Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 20 — February 26, 2010

Patrick Lam *version 1*

Structural Logic Coverage of Programs

Let's put what we saw last time into practice. Recall the four steps: identify predicates, compute reachability, calculate conditions for determination, and find needed variable values.

Extracting predicates. For programs, extract conditions from branches (if, while, etc.).

- Many conditions have only 1 clause, so predicate coverage suffices to cover all interesting cases.

Difficulties in achieving logic coverage. Let's talk a bit more about how to achieve logic coverage.

- (reachability) must get to appropriate program point—graph coverage;
- (internal variable problem) must set program values properly to get desired predicate value; e.g. for predicate `x < y`, we must be able to ensure that `x` and `y` have appropriate values.

  Ensuring appropriate values is easier in unit testing than in integration testing, since you have more control over the inputs. Potential complications include nondeterminism, causing time-sensitivity, or input/output.

Concrete Example

The canonical example for logic coverage is `TriTyp`, as seen in the book. I looked through source code on my hard disk and found the following excerpt from `fontconfig`, a font management library for Unix systems. It is written in C, which tends to have more complicated predicates than Java.

```
1  FcBool
2  FcConfigUptoDate (FcConfig *config)
3  {
4      FcFileTime  config_time, config_dir_time, font_time;
5      time_t      now = time(0);
6      if (!config)
7      {
8          config = FcConfigGetCurrent ();
9          if (!config)
10             return FcFalse;
11     }
12     config_time = FcConfigNewestFile (config->configFiles);
13     config_dir_time = FcConfigNewestFile (config->configDirs);
14     font_time = FcConfigNewestFile (config->fontDirs);
```

```
15      if ((config_time.set && config_time.time - config->rescanTime > 0) ||
16          (config_dir_time.set && (config_dir_time.time - config->rescanTime) > 0) ||
17          (font_time.set && (font_time.time - config->rescanTime) > 0))
18      {
19          /* We need to check for potential clock problems here (OLPC ticket #6046) */
20          if ((config_time.set && (config_time.time - now) > 0) ||
21              (config_dir_time.set && (config_dir_time.time - now) > 0) ||
22              (font_time.set && (font_time.time - now) > 0))
23          {
24              fprintf (stderr,
25                       "Fontconfig warning: Directory/file mtime in the future. "
26                       "New fonts may not be detected\\n");
27              config->rescanTime = now;
28              return FcTrue;
29          }
30          else
31              return FcFalse;
32      }
33      config->rescanTime = now;
34      return FcTrue;
35  }
```

Predicates

The `FcConfigUptoDate` function contains four predicates:

- `config` (line 6)
- `config` (line 9)
- `(config_time.set && config_time.time - config->rescanTime > 0) ||`
  `(config_dir_time.set && (config_dir_time.time - config->rescanTime) > 0) ||`
  `(font_time.set && (font_time.time - config->rescanTime) > 0))` (lines 15–17)
- `(config_time.set && (config_time.time - now) > 0) ||`
  `(config_dir_time.set && (config_dir_time.time - now) > 0) ||`
  `(font_time.set && (font_time.time - now) > 0))` (lines 20–22)

Reachability

We discuss the conditions for reachability for each of these predicates in turn.

- line 6: always reachable.
- line 9: parameter `config` must be initially `NULL` to reach line 9.
- line 15: reachable if `config` is non-`NULL` (or if `FcConfigGetCurrent()` returns a non-`NULL` value).
- line 20: one of the three disjuncts at line 15 must be true; for instance, there exist at least one configuration file and its modification time must be newer than the last time the configuration was scanned.

The first three predicates are easy to reach. The last predicate is more difficult to reach: a test case must initialize the `fontconfig` library and then touch (update the modification time of) a configuration file, directory, or font.

Note that to reach the predicate at line 20, we require the predicate at line 15 to be true. Sometimes you need to fix values of multiple predicates to reach a particular program point. Drawing the control-flow graph can help a lot.

Determination Analysis

This function contains two nontrivial clauses, at lines 15 and 20. Let's rewrite the one at line 15 for the determination analysis as follows:

$$(c_s \wedge c_{tr}) \vee (c_d \wedge c_{dtr}) \vee (f_s \wedge f_{tr});$$

the predicate at line 20 has the same shape, but uses `now` instead of `rescanTime`.

For c_s to determine the predicate: c_{tr} must be true; either c_d or c_{dtr} must be false; and either f_s or f_{tr} must be false. The other cases are symmetrical.

Imposing Variable Values

The next step is to find ways to make the clauses true and false at will. With such information, we ought to be able to meet test requirements for our logic coverage criteria.

- `config`, line 6: trivial, it's a parameter.
- `config`, line 9: need to pass in `NULL` and then make `FcConfigGetCurrent()` return desired value; calling `FcConfigDestroy()` will set `config` to `NULL`.
- `config_time.set`, `config_dir_time.set`, `font_time.set`: boolean, evaluates to true if the sets `config->configFiles`, `config->configDirs`, `config->fontDirs` are non-empty, false otherwise;
- `config_time.time - config->rescanTime > 0`, etc: true if the newest config file, config dir, or font is newer than the last scan time, or the current time, as appropriate.

