Table of Contents

Coverage Package Documentation

This package provides code coverage tracking utilities for MoonBit programs. It includes tools for measuring which parts of your code are executed during testin g and generating coverage reports.

Coverage Counter

The core component for tracking code execution:

```
1
2   test "coverage counter basics" {
3
4     let counter = CoverageCounter::new(5)
5
6
7     inspect(counter to_string(), content="[0, 0, 0, 0, 0]")
8
9
10    counter incr(0)
11    counter incr(2)
12    counter incr(0)
13
14
15    inspect(counter to_string(), content="[2, 0, 1, 0, 0]")
16  }
```

Tracking Code Execution

Use coverage counters to track which code paths are executed:

```
1
2   test "tracking execution paths" {
3     let counter = CoverageCounter::new(3)
4     fn conditional_function(x : Int, coverage : CoverageCounter) -> String
5       if x > 0 {
6         coverage incr(0)
7         "positive"
8       } else if x < 0 {
9         coverage incr(1)
10        "negative"
11      } else {
12        coverage incr(2)
13        "zero"
14      }
15    }
16
17
18    let result1 = conditional_function(5, counter)
19    inspect(result1, content="positive")
20    let result2 = conditional_function(-3, counter)
21    inspect(result2, content="negative")
22    let result3 = conditional_function(0, counter)
23    inspect(result3, content="zero")
24
25
26    inspect(counter to_string(), content="[1, 1, 1]")
27  }
```

Loop Coverage Tracking

Track coverage in loops and iterations:

```
1
2   test "loop coverage" {
3     let counter = CoverageCounter::new(2)
4     fn process_array(arr : Array[Int], coverage : CoverageCounter) -> Int
5       let mut sum = 0
6       for x in arr {
7         if x % 2 == 0 {
8           coverage incr(0)
9           sum = sum + x
10        } else {
11          coverage incr(1)
12          sum = sum + x * 2
13        }
14      }
15      sum
16    }
17
18    let test_data = [1, 2, 3, 4, 5]
19    let result = process_array(test_data, counter)
20
21
22    inspect(result, content="24")
23
24
25    let coverage_str = counter to_string()
26    inspect(coverage_str length() > 5, content="true")
27  }
```

Function Coverage

Track coverage across different functions:

```
1
2   test "function coverage" {
3     let counter = CoverageCounter::new(4)
4     fn math_operations(
5       a : Int,
6       b : Int,
7       op : String,
8       coverage : CoverageCounter,
9     ) -> Int {
10      match op {
11        "add" => {
12          coverage incr(0)
13          a + b
14        }
15        "sub" => {
16          coverage incr(1)
17          a - b
18        }
19        "mul" => {
20          coverage incr(2)
21          a * b
22        }
23        _ => {
24          coverage incr(3)
25          0
26        }
27      }
28    }
29
30
31    let add_result = math_operations(10, 5, "add", counter)
32    inspect(add_result, content="15")
33    let sub_result = math_operations(10, 5, "sub", counter)
34    inspect(sub_result, content="5")
35    let unknown_result = math_operations(10, 5, "unknown", counter)
36    inspect(unknown_result, content="0")
37
38
39    let final_coverage = counter to_string()
40    inspect(final_coverage, content="[1, 1, 0, 1]")
41  }
```

Coverage Analysis

Analyze coverage data to understand code execution:

```
1
2   test "coverage analysis" {
3     let counter = CoverageCounter::new(6)
4     fn complex_function(input : Int, coverage : CoverageCounter) -> String
5       coverage incr(0)
6       if input < 0 {
7         coverage incr(1)
8         return "negative"
9       }
10      coverage incr(2)
11      if input == 0 {
12        coverage incr(3)
13        return "zero"
14      }
15      coverage incr(4)
16      if input > 100 {
17        coverage incr(5)
18        "large"
19      } else {
20        "small"
21      }
22    }
23
24
25    let result1 = complex_function(-5, counter)
26    inspect(result1, content="negative")
27    let result2 = complex_function(0, counter)
28    inspect(result2, content="zero")
29    let result3 = complex_function(50, counter)
30    inspect(result3, content="small")
31
32
33    let coverage = counter to_string()
34
35    inspect(coverage length() > 10, content="true")
36  }
```

Integration with Testing

Coverage tracking integrates with MoonBit's testing system:

```
1
2   test "testing integration" {
3
4
5
6     fn test_function_with_coverage() -> Bool {
7
8       let counter = CoverageCounter::new(2)
9       fn helper(condition : Bool, cov : CoverageCounter) -> String {
10        if condition {
11          cov incr(0)
12          "true_branch"
13        } else {
14          cov incr(1)
15          "false_branch"
16        }
17      }
18
19
20      let result1 = helper(true, counter)
21      let result2 = helper(false, counter)
22      result1 == "true_branch" && result2 == "false_branch"
23    }
24
25    let test_passed = test_function_with_coverage()
26    inspect(test_passed, content="true")
27  }
```

Coverage Reporting

Generate and analyze coverage reports:

```

test "coverage reporting" {
  let counter = CoverageCounter::new(3)


  counter incr(0)
  counter incr(0)
  counter incr(2)


  let report = counter to_string()
  inspect(report, content="[2, 0, 1]")


  fn analyze_coverage(_coverage_str : String) -> (Int, Int) {


    (2, 3)
  }

  let (covered, total) = analyze_coverage(report)
  inspect(covered, content="2")
  inspect(total, content="3")
}
```

Best Practices

1. Automatic Coverage Generation

In real applications, coverage tracking is typically generated automatically:

```


fn example_function(x : Int) -> String {

  if x > 0 {

    "positive"
  } else {

    "non-positive"
  }

}


test "automatic coverage concept" {
  let result = example_function(5)
  inspect(result, content="positive")
}
```

2. Coverage-Driven Testing

Use coverage information to improve test quality:

```
1
2   test "coverage driven testing" {
3
4     fn multi_branch_function(a : Int, b : Int) -> String {
5       if a > b {
6         "greater"
7       } else if a < b {
8         "less"
9       } else {
10        "equal"
11      }
12    }
13
14
15    inspect(multi_branch_function(5, 3), content="greater")
16    inspect(multi_branch_function(2, 7), content="less")
17    inspect(multi_branch_function(4, 4), content="equal")
18
19
20  }
```

Integration with Build System

Coverage tracking integrates with MoonBit's build tools:

- Use moon test to run tests with coverage tracking
- Use moon coverage analyze to generate coverage reports
- Coverage data helps identify untested code paths
- Supports both line coverage and branch coverage analysis

Performance Considerations

- Coverage tracking adds minimal runtime overhead
- Counters use efficient fixed arrays for storage
- Coverage instrumentation is typically removed in release builds
- Use coverage data to optimize test suite performance

Common Use Cases

- **Test Quality Assessment**: Ensure comprehensive test coverage
- **Dead Code Detection**: Find unused code paths
- **Regression Testing**: Verify that tests exercise the same code paths
- **Performance Analysis**: Identify frequently executed code for optimization
- **Code Review**: Understand which parts of code are well-tested

The coverage package provides essential tools for maintaining high-quality, well-tested MoonBit code through comprehensive coverage analysis.