Implementing Mutation Testing for VDM-SL in ViennaTalk

Tomohiro Oda

Software Research Associates, Inc.

Agenda

- Mutation Testing
- Add-one Mutation
- Negate Mutation
- UI & demo
- Experiment
- Discussion and Conclusion

Mutation Testing

- How do you confirm your `isOdd` function works as intended?
  - `assert isOdd(1)`
  - `assert not isOdd(2)`

- How do you confirm your tests work as intended?
  - `assert TEST(correct_isOdd)` ... you don't have `correct_isOdd`.
  - `assert not TEST(buggy_isOdd)` ... you can generate `buggy_isOdd` by injecting a bug.

  test whether or not your test can detect bugs generated by mutation.

  mutation score = number of red mutants / total number of mutants

Add-one mutation

replace `expr` with `expr + 1` where `expr` is statically typed `real`.

```
isOdd : nat -> nat
isOdd(x) == x mod 2 = 1
```

- `(x+1) mod 2 = 1`
- `x mod (2+1) = 1`
- `x mod 2 = 1+1`

impacts on numeric computation, sequence indexing, cardinality of `set`/`seq`, and `for` ... `to` ... `do` loop.

Negate mutation

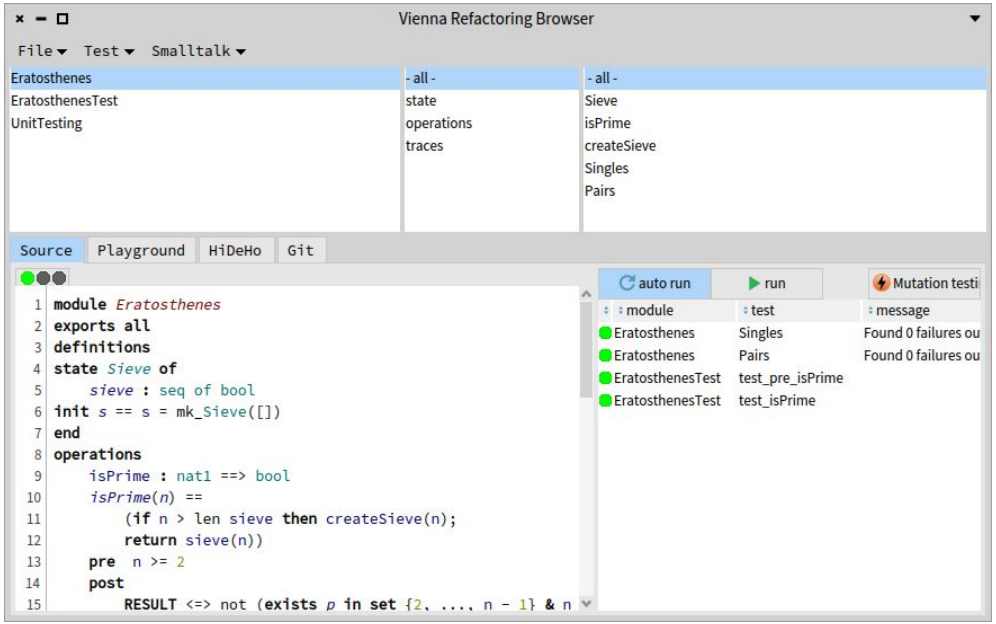
replace `expr` with `not expr` where `expr` is statically typed `bool`.

```
isOdd : nat -> nat
isOdd(x) == x mod 2 = 1
```

- `not (x mod 2 = 1)`

impacts on boolean expression, `if` branch, `while` ... `do` loop, and assertion

UI (demo)


