Softwaretechnik

Design by Contract

Software Engineering
Albert-Ludwigs-University Freiburg

June 29, 2011

Road Map

- Contracts for object-oriented programs
- Contract monitoring
- Program verification
- Automatic program verification

Road Map

- Contracts for object-oriented programs

Recall: Contracts for Procedural Programs

- Goal: Specification of imperative procedures
- Approach: give assertions about the procedure (contract)
  - Precondition
    - must be true on entry
    - ensured by caller of procedure
  - Postcondition
    - must be true on exit
    - ensured by procedure if it terminates
- $\textbf{Precondition}(\mathit{State}) \Rightarrow \textbf{Postcondition}(\textbf{procedure}(\mathit{State}))$
- Notation: $\{\textbf{Precondition}\}$ **procedure** $\{\textbf{Postcondition}\}$
- Assertions stated in first-order predicate logic

An Example

```
class TABLE {
  int capacity;   // size of table
  int count;      // number of elements in table
  T get (String key) {...}
  void insert (T element, String key);
}
```

Insert an element in a table of fixed size

Precondition: table is not full

$$\texttt{count} < \texttt{capacity}$$

Postcondition: new element in table, count updated

$$\begin{aligned} & \texttt{count} \leq \texttt{capacity} \\ \wedge \quad & \texttt{get}(\texttt{key}) = \texttt{element} \\ \wedge \quad & \texttt{count} = \textbf{old}\ \texttt{count} + 1 \end{aligned}$$

Inheritance and Dynamic Binding

- Subclass may override a method definition
- Effect on specification:
  - Subclass may have different invariant
  - Redefined methods may
    - have different pre- and postconditions
    - raise different exceptions

    $\Rightarrow$ method specialization
- Relation to invariant and pre-, postconditions in base class?
- Main guideline: No surprises requirement (Wing, FMOODS 1997)
  Properties that users rely on to hold of an object of type T should hold even if the object is actually a member of a subtype S of T.

Invariant of a Subclass

```
class MYTABLE extends TABLE ...
```

- each property expected of a TABLE object should also be granted by a MYTABLE object
- if o has type MYTABLE then $\mathit{INV}_{\texttt{TABLE}}$ must hold for o

$\Rightarrow$ $\mathit{INV}_{\texttt{MYTABLE}} \Rightarrow \mathit{INV}_{\texttt{TABLE}}$

- Example: MYTABLE might be a hash table with invariant

$$\mathit{INV}_{\texttt{MYTABLE}} \equiv \texttt{count} \leq \texttt{capacity}/3$$

Method Specialization

If `MYTABLE` redefines `insert` then ...

- the new precondition must be weaker and
- the new postcondition must be stronger

because in

```
TABLE cast = new MYTABLE (150);
...
cast.insert (new Terminator (3), "Arnie");
```

the caller

- guarantees only $\mathbf{Pre}_{\texttt{insert},\texttt{TABLE}}$
- expects $\mathbf{Post}_{\texttt{insert},\texttt{TABLE}}$

Requirements for Method Specialization

Suppose class T defines method m with assertions $\mathbf{Pre}_{T,m}$ and $\mathbf{Post}_{T,m}$ throwing exceptions $\mathbf{Exc}_{T,m}$. If class S extends class T and redefines m then the redefinition is a sound method specialization if

- $\mathbf{Pre}_{T,m} \Rightarrow \mathbf{Pre}_{S,m}$ and
- $\mathbf{Post}_{S,m} \Rightarrow \mathbf{Post}_{T,m}$ and
- $\mathbf{Exc}_{S,m} \subseteq \mathbf{Exc}_{T,m}$
  each exception thrown by $S.m$ may also be thrown by $T.m$

Example: `MYTABLE.insert`

- $\textbf{Pre}_{\texttt{MYTABLE,insert}} \equiv \texttt{count} < \texttt{capacity}/3$
  not a sound method specialization because it is not implied by $\texttt{count} < \texttt{capacity}$.
- `MYTABLE` may automatically resize the table, so that $\textbf{Pre}_{\texttt{MYTABLE,insert}} \equiv \textbf{true}$ a sound method specialization because $\texttt{count} < \texttt{capacity} \Rightarrow \textbf{true}$!
- Suppose `MYTABLE` adds a new instance variable `T lastInserted` that holds the last value inserted into the table.

$$
\begin{aligned}
\textbf{Post}_{\texttt{MYTABLE,insert}} \equiv & \quad \texttt{get(key)} = \texttt{element} \\
& \wedge \quad \texttt{count} = \textbf{old } \texttt{count} + 1 \\
& \wedge \quad \texttt{lastInserted} = \texttt{element}
\end{aligned}
$$

is sound method specialization because $\textbf{Post}_{\texttt{MYTABLE,insert}} \Rightarrow \textbf{Post}_{\texttt{TABLE,insert}}$

