Comparing 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?
  – **Substitutability:**
    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;

How does overflow affect answers?

<table>
<thead>
<tr>
<th>Spec1</th>
<th>Spec2</th>
<th>Spec3</th>
<th>Spec4</th>
<th>Spec5</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Review: Specification strength and substitutability

• A stronger specification promises more
  – It constrains the implementation more
  – The client can make more assumptions
  – *Weaker* preconditions (“contravariance”)  
  – Stronger postconditions

• Substitutability
  – A stronger specification can always be substituted for a weaker one
Procedure specifications

Example of a procedure specification:

```java
// 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:

```java
// requires a logical formula (a Boolean expression)
// modifies a list of (Java) expressions
// throws a list of exceptions, each with a condition
// effects a logical formula (a Boolean expression)
// returns (a condition on) the return value; like “// effects result = ...”
```
How to compare specifications

Three ways to compare

1. By hand; examine each clause
   Advantage: can be checked manually

2. Logical formulas representing the specification
   Advantage: mechanizable in tools

3. Transition relations
   Advantage: captures intuition of “stronger = smaller”
   a. Full transition relations
   b. Abbreviated transition relations

Use whichever is most convenient
Technique 1: Comparing by hand

Idea: compare the specification field-by-field

$S_2$ is **stronger** than $S_1$ if

- $S_2$ **requires** is easier to satisfy (**weaker** requires)
  - Preconditions are *contravariant* (other clauses are *covariant*)
- $S_2$ **modifies** is smaller (**stronger** modifies)
- $S_2$ **effects** is harder to satisfy (**stronger** effects)
- $S_2$ **throws** guarantees more (**stronger** throws)
- $S_2$ **returns** guarantees more (**stronger** returns)

Trivia:

The **strongest** (most constraining) spec has the following:

- **requires** clause: true  (equivalently, “**requires** nothing”)
- **modifies** clause: $\emptyset$  (equivalently, “**modifies** nothing”)
- **effects** clause: false
- **throws** clause: nothing
- **returns** clause:  *(there is no strongest returns clause)*

(This particular spec is so strong as to be useless.)
Technique 2: Comparing logical formulas

Essentially the same as technique 1 (comparing by hand).

Technique 1:
• 5 small comparisons
• Combine them to determine whether $S_2$ is stronger than $S_1$

Technique 2:
• One big comparison

Why do we care? Why should we learn another technique?
• Good for automated tools (you are unlikely to use it manually)
• Gives another perspective
• Helps to explicate rules (explains contravariance)
Technique 2: Comparing logical formulas

Specification S2 is stronger than S1 iff:
\[ \forall \text{implementation } P, \ (P \text{ satisfies } S2) \Rightarrow (P \text{ satisfies } S1) \]
If each specification is a logical formula, this is equivalent to:
\[ S2 \Rightarrow S1 \]
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:
\[ R \Rightarrow (E \land (\text{nothing but } M \text{ is modified})) \]

What about `throws` and `returns`? Absorb them into `effects`.

Final result: S2 is stronger than S1 iff
\[ (R_2 \Rightarrow (E_2 \land \text{only-modifies-M}_2)) \Rightarrow (R_1 \Rightarrow (E_1 \land \text{only-modifies-M}_1)) \]
Convert spec to formula, step 1: absorb **throws** and **returns** into **effects**

CSE 331 style:
- requires (unchanged)
- modifies (unchanged)
- throws
effects  } correspond to resulting "effects"
- returns

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

```java
// 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**:

```java
// 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 **requires, modifies**

Single logical formula

```latex
\text{requires } \Rightarrow (\text{effects } \land (\text{not-modified}))
```

"not-modified" preserves every field not in the **modifies** clause

Logical fact: If precondition is false, formula is true

Recall: For any $x$ and $y$: $x \Rightarrow true; \ false \Rightarrow x; \ (x \Rightarrow y) \equiv (\neg x \lor y)$

Example:

```c
// requires: true
// modifies: this[index]
// effects: E
T set(int index, T element)
```

Result:

```latex
true \Rightarrow (E \land (\forall i \neq \text{index}. \ this_{pre}[i] = this_{post}[i]))
```
Technique 3: Comparing transition relations

Transition relation relates **prestates** to **poststates**
Includes all possible behaviors
Transition relation maps procedure arguments to results

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

// requires: a ≥ 0
```java
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 ($S_{sqrt}$)

```c
int sqrt(int x) {
    // requires x is a perfect square
    // returns positive or negative square root
    return x;
}
```

Transition relation: $\langle 0,0 \rangle, \langle 1,1 \rangle, \langle 1,-1 \rangle, \langle 4,2 \rangle, \langle 4,-2 \rangle, \ldots$

sqrt code ($P_{sqrt}$)

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

Transition relation: $\langle 0,0 \rangle, \langle 1,1 \rangle, \langle 4,2 \rangle, \ldots$

$P_{sqrt}$ 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!)

```c
anyOdd specification (S\textsubscript{anyOdd})

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

**Abbreviated** transition relation: \( \langle 0,1 \rangle, \langle 0,3 \rangle, \langle 0,5 \rangle, \langle 0,7 \rangle, \ldots \)

```c
anyOdd code (P\textsubscript{anyOdd})

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

Transition relation: \( \langle 0,3 \rangle, \langle 1,3 \rangle, \langle 2,3 \rangle, \langle 3,3 \rangle, \ldots \)

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

\( P\textsubscript{anyOdd} \) is not a subset of \( S\textsubscript{anyOdd} \)
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 ($S_{\text{anyOdd}}$):

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

Full transition relation: $\langle 0,1 \rangle, \langle 0,3 \rangle, \langle 0,5 \rangle, \langle 0,7 \rangle, \ldots$

$\langle 1,0 \rangle, \langle 1,1 \rangle, \langle 1,2 \rangle, \ldots, \langle 1,\text{exception} \rangle, \langle 1,\text{infinite loop} \rangle, \ldots$

$\langle 2,0 \rangle, \langle 2,1 \rangle, \langle 2,2 \rangle, \ldots, \langle 2,\text{exception} \rangle, \langle 2,\text{infinite loop} \rangle, \ldots$

anyOdd code ($P_{\text{anyOdd}}$):

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

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

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 \textit{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 \textit{domain} to a \textit{range}.)

\begin{verbatim}
anyOdd specification (S_{anyOdd})
  // requires x = 0
  // returns any odd integer
  int anyOdd(int x)

Abbreviated transition relation: \( \langle 0,1 \rangle, \langle 0,3 \rangle, \langle 0,5 \rangle, \langle 0,7 \rangle, \ldots \)

anyOdd code (P_{anyOdd})
  int anyOdd(int x) {
    return 3;
  }

Transition relation: \( \langle 0,3 \rangle, \langle 1,3 \rangle, \langle 2,3 \rangle, \langle 3,3 \rangle, \ldots \)

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 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” intuition work
Review: ways to compare specifications

A stronger specification is satisfied by fewer implementations
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

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

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`
Should preconditions be checked?

Checking preconditions
  – makes an implementation more robust
  – provides better feedback to the client (fail fast)
  – avoids silent failures, avoids delayed failures

Preconditions are common in “helper” methods/classes
  – In public APIs, no precondition ⇒ handle all possible input
  – Why does binarySearch impose a precondition?

Rule of thumb: Check if it is cheap to do so
  – Example: list must be non-null ⇒ check
  – Example: list must be sorted ⇒ don’t check

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