Vienna Refactoring Browser
File
Test
Smalltalk
Eratosthenes
EratosthenesTest
UnitTesting
- all -
state
operations
traces
- all -
Sieve
isPrime
createSieve
Singles
Pairs
Source
Playground
HiDeHo
Git
auto run
run
Mutation testi
module
test
message
Eratosthenes
Singles
Found 0 failures ou
Eratosthenes
Pairs
Found 0 failures ou
EratosthenesTest
test_pre_isPrime
EratosthenesTest
test_isPrime
1 module Eratosthenes
2 exports all
3 definitions
4 state Sieve of
5 sieve : seq of bool
6 init s == s = mk_Sieve([])
7 end
8 operations
9 isPrime : nat1 ==> bool
10 isPrime(n) ==
11 (if n > len sieve then createSieve(n);
12 return sieve(n))
13 pre n >= 2
14 post
15 RESULT <=> not (exists p in set {2, ..., n - 1} & n

Experiment: Sieve of Eratosthenes

```
state Sieve of
        sieve : seq of bool
init s == s = mk_Sieve([])
end
operations
isPrime : nat1 ==> bool
isPrime(n) ==
        (if n > len sieve then createSieve(n);
        return sieve(n))
pre     n >= 2
post    RESULT <=>
            not (exists p in set {2, ..., n - 1} & n mod p = 0);
```

```
createSieve : nat1 ==> ()
createSieve(size) ==
        (dcl    newSieve:seq of bool := [true | i in set {1, ..., size}],
                n:nat1 := 2;
        newSieve(1) := false;
        while n  * n <= size do
                (if newSieve(n) then
                        for m = n  * 2 to  size by  n do
                                newSieve(m) := false;
                n := n + 1);
        sieve := newSieve)
pre     size >= 2
post    len sieve = size
        and (forall n in set {2, ..., size} &
           sieve(n) <=>
              not (exists p in set {2, ..., n - 1} & n mod p = 0));
```

Experiment: Sieve of Erathostenes ... tests

```
traces
    Singles:
        let n in set {2, ..., 50} in
                isPrime(n);
    Pairs:
        let n1, n2 in set {2, ..., 50} in
                (isPrime(n1);
                isPrime(n2));
```

```
test_pre_isPrime : () ==> ()
test_pre_isPrime() ==
        (assert(not Eratosthenes`pre_isPrime(1, no_sieve), "should not give 1");
        assert(not Eratosthenes`pre_isPrime(1, sieve), "should not give 1");
        assert(Eratosthenes`pre_isPrime(2, no_sieve), "2 without sieve is OK");
        assert(Eratosthenes`pre_isPrime(2, sieve), "2 with sieve is OK");
        assert(Eratosthenes`pre_isPrime(4, no_sieve), "4 without sieve is OK");
        assert(Eratosthenes`pre_isPrime(4, sieve), "4 with sieve is OK");
        skip);
test_isPrime : () ==> ()
test_isPrime() ==
        (assert(Eratosthenes`isPrime(2), "2 is prime");
        assert(Eratosthenes`isPrime(3), "3 is prime");
        assert(Eratosthenes`isPrime(5), "5 is prime");
        assert(Eratosthenes`isPrime(5), "5 is prime");
        assert(not Eratosthenes`isPrime(6), "6 is not prime");
        assert(Eratosthenes`isPrime(5), "5 is prime"));
```

mutation scores from various test conditions

test condition	killed	surviving	mutation score
unit testing without mutating pre/post conditions	20	9	0.6897
unit testing including mutated pre/post conditions	55	14	0.7971
unit testing without post conditions	16	13	0.5517
combinatorial testing without mutating pre/post conditions	25	4	0.8621
combinatorial testing including mutated pre/post conditions	60	9	0.8696
combinatorial testing without post conditions	7	22	0.2414

Discussion and Conclusion

- Do we want to mutate pre/post/inv in the spec?
  - Probably yes because assertions are the very core parts of the specification.
  - If we don't, our tests are against only the explicit statements/expressions.

- Lower mutation score means loose specification?
  - Probably yes if you give enough amount of tests.

Further investigation is needed.

- more kinds of mutation operators
- more kinds of VDM specifications