
Comment

In general, the result of a symbolic execution is a set of strings (symbols) representing the values of the program variables. These strings often grow uncontrollably during the execution. Thus the results may not be of much use unless the symbolic execution system is capable of simplifying these strings automatically.

Such a simplifier basically requires the power of a mechanical theorem prover. Therefore, a symbolic execution system is a computationally intensive software system, and is relatively difficult to build.
Code Inspection

Code inspection (walk-through) is a process designed to assure high quality of the software produced. It should be carried out after the first clean compilation of the code to be inspected, and before any formal testing is done on that code.
Objectives

(a) to find logic errors,
(b) to verify the technical accuracy and completeness of the code,
(c) to verify that the programming language definition used conforms to that of the compiler to be used by the customer,
Objectives (continued)

(d) to ensure that no conflicting assumptions or design decisions have been made in different parts of the code, and

(e) to ensure that good coding practices and standards are used, and the code is easily understandable.
The team should include

(a) the designer who will answer any question,
(b) the moderator who ensures that any discussion is topical and productive,
(c) the paraphraser who steps through the code and paraphrase it in English, and
(d) the librarian or recorder.
Material needed

(a) program listings and design documents,
(b) a list of assumptions and decisions made in coding, and
(c) a participant-prepared list of problems and minor errors.
Comment

The purpose of a code inspection should not be to evaluate the competence of the author of the code, or to unnecessarily criticize coding style. The style of the code should not be discussed unless it prevents the code from meeting the objectives of the code inspection.
Products

(a) a summary report which briefly describes the problems found during the inspection,

(b) a form for listing each problem found so that its disposition or resolution can be recorded, and

(c) a list of updates made to the specifications and changes made to the code.
Reinspect when

(a) a nontrivial change to the code is required, or
(b) the number of problems found exceeds one for every 25 non-commentary lines of the code.
Reschedule when

(a) any mandatory participant can not be in attendance,
(b) the material needed for inspection is not made available to the participants in time for preparation,
(c) there is a strong evidence to indicate that the participants are not properly prepared,
(d) the moderator can not function effectively for some reason, or
(e) material given to the participants is found to be not up-to-date.
Comment

The process described above is to be carried out manually. Some part of which, however, can be done more readily if proper tools are available.

For example, in preparation for a code inspection, if the programmer find it difficult to understand certain parts of the source code, software tools can be used to facilitate understanding. Such tools can be built based on the program analysis method described in Sec. 1.6, and the technique of program slicing outlined in the next section.
Program slicing

Program slicing is a method for abstracting from a program. Given a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior.

The reduced program, called a *slice*, is an independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behavior.
Example program P

1 begin
2    read(x, y);
3    total := 0.0;
4    sum := 0.0;
5    if x <= 1
6      then sum := y
7      else begin
8         read(z);
9         total := x*y
10      end;
11    write(total, sum)
12  end.
Example slice $S_1$

Slice on the value of $z$ at statement 12:

```plaintext
1    begin
2      read(x, y);
5      if x <= 1
6        then
7        else begin
8          read(z);
10        end;
12    end.
```
Example slice $S_2$

Slice on the value of total at statement 12:

1    begin
2       read(x, y);
3       total := 0.0;
5       if x <= 1
6           then
7           else begin
9               total := x*y
10              end;
12       end.
Example slice $S_3$

Slice on the value of $x$ at statement 9:

1    begin
2       read($x$, $y$);
12     end.
Slicing criterion

The specification of a subset of program behavior is called a slicing criterion.

It consists of a specific statement in the program (that fixes a point in the control flow) and a set of variables.
DEF and REF sets

**Definition:** Let P be a program, and suppose that the statements are numbered consecutively. Then for each statement n in P we can define two sets: \( \text{REF}(n) \) is the set of all variables referenced at n, and \( \text{DEF}(n) \) is the set of all variables defined at n.
Value trace

**Definition:** A value trace of a program P is a finite list of ordered pairs

\[(n_1, s_1)(n_2, s_2) \ldots (n_k, s_k)\]

where each \(n_i\) denotes a statement in P, and each \(s_i\) is a vector of values of all variables in P immediately before the execution of \(n_i\).
Example

Consider the program listed in the next slide in which the vector of variables used is

\[<x, y, z, \text{sum, total}>\]
Example program

begin
read(x, y);
total := 0.0;
sum := 0.0;
if x <= 1
then sum := y
else begin
read(z);
total := x*y
end;
write(total, sum)
end.
A value trace

\[ T_1: (1, <?, ?, ?, ?, ?>) \]
\[ (2, <?, ?, ?, ?, ?>) \]
\[ (3, <X, Y, ?, ?, ?>) \]
\[ (4, <X, Y, ?, ?, 0.0>) \]
\[ (5, <X, Y, ?, 0.0, 0.0>) \]
\[ (6, <X, Y, ?, 0.0, 0.0>) \]
\[ (11, <X, Y, ?, Y, 0.0>) \]
\[ (12, <X, Y, ?, Y, 0.0>) \]
Another possible value trace

\[ T_2: \]
\[ (1, <?, ?, ?, ?, ?>) \]
\[ (2, <?, ?, ?, ?, ?>) \]
\[ (3, <X, Y, ?, ?, ?>) \]
\[ (4, <X, Y, ?, ?, 0.0>) \]
\[ (7, <X, Y, ?, 0.0, 0.0>) \]
\[ (8, <X, Y, ?, 0.0, 0.0>) \]
\[ (9, <X, Y, Z, 0.0, 0.0>) \]
\[ (10, <X, Y, Z, 0.0, X*Y>) \]
\[ (11, <X, Y, Z, 0.0, X*Y>) \]
\[ (12, <X, Y, Z, 0.0, X*Y>) \]
Remark

