Testing for Design Faults

He Jifeng and B.K. Aichernig
Shanghai Key Laboratory of Trustworthy Computing

Background

Fault-based testing was born in practice when testers started to asses the adequacy of their test case by first injecting faults into their program, and then by observing if the test cases could detect these faults.

- R.G. Hamlet. *Testing programs with the aid of a compiler*
- R. DeMilo, R. Lipton and F. Sayward. *Hints on test data selection: Help for practicing programmer*

Related Work

T. Budd and A. Gopal. *Program testing by specification mutation*

K.-C. Tai and H.-K. Su. *Test generation for Boolean expressions*

K.-C. Tai. *Theory of fault-based predicate testing for computer programs.*

P.A. Stock. *Applying formal methods to software testing.*

S. Burton. *Automated Testing for Z Specifications.*

Our Contributions

- Presenting a denotational model for test cases
- Providing a constructive test case generation law for designs
- Lifting the test case generation process to the syntactical level

Roles of a Programming Theory

- Linking programs with specifiations
- Exploring algebraic properties of programs
- Supporting sound developing methods
- Providing a theory of testing

Introducing faults into Specifications

We aim to generate test cases on the basis of possible errors during the design of software. Examples of such errors might be

- a missing ot misunderstood requirement
- a wrongly implemented requirement
- simple coding errors

To represent these erros we will introduce faults into formal specifications by deliberately changing a design

Design

Designs are a special form of predicates

- with a pre- and postcondition part,
- together with an alphabet, which is a set of variables that declares the observation space.

Definition Design Let P and Q be predicates not containg variables ok and ok'

$$P \vdash Q \ =_{df} \ (ok \wedge P) \Rightarrow (ok' \wedge Q)$$

Programs as Designs

- $\texttt{skip} \;=_{df}\; \mathbf{true} \vdash (x' = x \wedge y' = y \wedge ... \wedge z' = z)$
- $\bot \;=_{df}\; (\mathbf{false} \vdash \mathbf{true})$
- $x := e \;=_{df}\; \mathbf{wf}(e) \vdash (x' = x \wedge y' = y \wedge ... \wedge z' = z)$
  where **wf** is the predicate defining the well-formedness of expression e.
- $P \lhd b \rhd Q \;=_{df}\; \mathbf{wf}(b) \vdash (b \wedge P \vee \neg b \wedge Q)$

Programs as Designs

- **Sequential Composition**

$$P(v'); Q(v) \;=_{df}\; \exists v_0 \bullet P(v_0) \wedge Q(v_0)$$

- **Nondeterministic Choice**

$$P \sqcap Q \;=_{df}\; P \vee Q$$

Design Calculus

Let D_1 and D_2 be designs with alphabet A. D_2 is a refinement of D_1 if it behaves less nondeterministic than D_1

$$D_1 \sqsubseteq D_2 \;=_{df}\; \forall v,\, w.. \in A \bullet (D_2 \Rightarrow D_1)$$

Theorem

(1) $(P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2) \;=\; (P_1 \wedge P_2) \vdash (Q_1 \vee Q_2)$

(2) $(P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2) \;=$
$(P_1 \lhd b \rhd P_2) \vdash (Q_1 \lhd b \rhd Q_2)$

(3) $(P_1 \vdash Q_1); (P_2 \vdash Q_2) \;=$
$(\neg(\neg P_1; true) \wedge \neg(Q_1; \neg P_2)) \vdash (Q_1; Q_2)$

Design Fault

Faults appear on different levels of abstraction in the refinement hierarchy ranging from requirements to implementations. Refinement is the central notion in order to discuss the roles and consequences of certain faults.

Definition Faulty mutation Given an intended design D and a unintended design D^m during which creation an error had been made (m stands for mutation). Then we define a design fault in design D^m as the syntactical difference to D if $\neg(D \sqsubseteq D^m)$.

We call D^m a faulty design or a faulty mutation of D.

