Mutation Testing in UTP

Test Cases, Faults, Mutation Testing for Designs

Bernhard K. Aichernig

Institute for Software Technology
Graz University of Technology
Graz, Austria

UNU-IIST: United Nations University
International Institute for Software Technology
Macau S.A.R. China

PSSE 2007

Outline

- Introduction
- Unifying Theories of Programming (UTP)
- Formalisation: Faults and Test Cases
- Mutation Testing for Pre-postcondition Contracts
- Mutation Testing for Programs
- Mutation Testing for Protocol Specifications

UTP: Theory of Designs

Designs

Let p and Q be predicates not containing ok or ok'

$$p \vdash Q \quad =_{df} \quad (ok \wedge p) \;\Rightarrow\; (ok' \wedge Q)$$

A design is a relation whose predicate is (or could be) expressed in this form (+ healthiness conditions).

Refinement

Correctness is defined via implication:

$$\forall v, w, \cdots \in A \bullet P \;\Rightarrow\; S\,, \qquad \text{for all } P \text{ with alphabet } A.$$

we write $[P \;\Rightarrow\; S]$ or $S \sqsubseteq P$

UTP: Theory of Designs

Designs

Let p and Q be predicates not containing ok or ok'

$$p \vdash Q \quad =_{df} \quad (ok \wedge p) \;\Rightarrow\; (ok' \wedge Q)$$

A design is a relation whose predicate is (or could be) expressed in this form (+ healthiness conditions).

Refinement

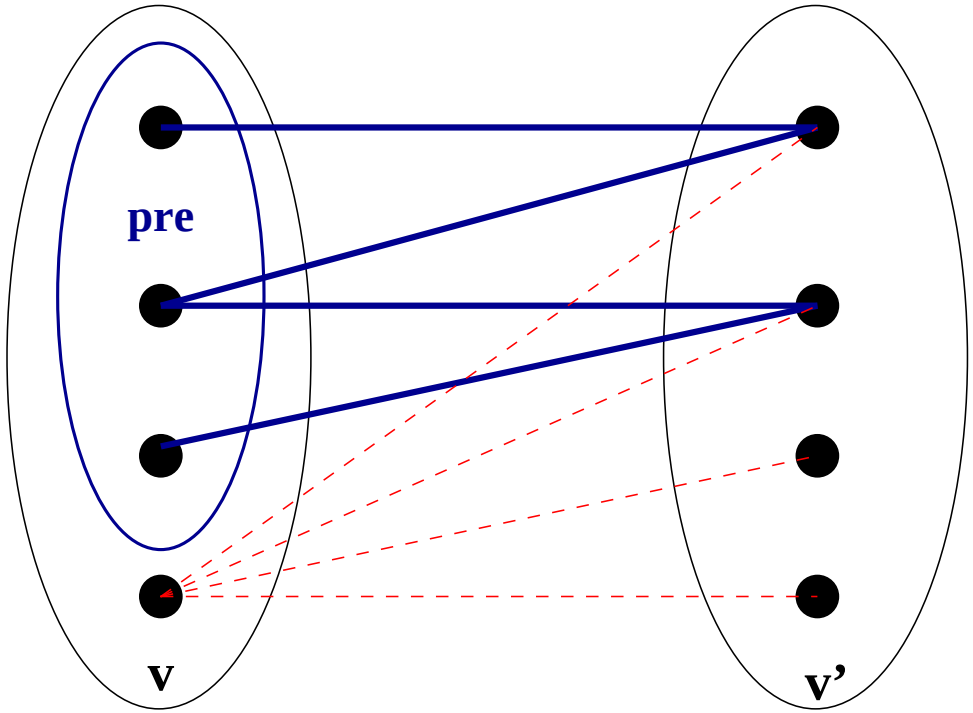
Correctness is defined via implication:

$$\forall v, w, \cdots \in A \bullet P \;\Rightarrow\; S\,, \qquad \text{for all } P \text{ with alphabet } A.$$

we write $[P \;\Rightarrow\; S]$ or $S \sqsubseteq P$

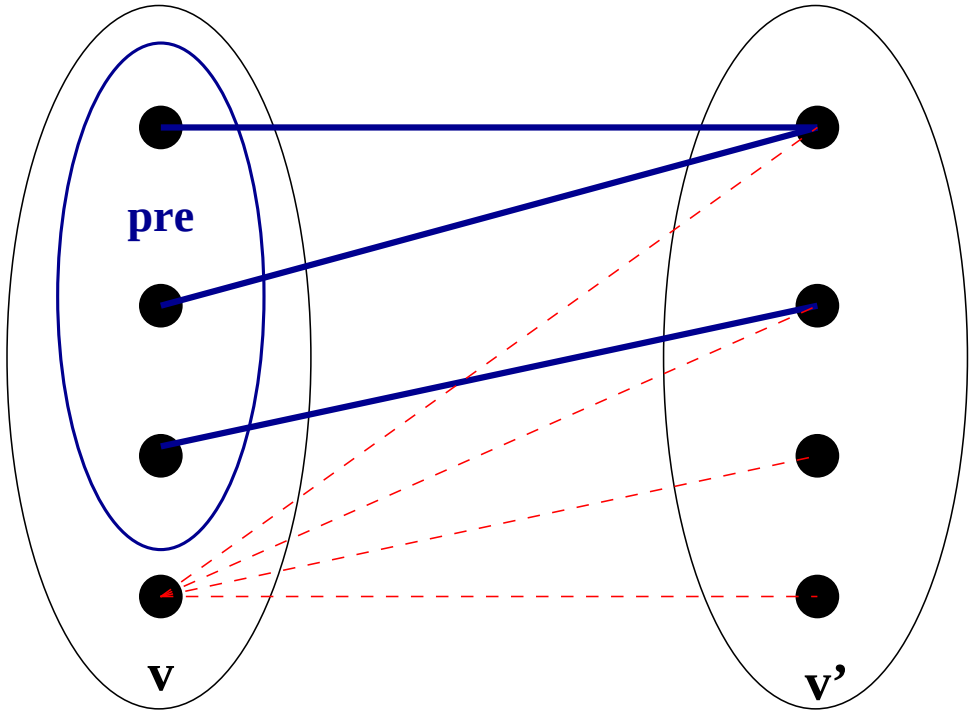
Test Case Semantics

Designs represent relations:



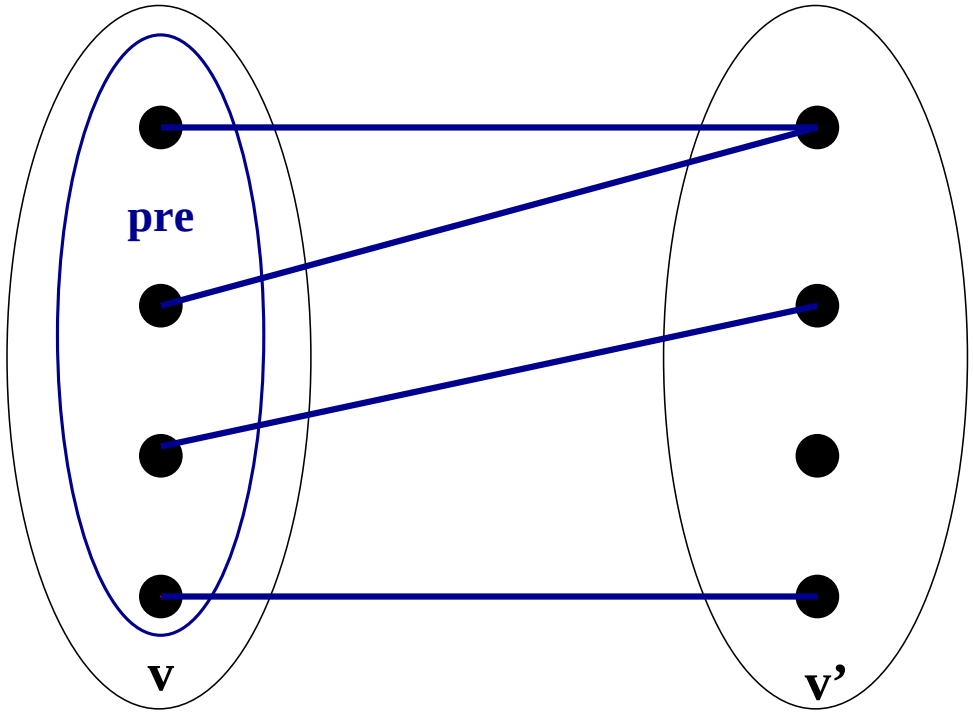
Test Case Semantics

Refinement 1: postcondition strengthening



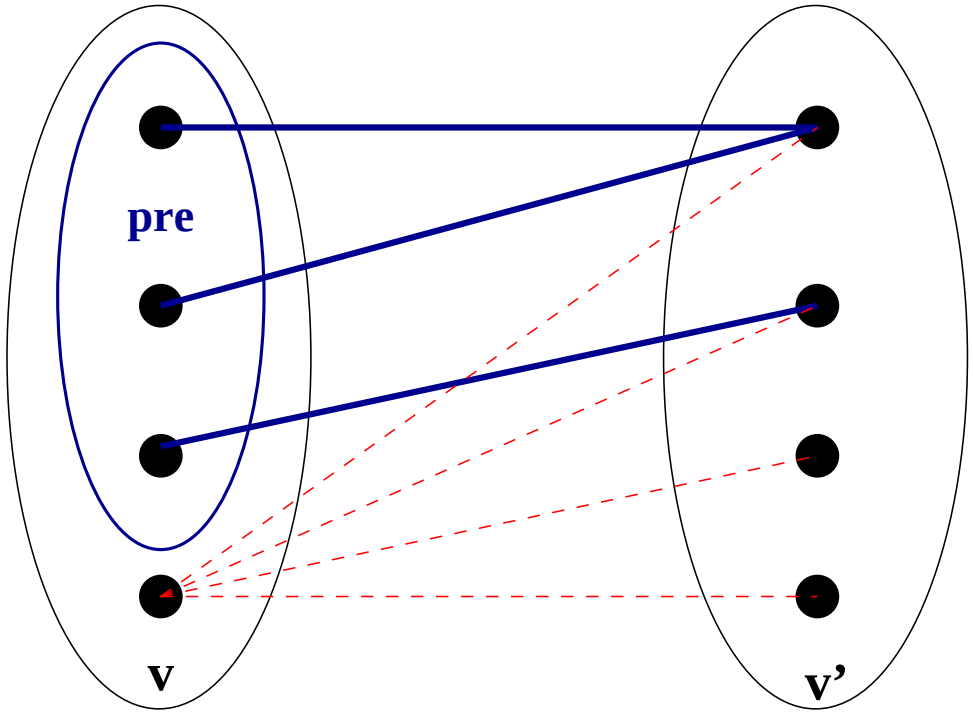
Test Case Semantics

Refinement 2: precondition weakening



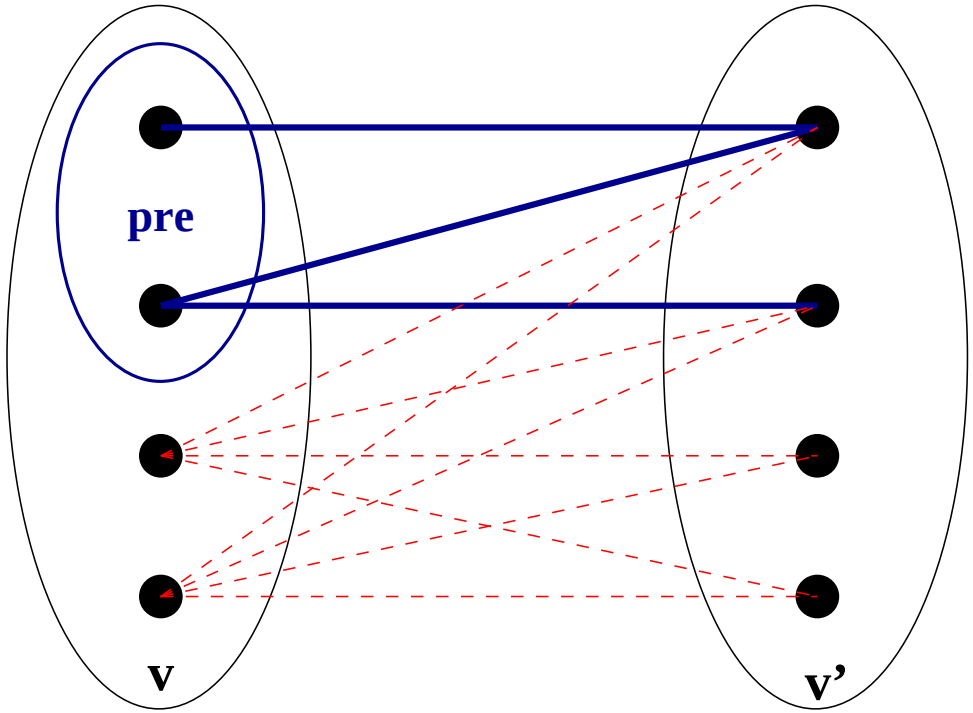
Test Case Semantics

Abstraction 1: precondition strengthening



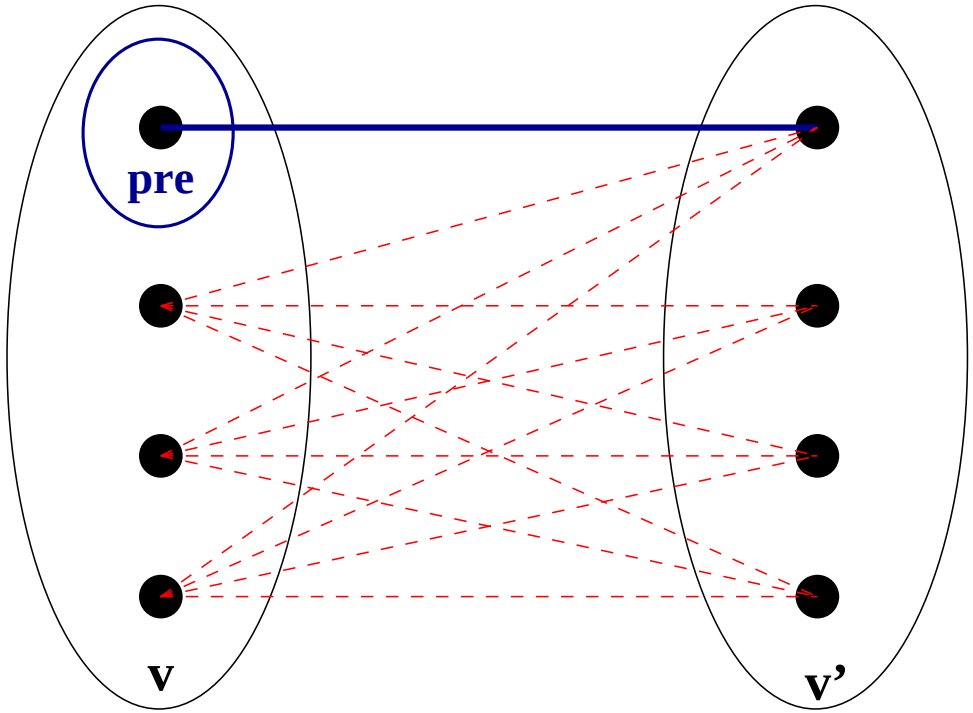
Test Case Semantics

Abstraction 2: further precondition strengthening



Test Case Semantics

Abstraction 2: precondition strengthening to single input



Notice! this represents a test case (input-output relation)

Test Case Semantics

In our theory, test cases are interpreted as designs (specifications).

Test Case, deterministic

$$t(i, o) \quad =_{df} \quad v = i \vdash v' = o$$

Test Case, non-deterministic

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

The relation between a set of test cases T, a specification S and its implementation I is defined by

$$T \sqsubseteq S \sqsubseteq I$$

Test Case Semantics

In our theory, test cases are interpreted as designs (specifications).

