6.1 Introduction

By static analysis here we mean a process in which we attempt to find faults in a program by analyzing the source code of the program without test-executing it on a computer.

Static analysis can be used to detect structural flaws in a program module, flaws in module interface, and errors in event sequencing.

Types of structural flaw detectable by static analysis include

- Certain syntactically correct but logically suspicious constructs
  For example, in C++, a beginner may write

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

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

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

  Another example: Because of an extraneous semicolon at the end of a FOR statement like the one shown below, its body will never be executed until the empty FOR statement is terminated.

  ```
  for ( i = 0, j = n – 1; i < j; i++, j-- ){
    char temp = a[i];
    a[i] = a[j];
    a[j] = temp;
  }
  ```

- Improper loop nesting
  This could happen if a GOTO statement is used in nested loops.

- Unreferenced labels
  They do not necessarily lead to any faults, but they definitely indicate that the programmer has overlooked something.

- Unreachable statements
Improper placement of a RETURN statement or incorrect formulation of a branch predicate (which becomes a tautology or contradiction) may cause of segment of code to become unreachable.

- Transfer of control into a loop
  This could happen if the use of GOTO statement is allowed.

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

Types of event-sequencing error detectable:
- Priority-interrupt handling conflict
- Error in file handling
- Dataflow anomaly
- Anomaly in concurrent programs

### 6.2 Dataflow Anomaly

**Basic Idea**

In execution a program component may act on a variable (datum) in three different ways, namely, *define*, *reference*, and *undefine*.

For example, when statement \( x := x + y - z \) is executed, \( y \) and \( z \) are referenced whereas \( x \) is referenced first and then defined. A local variable of a subroutine becomes undefined when return statement is executed.

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 [FOOS76].

The presence of a dataflow anomaly in the program is only a symptom of possible programming error. The program may or may not be in error.

**Anomaly Detection in Concurrent Programs:**

Possible events that may occur (or, possible actions that may be taken by a program) [TAOS80]:
- define
- reference
- undefine
- schedule
- unschedule (not scheduled)
Possible types of anomaly include:

a) referencing an uninitialized variable
b) a dead definition of a variable
c) waiting for a process not scheduled
d) scheduling a process in parallel with itself
e) waiting for a process guaranteed to have terminated previously
f) referencing a variable which is being defined by a parallel process
g) referencing a variable whose value is indeterminate

An example HAL/S program with several dataflow and synchronization anomalies [TAOS80]:

```hal/s
program Main;

declare integer x, y;
/* x and y are global variables known throughout the main programs and all tasks */
declare boolean flag;

T1: task;
    write x;
    wait for T3;
    close T1;

T2: task;
    x = 5;
    y = 6;
    close T2;

T3: task;
    read x;
    close T3;

/* end of declarations */
schedule T1; /* first executable statement of Main */
schedule T2;
read flag;
if flag then x = 8;
write x;
y = 9;
wait for T2;
if flag then y = 10;
write y;
wait for T2;
```
The process-augmented flowgraph for the above program:

Possible anomalies in the example HAL/s program:

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

b) 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.

c) Variable y is defined by two processes that may be executed concurrently, and thus the reference at line 23 may be to an indeterminate value.
d) Variable x is assigned a value by task T2 (line 9) while simultaneously being referenced by the main program at line 19.
e) 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.
f) The wait at line 24 is unnecessary, as T2 was guaranteed to have terminated at line 21, and it has not been scheduled subsequently.
g) The wait at line 6 will never be satisfied as T3 was never scheduled.

6.3 Symbolic Evaluation (Execution)

If we test a program for an input, we can only determine if it will work correctly for that particular case.

Now if we can find some way to execute the program in such a way that it can accept symbolic inputs and produce symbolic formulae as output, we should be able to determine the correctness of the program for a more general case. This argument led to the development of symbolic execution [KING75, KING76].