Test Cases as Designs

Definition Deterministic Test Cases Let i be the input vector and o be the expected output vactor., both being lists of values, having the same length as the variable lists v and v' respectively.

$$t_d(i,\, o) \;=_{df}\; (v = i) \vdash (v' = o)$$

Definition General Test Cases

$$t_{\sqcap}(i,\, c) \;=_{df}\; (v = i) \vdash c(v')$$

where c is a condition on the after state space defining the possibly infinite set of expected outcome vectors.

Specification, Implementation and Test

Definition Let T be a set of test cases, S a specification and I an implementation, all being designs, and for all $t \in T$

$$t \sqsubseteq S \sqsubseteq I$$

we define

- T as a *correct test set* with respect to S
- Implementation I *passes* the test cases in T
- Implementation I *conforms* to specification S

Faulting-detecting Test Case

Finding a test case that detects a given fault is the central strategy in fault-based testing.

Definition t is a fault-detecting case with respect to the intended design D and its faulty version D^m if

$$t \sqsubseteq D \qquad \text{and} \qquad \neg(t \sqsubseteq D^m)$$

In this case we say t distinguishes D and D^m. In the language of mutation testing, t *kills* the mutant D^m.

All the test cases that detects a certain fault form a *fault-detecting equivalence class.*

Equivalence Class Testing

To reduce test cases, a common technique is the partitioning of the input domaion (or output range) into equivalence classes. A well-known method is DNF partitioning: the rewriting of a specification into a disjunctive normal form

Definition Test Equivalence Class Given a design $D = (b \vdash Q)$, we define a test equivalence cladd T for testing D as a design of the form

$$d_\perp ; D \quad \text{such that} \quad \forall v \bullet (d \Rightarrow b)$$

where $d_\perp =_{df} (\texttt{skip} \lhd d \rhd \perp)$

A Representive Test Case

A test case $(v = i) \vdash c(v')$ is a *representive test case* of a test equivalence class $d_{\perp}; (b(v) \vdash Q(v, v'))$ if

$$d(i) \wedge [Q(i, v') \equiv c(v')]$$

This definition ensures that the output condition of a representive test case is not *weaker* than its test equivalence class specifies.

Question How to choose d such that its representive test case can detect the faulty design D^m

Fault-detecting Equivalence Class

Theorem 1

Given a design $D = (p \vdash Q)$ and its faulty design $D^m = (p^m \vdash Q^m)$

Then every representative test case of $d_\perp ; D$ with

$$d \;=\; (p \wedge \neg p^m) \vee (p \wedge p^m \wedge \exists v' \bullet (\neg Q \wedge Q^m))$$

is able to detect the fault in D^m

For a test case t being able to distinguish D from D^m, refinement between t and D^m must not hold. For designs one may observe two cases where refinement may be violated:

- The first class are test inputs that work for D but cause D^m to abort.
- The second class are the common input states that produces different output values.

Testing for Program Faults

$$
\begin{aligned}
P \;::=\; & \bot \\
| \quad & < variable\ list > := < expression\ list > \\
| \quad & P \lhd < Boolean\ Expression > \rhd P \\
| \quad & P; P \\
| \quad & P \sqcap P \\
| \quad & < recursive\ identifier > \\
| \quad & \mu < recursive\ identifier > \bullet P
\end{aligned}
$$

Normal Form

Algebraic laws, expressing familiar properties of the operators in the language, can be used to reduce every program to a *normal form*. Our idea is to use normal form to decide if two programs, the original one and the mutant can be distinguished by a test c

- If the normal forms of both are equivalent, then the error did not lead to an observable fault. This solves the problem of equivalent mutants in mutation testing
- Otherwise, the normal form will be used for deriving fault-detecting equivalence classes on a purely syntactic base.

