Testing with mutants

@janstepien

Ihre E-Ticket-BestÃ¤tigung

Sehr geehrte(r) Herr Stepie

Buchungsreferenz:

Vielen Dank fÃ¼r Ihre Buchung bei 😂😂😂

Ticketart: E-Ticket

Sie erhalten hiermit Ihre E-Ticket-Buchungsbestï¿½tigung. Ihr Ticket ist in unserem System gespeichert. Sie erhalten kein Papierticket fï¿½r Ihre Buchung.

We write tests

lein new project

```
(ns project.core-test
  (:require [clojure.test :refer :all]
            [project.core :refer :all]))

(deftest a-test
  (testing "FIXME, I fail."
    (is (= 0 1))))
```

Generative testing

```
(require '[clojure.test.check
           [generators :as gen]
           [properties :as prop]])

(def prop-sort-idempotency
  (prop/for-all [coll (gen/vector gen/int)]
    (= ((comp sort sort) coll)
       (sort coll))))
```

We *write* tests

Who is testing our tests?

@janstepien

Mutation testing

Competent programmer hypothesis

Coupling effect hypothesis

Mutants which don't get *killed* become *survivors*

An Analysis and Survey of the Development of Mutation Testing

Yue Jia *Student Member, IEEE,* and Mark Harman *Member, IEEE*

***Abstract*— Mutation Testing is a fault–based software testing technique that has been widely studied for over three decades. The literature on Mutation Testing has contributed a set of approaches, tools, developments and empirical results. This paper provides a comprehensive analysis and survey of Mutation Testing. The paper also presents the results of several development trend analyses. These analyses provide evidence that Mutation Testing techniques and tools are reaching a state of maturity and applicability, while the topic of Mutation Testing itself is the subject of increasing interest.**

***Index Terms*— mutation testing, survey**

I. Introduction

Mutation Testing is a fault-based testing technique which provides a testing criterion called the "mutation adequacy score". The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults.

The general principle underlying Mutation Testing work is that the faults used by Mutation Testing represent the mistakes that programmers often make. By carefully choosing the location and type of mutant, we can also simulate any test adequacy criteria. Such faults are deliberately seeded into the original program, by simple syntactic changes, to create a set of faulty programs called

Besides using Mutation Testing at the software implementation level, it has also been applied at the design level to test the specifications or models of a program. For example, at the design level Mutation Testing has been applied to Finite State Machines [20], [28], [88], [111], State charts [95], [231], [260], Estelle Specifications [222], [223], Petri Nets [86], Network protocols [124], [202], [216], [238], Security Policies [139], [154], [165], [166], [201] and Web Services [140], [142], [143], [193], [245], [259].

Mutation Testing has been increasingly and widely studied since it was first proposed in the 1970s. There has been much research work on the various kinds of techniques seeking to turn Mutation Testing into a practical testing approach. However, there is little survey work in the literature on Mutation Testing. The first survey work was conducted by DeMillo [62] in 1989. This work summarized the background and research achievements of Mutation Testing at this early stage of development of the field. A survey review of the (very specific) sub area of Strong, Weak, and Firm mutation techniques was presented by Woodward [253], [256]. An introductory chapter on Mutation Testing can be found in the book by Mathur [155] and also in the book by Ammann and Offutt [11]. The most recent survey work was

Off the shelf

- **Mutant** for Ruby
- **PIT** for Java
- **Stryker** for JavaScript

...and many more

Mutant

Mutation testing for Clojure

@janstepien

- Read all source files in a directory
- Generate *mutants* for all top-level forms
- Run the test suite for every *mutant*
- Report all *survivors*

Generating mutants

```
[rewrite-clj "0.6.0"]
```

```clojure
(and x y)
```

```clojure
(or x y)
```

```clojure
(< x 1)
```

```clojure
(<= x 1)
```

```clojure
(empty? coll)
```

```clojure
(seq coll)
```

```clojure
(defn f [a b]
  (+ a b))
```

```clojure
(defn f [a b]
  )
```

Reevaluating the code

```
[org.clojure/tools.namespace "0.2.11"]
```

Running tests

```
8 survivors out of 61 mutants

(ns mutant.mutations)
(defn- rm-args [node]
  (let [sexpr (z/sexpr node)]
    (if (seq? sexpr)
      (let [[defn name args & more] sexpr]
        (if ([-and-]{+or+} (#{'defn 'defn-} defn)
                   (vector? args))
          (for [arg args]
            (-> node z/down z/right z/right
                (z/edit (partial filterv (complement #{arg})))
                (z/up)))))))))

(ns mutant.internals)
(defn- dependants [graph ns]
  [-(letfn [(rec [sym]
              (if-let [deps (seq (dep/dependents graph sym))]
                (reduce into [] (conj (mapv rec deps) deps))))]
    (reverse (distinct (rec ns))))-])
```

Problems

It’s slow

Do fewer, do faster,
or do smarter.

— Offutt and Untch, 2001

Do fewer

- Select what to mutate
- Select your mutation operators
- Sample your mutants

Do faster

- Don’t restart the virtual machine
- Run tests in parallel

Do smarter

- Execute only relevant tests
- Reorder the test suite

It might not terminate

- JVM cannot stop threads
- JVM cannot fork
- Starting new JVM is too slow

Mutate continuously

@janstepien

```
git diff master~..master
```

Introduce it gradually

@janstepien

```
[lein-mutate "0.1.0"]
```

github.com/jstepien/mutant

Testing with mutants

jan.stepien@innoq.com

@janstepien