Test Case, deterministic

$$t(i, o) \quad =_{df} \quad v = i \vdash v' = o$$

Test Case, non-deterministic

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

The relation between a set of test cases T, a specification S and its implementation I is defined by

$$T \sqsubseteq S \sqsubseteq I$$

Order of Test Cases

Theorem

$$[t_{\sqcap}(i, c) \Rightarrow t_{\sqcap}(i, d)] \quad \textbf{iff} \quad [c \Rightarrow d]$$

Border (Test) Cases

Definition (Explorative Test Case)

$$T_?(i) \quad =_{df} \quad t_\sqcap(i, \mathbf{true})$$

Definition (Infeasible Test Case)

$$T_\varnothing(i) \quad =_{df} \quad t_\sqcap(i, \mathbf{false})$$

- Infeasible test cases: an input i is not accepted,
- since $t_\sqcap(i, \mathbf{false}) = \neg(ok \wedge v = i)$

Order of Test Cases (cont.)

Theorem (Order of Test Cases)

For a given input vector i, output vector o and condition c

$$\bot \sqsubseteq T_?(i) \sqsubseteq t_\sqcap(i, c) \sqsubseteq t(i, o) \sqsubseteq T_\varnothing(i) \sqsubseteq \top$$

provided $c(o)$ holds.