Variety of Normal Forms

- Assignment Normal Form
- Nondeterministic Normal Form
- Nontermination Normal Form

Assignment Normal Form

The normal form for assignments is the *total assignment*, in which all the variables of the program appear on the left hand side in some standard order:

$$x,\, y,\, \ldots z := e,\, f,\, \ldots g$$

A non-total assignment can be converted to a total assignment by

1. addition of identity assignments

$x := e) \;=\; (x,\, y := e,\, y)$

2. reording of variables with their associated expression

$$(x,\, y := e,\, f) \;=\; (y,\, x := f,\, e)$$

Refinement of Assignment Normal Form

We can eliminate sequential composition between total assignments by using the following law:

Law $(v := g; v := h(v)) \;=\; (v := h(g))$

For assignments are deterministic, the question of refinement becomes a simple question of equality:
Two assignment normal forms are equal if and only if all the expressions in the total assignment are equal:

$$(v := g) \;\equiv\; (v := h) \quad \textbf{iff} \quad [g = h]$$

Non-derterministic Normal Form

A nondeterministic normal form is defined to be a non-deterministic choice of guarded total assignments

$$(g_1 \wedge v := f_1) \sqcap ... \sqcap (g_n \wedge v := f_n)$$

where $(g_1 \vee ... \vee g_n) \ = \ true$

Let A be a set of guarded total assignment, we write the normal form as $\bigsqcap A$.

A total assignment can be easily expressed in this new normal form

$$v := g \ = \ \bigsqcap \{true \wedge (v := g)\}$$

Elimination of Programming Combinators

The following laws can be used to remove programming operators between two non-deterministic normal forms

Law $(\prod A) \sqcap (\prod B) = \prod (A \cup B)$

Law $(\bigcap A) \lhd d \rhd (\prod B) =$

$\prod \{(d \wedge b) \wedge P \mid (b \wedge P) \in A\} \cup \{(\neg d \wedge c) \wedge Q \mid c \wedge Q \in B\}$

Law $(\prod A); (\prod B) =$

$\prod \{(b \wedge c(g)) \wedge (v := h(g)) \mid b \wedge (v := g) \in A \wedge c \wedge (v := h) \in B\}$

Non-termination Normal Form

A non-termination normal form is a program represented as a disjunction

$$b \vee (\bigsqcap A)$$

Any nondeterministic normal form P can be expressed as

$$false \ \vee \ P$$

and the chaotic program $\perp$ as

$$true \ \vee \ (v := v)$$

Test Case Generation from Normal Forms

Theorem 2 Let P be a program and P^m a faulty mutation of this program with normal forms

$$P \;=\; c \vee \bigsqcap\{(a_j \wedge v := f_j) \mid 1 \leq j \leq m\}$$

$$P^m \;=\; d \vee \bigsqcap\{(b_k \wedge v := h_k) \mid 1 \leq k \leq n\}$$

Then, every representative test case of the test equivalence class $T \;=\; d; P$ where

$$d \;=\; (\neg c \wedge d) \vee \bigvee_k (\neg c \wedge b_k \wedge \bigwedge_j (\neg a_j \vee (f_j \neq h_k)))$$

Example

Min is a program for computing the minumum of x and y

$$Min \;=_{df}\; (z := x \lhd x \leq y \rhd (z := y)$$

$$= \quad ((x,\, y,\, z := x,\, y,\, x) \lhd x \leq y \rhd (x,\, y,\, z := x,\, y,\, y)$$

$$= \quad (x \leq y) \wedge (x,\, y,\, z := x,\, y,\, x) \sqcap (x > y) \wedge (x,\, y,\, z := x,\, y,\, y)$$

The mutant Min^m models a mix-operator error

$$Min^m \;=_{df}\; (z := x) \lhd x \geq y \rhd (z := y)$$

$$= \quad (x \geq y) \wedge (x,\, y,\, z := x,\, y,\, x) \sqcap (x < y) \wedge (x,\, y,\, z := x,\, y,\, y$$

Example (cont'd)

$$
\begin{aligned}
d \;=\;& (\neg false \wedge false) \vee \\
& b_1 \wedge (\neg a_1 \vee f_1 \neq h_1) \wedge (\neg a_2 \vee f_2 \neq h_1) \vee \\
& b_2 \wedge (\neg a_1 \vee f_1 \neq h_2) \wedge (\neg a_2 \vee f_2 \neq h_2) \\
=\;& (x \geq y) \wedge (x > y \vee false) \wedge (x \leq y \vee x \neq y) \vee \\
& (x < y) \wedge (x > y \vee x \neq y) \wedge (x \leq y \vee false) \\
=\;& x > y \vee x < y
\end{aligned}
$$

