Software Engineering

Lecture 13: Design by Contract

Peter Thiemann

University of Freiburg, Germany

17.06.2013

Table of Contents

Contracts for Procedural Programs

Underlying Idea

Transfer the notion of contract between business partners to software engineering.

What is a contract?

A binding agreement that explicitly states the **obligations** and the **benefits** of each partner.

Example: Contract between Builder and Landowner

	Obligations	Benefits
Landowner	Provide 5 acres of land; pay for building if completed in time	Get building in less than six months
Builder	Build house on provided land in less than six month	No need to do anything if provided land is smaller than 5 acres; Receive payment if house finished in time

Who are the contract partners in SE?

Partners can be modules/procedures, objects/methods, components/operations, ...
In terms of software architecture, the partners are the components and each connector may carry a contract.

Contracts for Procedural Programs

- Goal: Specification of imperative procedures
- Approach: give **assertions** about the procedure
  - 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

Example

Consider the following procedure:

```
/**
 * @param a an integer
 * @return integer square root of a
 */
int root (int a) {
  int i = 0;
  int k = 1;
  int sum = 1;
  while (sum <= a) {
    k = k+2;
    i = i+1;
    sum = sum+k;
  }
  return i;
}
```

Specification of `root`

- types guaranteed by compiler: $\mathtt{a} \in \mathtt{integer}$ and $\mathtt{root} \in \mathtt{integer}$ (the result)

1. `root` as a partial function

   Precondition: $\mathtt{a} \geq 0$

   Postcondition: $\mathtt{root} * \mathtt{root} \leq \mathtt{a} < (\mathtt{root} + 1) * (\mathtt{root} + 1)$

2. `root` as a total function

   Precondition: **true**

   Postcondition:

$$
\begin{array}{l}
(\mathtt{a} \geq 0 \quad \Rightarrow \quad \mathtt{root} * \mathtt{root} \leq \mathtt{a} < (\mathtt{root} + 1) * (\mathtt{root} + 1)) \\
\wedge \\
(\mathtt{a} < 0 \quad \Rightarrow \quad \mathtt{root} = 0)
\end{array}
$$

Weakness and Strength

Goal:

- find weakest precondition
  a precondition that is implied by all other preconditions
  highest demand on procedure
  largest domain of procedure
  (Q: what if precondition = **false**?)
- find strongest postcondition
  a postcondition that implies all other postconditions
  smallest range of procedure
  (Q: what if postcondition = **true**?)

Met by "root as a total function":

- **true** is weakest possible precondition
- "defensive programming"

Example (Weakness and Strength)

Consider `root` as a function over integers

Precondition: **true**

Postcondition:

$$\begin{array}{l} (\mathtt{a} \geq 0 \quad \Rightarrow \quad \mathtt{root} * \mathtt{root} \leq \mathtt{a} < (\mathtt{root} + 1) * (\mathtt{root} + 1)) \\ \wedge \\ (\mathtt{a} < 0 \quad \Rightarrow \quad \mathtt{root} = 0) \end{array}$$

- **true** is the weakest precondition
- The postcondition can be strengthened to

$$\begin{array}{lcl} (\mathtt{root} \geq 0) \ \wedge & & \\ (\mathtt{a} \geq 0 & \Rightarrow & \mathtt{root} * \mathtt{root} \leq \mathtt{a} < (\mathtt{root} + 1) * (\mathtt{root} + 1)) \ \wedge \\ (\mathtt{a} < 0 & \Rightarrow & \mathtt{root} = 0) \end{array}$$

An Example

Insert an element in a table of fixed size

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

Precondition: table is not full

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

Postcondition: new element in table, count updated

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

	Obligations	**Benefits**
Caller	Call `put` only on non-full table	Get modified table in which `element` is associated with `key`
Procedure	Insert `element` in table so that it may be retrieved through `key`	No need to deal with the case where table is full before insertion

Contracts for Object-Oriented Programs

Contracts for Object-Oriented Programs

Contracts for methods have additional features

- local state
  receiving object's state must be specified
- inheritance and dynamic method dispatch
  receiving object's type may be different than statically expected;
  method may be overridden

Local State: Class Invariant

- class invariant *INV* is predicate that holds for all objects of the class

$\Rightarrow$ must be established by all constructors

$\Rightarrow$ must be maintained by all public methods

Pre- and Postconditions for Methods

- constructor methods c

$$\{\mathbf{Pre}_c\}\ c\ \{INV\}$$

- visible methods m

$$\{\mathbf{Pre}_m \land INV\}\ m\ \{\mathbf{Post}_m \land INV\}$$

Table example revisited

- `count` and `capacity` are instance variables of class `TABLE`
- $INV_{\texttt{TABLE}}$ is $\texttt{count} \le \texttt{capacity}$
- specification of `void put (T element, String key)`