In the above we use a question mark (?) to denote an undefined value, and a variable name in upper case to denote the value of that variable obtained through an input statement in the program.
Slicing criterion

**Definition:** A slicing criterion of a program $P$ is an ordered pair $(i, V)$, where $i$ is a statement in $P$ and $V$ is a subset of the variables in $P$. 
Example slicing criteria

\[ C_1: (12, \{z\}), \]

\[ C_2: (12, \{\text{total}\}), \text{ and} \]

\[ C_3: (9, \{x\}). \]
Projection

Definition: Given a slicing criterion \( C = (i, V) \) and a value trace \( T \), we can define a projection function \( \text{Proj}(C, T) \) that deletes from a value trace all ordered pairs except those with \( i \) as the left component, and from the right components of the remaining pairs all values except those of variables in \( V \).
Example projection

\[
\text{Proj}(C_1, T_1) = \text{Proj}((12, \{z\}), T_1)
\]
\[
= \text{Proj}((12, \{z\}), (1, <?, ?, ?, ?, ?>)
\]
\[
(2, <?, ?, ?, ?, ?>)
\]
\[
(3, <X, Y, ?, ?, ?>)
\]
\[
(4, <X, Y, ?, ?, 0.0>)
\]
\[
(5, <X, Y, ?, 0.0, 0.0>)
\]
\[
(6, <X, Y, ?, 0.0, 0.0>)
\]
\[
(11, <X, Y, ?, Y, 0.0>)
\]
\[
(12, <X, Y, ?, Y, 0.0>)
\]
\[
= (12, <?>)
\]
Another example projection

\[
\text{Proj}(C_2, T_1) = \text{Proj}((12, \{\text{total}\}), T_1) \\
= \text{Proj}((12, \{\text{total}\}), (1, <?, ?, ?, ?, ?>) \\
(2, <?, ?, ?, ?, ?>) \\
(3, <X, Y, ?, ?, ?>) \\
(4, <X, Y, ?, ?, 0.0>) \\
(5, <X, Y, ?, 0.0, 0.0>) \\
(6, <X, Y, ?, 0.0, 0.0>) \\
(11, <X, Y, ?, Y, 0.0>) \\
(12, <X, Y, ?, Y, 0.0>) \\
= (12, <0.0>)
\]
Yet another example projection

\[
\text{Proj}(C_3, T_2) = \text{Proj}((9, \{x\}), T_2) \\
= \text{Proj}((9, \{x\}), (1, <?, ?, ?, ?, ?>) \\
(2, <?, ?, ?, ?, ?>) \\
(3, <X, Y, ?, ?, ?>) \\
(4, <X, Y, ?, ?, 0.0>) \\
(7, <X, Y, ?, 0.0, 0.0>) \\
(8, <X, Y, ?, 0.0, 0.0>) \\
(9, <X, Y, Z, 0.0, 0.0>) \\
(10, <X, Y, Z, 0.0, X*Y>) \\
(11, <X, Y, Z, 0.0, X*Y>) \\
(12, <X, Y, Z, 0.0, X*Y>) \\
= (9, <X>)
\]
Formal definition of a slice

**Definition**: A slice \( S \) of a program \( P \) on a slicing criterion \( C = (i, V) \) is any executable program satisfying the following two properties:

(a) S can be obtained from \( P \) by deleting zero or more statement from \( P \).

(b) Whenever \( P \) halts on a input \( I \) with value trace \( T \), \( S \) also halts on input \( I \) with value trace \( T' \), and \( \text{Proj}(C, T) = \text{Proj}(C', T') \), where \( C' = (i', V) \), and \( i' = i \) if statement \( i \) is in the slice, or \( i' \) is the nearest successor to \( i \) otherwise.
Example

Again, consider P, the example program listed in the next slide, and the slicing criterion \( C_1 = (12, \{z\}) \). According to the above definition, \( S_1 \) is a slice because if we execute P with any input \( x = X \) such that \( X \leq 1 \), it will produce the value trace \( T_1 \), and as given previously, \( \text{Proj}(C_1, T_1) = (12, \langle ? \rangle) \).
Example program P

1  begin
2    read(x, y);
3    total := 0.0;
4    sum := 0.0;
5    if x <= 1
6      then sum := y
7      else begin
8          read(z);
9          total := x*y
10        end;
11    write(total, sum)
12  end.
Example (continued)

Now if we execute $S_1$ with the same input, it should yield the following value trace:

$T'_1$: 
(1, $? , ?, ?, ?, ?>) 
(5, <X, Y, ?, ?, ?>) 
(6, <X, Y, ?, ?, ?>) 
(12, <X, Y, ?, ?>)
Example (continued)

Since statement 12 exists in P as well as $S_1$, $C_1 = C'_1$, and

\[
\text{Proj}(C'_1, T'_1) = ((12, \{z\}), T'_1) = (1, <?, ?, ?, ?, ?>) \\
(2, <?, ?, ?, ?, ?>) \\
(5, <X, Y, ?, ?, ?>) \\
(6, <X, Y, ?, ?, ?>) \\
(12, <X, Y, ?, ?, ?>) = (12, <?>) = \text{Proj}(C_1, T_1)
\]
Example (continued)

Hence \( S_1 \) is a slice of \( P \).

As yet another example in which \( C \subseteq C' \), consider \( C = (11, \{z\}) \). Since statement 11 is not in \( S_1 \), \( C' \) will have to be set to \( (12, \{z\}) \) instead because statement 12 is the nearest successor of 11.
Comment

There can be many different slices for a given program and slicing criterion. There is always at least one slice for a given slicing criterion -- the program itself.
Comment

The above definition of a slice is not constructive in that it does not say how to find one. The smaller the slice the better. However, finding minimal slices is equivalent to solving the halting problem -- it is impossible.