Infinite Normal Form

An infinite normal form for recursive program is a program represented as least upper bound of decending chain of finite normal form

$$\bigsqcup\{(c_n \vee Q_n) |\ n \in Nat\}$$

where

(1) $\forall v \bullet (c_{n+1} \Rightarrow c_n)$ fior all $n \in Nat$

(2) Q_n is a non-deterministic normal form

Theorem

$(\bigsqcup S) \sqsubseteq (\bigsqcup T)$ if and only if $\forall n \bullet (S_n \sqsubseteq \bigsqcup T)$

Test Case Generation for Recursion

The central idea to deal with recursive program in our test case generation approach is to approximate the normal form of both the program and the mutant until non-refinement can be detected. For equivalent mutants an upper limit n will determine when to stop the search.

Example

Assume that we want to find an index pointing to the smallest element in an array $A[1, .., n]$.

$$MIN \;=_{df}\; k := 2; t := 1; \mu X \bullet F(X)$$

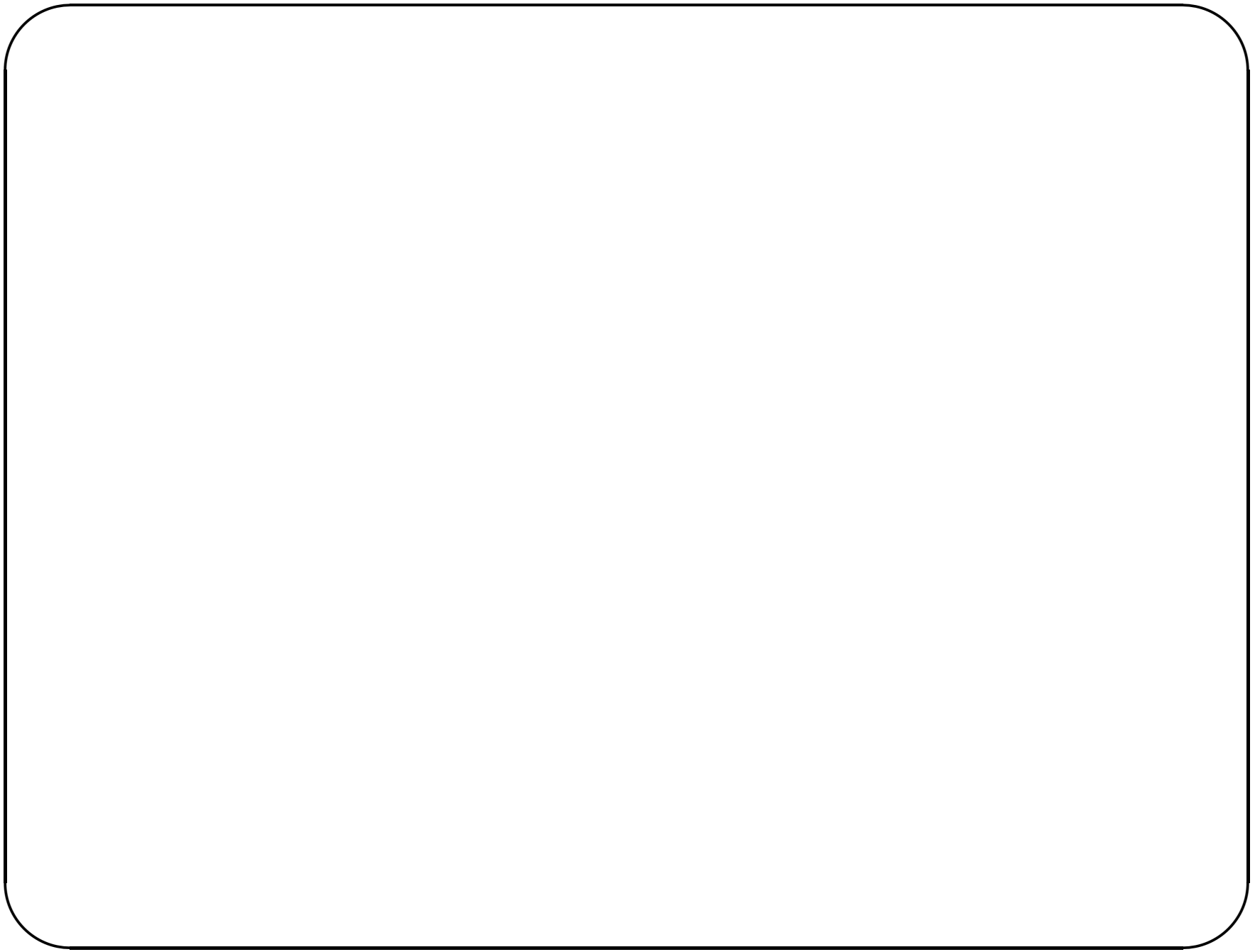
$$F(X) \;=_{df}\; (B; X) \lhd k \leq n \rhd k,\, t := k,\, t$$