Road Map

- Contracts for object-oriented programs
- Contract monitoring
- Program verification
- Automatic program verification

Road Map

- Contract monitoring

Contract Monitoring

- What happens if a system's execution violates an assertion at run time?
- A violating execution runs outside the system's specification.
- The system's reaction may be arbitrary
  - crash
  - continue
  - contract monitoring: evaluate assertions at runtime and raise an exception indicating any violation
- Why monitor?
  - Debugging (with different levels of monitoring)
  - Software fault tolerance (e.g., α and β releases)

What can go wrong?

precondition: evaluate assertion on entry
identifies problem in the caller

postcondition: evaluate assertion on exit
identifies problem in the callee

invariant: evaluate assertion on entry and exit
problem in the callee's class

hierarchy: unsound method specialization
need to check (for all superclasses T of S)

- $\mathbf{Pre}_{T,m} \Rightarrow \mathbf{Pre}_{S,m}$ on entry and
- $\mathbf{Post}_{S,m} \Rightarrow \mathbf{Post}_{T,m}$ on exit

how?

Hierarchy Checking

Suppose `class` S `extends` T and overrides a method m.
Let $T\ x = \texttt{new}\ S()$ and consider $x.m()$

- on entry
  - if $\mathbf{Pre}_{T,m}$ holds, then $\mathbf{Pre}_{S,m}$ must hold, too
  - $\mathbf{Pre}_{S,m}$ must hold
- on exit
  - $\mathbf{Post}_{S,m}$ must hold
  - if $\mathbf{Post}_{S,m}$ holds, then $\mathbf{Post}_{T,m}$ must hold, too
- in general: cascade of implications between S and T

Examples

```
interface IConsole {
  int getMaxSize();
    @post { getMaxSize > 0 }
  void display (String s);
    @pre { s.length () < this.getMaxSize() }
}

class Console implements IConsole {
  int getMaxSize () { ... }
    @post { getMaxSize > 0 }
  void display (String s) { ... }
    @pre { s.length () < this.getMaxSize() }
```

A Good Extension

```
class RunningConsole extends Console {
  void display (String s) {
    ...
    super.display
        (String. substring (s, ..., ... + getMaxSize()))
    ...
  }
    @pre { true }
}
```

A Bad Extension

```
class PrefixedConsole extends Console {
  String getPrefix() {
    return ">> ";
  }
  void display (String s) {
    super.display (this.getPrefix() + s);
  }
    @pre { s.length() <
        this.getMaxSize() - this.getPrefix().length() }
}
```

- caller may only guarantee `IConsole`'s precondition
- blame the programmer of `PrefixedConsole`!

Properties of Monitoring

- Assertions can be arbitrary side effect-free boolean expressions
- Monitoring can only prove the presence of violations, not their absence
- Absence of violations can only be guaranteed by formal verification

Road Map

- Contracts for object-oriented programs
- Contract monitoring
- Program verification
- Automatic program verification

Road Map

- Program verification

Verification of Contracts

- Given: Specification of imperative **procedure** by **Precondition** and **Postcondition**
- Goal: Formal proof for
  Precondition(*State*) $\Rightarrow$ **Postcondition**(**procedure**(*State*))
- Method: Hoare Logic, i.e., a proof system for Hoare triples of the form

  $$\{\textbf{Precondition}\}\ \textbf{procedure}\ \{\textbf{Postcondition}\}$$

- named after C.A.R. Hoare, the inventor of Quicksort, CSP, and many other
- here: method bodies, no recursion, no pointers (extensions exist)

Syntax

$$
\begin{array}{lcll}
E, F & ::= & c \mid x \mid E + F \mid \ldots & \text{expressions} \\
B, P, Q & ::= & \neg B \mid P \wedge Q \mid P \vee Q & \text{boolean expressions} \\
 & \mid & E = F \mid E \leq F \mid \ldots & \\
C, D & ::= & \texttt{skip} & \text{statements} \\
 & \mid & x{=}E & \text{assignment} \\
 & \mid & C;D & \text{sequence} \\
 & \mid & \texttt{if } B \texttt{ then } C \texttt{ else } D & \text{conditional} \\
 & \mid & \texttt{while } B \texttt{ do } C & \text{iteration} \\
 & & & \\
\mathcal{H} & ::= & \{P\}\ C\ \{Q\} & \text{Hoare triples}
\end{array}
$$

- (boolean) expressions are free of side effects

Semantics — Domains and Types

$$
\begin{array}{lcl}
\mathit{BValue} & = & \texttt{true} \mid \texttt{false} \\
\mathit{IValue} & = & 0 \mid 1 \mid \ldots \\
\sigma \in \mathit{State} & = & \mathit{Variable} \to \mathit{Value}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}[\![]\!] & : & \mathit{Expression} \times \mathit{State} \to \mathit{IValue} \\
\mathcal{B}[\![]\!] & : & \mathit{BoolExpression} \times \mathit{State} \to \mathit{BValue} \\
\mathcal{S}[\![]\!] & : & \mathit{State}_\perp \to \mathit{State}_\perp
\end{array}
$$

