

Graph Coverage for Specifications

We'll move further up the abstraction chain now and talk about testing based on specifications. Specification-based testing can help ensure that a system doesn't drift too far from its intended behaviour, and tends to focus on higher-level concerns than the testing we've seen so far, which is based on properties of the source code.

Sequencing Constraints

Libraries often come with constraints on permissible actions; we call these constraints sequencing constraints.

Java Iterators. Here are two possible constraints.

The general format of sequencing constraints is “do X before Y”, or “don't do W before A”. Here is another sequencing constraint, this one implicit.

```
/* @requires size() > 0 */
public int dequeue() { ... }

/* @ensures size() = \old(size()) + 1 */
public void enqueue (int x) { ... }
```

Assuming that the only way to increase `size()` is by calling `enqueue()`, then a client had better call `enqueue()` before calling `dequeue()`.

It's generally useful to look for such constraints in code that you're developing or testing, even if it's hard to identify them. These constraints specify partial program properties.

Using Sequencing Constraints. Our approach will be to verify that sequencing constraints hold (or are never violated) on all paths through a graph modelling the system under test.

Let's consider the following three methods:

- `open(String fname)`
- `close()`
- `write(String line)`

and the following rules:

1. `open(fn)` must be executed before any call to `write(t)`;
2. `open(fn)` must be executed before any call to `close()`;
3. `write(t)` must not execute after `close()` unless `open(fn)` has executed in between;
4. `write(t)` should be executed before every `close()`;
5. `close()` must not execute after `close()` unless `open(fn)` executes in between;
6. `open(fn)` must not execute after `open(fn)` unless `close()` executes in between.

How does this simplify reality?

We will check these sequencing constraints on graphs, typically CFGs.

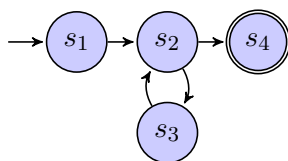


Figure 1: Simple Control-Flow Graph

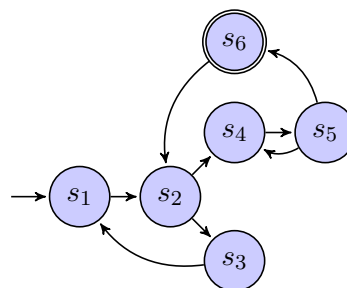


Figure 2: More Complicated Control-Flow Graph

Static approach. We can deduce, without running anything (i.e. statically), that the program in Figure 1 must satisfy all of the properties except for the one which requires that `write` occur before `close`. The model of the program in Figure 2 shows that it may violate the property that requires no writes to occur after closes. However, the back edge (s_5, s_4) might be infeasible.

Dynamic approach. By running the programs with suitable test cases, we could explicitly identify violating cases. If we are unable to find any violating cases, we can have some confidence that, in practice, the program never violates the properties.

Test Requirements

We can use test requirements to verify sequencing constraints in two ways: (1) we can create tests that satisfy various coverage criteria, thus suggesting that the program probably doesn't violate properties; or (2) we can set up test requirements, if satisfied, exhibit executions where constraints are violated. We might, for instance, set up the following test requirements:

- cover every path from the start to all `write()` nodes that don't go through any `open()` nodes;
- cover every path from the start to all `close()` nodes that don't go through any `open()` nodes;
- etc.

We are hoping that such test requirements will be infeasible; any program that meets the specification will be unable to satisfy these test requirements.