$$B \;=_{df}\; (t := k,\, k := k + 1) \lhd A[k] < A[t] \rhd k := k + 1$$

A common error is to get the loop termination condition wrong.

$$MIN \;=_{df}\; k := 2; t := 1; \mu X \bullet G(X)$$

$$G(X) \;=_{df}\; (B; X) \lhd k < n \rhd k,\, t := k,\, t$$



First Approximation

$$
\begin{aligned}
S_1 &=_{df} F(\mathbf{true}) \\
&= (k \leq n) \vee ((k > n) \wedge (k, t := k, t)) \\
S_1^m &=_{df} G(\mathbf{true}) \\
&= (k < n) \vee ((k \geq n) \wedge (k, t := t, k))
\end{aligned}
$$

we find that such a test case does not exist, because $S_1 \sqsubseteq S_1^m$

$$
\begin{aligned}
d_1 &= (\neg(k \leq n) \wedge k < n) \vee \\
&\quad (\neg(k \leq n) \wedge k \geq n \wedge (\neg(k > n) \vee \mathbf{false})) \\
&= \mathbf{false}
\end{aligned}
$$

Second Approximation

$$
\begin{aligned}
S_2 \;&=_{df}\; F(S_1) \\
&=\; (k < n) \vee \\
&\quad (((k = n) \wedge A[k] < A[t]) \wedge (k,\, t := k+1,\, k)) \sqcap \\
&\quad (((k = n) \wedge A[k] \geq A[t]) \wedge (k,\, t := k+1,\, t)) \sqcap \\
&\quad ((k > n) \wedge (k,\, t := k,\, t)) \\
S_2^m \;&=\; (k+1 < n) \vee \\
&\quad (((k+1 = n) \wedge A[k] < A[t]) \wedge (k,\, t := k+1,\, k)) \sqcap \\
&\quad (((k+1 = n) \wedge A[k] \geq A[t]) \wedge (k,\, t := k+1,\, t)) \sqcap \\
&\quad ((k \geq n) \wedge (k,\, t := k,\, t))
\end{aligned}
$$

Test Equivalence Class

$$d_2 \;=\; (k \geq n) \wedge (k \leq n) \;=\; (k = n)$$

By substitutiong the initialisation values

$$k = 2 \;\text{ and }\; t = 1$$

the concretye fault detecting test equivalence class is

$$T^2 \;=\; (n = 2)_{\perp}; MIN$$

It says that everry array with two elements can serve as a test case to detect the error. The reason is that two program versions can be distinguished by their different values for k (3 vs 2)

Discussion

- The present theory is far from being final or stable
- The next step is to improve the existing constraint solver for generating test cases from mutated OCL specifications based on Theorem 1.
- It is another step in our research aim to establish a unifying theory of test, which can provide semantic link between different testing theories and models.