- $\mathit{State}_\perp := \mathit{State} \cup \{\perp\}$
- result $\perp$ indicates non-termination

Semantics — Expressions

$$
\begin{array}{lcl}
\mathcal{E}[\![c]\!]\sigma & = & c \\
\mathcal{E}[\![x]\!]\sigma & = & \sigma(x) \\
\mathcal{E}[\![E{+}F]\!]\sigma & = & \mathcal{E}[\![E]\!]\sigma + \mathcal{E}[\![F]\!]\sigma \\
\ldots & & \\
\mathcal{B}[\![E{=}F]\!]\sigma & = & \mathcal{E}[\![E]\!]\sigma = \mathcal{E}[\![F]\!]\sigma \\
\mathcal{B}[\![\neg B]\!]\sigma & = & \neg\mathcal{B}[\![B]\!]\sigma \\
\ldots & &
\end{array}
$$

Semantics — Statements

$$
\begin{array}{rcl}
\mathcal{S}[\![C]\!]\bot & = & \bot \\
\mathcal{S}[\![\texttt{skip}]\!]\sigma & = & \sigma \\
\mathcal{S}[\![x{=}E]\!]\sigma & = & \sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma] \\
\mathcal{S}[\![C;D]\!]\sigma & = & \mathcal{S}[\![D]\!](\mathcal{S}[\![C]\!]\sigma) \\
\mathcal{S}[\![\texttt{if } B \texttt{ then } C \texttt{ else } D]\!]\sigma & = & \mathcal{B}[\![B]\!]\sigma = \texttt{true} \rightarrow \mathcal{S}[\![C]\!]\sigma \ , \ \mathcal{S}[\![D]\!]\sigma \\
\mathcal{S}[\![\texttt{while } B \texttt{ do } C]\!]\sigma & = & F(\sigma) \\
\textit{where} \quad F(\sigma) & = & \mathcal{B}[\![B]\!]\sigma = \texttt{true} \rightarrow F(\mathcal{S}[\![C]\!]\sigma) \ , \ \sigma
\end{array}
$$

Proving a Hoare triple

$$\{P\}\ C\ \{Q\}$$

- holds if $(\forall \sigma \in \mathit{State})\ P(\sigma) \Rightarrow (Q(\mathcal{S}[\![C]\!]\sigma) \vee \mathcal{S}[\![C]\!]\sigma = \bot)$ (partial correctness)
- alternative reading: $P, Q \subseteq \mathit{State}$
  $\{P\}\ C\ \{Q\} \equiv \mathcal{S}[\![C]\!]P \subseteq Q \cup \bot$

Proof Rules for Hoare Triples

- Proving that $\{P\}\ C\ \{Q\}$ holds directly from the definition is tedious
- Instead: define axioms and inferences rules
- Construct a derivation to prove the triple
- Choice of axioms and rules guided by structure of C

Skip Axiom

$$\{P\}\ \texttt{skip}\ \{P\}$$

Assignment Axiom

$$\{P[x \mapsto E]\}\ x = E\ \{P\}$$

Examples:

- $\{1 == 1\}\ x = 1\ \{x == 1\}$
- $\{odd(1)\}\ x = 1\ \{odd(x)\}$
- $\{x == 2 * y + 1\}\ y = 2 * y\ \{x == y + 1\}$

Sequence Rule

$$\frac{\{P\}\ C\ \{R\} \qquad \{R\}\ D\ \{Q\}}{\{P\}\ C;D\ \{Q\}}$$

Example:

$$\frac{\{x == 2*y+1\}\ y = 2*y\ \{x == y+1\} \qquad \{x == y+1\}\ y = y+1\ \{x == y\}}{\{x == 2*y+1\}\ y = 2*y; y = y+1\ \{x == y\}}$$

Conditional Rule

$$\frac{\{P \wedge B\}\ C\ \{Q\} \qquad \{P \wedge \neg B\}\ D\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ D\ \{Q\}}$$

Conditional Rule — Issues

Examples:

$$\frac{\{P \land x < 0\}\ z = -x\ \{z == |x|\} \qquad \{P \land x \geq 0\}\ z = x\ \{z == |x|\}}{\{P\}\ \texttt{if}\ x < 0\ \texttt{then}\ z = -x\ \texttt{else}\ z = x\ \{z == |x|\}}$$

- incomplete!
- precondition for $z = -x$ should be $(z == |x|)[z \mapsto -x] \equiv -x == |x|$