Again, we can force predicate values by updating the last modification times of a configuration file, a configuration directory, or a font file.

Meeting Logic Coverage Criteria

We now have enough information to build appropriate test suites. Let's pick a couple of representative cases.

Predicate coverage. We need to force each of the predicates to be true and false. For line 6, we simply need to have a test case with a `NULL` parameter and one with a non-`NULL` parameter. For line 15, we can get by with one case where everything is up-to-date—this test case simply needs to call `FcInit()`, which initializes the `fontconfig` library, and then call `FcConfigUptoDate()`, and one case where something is not up-to-date, for instance by calling `FcInit()` and then touching a configuration file.

CACC. Let's consider the predicate at line 15. (The predicate at line 20 could be tested by setting modification times in the future). For c_{tr} to determine the predicate, we need c_s to be true, so there must be at least one configuration file; c_{dtr} should be false, so all directories must be older than the last rescan time; and f_{tr} should also be false, so all fonts must be older than the last rescan time.

Test cases which exercise this predicate for CACC include a test case where c_{tr} is true, i.e. a config file has a newer date than the last scan time, and a test case where c_{tr} is false, i.e. all config files are older than the last scan time.

Summary. It is straightforward, although perhaps tedious, to get complete logic criterion coverage. The only complication which I haven't addressed in this example is in dealing with loops. We only require one path in the test case to reach the predicate with the desired value.

Other Sources of Predicates

Besides programs, other sources of predicates are:

The three main kinds of specifications are preconditions, postconditions and invariants—together, these specifications make up method contracts. The basic idea of programming by contract requires you to specify contracts for each method, and the language promises to verify these contracts at runtime. (One can also try to verify these contracts at compile time, or statically).

Example. The textbook contains a `cal()` method, which calculates the number of days between two dates in the same year (assuming the Gregorian calendar). I haven't excerpted it here.

Note that the method implementation assumes that the first day is before the second day, but the precondition allows some situations where day 1 is before day 2, for instance day 1 = Jan 31, day 2 = Jan 15. It also allows bogus days, like February 30.

What happens when you break the contract at runtime? The JML compiler will create code to test the contracts, and throw an exception when a contract doesn't hold.

Here is the JML precondition for the `cal()` method:

```
1 /* @public normal_behavior
2     requires month1 >= 1 && month1 <= 12 && month2 >= 1 && month2 <= 12 && month1 <= month2
3           && day1 >= 1 && day1 <= 31 && day2 >= 1 && day2 <= 31
4           && year >= 1 && year <= 10000; @*/
```

We can also write this precondition in predicate form:

$$\begin{aligned} & m_1 \geq 1 \wedge m_1 \leq 12 \wedge m_2 \geq 1 \wedge m_2 \leq 12 \wedge m - 1 \leq m - 2 \wedge d_1 \geq 1 \wedge d_1 \leq 31 \\ \wedge \quad & d_2 \geq 1 \wedge d_2 \leq 31 \wedge y \geq 1 \wedge y \leq 10000 \end{aligned}$$

Although there are 11 clauses, this predicate is simple, because the clauses are linked by "and".

Predicate coverage. True: make all clauses true; e.g. $(m_1 = 4, m_2 = 4, d_1 = 12, d_2 = 30, y = 1911)$; false: make any clause false; e.g. $(m_1 = 6, m_2 = 4, d_1 = 12, d_2 = 30, y = 1961)$. Note that nothing interesting happens when the precondition fails.

Clause coverage. Make each clause true, and make each clause false. Sometimes it's possible to achieve clause coverage with two tests (all clauses true and all clauses false), but we need more than that here. (Why?)

The following test set should work:

- all true: $(m_1 = 1, m_2 = 1, d_1 = 1, d_2 = 1, y = 1)$
- half false: $(m_1 = 0, m_2 = -1, d_1 = 0, d_2 = 0, y = 0)$
- other half false: $(m_1 = 13, m_2 = 13, d_1 = 32, d_2 = 32, y = 10001)$

Active clause coverage. We can follow the following pattern to achieve active clause coverage:

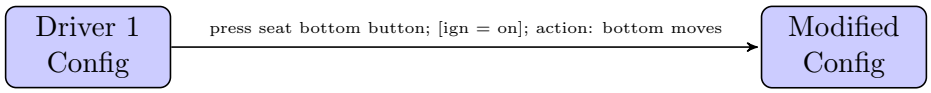
$$\begin{array}{cccccc} T & t & t & t & \cdots & t \\ F & t & t & t & \cdots & t \\ t & F & t & t & \cdots & t \\ t & t & F & t & \cdots & t \\ \ldots \text{etc.} & & & & & \end{array}$$

(Capitalized letters indicate the determining clause.) We can obviously find values for these truth assignments.

If you think about it, ACC doesn't really bring much here.

Finite State Machines

Here is a simplified FSM for a car memory seat.



Predicates are guards and triggers from the edges.

Satisfying criteria is therefore straightforward. Issues include reachability: how do you get to desired pre-state? Use depth-first search to come up with the prefix. Postfix values help get to the exit state.