Test Suites

Definition (Test Suite)

Given a set s of test cases $t_1, \ldots, t_n$

$$TS(s) \quad =_{df} \quad t_1 \sqcup \ldots \sqcup t_n$$

- $\sqcup \quad =_{df} \quad \wedge$
- Test suite is conjunction of all test cases
- Contradicting test cases?: they are infeasible (equals magic).

Building Test Suites

- Adding test cases is refinement:

Theorem

Let T_1, T_2 be test cases of any type

$$T_i \sqsubseteq T_1 \sqcup T_2\,, \qquad i \in \{1, 2\}$$

- follows immediately from Lattice Theory

Exhaustive Test Suites

Definition (Exhaustive Test Suite)

Let D be a design, its set of *exhaustive test suites* is defined as

$$TS_{exhaustive} =_{df} \{TS(s) \mid TS(s) = D\}$$

- Note: Exhaustive with respect to a design!
  - skips inputs outside the precondition.

Exhaustive Test Suites (cont.)

Adding explorative test cases is useless:

Theorem

Given a design $D = p \vdash Q$ and one of its exhaustive test suites $ts_{exhaustive} \in TS_{exhaustive}$

$$ts_{exhaustive} \sqcup T_?(i) = ts_{exhaustive}, \quad \text{provided } p(i) \text{ holds.}$$

Proof. via lattice theory

$$\begin{aligned}
ts_{exhaustive} \sqcup T_?(i) = & \quad \{\text{by definition of exhaustive test suites}\} \\
& D \sqcup T_?(i) = D \\
= & \quad \{\text{by lattice theory}\} \\
& T_?(i) \sqsubseteq D \\
= & \quad \{\text{by refinement laws}\} \\
& [v = i \Rightarrow p] \wedge [(v = i \wedge Q) \Rightarrow \mathbf{true}] \\
= & \quad \{\text{since p(i) holds}\} \\
& true
\end{aligned}$$

Exhaustive Test Suites (cont.)

Adding more test cases for a design D is useless:

Theorem

Given a design D and an exhaustive test suite $ts_{exhaustive} \in TS_{exhaustive}$. Furthermore, we have a test case $t \sqsubseteq D$ expressing the fact that t has been derived from D. Then,

$$ts_{exhaustive} \sqcup t = ts_{exhaustive}$$

Proof. by lattice theory

$$\begin{aligned} ts_{exhaustive} \sqcup t &= \{\text{by definition of exhaustive test suites}\} \\ & D \sqcup t \\ &= \{\text{by lattice theory, since } t \sqsubseteq D\} \\ & D \\ &= \\ & ts_{exhaustive} \end{aligned}$$

Conformance, Refinement, Abstraction

Definition

Let T be a test suite, S a specification, and I an implementation, all being designs, and

$$T \sqsubseteq S \sqsubseteq I$$

we define

- T as a *correct test suite* with respect to S,
- all test cases in T as *correct test cases* with respect to S,
- implementation I *passes* a test suite (test case) T,
- implementation I *conforms* to specification S.

Conformance, Refinement, Abstraction (cont.)

Theorem

$$t(i, o) \sqsubseteq D \quad \textbf{iff} \quad v := o \sqsubseteq (v := i; D)$$

$$t_{\sqcap}(i, c) \sqsubseteq D \quad \textbf{iff} \quad c(v') \sqsubseteq (v := i; D)$$

Observable Faults in Design

Definition (Faulty Design)

- Let D be a design, and D^m its mutated version, meaning that D^m has been produced by slightly altering D.
- Furthermore, let the mutation represent a fault model.
- Then, the mutated design D^m is defined to be a *faulty design* (or a *faulty mutation*) of D, if

$$D \not\sqsubseteq D^m$$

- (or $\neg(D \sqsubseteq D^m)$).

Equivalent Mutants

- informal: cannot be killed by any test case
- formal: $D \sqsubseteq D^m$
- Note: Does not mean mathematical equivalence,
- but refining mutants:
  - mutation outside the precondition (additional observations)
  - mutation strengthening the postcondition

Outline

- Introduction
- Unifying Theories of Programming (UTP)
- Formalisation: Faults and Test Cases
- Mutation Testing for Pre-postcondition Contracts
- Mutation Testing for Programs
- Mutation Testing for Protocol Specifications

Observable Faults and Test Cases

Theorem

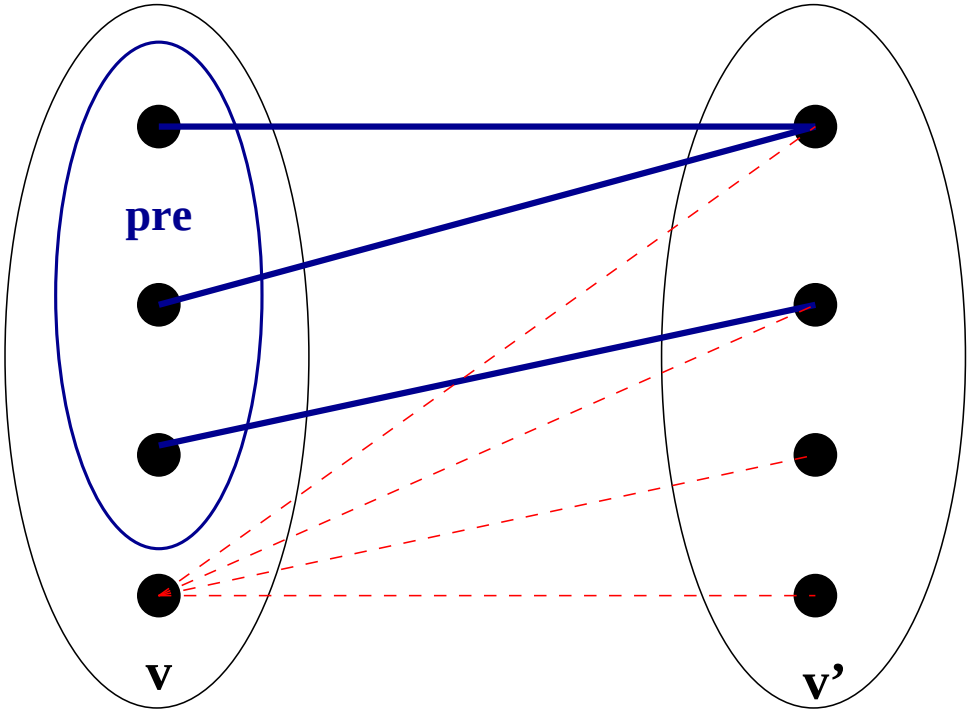
Given a design D, and a faulty design D^m, then there exists a test case t, with $t \sqsubseteq D$, such that $t \not\sqsubseteq D^m$.

