Table of Contents

Design by Contract

Contracts for Procedural Programs
Contracts for Object-Oriented Programs
Contract Monitoring
Verification of Contracts
Contracts for Procedural Programs
Reminder: 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

<table>
<thead>
<tr>
<th></th>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Landowner</td>
<td>Provide 5 acres of land; pay for building if completed in time</td>
<td>Get building in less than six months</td>
</tr>
<tr>
<td>Builder</td>
<td>Build house on provided land in less than six months</td>
<td>No need to do anything if provided land is smaller than 5 acres; Receive payment if house finished in time</td>
</tr>
</tbody>
</table>
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*

- **Precondition**($State$) $\Rightarrow$ **Postcondition**(procedure($State$))

- **Notation:** \{**Precondition**\} procedure \{**Postcondition**\}

- Assertions stated in first-order predicate logic
Example

Consider the following procedure:

```c
/**
 * @param a an integer
 * @returns 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: $a \in \text{integer}$ and $\text{root} \in \text{integer}$ (the result)

1. root as a partial function
   
   **Precondition:** $a \geq 0$
   
   **Postcondition:** $\text{root} \times \text{root} \leq a < (\text{root} + 1) \times (\text{root} + 1)$

2. root as a total function
   
   **Precondition:** **true**
   
   **Postcondition:**
   
   $$\left(a \geq 0 \Rightarrow \text{root} \times \text{root} \leq a < (\text{root} + 1) \times (\text{root} + 1)\right) \land \left(a < 0 \Rightarrow \text{root} = 0\right)$$
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:**

\[
(a \geq 0 \Rightarrow root \times root \leq a < (root + 1) \times (root + 1)) \\
\land \\
(a < 0 \Rightarrow root = 0)
\]

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

\[
(root \geq 0) \land \\
(a \geq 0 \Rightarrow root \times root \leq a < (root + 1) \times (root + 1)) \land \\
(a < 0 \Rightarrow root = 0)
\]
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

```
count < capacity
```

**Postcondition:** new element in table, count updated

```
count ≤ capacity
\∧  get(key) = element
\∧  count = old count + 1
```
<table>
<thead>
<tr>
<th></th>
<th><strong>Obligations</strong></th>
<th><strong>Benefits</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Caller</strong></td>
<td>Call put only on non-full table</td>
<td>Get modified table in which element is associated with key</td>
</tr>
<tr>
<td><strong>Procedure</strong></td>
<td>Insert element in table so that it may be retrieved through key</td>
<td>No need to deal with the case where table is full before insertion</td>
</tr>
</tbody>
</table>
Contracts for Object-Oriented Programs
Contracts for Object-Oriented Programs

Contracts for methods have additional complications

- 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 $\Rightarrow$ Class Invariant

- class invariant $INV$ is predicate that holds for all objects of the class
- must be established by all constructors
- must be maintained by all visible methods
Pre- and Postconditions for Methods

- constructor methods $c$

\[ \{ \text{Pre}_c \} \ c \ \{ \text{INV} \} \]

- visible methods $m$

\[ \{ \text{Pre}_m \land \text{INV} \} \ m \ \{ \text{Post}_m \land \text{INV} \} \]
Table example revisited

- count and capacity are instance variables of class TABLE
- $INV_{TABLE}$ is $count \leq capacity$
- specification of void put (T element, String key)
  
  Precondition:
  
  \[ count < capacity \]

  Postcondition:

  \[ get(key) = element \land count = old\ count + 1 \]
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 $INV_{\text{TABLE}}$ must hold for o

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

- Example: MYTABLE might be a hash table with invariant

$$INV_{\text{MYTABLE}} \equiv \text{count} \leq \text{capacity}/3$$
Method Specialization

If MYTABLE redefines put then ... 

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

because in

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

the caller

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

- $\text{Pre}_{T,m} \Rightarrow \text{Pre}_{S,m}$ and
- $\text{Post}_{S,m} \Rightarrow \text{Post}_{T,m}$ and
- $\text{Exc}_{S,m} \subseteq \text{Exc}_{T,m}$

each exception thrown by $S.m$ may also be thrown by $T.m$
Example: MYTABLE.put

- $\textbf{Pre}_{\text{MYTABLE}} \equiv \text{count} < \text{capacity}/3$
  not a sound method specialization because it is not implied by $\text{count} < \text{capacity}$.

- MYTABLE may automatically resize the table, so that $\textbf{Pre}_{\text{MYTABLE}} \equiv \text{true}$
  a sound method specialization because $\text{count} < \text{capacity} \Rightarrow \text{true}$!

- Suppose MYTABLE adds a new instance variable $T \text{lastInserted}$ that holds the last value inserted into the table.

  $$\begin{align*}
  \textbf{Post}_{\text{MYTABLE}} \equiv & \quad \text{item(key)} = \text{element} \\
  & \quad \text{count} = \text{old count} + 1 \\
  & \quad \text{lastInserted} = \text{element}
  \end{align*}$$

  is sound method specialization because
  $\textbf{Post}_{\text{MYTABLE}} \Rightarrow \textbf{Post}_{\text{TABLE}}$,insert
Interlude: Method Specialization in Java 5

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

Example: Assume A extends B