$\Rightarrow$ need logical rules

Logical Rules

- strengthen precondition

$$\frac{P' \Rightarrow P \qquad \{P\}\ C\ \{Q\}}{\{P'\}\ C\ \{Q\}}$$

- weaken postcondition

$$\frac{\{P\}\ C\ \{Q\} \qquad Q \Rightarrow Q'}{\{P\}\ C\ \{Q'\}}$$

Correctness obvious

- Example needs strengthening: $P \wedge x < 0 \Rightarrow -x == |x|$
- holds if $P \equiv$ **true**!
- similarly: $P \wedge x \geq 0 \Rightarrow x == |x|$

Completed example:

$$\mathcal{D}_1 = \frac{x < 0 \Rightarrow -x == |x| \qquad \{-x == |x|\}\ z = -x\ \{z == |x|\}}{\{x < 0\}\ z = -x\ \{z == |x|\}}$$

$$\mathcal{D}_2 = \frac{x \geq 0 \Rightarrow x == |x| \qquad \{x == |x|\}\ z = x\ \{z == |x|\}}{\{x \geq 0\}\ z = x\ \{z == |x|\}}$$

$$\frac{\dfrac{\mathcal{D}_1}{\{x < 0\}\ z = -x\ \{z == |x|\}} \qquad \dfrac{\mathcal{D}_2}{\{x \geq 0\}\ z = x\ \{z == |x|\}}}{\{\textbf{true}\}\ \texttt{if}\ x < 0\ \texttt{then}\ z = -x\ \texttt{else}\ z = x\ \{z == |x|\}}$$

While Rule

$$\frac{\{P \land B\}\ C\ \{P\}}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ C\ \{P \land \neg B\}}$$

- P is loop invariant

Example: try to prove

```
{ a>=0 /\ i==0 /\ k==1 /\ sum==1 }
while sum <= a do
  k = k+2;
  i = i+1;
  sum = sum+k
{ i*i <= a /\ a < (i+1)*(i+1) }
```

$\Rightarrow$ while rule not directly applicable ...

While Rule

Step 1: Find the loop invariant

```
a>=0 /\ i==0 /\ k==1 /\ sum==1
=>
i*i<=a /\ i>=0 /\ k==2*i+1 /\ sum==(i+1)*(i+1)
```

- $P \equiv i * i \leq a \wedge i \geq 0 \wedge k == 2 * i + 1 \wedge sum == (i + 1) * (i + 1)$ holds on entry to the loop
- To prove that P is an invariant, requires to prove that $\{P \wedge sum \leq a\}\ k = k + 2; i = i + 1; sum = sum + k\ \{P\}$
- It follows by the sequence rule and weakening:

Proof of loop invariance

```
{ i*i<=a /\ i>=0   /\ k==2*i+1     /\ sum==(i+1)*(i+1) /\ sum<=a }
{           i>=0   /\ k+2==2+2*i+1 /\ sum==(i+1)*(i+1) /\ sum<=a }
k = k+2
{           i>=0   /\ k==2+2*i+1   /\ sum==(i+1)*(i+1) /\ sum<=a }
{           i+1>=1 /\ k==2*(i+1)+1 /\ sum==(i+1)*(i+1) /\ sum<=a }
i = i+1
{           i>=1   /\ k==2*i+1     /\ sum==i*i         /\ sum<=a }
{ i*i<=a /\ i>=1   /\ k==2*i+1     /\ sum+k==i*i+k     /\ sum+k<=a+k }
sum = sum+k
{ i*i<=a /\ i>=1   /\ k==2*i+1     /\ sum==i*i+k       /\ sum<=a+k }
{ i*i<=a /\ i>=1   /\ k==2*i+1     /\ sum==i*i+2*i+1   /\ sum<=a+k }
{ i*i<=a /\ i>=1   /\ k==2*i+1     /\ sum==(i+1)*(i+1) /\ sum<=a+k }
{ i*i<=a /\ i>=0   /\ k==2*i+1     /\ sum==(i+1)*(i+1) }
```

Step 2: Apply the while rule

$$\frac{\{P \wedge sum \leq a\}\ k = k + 2; i = i + 1; sum = sum + k\ \{P\}}{\{P\}\ \texttt{while}\ sum \leq a\ \texttt{do}\ k = k + 2; i = i + 1; sum = sum + k\ \{P \wedge sum > a\}}$$

Now, $P \wedge sum > a$ is

```
  { i*i<=a /\ i>=0   /\ k==2*i+1   /\ sum==(i+1)*(i+1) /\ sum>a }
implies
  { i*i<=a /\ a<(i+1)*(i+1) }
```

Properties of Formal Verification

- requires more restrictions on assertions (e.g., use a certain logic) than monitoring
- full compliance of code with specification can be guaranteed
- scalability is a challenging research topic:
  - full automatization
  - manageable for small/medium examples
  - large examples require manual interaction