Observable Faults and Test Cases: Proof

- Assume that such a test case does not exist and for all test cases $t \sqsubseteq D$ also $t \sqsubseteq D^m$ holds.
- This set of all possible test cases $t \sqsubseteq D$ defines an exhaustive test suite $ts_{exhaustive}$ of D.
- This is obvious, since the least upper bound of all such t is the design D.
- Hence, by definition of exhaustive test suites we have $ts_{exhaustive} = D$ and by assumption $ts_{exhaustive} \sqsubseteq D^m$.
- From this follows that $D \sqsubseteq D^m$. This is a contradiction to our assumption that D^m is a faulty design. Consequently, the theorem holds.

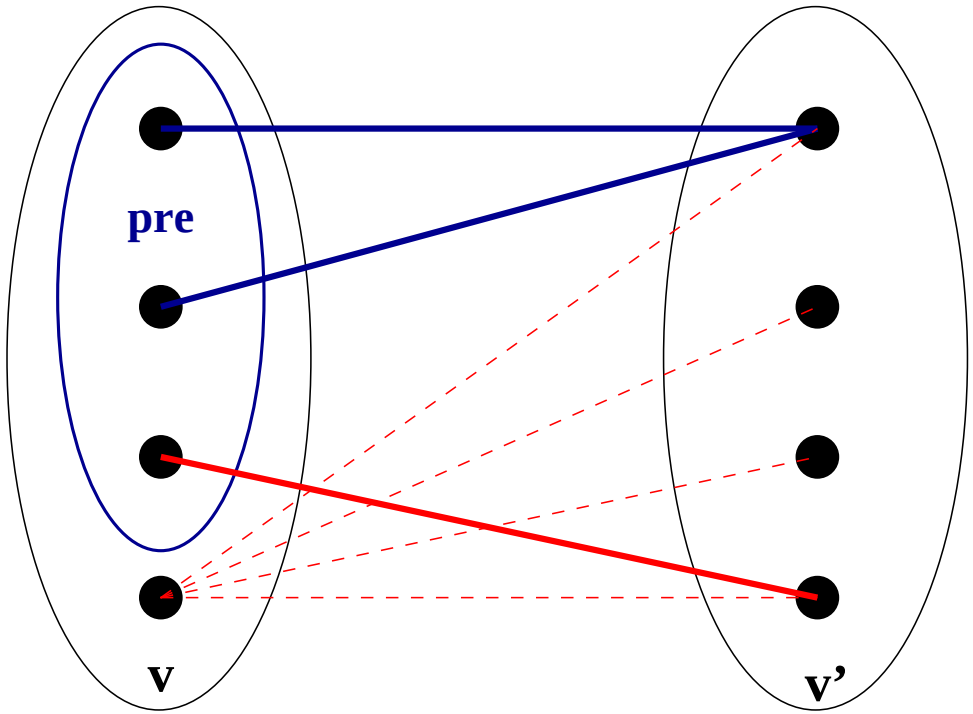
Fault-detecting Test Cases

Given a design (specification) D:



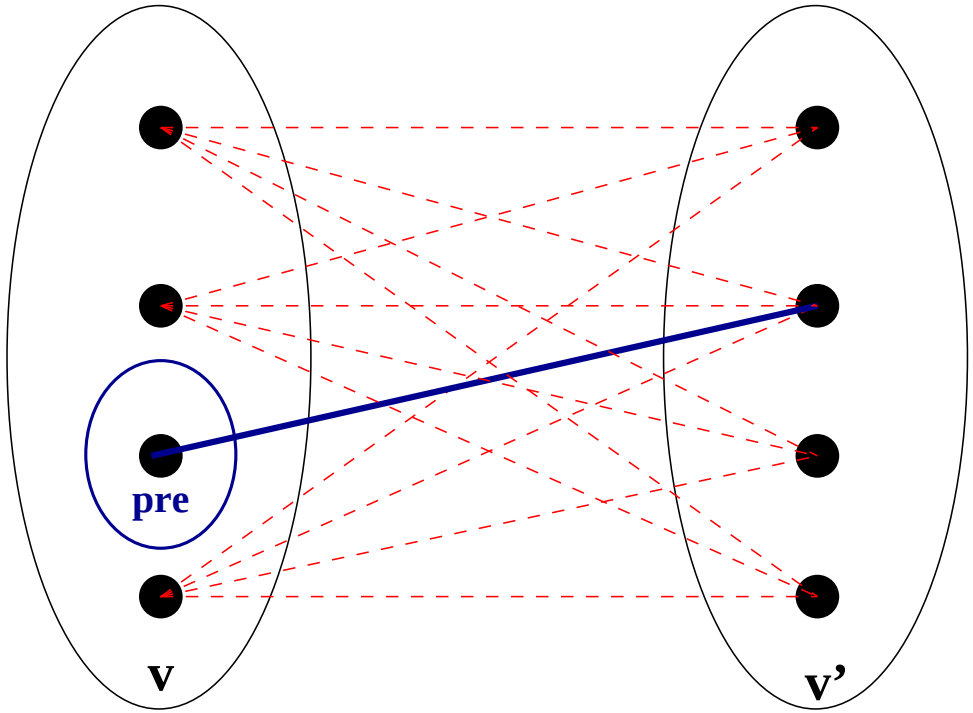
Fault-detecting Test Cases

and a faulty design D^m (created by mutating D) and $D \not\sqsubseteq D^m$:



Fault-detecting Test Cases

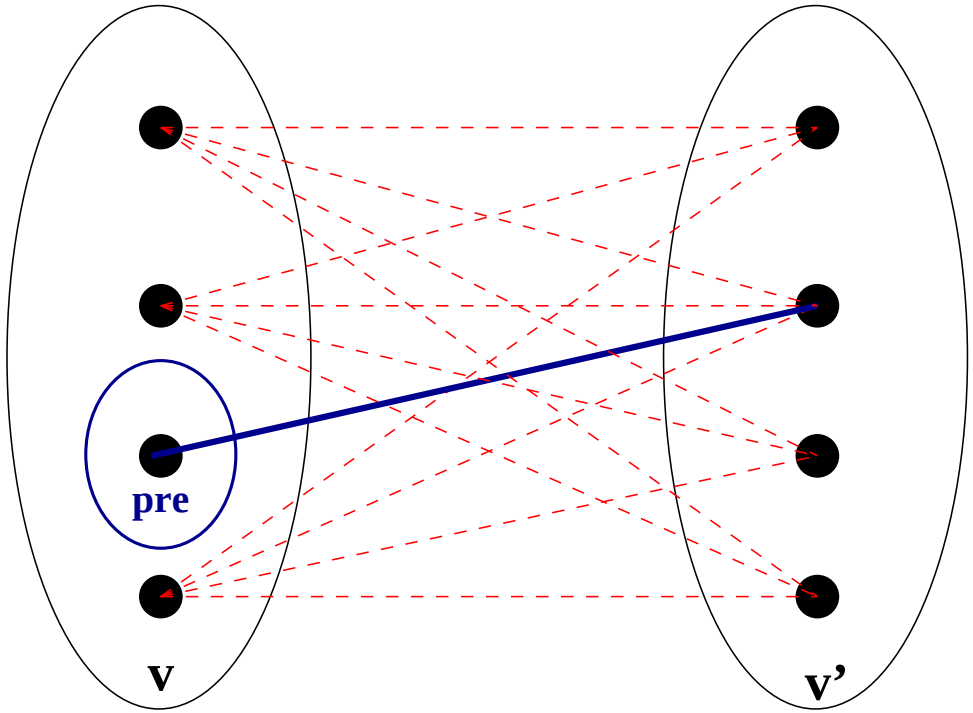
a test case t shall be able to distinguish D and D^m:



Test case property: $t \sqsubseteq D \quad \wedge \quad (t \not\sqsubseteq D^m)$

Fault-detecting Test Cases

a test case t shall be able to distinguish D and D^m:



Test case property: $t \sqsubseteq D \quad \wedge \quad (t \not\sqsubseteq D^m)$

Fault-detecting Test Cases: Definition

Definition (Fault-detecting Test Case)

- Let t be either a deterministic or non-deterministic input-output test case.
- Furthermore, D is a design and D^m its faulty version.
- Then, t is a *fault-detecting test case* when

$$(t \sqsubseteq D) \quad \text{and} \quad (t \not\sqsubseteq D^m)$$

- Fault-detecting test case *detects* the fault in D^m.
- Test case *distinguishes* D and D^m.
- Context of mutation testing: t *kills* the mutant D^m.

Fault-detecting Test Cases: Method

- Given a specification D and a possible fault represented as a mutant D^m.
- Take a fault-detecting test case t.
- Apply fault-detecting test case on an implementation I of D.

$$D \sqsubseteq I$$

- in order to prevent the fault from being implemented.
- Note: $D \sqsubseteq I \wedge D^m \sqsubseteq I$ possible.

Fault-detecting Test Cases: Method

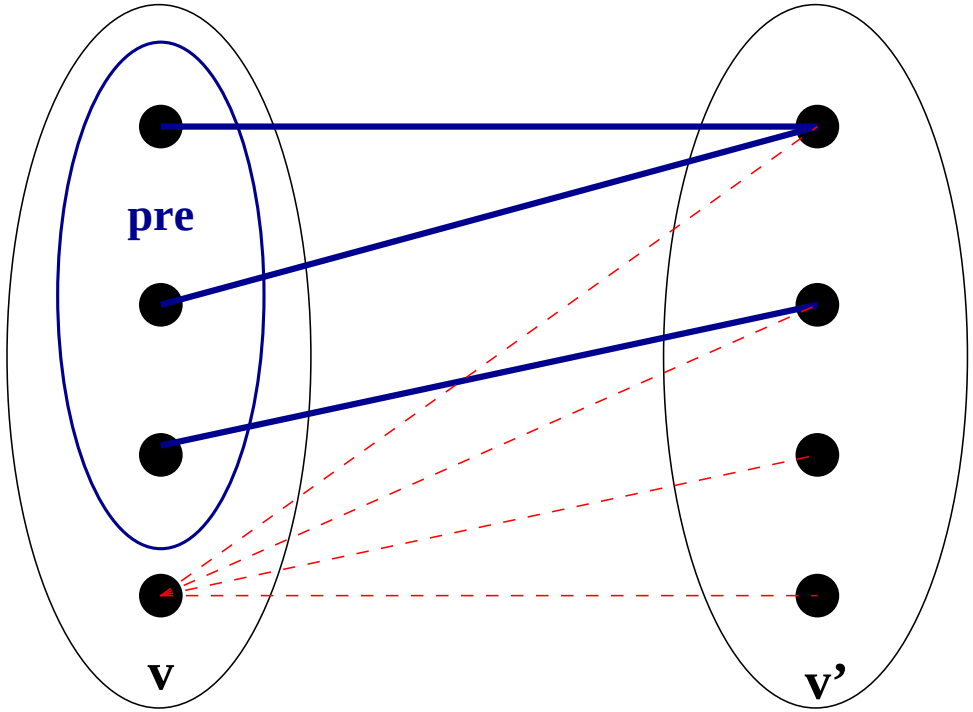
- Given a specification D and a possible fault represented as a mutant D^m.
- Take a fault-detecting test case t.
- Apply fault-detecting test case on an implementation I of D.

$$D \sqsubseteq I$$

- in order to prevent the fault from being implemented.
- Note: $D \sqsubseteq I \wedge D^m \sqsubseteq I$ possible.

