Procedure specifications

CSE 331
University of Washington

Michael Ernst

Outline

- Satisfying a specification; substitutability
- Stronger and weaker specifications
  - Comparing by hand
  - Comparing via logical formulas
  - Comparing via transition relations
    - Full transition relations
    - Abbreviated transition relations
- Specification style; checking preconditions

Satisfaction of a specification

- Let P be an implementation and S a specification
- *P satisfies S* iff
  - Every behavior of P is permitted by S
  - “The behavior of P is a subset of S”
- The statement “P is correct” is meaningless
  - Though often made!
- If P does not satisfy S, either (or both!) could be “wrong”
  - *“One person’s feature is another person’s bug.”*
  - It’s usually better to change the program than the spec

Why compare specifications?

We wish to compare procedures to specifications

- Does the procedure satisfy the specification?
- Has the implementer succeeded?

We wish to compare specifications to one another

- Which specification (if either) is stronger?
- A procedure satisfying a stronger specification can be used anywhere that a weaker specification is required

A specification denotes a set of procedures

- Some set of procedures satisfies a specification
- Suppose a procedure takes an integer as an argument
  - Spec 1: "returns an integer ≥ its argument"
  - Spec 2: "returns a non-negative integer ≥ its argument"
  - Spec 3: "returns argument + 1"
  - Spec 4: "returns argument^2"
  - Spec 5: "returns Integer.MAX_VALUE"
- Consider these implementations:
  - Code 1: `return arg * 2;`
  - Code 2: `return abs(arg);`
  - Code 3: `return arg + 5;`
  - Code 4: `return arg * arg;`
  - Code 5: `return Integer.MAX_VALUE;`

Spec1	Spec2	Spec3	Spec4	Spec5

Specification strength and substitutability

- A stronger specification promises more
  - It constrains the implementation more
  - The client can make more assumptions
- Substitutability
  - A stronger specification can always be substituted for a weaker one

Procedure specifications

Example of a procedure specification:

```
// requires i > 0
// modifies nothing
// returns true iff i is a prime number
public static boolean isPrime (int i)
```

General form of a procedure specification:

```
// requires
// modifies
// throws
// effects
// returns
```

How to compare specifications

Three ways to compare

1. By hand; examine each clause
2. Logical formulas representing the specification
3. Transition relations
   a) Full transition relations
   b) Abbreviated transition relations

Use whichever is most convenient

Technique 1: Comparing by hand

We can weaken a specification by

- Making requires harder to satisfy (strengthening requires)
  - Preconditions are *contravariant* (other clauses are *covariant*)
- Adding things to modifies clause (weakening modifies)
- Making effects easier to satisfy (weakening effects)
- Guaranteeing less about throws (weakening throws)
- Guaranteeing less about returns value (weakening returns)

The strongest (most constraining) spec has the following:

- requires clause: true
- modifies clause: nothing
- effects clause: false
- throws clause: nothing
- returns clause: false
- (This particular spec is so strong as to be useless.)

Technique 2: Comparing logical formulas

Specification S1 is stronger than S2 iff:

$\forall$ P, (P satisfies S1) $\Rightarrow$ (P satisfies S2)

If each specification is a logical formula, this is equivalent to:

S1 $\Rightarrow$ S2

So, convert each spec to a formula (in 2 steps, see following slides)

This specification:

// requires *R*

// modifies *M*

// effects *E*

is equivalent to this single logical formula:

$R \Rightarrow (E \wedge$ (nothing but M is modified))

What about throws and returns? Absorb them into effects.

Final result: S1 is stronger than S2 iff

$(R_1 \Rightarrow (E_1 \wedge \text{only-modifies-}M_1)) \Rightarrow (R_2 \Rightarrow (E_2 \wedge \text{only-modifies-}M_2))$

Convert spec to formula, step 1: absorb throws, returns

CSE 331 style:

- requires (unchanged)
- modifies (unchanged)
- throws
- effects
- returns

correspond to resulting "effects"

Example (from `java.util.ArrayList<T>`):

// requires: true
// modifies: this[index]
// throws: IndexOutOfBoundsException if index < 0 || index ≥ size()
// effects: $this_{post}$[index] = element
// returns: $this_{pre}$[index]
`T set(int index, T element)`

Equivalent spec, after absorbing throws and returns into effects:

// requires: true
// modifies: this[index]
// effects: if index < 0 || index ≥ size() then throws IndexOutOfBoundsException
//   else $this_{post}$[index] = element && returns $this_{pre}$[index]
`T set(int index, T element)`

Convert spec to formula, step 2: eliminate <u>requires</u>, <u>modifies</u>

Single logical formula

requires $\Rightarrow$ (effects $\wedge$ (*not-modified*))

“not-modified” preserves every field not in the <u>modifies</u> clause

Logical fact: If precondition is false, formula is true

Recall: $\forall x.\ x \Rightarrow \text{true}$; $\forall x.\ \text{false} \Rightarrow x$; $(x \Rightarrow y) \equiv (\neg x \vee y)$

Example:

// requires: true

// modifies: this[index]

// effects: *E*

```
T set(int index, T element)
```

Result:

true $\Rightarrow$ ($E \wedge$ ($\forall$i≠index. $\text{this}_{\text{pre}}[\text{i}] = \text{this}_{\text{post}}[\text{i}]$))

Technique 3: Comparing transition relations

Transition relation relates prestates to poststates

Contains all possible ⟨input,output⟩ pairs

Transition relation maps procedure arguments to results

```
int increment(int i) {
  return i+1;
}

double mySqrt(double a) {
  if (Random.nextBoolean())
    return Math.sqrt(a);
  else
    return - Math.sqrt(a);
}
```