To fix the idea, let us consider the following example program:

\[
\begin{align*}
\text{read}(x, y); \\
z &:= x + y; \\
x &:= x - y; \\
z &:= x \times z; \\
\text{write}(z);
\end{align*}
\]

This program should produce the following value trace if it is executed in an ordinary manner with \( x = 2 \) and \( y = 4 \).

\[
\begin{array}{ccc}
\text{value trace} \\
x & y & z \\
\hline \\
\text{read}(x, y); & 2 & 4 & \text{undefined} \\
z &:= x + y; & 2 & 4 & 6 \\
x &:= x - y; & -2 & 4 & 6 \\
z &:= x \times z; & -2 & 4 & -12 \\
\text{write}(z); & -2 & 4 & -12 \\
\end{array}
\]

Now if we execute this program symbolically with \( x = a \) and \( y = b \), the following value trace will result.

\[
\begin{array}{ccc}
\text{value trace} \\
x & y & z \\
\hline \\
\text{read}(x, y); & 2 & 4 & \text{undefined} \\
z &:= x + y; & 2 & 4 & 6 \\
x &:= x - y; & -2 & 4 & 6 \\
z &:= x \times z; & -2 & 4 & -12 \\
\text{write}(z); & -2 & 4 & -12 \\
\end{array}
\]
As one can see from the above example, when an expression is symbolically evaluated, the symbolic values of variables in the expression are substituted into the expression. If the expression constitutes the right part of an assignment statement, the resulting symbolic value becomes the new symbolic value of the variable on the left of the assignment operator.

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 predicate over the symbolic values. For the purpose of computing \( pc \), it is useful to think of it as the accumulator of conditions that the inputs must satisfy in order for a particular path to be followed. Each symbolic execution starts with \( pc \) initialized to TRUE. As assumptions about the inputs are made in choosing between alternative paths, those assumptions are added conjunctively to \( pc \).

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. For example, consider the following C program:

```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);
}
```

The execution tree of this program is shown below:
Note that the execution tree of a program with loop constructs is infinite.

A symbolic execution of the above program would not give much clue as to what it does. Generally speaking, the usefulness of symbolic execution is limited to numerical programs designed to compute a function describable by a closed formula. For example, the technique is more useful to the following Fortran program designed to solve quadratic equations by using the formula:

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]
C OBTAINS SOLUTIONS OF THE EQUATION A*X**2 + B*X + C = 0
C
10 READ (5, 11) A, B, C
11 FORMAT (3F10, 0)
   WRITE (6, 12) A, B, C
12 FORMAT (' 0A =', 1PE16.6, ', B =', 1PE16.6, ', C =', 1PE16.6)
   IF (A .EQ. 0.0 .AND. B .EQ. 0.0 .AND. C .EQ. 0.0) STOP
   IF (A .NE. 0.0 .OR. B .NE. 0.0) GOTO 20
   WRITE (6, 13) C
13 FORMAT (' EQUATION SAYS', 1PE16.6, ' = 0')
   GOTO 90
20 IF (A .NE. 0.0) GOTO 30
   R1 = -C/B
   WRITE (6, 21) R1
21 FORMAT (' ONE ROOT. R = ', 1PE16.6)
   GOTO 90
C A IS NOT ZERO
30 IF (C .NE. 0.0) GOTO 40
   R1 = -B/A
   R2 = 0.0
   WRITE (6, 31) R1, R2
31 FORMAT (' R1 =', 1PE16.6, ' , R2 =', 1PE16.6)
   GOTO 90
C GENERAL CASE: A, C NON-ZERO
40 RREAL = -B/(2.0*A)
   DISC = B**2 - 4.0*A*C
   RIMAG = SQRT(ABS(DISC))/(2.0*A)
   IF (DISC .LT. 0.0) GOTO 50
   R1 = RREAL + RIMAG
   R2 = RREAL - RIMAG
   WRITE (6, 31) R1, R2
   GOTO 90
50 R1 = -RIMAG
   WRITE (6, 51) RREAL, RIMAG, RREAL, R1
51 FORMAT (' R1 = (', 1PE16.6, ', ', 1PE16.6,  ')',
      ', R2 = (', 1PE16.6, ', ', 1PE16.6, ', ')')
90 GOTO 10
END

Listed below is the trace subprogram representing a possible execution path in this program.

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)
\ (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 its canonical form first,

READ (5, 11) A, B, C
\ (A .NE. 0.0 .OR. B .NE. 0.0 .OR. C .NE. 0.0)
and then perform a symbolic execution (by using the name of each variable as its symbolic value) along this path to yield

\[
\begin{align*}
R1 &= \frac{-B}{2.0 \times A} + \frac{\sqrt{\text{ABS}(B^2 - 4.0 \times A \times C)}}{2.0 \times A} \\
R2 &= \frac{-B}{2.0 \times A} - \frac{\sqrt{\text{ABS}(B^2 - 4.0 \times A \times C)}}{2.0 \times A}
\end{align*}
\]

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

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 unintrolably 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 computationlly intensive software system, and is relative difficult to build.