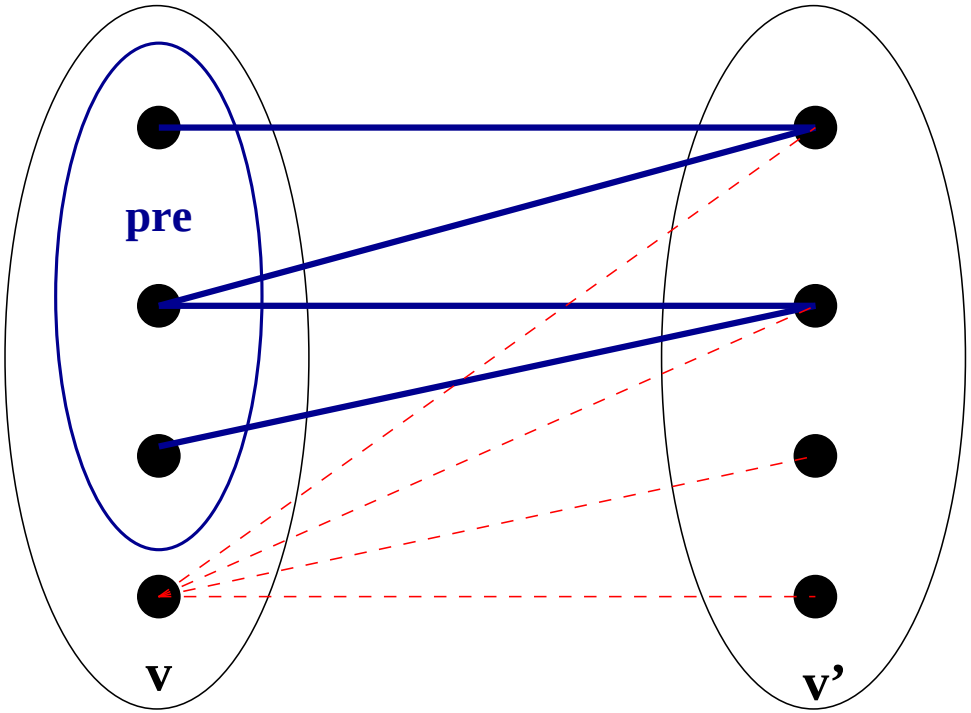
Implementing Original AND Mutant: Example

Given a design (specification) D:



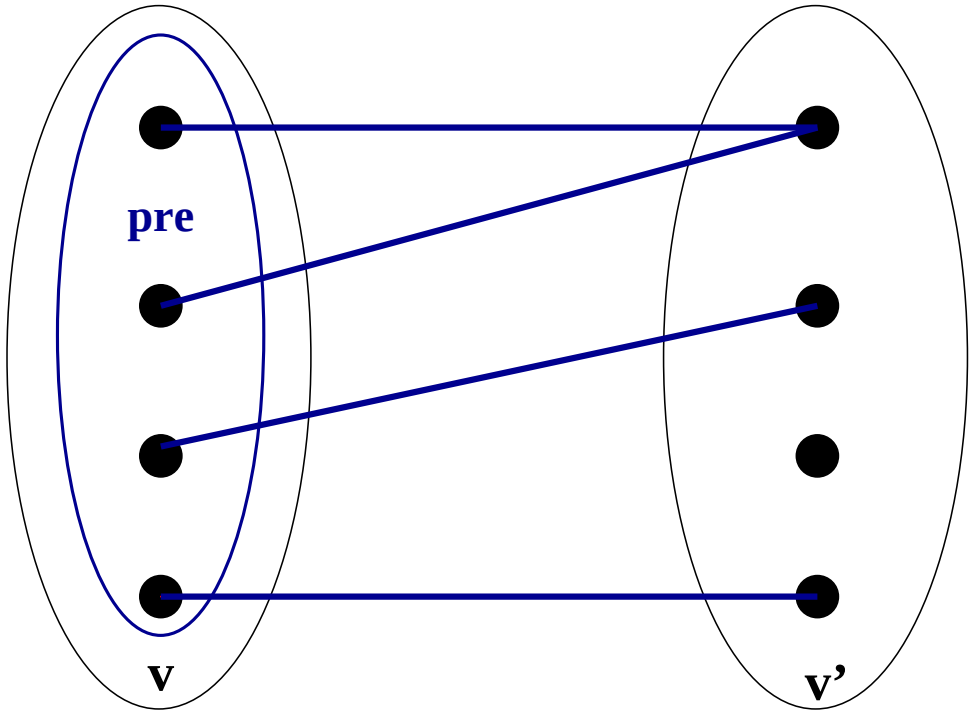
Implementing Original AND Mutant: Example

Given a mutant D^m by abstracting from D:



Implementing Original AND Mutant: Example

Implementation I implements both D^m and D:



Test Equivalence Class

Definition (Assertion)

$$b_\perp \quad =_{df} \quad true \vdash ((v = v') \lhd b \rhd \perp)$$

Definition (Test Equivalence Class)

Given a design $D = (p \vdash Q)$, we define a *test equivalence class* $T_\sim$ for testing D as a design of form $T_\sim = d_\perp ; D$ such that

- $[d \Rightarrow p]$.
- condition d represents an equivalence class with respect to some test coverage goal.

