

6.170 Lecture 6 Procedure specifications



MIT EECS



Outline

Satisfying a specification; substitutability

Stronger and weaker specifications

Comparing by hand

Comparing via logical formulas

Comparing via transition relations

Specification style; checking preconditions

MIT 6.170

Slide 2



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

MIT 6.170

Slide 3



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

MIT 6.170

Slide 4



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 \geq its argument”

Spec 2: “returns a non-negative integer \geq its argument”

Spec 3: “returns argument + 1”

Spec 4: “returns argument²”

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;`

MIT 6.170

Slide 5



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

MIT 6.170

Slide 6



Comparing specifications and procedures


We wish to compare procedures to specifications
 Determine whether the procedure satisfies the specification
 This indicates whether the implementer has succeeded

We wish to compare specifications to one another
 Determine which specification (if either) is stronger
 A procedure satisfying a stronger specification can be used anywhere that a weaker specification is required

Three ways to compare (use whichever is most convenient)

1. By hand; examine each clause
2. Logical formulas representing the specification
3. Transition relations

MIT 6.170 Slide 7




Comparing by hand (comparison technique 1)

We can weaken a specification by
 Making requires harder to satisfy (**strengthening requires**)
 Preconditions: *contravariant*, all other clauses: *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.)

MIT 6.170 Slide 8



Comparing logical formulas (comparison technique 2)

Specification S1 is stronger than S2 iff:
 $\forall P, (P \text{ satisfies } S1) \Rightarrow (P \text{ satisfies } S2)$

If each specification is a logical formula, this is equivalent to:
 $S1 \Rightarrow S2$


So, convert each spec to a formula (see following slides)
 This specification:

```
// requires R
// modifies M
// effects E
```

 is equivalent to this single logical formula:
 $R \Rightarrow (E \wedge (\text{nothing but } M \text{ is modified}))$
 What about throws and returns? Absorb them into effects.

Final result: S1 is stronger than S2 iff
 $(R_1 \Rightarrow (E_1 \wedge \text{only-modifies-}M_1)) \Rightarrow (R_2 \Rightarrow (E_2 \wedge \text{only-modifies-}M_2))$

MIT 6.170 Slide 9



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

6.170 style:
 requires (unchanged)
 modifies (unchanged)
 throws } correspond to resulting "effects"
 effects }
 returns }


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)
```

MIT 6.170 Slide 10



Convert spec to formula: eliminate requires, modifies


Single logical formula
 requires $\Rightarrow ((\text{not-modified}) \wedge \text{effects})$
 "not-modified" preserves every field not in modifies 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:
 $\text{true} \Rightarrow ((\forall i \neq \text{index}. \text{this_pre}[i] = \text{this_post}[i]) \wedge E)$

MIT 6.170 Slide 11



Transition relations (comparison technique 3)

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);
}
```

Specifications have transition relations, too
 Contains just as much information as other forms of specification

MIT 6.170 Slide 12



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: $\langle 0,0 \rangle, \langle 1,1 \rangle, \langle 1,-1 \rangle, \langle 4,2 \rangle, \langle 4,-2 \rangle, \dots$

Sqrt code (P_{sqrt})
 int sqrt (int x) {
 // ... always returns positive square root
 }
 Transition relation: $\langle 0,0 \rangle, \langle 1,1 \rangle, \langle 4,2 \rangle, \dots$

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

MIT 6.170 Slide 13



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: $\langle 0,1 \rangle, \langle 0,3 \rangle, \langle 0,5 \rangle, \langle 0,7 \rangle, \dots$

anyOdd code (P_{anyOdd})
 int anyOdd (int x) {
 return 3;
 }
 Transition relation: $\langle 0,3 \rangle, \langle 1,3 \rangle, \langle 2,3 \rangle, \langle 3,3 \rangle, \dots$

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

MIT 6.170 Slide 14



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: $\langle 0,1 \rangle, \langle 0,3 \rangle, \langle 0,5 \rangle, \langle 0,7 \rangle, \dots$ // on previous slide
 $\langle 1,0 \rangle, \langle 1,1 \rangle, \langle 1,2 \rangle, \dots, \langle 1, \text{exception} \rangle, \langle 1, \text{infinite loop} \rangle, \dots$ // new
 $\langle 2,0 \rangle, \langle 2,1 \rangle, \langle 2,2 \rangle, \dots, \langle 2, \text{exception} \rangle, \langle 2, \text{infinite loop} \rangle, \dots$ // new

anyOdd code (P_{anyOdd}) // same as before
 int anyOdd (int x) {
 return 3;
 }
 Transition relation: $\langle 0,3 \rangle, \langle 1,3 \rangle, \langle 2,3 \rangle, \langle 3,3 \rangle, \dots$ // 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*.)

MIT 6.170 Slide 15



Satisfaction via abbreviated transition relations (option 2)


New rule: P satisfies S iff $P \mid (\text{Domain of S})$ is a subset of S
 where “ $P \mid D$ ” = “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})
 // requires $x = 0$
 // returns any odd integer
 int anyOdd (int x)
Abbreviated transition relation: $\langle 0,1 \rangle, \langle 0,3 \rangle, \langle 0,5 \rangle, \langle 0,7 \rangle, \dots$

anyOdd code (P_{anyOdd})
 int anyOdd (int x) {
 return 3;
 }
 Transition relation: $\langle 0,3 \rangle, \langle 1,3 \rangle, \langle 2,3 \rangle, \langle 3,3 \rangle, \dots$

Domain of S = { 0 }
 $P \mid (\text{domain of S}) = \langle 0,3 \rangle$, 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 6.170

MIT 6.170 Slide 16




Abbreviated transition relations, summary

The abbreviated version of the transition relation can be misleading
 The true transition relation contains all the pairs

When doing comparisons
 Use the expanded transition relation, or
 Restrict the domain when comparing

Either approach makes the “smaller is stronger rule” work

MIT 6.170 Slide 17



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)

MIT 6.170 Slide 18




Specification style

Typically have only one of effects and returns
A procedure has a side effect or is called for its value
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 (bad example: `HashMap.get`)
strong enough (to do something useful, to make guarantees)
weak enough (to permit (efficient) implementation)

MIT 6.170 Slide 19



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

MIT 6.170 Slide 20