### 6.4 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**

The main objectives of code inspection are

1. to find logic errors,
2. to verify the technical accuracy and completeness of the code,
3. to verify that the programming language definition used conforms to that of the compiler to be used by the customer,
4. to ensure that no conflicting assumptions or design decisions have been made in different parts of the code, and
5. to ensure that good coding practices and standards are used, and the code is easily understandable.
NOTE: The purpose of a code inspection should not be to evaluate the competence of the author of the code, or to criticize coding style unnecessarily. The style of the code should not be discussed unless it prevents the code from meeting the objectives of the code inspection.

Procedure

A code inspection should involve at least three persons. The inspection team should include

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

The material needed for inspection includes

(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.

These should ideally be distributed well before the inspection so that each participant has a good understanding of the purpose of the code, and how that purpose is achieved.

In a code inspection the paraphraser walks through the code, paraphrasing or enunciating it in English, with the other participants following along with him or her. The main function of the paraphraser is to make sure that all the code is covered, and that every participant is focused on the same part of the code at the same time.

The moderator leads the discussion to ensure that the objectives of the code inspection are effectively and efficiently met.

Major problems should not be resolved during the code inspection. Points should be brought up, clarified, and noted, and that is all. Afterwards, the designer, with any required consultation, can resolve the problems without burdening the participants unnecessarily.

A participant -- can be anyone other than the paraphraser -- should be assigned to take notes to record required changes. A copy of these notes or a list of changes which resolve the points raised should be made available to each participant.

The following types of report should be prepared as the results of a code inspection:

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

The code should be reinspected if

(a) a nontrivial change to the code was required, or
(b) the rate of problems found in the program exceeded a certain limit prescribed by the organization, say, one for every 25 non-commentary lines of the code.

A code inspection should be terminated and rescheduled if

(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.

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 found 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 constraint-based program analysis method [HUAN90] described elsewhere and the technique of program slicing outlined in the next section.

### 6.5 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 that 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 [WEIS84].

To illustrate, let us consider the following 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.
```

Slice on the value of \(z\) at statement 12:

```
S_1:
1 begin
2   read(x, y);
5   if x <= 1
```
then
else begin
    read(z);
end;
end.

Slice on the value of total at statement 12:

S₂:
begin
    read(x, y);
    total := 0.0;
    if x <= 1 then
        else begin
            total := x*y
        end;
end.

Slice on the value of x at statement 9:

S₃:
begin
    read(x, y);
end.

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. A slice can be defined as follows.

**Definition 6.5.1:** 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: REF(n) is the set of all variables referenced at n, and DEF(n) is the set of all variables defined at n.

**Definition 6.5.2:** A slicing criterion of 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.

For example, three slicing criteria were used to construct three example slices at the beginning of this section. By the above definition, they can be expressed as

- \( C₁ : (12, \{z\}) \),
- \( C₂ : (12, \{total\}) \), and
- \( C₃ : (9, \{x\}) \).

Given a slice criterion (n, V) of program P, a slice can be obtained by deleting all statements that are irrelevant to computation of value of any variable in V at statement n. We can search for such statements backward starting at statement n. Let REF(n) = V and let statement m be an immediate predecessor of statement n. If REF(n) \( \cap \) DEF(m) \( \neq \) \( \emptyset \), i.e., if there was a variable
used in statement n that was defined in statement m then statement m is relevant, and therefore must be included in the slice. Otherwise statement m can be excluded from the slice.

For example, consider the slicing criterion (12, {total}). Let R(12) = V = {total}. Statement 11 is the only predecessor, and D(11) = ∅. REF(12) ∩ DEF(11) = ∅, statement 11 is irrelevant and thus can be excluded. With statement 11 excluded, now statement 5 (a compound conditional statement that includes lines 5 through 10) becomes the immediate processor. DEF(5) = {sum, z, total}, REF(12) ∩ DEF(5) = {total} ≠ ∅. Therefore, statement 5 is relevant, and should be included in the slice.

Although the idea of a program slice is very simple, it is difficult to define it formally. In the original work on slice [WEIS84], it is formally defined as follows.

**Definition 6.5.3:** A value trace (of length k) of a program P is a finite list of ordered pairs

\[(n_1, v_1)(n_2, v_2) ... (n_k, v_k)\]

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

For example, the above program makes use of a vector of variables: <x, y, z, sum, total>. A possible value trace would be

\[T_1: (1, <?, ?, ?, ?, ?>)\]
\[(2, <?, ?, ?, ?, ?>)\]
\[(3, <X, Y, ?, ?, ?>)\]
\[(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>\)]

and another possible value trace would be

\[T_2: (1, <?, ?, ?, ?, ?>)\]
\[(2, <?, ?, ?, ?, ?>)\]
\[(3, <X, Y, ?, ?, ?>)\]
\[(4, <X, Y, ?, ?, 0.0>\)]
\[(5, <X, Y, ?, 0.0, 0.0>\)]
\[(6, <X, Y, ?, 0.0, 0.0>\)]
\[(7, <X, Y, ?, 0.0, 0.0>\)]
\[(8, <X, Y, ?, 0.0, 0.0>\)]
\[(9, <X, Y, Z, 0.0, 0.0>\)]
\[(10, <X, Y, Z, 0.0, X*Y>)\]
\[(11, <X, Y, Z, 0.0, X*Y>)\]
\[(12, <X, Y, Z, 0.0, X*Y>)\]
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.

**Definition 6.5.4:** 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 \( V \).

Thus

\[
\text{Proj}(C_1, T_1) = \text{Proj}((12, \{z\}), T_1)
\]

\[
= (1, ?, ?, ?, ?, ?)
\]

\[
(2, ?, ?, ?, ?, ?)
\]

\[
(3, ?<X, Y, ?, ?, ?>)
\]

\[
(4, ?<X, Y, ?, 0.0, ?>)
\]

\[
(5, ?<X, Y, ?, 0.0, 0.0>)
\]

\[
(6, ?<X, Y, ?, 0.0, 0.0>)
\]

\[
(11, ?<X, Y, ?, Y, 0.0>)
\]

\[
(12, ?<X, Y, ?, Y, 0.0>)
\]

\[
= (12, <?>)
\]

\[
\text{Proj}(C_2, T_1) = \text{Proj}((12, \{\text{total}\}), T_1)
\]

\[
= (1, ?, ?, ?, ?, ?)
\]

\[
(2, ?, ?, ?, ?, ?)
\]

\[
(3, ?<X, Y, ?, ?, ?>)
\]

\[
(4, ?<X, Y, ?, 0.0, ?>)
\]

\[
(5, ?<X, Y, ?, 0.0, 0.0>)
\]

\[
(6, ?<X, Y, ?, 0.0, 0.0>)
\]

\[
(11, ?<X, Y, ?, Y, 0.0>)
\]

\[
(12, ?<X, Y, ?, Y, 0.0>)
\]

\[
= (12, <0.0>)
\]

and

\[
\text{Proj}(C_3, T_2) = \text{Proj}((9, \{x\}), T_2)
\]

\[
= (1, ?, ?, ?, ?, ?)
\]

\[
(2, ?, ?, ?, ?, ?)
\]

\[
(3, ?<X, Y, ?, ?, ?>)
\]

\[
(4, ?<X, Y, ?, 0.0, ?>)
\]

\[
(7, ?<X, Y, ?, 0.0, 0.0>)
\]

\[
(8, ?<X, Y, ?, 0.0, 0.0>)
\]
Definition 6.5.5: 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 an 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.

For example, consider $P$, the example program listed at the beginning of this section, 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, $\text{Proj}(C_1, T_1) = (12, <\_>)$. Now if we execute $S_1$ with the same input, it should yield the following value trace

$T'_1$: (1, <?, ?, ?, ?, ?>)  
(2, <?, ?, ?, ?, ?>)  
(5, <X, Y, ?, ?, ?>)  
(6, <X, Y, ?, ?, ?>)  
(12, <X, Y, ?, ?, ?>)

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

$$\text{Proj}(C_1, T'_1) = ((12, \{z\}), T'_1)$$

$$= (1, <?, ?, ?, ?, ?>)  
(2, <?, ?, ?, ?, ?>)  
(5, <X, Y, ?, ?, ?>)  
(6, <X, Y, ?, ?, ?>)  
(12, <X, Y, ?, ?, ?>)$$

$$= (12, <\_>)$$

$$= \text{Proj}(C_1, T_1)$$

As an example in which $C \neq C'$, 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.

Exercise: What is the condition under which the projection of a trace consists of a sequence of more than one pair?
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.

The above definition of a slice is not constructive in that it does not say how to find one. A constructive definition of slice can be found in [LAVI97].

The smaller the slice the easier it is to understand. Hence it is of practical value to be able to find the minimum slice of a program. It has been shown, however, that finding minimal slices of a program is equivalent to solving the halting problem -- an unsolvable problem in the sense that no single algorithm can be found for this purpose [WEIS84].