```java
class C {
    A m () {
        return new A();
    }
}
class D extends C {
    B m () { // overrides method C.m()
        return new B();
    }
}
```
Contract Monitoring
Contract Monitoring

- What happens if a system’s execution violates an assertion at runtime?
- 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., $\alpha$ and $\beta$ 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 $S$)

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

how?
Hierarchy Checking

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

- on entry
  - if $\text{Pre}_{T,m}$ holds, then $\text{Pre}_{S,m}$ must hold, too
  - $\text{Pre}_{S,m}$ must hold

- on exit
  - $\text{Post}_{S,m}$ must hold
  - if $\text{Post}_{S,m}$ holds, then $\text{Post}_{T,m}$ must hold, too

- in general: cascade of implications between $S$ and $T$
- pre- and postcondition only checked for $S$!
- If the precondition of $S$ is not fulfilled, but the one of $T$ is, then this is a wrong method specialization.
Examples

```java
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

```java
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
- Console.display can be called with to long argument
- 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*)
- Method: **Hoare Logic**, *i.e.*, a proof system for **Hoare triples** of the form

\[
\{\text{Precondition}\}\ \text{procedure}\ \{\text{Postcondition}\}
\]

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

\[
E ::= c \mid x \mid E + E \mid \ldots \quad \text{expressions}
\]

\[
B, P, Q ::= \neg B \mid P \land Q \mid P \lor Q \quad \text{boolean expressions}
\]

\[
| E = E \mid E \leq E \mid \ldots
\]

\[
C, D ::= x = E \quad \text{assignment}
\]

\[
| C;D \quad \text{sequence}
\]

\[
| \text{if } B \text{ then } C \text{ else } D \quad \text{conditional}
\]

\[
| \text{while } B \text{ do } C \quad \text{iteration}
\]

\[
\mathcal{H} ::= \{ P \} C \{ Q \} \quad \text{Hoare triples}
\]

- (boolean) expressions are free of side effects
Semantics — Domains and Types

\[ BValue \ = \ \text{true} \mid \text{false} \]
\[ IValue \ = \ 0 \mid 1 \mid \ldots \]
\[ \sigma \in State \ = \ Variable \rightarrow Value \]

\[ \mathcal{E}[] : Expression \times State \rightarrow IValue \]
\[ \mathcal{B}[] : BoolExpression \times State \rightarrow BValue \]
\[ S[] : State_{\bot} \rightarrow State_{\bot} \]

- \( State_{\bot} := State \cup \{\bot\} \)
- \( \text{result } \bot \) indicates non-termination
Semantics — Expressions

\[
\begin{align*}
\mathcal{E}[c] \sigma &= c \\
\mathcal{E}[x] \sigma &= \sigma(x) \\
\mathcal{E}[E + F] \sigma &= \mathcal{E}[E] \sigma + \mathcal{E}[F] \sigma \\
\vdots \\
\mathcal{B}[E = F] \sigma &= \mathcal{E}[E] \sigma = \mathcal{E}[F] \sigma \\
\mathcal{B}[\neg B] \sigma &= \neg \mathcal{B}[B] \sigma \\
\vdots
\end{align*}
\]
Semantics — Statements

\[ S[C] \perp = \perp \]
\[ S[\text{skip}] \sigma = \sigma \]
\[ S[x=E] \sigma = \sigma[x \mapsto E] \]
\[ S[C;D] \sigma = S[D](S[C] \sigma) \]
\[ S[\text{if } B \text{ then } C \text{ else } D] \sigma = B[B] \sigma = \text{true} \rightarrow S[C] \sigma, S[D] \sigma \]
