Slice-based Testing

- Slice-based testing is a software testing technique where the program is analyzed and tested by examining slices of the code, usually derived through program slicing.
- It focuses on testing parts of a program (called slices) that affect a specific variable or output at a certain point.
- A program slice is the set of statements in a program that directly or indirectly affect the value of a target variable at a particular location.
- Program slicing involves breaking a program into smaller "slices" based on the values of certain variables or execution points. Each slice consists of the set of program statements that potentially affect the value of a variable at a particular point in the program.
- **Goal:** To ensure correctness and thorough testing by focusing on relevant code affecting a specific computation or variable.
- **Method:** Identify program slices related to particular features, behaviors, or outputs and test those independently.
- **Application:** Useful in debugging, regression testing, and impact analysis.

Benefits:

- Helps in isolating bugs quickly.
- Improves test efficiency by focusing on relevant parts.
- Useful in maintaining and understanding large codebases.

Types of Slicing:

1. **Static slicing:** Based on the program's source code without running it. It analyzes the entire program code to find all statements that might affect the value of a variable, regardless of input values or execution paths.
2. **Dynamic slicing:** It analyzes a specific execution of the program, using specific input values, and includes only the statements that were actually executed and affected the variable.

Parameter	Static Slicing	Dynamic Slicing
Scope	All possible paths	Only the path taken during execution
Useful for	Code understanding, maintenance	Debugging, test case analysis
Based on	Code analysis only	Code + runtime values
Result	May include unused paths	Precise, minimal set of actual dependencies
Input-dependent	No	Yes
Slice size	Larger (includes all possible influences)	Smaller (precise and input-specific)

For example:

```
int calculate_salary(hours_worked, hourly_rate) {
    bonus = 0
    if (hours_worked > 40)
       { bonus = (hours_worked - 40) * (hourly_rate * 1.5) }
    base_pay = hours_worked * hourly_rate
    total_salary = base_pay + bonus
    printf("Total Salary: %d”, total_salary)
    return total_salary
}
```

Static Program Slice

Target variable: total_salary

The slice includes:

- *bonus = 0*
- *if hours_worked > 40: bonus = (hours_worked - 40) * (hourly_rate * 1.5)*
- *base_pay = hours_worked * hourly_rate*
- *total_salary = base_pay + bonus*

This slice excludes unrelated lines (e.g., the print statement) since it doesn't affect the return value directly.

Now, create test cases specifically to validate the correctness of this slice.

Test Case 1 (No Overtime)
calculate_salary(30, 10)
# Expected: 30*10 = 300; no bonus
Test Case 2 (With Overtime)
calculate_salary(45, 10)
# Expected: base = 450, bonus = 5*15 = 75 → Total = 525

These test cases ensure the logic affecting total_salary is tested without worrying about other program behavior (e.g., formatting output, logging, or other features).

Dynamic Slicing

Unlike static slicing (which analyzes the code regardless of input), dynamic slicing considers a specific input and execution path. It includes only the statements that were actually executed and affected the variable at a particular point in that specific run.

Dynamic Slice for Input: calculate_salary(30, 10)

```
hours_worked = 30
hourly_rate = 10

# bonus = 0                  ← Executed
# if (hours_worked > 40)     → False (Skipped)
#     { bonus = ...          ← Skipped
base_pay = 30 * 10           ← Executed
total_salary = 300 + 0       ← Executed
return total_salary          ← Executed
```

Dynamic Slice (Input: 30, 10) includes:

- `bonus = 0`
- `base_pay = hours_worked * hourly_rate`
- `total_salary = base_pay + bonus`
- `return total_salary`

Dynamic Slice for Input: calculate_salary(45, 10)

```
hours_worked = 45
hourly_rate = 10

bonus = 0                                ← Executed
if (hours_worked > 40)                   ← True
   { bonus = (5) * (10 * 1.5) = 75 }     ← Executed
```

```
base_pay = 45 * 10 = 450                ← Executed
total_salary = 450 + 75 = 525           ← Executed
return total_salary                     ← Executed
```

Dynamic Slice (Input: 45, 10) includes:

- `bonus = 0`
- `if (hours_worked > 40)`
- `{ bonus = (hours_worked - 40) * (hourly_rate * 1.5) }`
- `base_pay = hours_worked * hourly_rate`
- `total_salary = base_pay + bonus`
- `return total_salary`