  Precondition:

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

  Postcondition:

  $$\texttt{get}(\texttt{key}) = \texttt{element} \wedge \texttt{count} = \textbf{old}\ \texttt{count} + 1$$

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?
- 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

Suppose

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 $INV_{\texttt{TABLE}}$ must hold for o

$\Rightarrow$ $INV_{\texttt{MYTABLE}} \Rightarrow INV_{\texttt{TABLE}}$

- Example: `MYTABLE` might be a hash table with invariant

$$INV_{\texttt{MYTABLE}} \equiv \texttt{count} \leq \texttt{capacity}/3$$

Method Specialization

If `MYTABLE` redefines put then ...

- the **precondition** in the subclass **must be weaker** and
- the **postcondition** in the subclass **must be stronger**

than in the superclass because in

```
TABLE personnel = new MYTABLE (150);
...
personnel.put (new Terminator (3), "Arnie");
```

the caller

- only guarantees $\mathbf{Pre}_{\mathtt{put},\mathtt{Table}}$
- and expects $\mathbf{Post}_{\mathtt{put},\mathtt{Table}}$

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.put

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

$$\mathbf{Post}_{\texttt{MYTABLE},\texttt{put}} \equiv \begin{array}{ll} & \texttt{item}(\texttt{key}) = \texttt{element} \\ \wedge & \texttt{count} = \textbf{old}\ \texttt{count} + 1 \\ \wedge & \texttt{lastInserted} = \texttt{element} \end{array}$$

  is a sound method specialization because
  $\mathbf{Post}_{\texttt{MYTABLE},\texttt{put}} \Rightarrow \mathbf{Post}_{\texttt{TABLE},\texttt{put}}$

Interlude: Method Specialization since Java 5

- Overriding methods in Java 5 only allows specialization of the result type. (It can be replaced by a subtype).
- The parameter types muss stay unchanged (why?)

Example : Assume B **extends** A

```
class Original {
  A m () {
    return new A();
  }
}
class Specialization extends Original {
  B m () { // overrides method Original.m()
    return new B();
  }
}
```

Interlude: NO Specialization

- Method specialization interferes with overloading in Java
- Class `Specialization` has two different methods

Example : Assume B **extends** A

```
class Original {
  void m (B x) {
    return;
  }
}
class Specialization extends Original {
  void m (A x) { // does NOT override method Original.m()
    return;
  }
}
```

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

- 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

- evaluates assertions at run time
- raises an exception indicating any violation
- assign **blame** for the violation

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

- evaluates assertions at run time
- raises an exception indicating any violation
- assign **blame** for the 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 in class S
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
- If the precondition of S is not fulfilled, but the one of T is, then this is a wrong method specialization.
- on exit
  - $\mathbf{Post}_{S,m}$ must hold
  - if $\mathbf{Post}_{S,m}$ holds, then $\mathbf{Post}_{T,m}$ must hold, too
- In general, with more than two classes:
  - Cascade of implications between S and T must be checked.
  - All intermediate pre- and postconditions must be checked.

Examples

```
interface IConsole {
  @post { getMaxSize > 0 }
  int getMaxSize();

  @pre { s.length () < this.getMaxSize() }
  void display (String s);
}

class Console implements IConsole {
  @post { getMaxSize > 0 }
  int getMaxSize () { ... }

  @pre { s.length () < this.getMaxSize() }
  void display (String s) { ... }
```

A Good Extension

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

A Bad Extension

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

- caller may only guarantee `IConsole`'s precondition
- Console.display can be called with argument that is too long
- blame the programmer of `PrefixedConsole`!

Properties of Monitoring

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

Verification of Contracts

Verification of Contracts

- Given: Specification of imperative **procedure** by **Precondition** and **Postcondition**
- Goal: Formal proof for
  Precondition(*State*) $\Rightarrow$ **Postcondition**(**procedure**(*State*))
  if **procedure**(*State*) terminates
- 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, inventor of Quicksort, CSP, and many other
- here: method bodies, no recursion, no pointers (extensions exist)

Syntax of While

A small language to illustrate verification

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

- (boolean) expressions are free of side effects

Proof Rules for Hoare Triples

- 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:

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

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

$\Rightarrow$ need *logical rules*

Logical Rules

- weaken precondition

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

- strengthen postcondition

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

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

Completed example:

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

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

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

While Rule

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

- 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) }
```

Soundness of the Rules

- Intuitively, the proof rules are ok.
- But are they sound?
- Is there a definition from which $\{P\}\ C\ \{Q\}$ can be proved directly?
- Answer: Yes!
- Each rule can be proved correct from this definition.

Semantics — Domains and Types

$$
\begin{array}{lcl}
BValue & = & \texttt{true} \mid \texttt{false} \\
IValue & = & 0 \mid 1 \mid \ldots \\
\sigma \in State & = & Variable \rightarrow Value \\
\\
\mathcal{E}[\![]\!] & : & Expression \times State \rightarrow IValue \\
\mathcal{B}[\![]\!] & : & BoolExpression \times State \rightarrow BValue \\
\mathcal{S}[\![]\!] & : & State_\perp \rightarrow State_\perp
\end{array}
$$

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

Semantics — Expressions

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

Semantics — Statements

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

- McCarthy conditional: $b \rightarrow e_1, e_2$

Proving a Hoare triple

Theorem

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

- 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/notation: $P, Q \subseteq \mathit{State}$
  $\{P\}\ C\ \{Q\} \equiv \mathcal{S}[\![C]\!]P \subseteq Q \cup \bot$
- reading predicates as boolean expressions
  $\mathcal{B}[\![P]\!]\sigma = \mathtt{true} \Rightarrow (\mathcal{B}[\![Q]\!](\mathcal{S}[\![C]\!]\sigma) = \mathtt{true} \vee \mathcal{S}[\![C]\!]\sigma = \bot)$

Proof

By induction on the derivation of $\{P\}\ C\ \{Q\}$:
For each Hoare rule, if the above hypothesis holds for the assumptions, then it holds for the conclusion.

Skip Axiom — Correctness

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

Correctness

- $\mathcal{S}[\![\texttt{skip}]\!]\sigma = \sigma$
- Assume $\mathcal{B}[\![P]\!]\sigma = \texttt{true}$. Then $\mathcal{B}[\![P]\!](\mathcal{S}[\![\texttt{skip}]\!]\sigma) = \mathcal{B}[\![P]\!]\sigma = \texttt{true}$

Assignment Axiom — Correctness

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

- Semantics: $\mathcal{S}[\![x{=}E]\!]\sigma = \sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]$
- Under assumption $\mathcal{B}[\![P[x \mapsto E]]\!]\sigma = \texttt{true}$ show that
  $(\mathcal{B}[\![P]\!](\mathcal{S}[\![x = E]\!]\sigma) = \texttt{true} \lor \mathcal{S}[\![x = E]\!]\sigma = \bot)$
  $\Leftrightarrow (\mathcal{B}[\![P]\!](\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]) = \texttt{true} \lor \mathcal{S}[\![x = E]\!]\sigma = \bot)$
- Requires induction on P:

Assignment Axiom — Correctness II

- Prove $\mathcal{B}[\![P[x \mapsto E]]\!]\sigma = \mathcal{B}[\![P]\!](\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma])$ by induction on P.
- Case $P \equiv \neg Q$:
  $\mathcal{B}[\![\neg Q[x \mapsto E]]\!]\sigma \stackrel{def}{=} \neg\mathcal{B}[\![Q[x \mapsto E]]\!]\sigma \stackrel{IH}{=} \neg\mathcal{B}[\![Q]\!](\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]) \stackrel{def}{=}$
  $\mathcal{B}[\![\neg Q]\!](\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma])$
- Cases $P \equiv Q \wedge Q'$ and $P \equiv Q \vee Q'$ analogously.
- Case $P \equiv E' = E''$:
  $\mathcal{B}[\![(E' = E'')[x \mapsto E]]\!]\sigma \stackrel{def}{=} (\mathcal{E}[\![E'[x \mapsto E]]\!]\sigma = \mathcal{E}[\![E''[x \mapsto E]]\!]\sigma)$
  - Need another lemma:
    $\mathcal{E}[\![E'[x \mapsto E]]\!]\sigma = \mathcal{E}[\![E']\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]$

  $= (\mathcal{E}[\![E']\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma] = \mathcal{E}[\![E'']\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma])$
  $\stackrel{def}{=} \mathcal{E}[\![E' = E'']\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]$