\[ S[\text{while } B \text{ do } C] \sigma = F(\sigma) \]

where \[ F(\sigma) = B[B] \sigma = \text{true} \rightarrow F(S[C] \sigma), \sigma \]

- McCarthy conditional: \( b \rightarrow e_1, e_2 \)
Proving a Hoare triple

\{P\} \mathcal{C} \{Q\}

- holds if (\forall \sigma \in \text{State}) \ P(\sigma) \Rightarrow (Q(S[C]\sigma) \lor S[C]\sigma = \bot)
  (partial correctness)
- alternative reading/notation: \ P, Q \subseteq \text{State}
  \{P\} \mathcal{C} \{Q\} \equiv S[C]P \subseteq Q \cup \bot
- reading predicates as boolean expressions
  \mathcal{B}[P]\sigma = \text{true} \Rightarrow (\mathcal{B}[Q](S[C]\sigma) = \text{true} \lor S[C]\sigma = \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\} \text{skip} \{P\}
Skip Axiom

\{P\} \text{skip} \{P\}

Correctness

- $S[\text{skip}]\sigma = \sigma$
- $S[\text{skip}]P = P$
Assignment Axiom

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

Examples:

- \( \{1 == 1\} \ x = 1 \ \{x == 1\} \)
- \( \{\text{odd}(1)\} \ x = 1 \ \{\text{odd}(x)\} \)
- \( \{x == 2 \ast y + 1\} \ y = 2 \ast y \ \{x == y + 1\} \)
Assignment Axiom — Correctness

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

- Semantics \( S[x=E] \sigma = \sigma[x \mapsto E] \)
- Have to show
  \( B[P[x \mapsto E]] \sigma = \text{true} \Rightarrow \)
  \( (B[P](S[x = E] \sigma) = \text{true} \lor S[x = E] \sigma = \bot) \)
- By induction on \( P \); result of \( B[E' \rho E''] \sigma \) must remain the same; result of \( E'[E'] \sigma \) must remain the same
- Sufficient to show \( E'[x \mapsto E]' \sigma = E'[E'] \sigma[x \mapsto E] \sigma \)
- Holds because \( E[x[x \mapsto E]] \sigma = E[E] \sigma = E[x] \sigma[x \mapsto E] \sigma \)
**Sequence Rule**

\[
\begin{array}{c}
\{P\} \quad C \quad \{R\} \quad \{R\} \quad D \quad \{Q\} \\
\{P\} \quad C;D \quad \{Q\}
\end{array}
\]

Example:

\[
\begin{array}{c}
\{x == 2 * y + 1\} \quad y = 2 * y \quad \{x == y + 1\} \quad \{x == y + 1\} \quad y = y + 1 \quad \{x == y\}
\end{array}
\]

\[
\begin{array}{c}
\{x == 2 * y + 1\} \quad y = 2 * y; y = y + 1 \quad \{x == y\}
\end{array}
\]
Sequence Rule

\[
\begin{array}{c}
\{ P \} C \{ R \} \quad \{ R \} D \{ Q \} \\
\{ P \} C;D \{ Q \}
\end{array}
\]

Example:

\[
\begin{array}{c}
\{ x == 2 * y + 1 \} y = 2 * y \{ x == y + 1 \} \quad \{ x == y + 1 \} y = y + 1 \{ x == y \} \\
\{ x == 2 * y + 1 \} y = 2 * y; y = y + 1 \{ x == y \}
\end{array}
\]

Correctness

- If \( \sigma \in P \) then \( \sigma' = S[C] \sigma \in R \cup \{ \perp \} \)
- If \( \sigma' = \perp \) then \( S[D] \perp = \perp \)
- If \( \sigma' \in R \) then \( S[D] \sigma' \in Q \cup \{ \perp \} \)
- Hence: \( \sigma \in P \Rightarrow S[C;D] \sigma \in Q \cup \{ \perp \} \)
Conditional Rule

\[
\begin{align*}
\{ P \land B \} & \; C \; \{ Q \} & \quad & \{ P \land \neg B \} & \; D \; \{ Q \} \\
\{ P \} & \; \text{if} \; B & \; \text{then} \; C & \; \text{else} \; D & \; \{ Q \}
\end{align*}
\]
Conditional Rule

\[
\begin{align*}
\{ P \land B \} & \quad C \quad \{ Q \} & \quad \{ P \land \neg B \} & \quad D \quad \{ Q \} \\
