Invariants and State in Testing and Formal Methods

Dick Hamlet

Portland State University

Supported by NSF CCR-0112654 and SFI E.T.S. Walton Fellowship

The Simplest Context

Meaning of a program P with persistent state:

- input domain D (*think:* `STDIN`)
- output domain R (*think:* `STDOUT`)
- state space H (*think:* permanent R/W file)

The Simplest Context

Meaning of a program P with persistent state:

- input domain D (*think:* `STDIN`)
- output domain R (*think:* `STDOUT`)
- state space H (*think:* permanent R/W file)

$$P : D \times H \to H \times R$$
$$(d, h') \mapsto (h, r)$$

State is Anomalous

<u>On the one hand...</u>

<u>On the other hand...</u>

State is Anomalous

On the one hand...

States are ‘inputs’ that influence program behavior

On the other hand...

State is Anomalous

On the one hand...

States are ‘inputs’ that influence program behavior

On the other hand...

States are ‘outputs’ that only the program creates

State is Anomalous

On the one hand...

States are ‘inputs’ that influence program behavior

On the other hand...

States are ‘outputs’ that only the program creates

(bottom line)

A state variable is not independent
– sample at your own risk!

Testing Viewpoint

Stateless case:

Black-box program $P : D \rightarrow R$.
Specification function $F : D \rightarrow R$.
Test point $x \in D$ fails if $P(x) \neq F(x)$.
Operational profile: Usage P.d.f. on D.

Testing Viewpoint

<u>Stateless case</u>:

Black-box program $P : D \to R$.
Specification function $F : D \to R$.
Test point $x \in D$ fails if $P(x) \neq F(x)$.
Operational profile: Usage P.d.f. on D.

<u>Persistent state</u>:

Replace D by D sequences $D^\infty = \bigcup_{k=1}^{\infty} D^k$.

$P, F : D^\infty \to R$. (*Sequence* profile)

Testing Viewpoint

Stateless case:

Black-box program $P : D \to R$.
Specification function $F : D \to R$.
Test point $x \in D$ fails if $P(x) \neq F(x)$.
Operational profile: Usage P.d.f. on D.

Persistent state:

Replace D by D sequences $D^\infty = \bigcup_{k=1}^{\infty} D^k$.

$P, F : D^\infty \to R$. (*Sequence* profile)

State is only implicit — tester may sample H...(?)

Proving Viewpoint

Specification is a first-order formula in values of program variables $d \in D, h \in H, r \in R$.

Type, Symbol	*Evaluation*	*Variables* (v' original)
Pre-cond B	before	d
Post-cond C	after	d', h', h, r
Assertion A	any	d', h', h, r
Invariant I	before/after	d, h

Proving Viewpoint

Specification is a first-order formula in values of program variables $d \in D, h \in H, r \in R$.

Type, Symbol	*Evaluation*	*Variables* (v' original)
Pre-cond B	before	d
Post-cond C	after	d', h', h, r
Assertion A	any	d', h', h, r
Invariant I	before/after	d, h

State variable h is explicit – specification is state-prescriptive...(?)

Invariants in Proofs

Room for confusion –
First-order formulas include implicit evaluation times; Hoare logic hides quantification.

For example, correctness of program P:

$$\forall d', d, h', h[B(d) \Rightarrow C(d', h', h, r)]$$
$$B\{P\}C$$

Invariants in Proofs

Room for confusion –
First-order formulas include implicit evaluation times; Hoare logic hides quantification.

For example, correctness of program P:

$$\forall d', d, h', h[B(d) \Rightarrow C(d', h', h, r)]$$

$$B\{P\}C$$

Invariant role filter out P-impossible states.

Pre-condition role filter out inputs humans agree not to use.

$$\forall d, h[I(d, h) \Rightarrow [B(d) \Rightarrow C(d', h', h, r)]]$$

Testing with Invariants

Stateless testing of P to approximate proof:

Sample D, and for each d such that $B(d)$, run P and check $C(d, r)$. (TestEra)

Testing with Invariants

Stateless testing of P to approximate proof:

Sample D, and for each d such that $B(d)$, run P and check $C(d, r)$. (TestEra)

With state it's more complicated.

First try: Sample $D \times H$. For each (d, h) such that $I(d, h) \wedge B(d)$, run P and check $C(d', h', h, r)$.

Testing with Invariants

Stateless testing of P to approximate proof:

Sample D, and for each d such that $B(d)$, run P and check $C(d, r)$. (TestEra)

With state it's more complicated.

First try: Sample $D \times H$. For each (d, h) such that $I(d, h) \wedge B(d)$, run P and check $C(d', h', h, r)$.

Better: Sample D^{∞}, say $d_0, d_1, ..., d_n$, such that $\forall i \in [0, n], B(d_i)$. Sample $h_0 \in H$.
If $I(d_0, h_0)$, run P on the sequence, obtaining state sequence $h_1, h_2, ..., h_n$ and check $I(d_n, h_n) \wedge C(d_n, h_{n-1}, h_n, r)$.

Proof-, Testing-like Formulas

Let R be a logical formula (invariant, post-condition, etc.) applied to a program.

Proof-, Testing-like Formulas

Let R be a logical formula (invariant, post-condition, etc.) applied to a program.

R *is Proof-like*: No test case can falsify R.

R *is Testing-like*: There is a low probability that test-case sequences drawn according to a given operational profile will falsify R.

Since profiles are arbitrary human specifications, proof-like and testing-like can be very different.

Proof-, Testing-like Formulas

Let R be a logical formula (invariant, post-condition, etc.) applied to a program.

R *is Proof-like*: No test case can falsify R.

R *is Testing-like*: There is a low probability that test-case sequences drawn according to a given operational profile will falsify R.

Since profiles are arbitrary human specifications, proof-like and testing-like can be very different.

R itself can be proof- or testing-like if it is obtained using all possibilities, or only those from a profile.

Daikon, TestEra, Etc.

`Daikon`	`TestEra`
Generates possible pre- and post-conditions from given testset.	Generates bounded exhaustive testset (BET) from given pre-condition; checks given post-condition.

Daikon, TestEra, Etc.

`Daikon`	`TestEra`
Generates possible pre- and post-conditions from given testset.	Generates bounded exhaustive testset (BET) from given pre-condition; checks given post-condition.

$\Downarrow$

+invariant +profile

Daikon, TestEra, Etc.

Daikon	TestEra
Generates possible pre- and post-conditions from given testset.	Generates bounded exhaustive testset (BET) from given pre-condition; checks given post-condition.

$\Downarrow$

+invariant +profile

$\searrow$ $\swarrow$

From invariant and profile, generate BET;
check invariant as post-condition.
Use BET to generate possible post-condition.

Summary

- Testing needs to recognize state and invariants
  - Sample state with care!
  - Drive sampling with invariants

Summary

- Testing needs to recognize state and invariants
  - Sample state with care!
  - Drive sampling with invariants
- Invariants are inherently prescriptive

Summary

- Testing needs to recognize state and invariants
  - Sample state with care!
  - Drive sampling with invariants
- Invariants are inherently prescriptive
- Operational profiles define ‘usage invariants’

Summary

- Testing needs to recognize state and invariants
  - Sample state with care!
  - Drive sampling with invariants
- Invariants are inherently prescriptive
- Operational profiles define 'usage invariants'
- Tools using first-order formulas with tests need specification-based invariants

Summary

- Testing needs to recognize state and invariants
  - Sample state with care!
  - Drive sampling with invariants
- Invariants are inherently prescriptive
- Operational profiles define ‘usage invariants’
- Tools using first-order formulas with tests need specification-based invariants