A specification has a transition relation, too

Contains just as much information as other forms of specification

Satisfaction via transition relations

A stronger specification has a smaller transition relation

Rule: P satisfies S iff P is a subset of S

(when both are viewed as transition relations)

sqrt specification (S_{sqrt})

```
// requires x is a perfect square
// returns positive or negative square root
int sqrt (int x)
```

Transition relation: ⟨0,0⟩, ⟨1,1⟩, ⟨1,-1⟩, ⟨4,2⟩, ⟨4,-2⟩, …

sqrt code (P_{sqrt})

```
int sqrt (int x) {
    // … always returns positive square root
}
```

Transition relation: ⟨0,0⟩, ⟨1,1⟩, ⟨4,2⟩, …

P_{sqrt} satisfies S_{sqrt} because P_{sqrt} is a subset of S_{sqrt}

Beware transition relations in abbreviated form

"P satisfies S iff P is a subset of S" is a good rule

But it gives the wrong answer for transition relations in abbreviated form

(The transition relations we have seen so far are in abbreviated form!)

anyOdd specification (S_{anyOdd})

```
// requires x = 0
// returns any odd integer
int anyOdd (int x)
```

Abbreviated transition relation: $\langle 0,1\rangle$, $\langle 0,3\rangle$, $\langle 0,5\rangle$, $\langle 0,7\rangle$, …

anyOdd code (P_{anyOdd})

```
int anyOdd (int x) {
    return 3;
}
```

Transition relation: $\langle 0,3\rangle$, $\langle 1,3\rangle$, $\langle 2,3\rangle$, $\langle 3,3\rangle$, …

The code satisfies the specification, but the rule says it does not

P_{anyOdd} is not a subset of S_{anyOdd}

because $\langle 1,3\rangle$ is not in the specification's transition relation

We will see two solutions to this problem: full or abbreviated transition relations

Satisfaction via full transition relations (option 1)

The transition relation should make explicit everything an implementation may do

Problem: abbreviated transition relation for S does not indicate all possibilities

anyOdd specification (S_{anyOdd}): // same as before

```
// requires x = 0
// returns any odd integer
int anyOdd (int x)
```

Full transition relation: ⟨0,1⟩, ⟨0,3⟩, ⟨0,5⟩, ⟨0,7⟩, … // on previous slide

⟨1, 0⟩, ⟨1, 1⟩, ⟨1, 2⟩, …, ⟨1, exception⟩, ⟨1, infinite loop⟩, … // new

⟨2, 0⟩, ⟨2, 1⟩, ⟨2, 2⟩, …, ⟨2, exception⟩, ⟨2, infinite loop⟩, … // new

anyOdd code (P_{anyOdd}): // same as before

```
int anyOdd (int x) {
    return 3;
}
```

Transition relation: ⟨0,3⟩, ⟨1,3⟩, ⟨2,3⟩, ⟨3,3⟩, … // same as before

The rule "P satisfies S iff P is a subset of S" gives the right answer for full relations

Downside: writing the full transition relation is bulky and inconvenient

It's more convenient to make the implicit notational assumption:

For elements not in the domain of S, any behavior is permitted.

(Recall that a relation maps a *domain* to a *range*.)

Satisfaction via abbreviated transition relations (option 2)

New rule: P satisfies S iff P | (Domain of S) is a subset of S

where "P | D" = "P restricted to the domain D"

i.e., remove from P all pairs whose first member is not in D

(recall that a relation maps a *domain* to a *range*)

anyOdd specification (S_{anyOdd})

// <u>requires</u> x = 0

// <u>returns</u> any odd integer

```
int anyOdd (int x)
```

Abbreviated transition relation: ⟨0,1⟩, ⟨0,3⟩, ⟨0,5⟩, ⟨0,7⟩, …

anyOdd code (P_{anyOdd})

```
int anyOdd (int x) {
    return 3;
}
```

Transition relation: ⟨0,3⟩, ⟨1,3⟩, ⟨2,3⟩, ⟨3,3⟩, …

Domain of S = { 0 }

P | (domain of S) = ⟨0,3⟩, which is a subset of S, so P satisfies S

The new rule gives the right answer even for abbreviated transition relations

We'll use this version of the notation in CSE 331

Abbreviated transition relations, summary

True transition relation:

Contains all the pairs, all comparisons work

Bulky to read and write

Abbreviated transition relation

Shorter and more convenient

Naively doing comparisons leads to wrong result

How to do comparisons:

- Use the expanded transition relation, or
- Restrict the domain when comparing

Either approach makes the “smaller is stronger rule” work

Review: strength of a specification

A stronger specification is satisfied by fewer procedures

A stronger specification has

- *weaker* preconditions (note contravariance)
- stronger postcondition
- fewer modifications

Advantage of this view: can be checked by hand

A stronger specification has a (logically) stronger formula

Advantage of this view: mechanizable in tools

A stronger specification has a smaller transition relation

Advantage of this view: captures intuition of "stronger = smaller" (fewer choices)

Specification style

A procedure has a side effect *or* is called for its value

Bad style to have *both* effects and returns

Exception: return old value, as for **`HashMap.put`**

The point of a specification is to be helpful

Formalism helps, overformalism doesn't

A specification should be

- coherent: not too many cases
- informative: a bad example is **`HashMap.get`**
- strong enough: to do something useful, to make guarantees
- weak enough: to permit (efficient) implementation

Checking preconditions

Checking preconditions

- makes an implementation more robust
- provides better feedback to the client
- avoids silent errors

A quality implementation checks preconditions whenever it is *inexpensive* and *convenient* to do so