The condition d is called the *domain* of the test equivalence class.

- Our aim: equivalence class of inputs detecting a given fault.

Finding Fault-detecting Test Cases

- we are interested in the cases, where $D \not\sqsubseteq D^m$

Refinement Theorem

$$(p \vdash Q) \sqsubseteq (p^m \vdash Q^m) \quad \textbf{iff} \quad [p \Rightarrow p^m] \text{ and } [(p \wedge Q^m) \Rightarrow Q]$$

- negating this leads to

Fault-detecting Domain

$$p \wedge \neg p^m \quad \textbf{or} \quad p \wedge \exists v' \bullet (Q^m \wedge \neg Q)$$

Finding Fault-detecting Test Cases

- we are interested in the cases, where $D \not\sqsubseteq D^m$

Refinement Theorem

$$(p \vdash Q) \sqsubseteq (p^m \vdash Q^m) \quad \textbf{iff} \quad [p \Rightarrow p^m] \text{ and } [(p \wedge Q^m) \Rightarrow Q]$$

- negating this leads to

Fault-detecting Domain

$$p \wedge \neg p^m \quad \textbf{or} \quad p \wedge \exists v' \bullet (Q^m \wedge \neg Q)$$

Triangle Example

$$a < (b + c) \land b < (a + c) \land c < (a + b)$$
$$\vdash$$
$$r' = \text{equilateral} \lhd a = b \land b = c \rhd$$
$$(r' = \text{isosceles} \lhd a = b \lor a = c \lor b = c \rhd$$
$$r' = \text{scalene})$$

Triangle Mutant

$$a < (b + c) \land b < (a + c) \land c < (a + b)$$
$$\vdash$$
$$r' = \text{equilateral} \lhd a = a \land b = c \rhd$$
$$(r' = \text{isosceles} \lhd a = b \lor a = c \lor b = c \rhd$$
$$r' = \text{scalene})$$

Fault-Detecting Domain

$$
\begin{aligned}
& p \ \wedge \ \exists v' \bullet (Q^m \wedge \ \neg Q) \\
= \ & a < (b + c) \wedge b < (a + c) \wedge c < (a + b) \\
& \wedge \\
& \exists r' \bullet \\
& r' = \text{equilateral} \lhd a = a \wedge b = c \rhd \\
& \quad (r' = \text{isosceles} \lhd a = b \vee a = c \vee b = c \rhd \\
& \qquad r' = \text{scalene}) \\
& \wedge \\
& \neg (r' = \text{equilateral} \lhd a = b \wedge b = c \rhd \\
& \quad (r' = \text{isosceles} \lhd a = b \vee a = c \vee b = c \rhd \\
& \qquad r' = \text{scalene}))
\end{aligned}
$$

Fault-Detecting Domain

$$
\begin{aligned}
& p \ \wedge \ \exists v' \bullet (Q^m \wedge \ \neg Q) \\
= \ & a < (b + c) \wedge b < (a + c) \wedge c < (a + b) \\
& \wedge \\
& \exists r' \bullet \\
& r' = \text{equilateral} \lhd a = a \wedge b = c \rhd \\
& \quad (r' = \text{isosceles} \lhd a = b \vee a = c \vee b = c \rhd \\
& \qquad r' = \text{scalene}) \\
& \wedge \\
& r' \neq \text{equilateral} \lhd a = b \wedge b = c \rhd \\
& \quad (r' \neq \text{isosceles} \lhd a = b \vee a = c \vee b = c \rhd \\
& \qquad r' \neq \text{scalene}))
\end{aligned}
$$

Fault-Detecting Domain

$$
\begin{aligned}
& p \ \wedge \ \exists v' \bullet (Q^m \wedge \ \neg Q) \\
= \ & a < (b + c) \wedge b < (a + c) \wedge c < (a + b) \\
& \wedge \ \exists r' \bullet \\
& ((b = c \wedge r' = \text{equilateral}) \vee \\
& (b \neq c \wedge a = b \wedge r' = \text{isosceles}) \vee \\
& (b \neq c \wedge a = c \wedge r' = \text{isosceles}) \vee \\
& (b \neq c \wedge a \neq b \wedge a \neq b \wedge r' = \text{scalene})) \\
& \wedge \\
& ((a = b \wedge b = c \wedge r' \neq \text{equilateral}) \vee \\
& (a \neq b \wedge a = c \wedge r' \neq \text{isosceles}) \vee \\
& (a \neq b \wedge b = c \wedge r' \neq \text{isosceles}) \vee \\
& (b \neq c \wedge a = b \wedge r' \neq \text{isosceles}) \vee \\
& (a \neq b \wedge b \neq c \wedge c \neq a \wedge r' \neq \text{scalene}))
\end{aligned}
$$

Fault-Detecting Domain

$$
\begin{aligned}
& p \ \wedge \ \exists v' \bullet (Q^m \wedge \ \neg Q) \\
= \ & a < (b + c) \wedge b < (a + c) \wedge c < (a + b) \\
& \wedge \ \exists r' \bullet \\
& (a \neq b \wedge b = c \wedge r' = \text{equilateral} \wedge r' \neq \text{isosceles})
\end{aligned}
$$

Automated Test Case Generation

- Automated using a Constraint Solver:
  - Aichernig and Pari Salas, Test Case Generation by OCL Mutation and Constraint Solving, QSIC 2005.
- Tool takes OCL pre-postcondition specifications
- and generates for each mutation a test case t detecting the fault.
- We could proof that

$$t \sqsubseteq Spec \wedge t \not\sqsubseteq Spec^m$$

- Hence, our test cases are indeed fault adequate.