- Case $P \equiv E' \leq E''$ etc: analogously.

Assignment Axiom — Correctness III

Remains to show that $\mathcal{E}[\![E'[x \mapsto E]]\!]\sigma = \mathcal{E}[\![E']\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]$ by induction on E'.

- Case $E' \equiv x$:
  $\mathcal{E}[\![x[x \mapsto E]]\!]\sigma = \mathcal{E}[\![E]\!]\sigma = \mathcal{E}[\![x]\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]$
- Case $E' \equiv y$, $y \neq x$:
  $\mathcal{E}[\![y[x \mapsto E]]\!]\sigma = \mathcal{E}[\![y]\!]\sigma = \sigma(y) = \sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma](y) = \mathcal{E}[\![y]\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]$
- Case $E' \equiv -E''$: Immediate by induction.
  $\mathcal{E}[\![-E''[x \mapsto E]]\!]\sigma \overset{def}{=} -\mathcal{E}[\![E''[x \mapsto E]]\!]\sigma \overset{IH}{=} -\mathcal{E}[\![E'']\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma] \overset{def}{=} \mathcal{E}[\![-E'']\!]\sigma[x \mapsto \mathcal{E}[\![E]\!]\sigma]$
- Case $E' \equiv E'' + E'''$ etc: analogously.

Sequence Rule — Correctness

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

Proof

- Assume $\mathcal{B}[\![P]\!]\sigma = \mathtt{true}$
- Induction on $\{P\}\ C\ \{R\}$ yields
  $\mathcal{B}[\![R]\!](\mathcal{S}[\![C]\!]\sigma) = \mathtt{true} \vee \mathcal{S}[\![C]\!]\sigma = \perp$
- If $\mathcal{S}[\![C]\!]\sigma = \perp$ then the rule is correct because $\mathcal{S}[\![C;D]\!]\sigma = \perp$.
- Otherwise: induction on $\{R\}\ C\ \{Q\}$ yields
  $\mathcal{B}[\![Q]\!](\mathcal{S}[\![D]\!](\mathcal{S}[\![C]\!]\sigma)) = \mathtt{true} \vee \mathcal{S}[\![D]\!](\mathcal{S}[\![C]\!]\sigma) = \perp$
- Recall that $\mathcal{S}[\![D]\!](\mathcal{S}[\![C]\!]\sigma) \overset{def}{=} \mathcal{S}[\![C;D]\!]\sigma$
- If $\mathcal{S}[\![D]\!](\mathcal{S}[\![C]\!]\sigma) = \perp$ then the rule is correct because $\mathcal{S}[\![C;D]\!]\sigma = \perp$.
- Otherwise: $\mathcal{B}[\![Q]\!](\mathcal{S}[\![C;D]\!]\sigma) = \mathtt{true}$ QED

Conditional Rule — Correctness

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

Correctness

- Show: $\sigma \in P$ implies $\mathcal{S}[\![\texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ D]\!] \in Q \cup \{\bot\}$
- Exercise

Logical Rules — Correctness

- weaken precondition

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

- strengthen postcondition

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

Correctness

$P' \Rightarrow P$ iff $P' \subseteq P$ (as set of states)

While-Rule — Correctness

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

- Consider the semantics of while: $\mathcal{S}[\![\texttt{while}\ B\ \texttt{do}\ C]\!]\sigma = F(\sigma)$
  where $F(\bot) = \bot$ and $F(\sigma) = \mathcal{B}[\![B]\!]\sigma = \texttt{true} \to F(\mathcal{S}[\![C]\!]\sigma), \sigma$
- It is sufficient to show (fixpoint induction):
  If $(\forall \sigma \in P)$, $F(\sigma) \in P \land \neg B \lor \{\bot\}$
  then $(\forall \sigma \in P)$, $\mathcal{B}[\![B]\!]\sigma = \texttt{true} \to F(\mathcal{S}[\![C]\!]\sigma), \sigma \in P \land \neg B \lor \{\bot\}$
  - Case $\mathcal{B}[\![B]\!]\sigma = \texttt{true}$:
    By induction on $\{P \land B\}\ C\ \{P\}$,
    either $\mathcal{S}[\![C]\!]\sigma = \bot$ (then $F(\mathcal{S}[\![C]\!]\sigma) = F(\bot) = \bot$ completes the proof),
    or $\mathcal{S}[\![C]\!]\sigma \in P$ (then $F(\mathcal{S}[\![C]\!]\sigma) \in P \land \neg B \lor \{\bot\}$ completes the proof)
  - Case $\mathcal{B}[\![B]\!]\sigma = \texttt{false}$:
    Then $\sigma \in P \land \neg B$. QED

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 and expressivity are challenging research topics:
  - full automatization
  - manageable for small/medium examples
  - large examples require manual interaction
  - real programs use dynamic datastructures (pointers, objects) and concurrency