\{ P \} & \quad \text{if } B \text{ then } C \text{ else } D \quad \{ Q \}
\end{align*}
\]

Correctness

- **Show:** \( \sigma \in P \) implies \( S[\text{if } B \text{ then } C \text{ else } D] \in Q \cup \{ \bot \} \)
- **Exercize**
Conditional Rule — Issues

Examples:

\[
\begin{align*}
\{ P \land x < 0 \} & \quad z = -x \quad \{ z = |x| \} \quad \{ P \land x \geq 0 \} & \quad z = x \quad \{ z = |x| \}
\end{align*}
\]

\[
\{ P \} \quad \text{if } x < 0 \text{ then } z = -x \text{ else } z = x \quad \{ z = |x| \}
\]

▶ incomplete!

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

\( \Rightarrow \) need logical rules
Logical Rules

▶ weaken precondition

\[
P' \implies P \quad \{ P \} \ C \ \{ Q \} \\
\{ P' \} \ C \ \{ Q \}
\]

▶ strengthen postcondition

\[
\{ P \} \ C \ \{ Q \} \quad Q \implies Q' \\
\{ P \} \ C \ \{ Q' \}
\]

▶ Example needs strengthening: \( P \land x < 0 \implies -x = |x| \)

▶ holds if \( P \equiv \text{true} \)!

▶ similarly: \( P \land x \geq 0 \implies x = |x| \)
Logical Rules

▶ weaken precondition

\[
P' \Rightarrow P \\
\{P\} \ C \ \{Q\}
\]

\[
\{P'\} \ C \ \{Q\}
\]

▶ strengthen postcondition

\[
\{P\} \ C \ \{Q\} \\
Q \Rightarrow Q'
\]

\[
\{P\} \ C \ \{Q'\}
\]

▶ Example needs strengthening: \( P \land x < 0 \Rightarrow -x == |x| \)
▶ holds if \( P \equiv \text{true}! \)
▶ similarly: \( P \land x \geq 0 \Rightarrow x == |x| \)

Correctness

\( P' \Rightarrow P \) iff \( P' \subseteq P \) (as set of states)
Completed example:

\[ D_1 = \begin{array}{c} x < 0 \implies -x = |x| \\ \{ x < 0 \} \ z = -x \ {z = |x|} \end{array} \]

\[ D_2 = \begin{array}{c} x \geq 0 \implies x = |x| \\ \{ x \geq 0 \} \ z = x \ {z = |x|} \end{array} \]

\[ \{ x < 0 \} \ z = -x \ {z = |x|} \]
\[ \{ x \geq 0 \} \ z = x \ {z = |x|} \]

\[ \{ \text{true} \} \text{ if } x < 0 \text{ then } z = -x \text{ else } z = x \ {z = |x|} \]
While Rule

\[
\begin{align*}
\{P \land B\} & \ C \ \{P\} \\
\{P\} \ \text{while} \ B \ \text{do} \ C \ \{P \land \neg B\}
\end{align*}
\]

- \( P \) is \textit{loop invariant}

Example: try to prove

\{ a \geq 0 \land i = 0 \land k = 1 \land \text{sum} = 1 \}

while \( \text{sum} \leq a \) do

\begin{align*}
&k = k + 2; \\
i &= i + 1; \\
\text{sum} &= \text{sum} + k \\
\{ i \cdot i \leq a \land a < (i + 1) \cdot (i + 1) \}
\end{align*}

\( \Rightarrow \) while rule not directly applicable …
While Rule

Step 1: Find the loop invariant

\[ a \geq 0 \land i = 0 \land k = 1 \land \text{sum} = 1 \]
\[ \Rightarrow \]
\[ i \times i \leq a \land i \geq 0 \land k = 2 \times i + 1 \land \text{sum} = (i + 1) \times (i + 1) \]

- \[ P \equiv i \times i \leq a \land i \geq 0 \land k = 2 \times i + 1 \land \text{sum} = (i + 1) \times (i + 1) \]
  holds on entry to the loop

- To prove that \( P \) is an invariant, requires to prove that
  \[ \{ P \land \text{sum} \leq a \} \ k = k + 2; i = i + 1; \text{sum} = \text{sum} + k \ \{ P \} \]

- It follows by the sequence rule and weakening:
Proof of loop invariance

\{ i \cdot i \leq a \land i \geq 0 \land k = 2 \cdot i + 1 \land \sum = (i+1) \cdot (i+1) \land \sum \leq a \}
\{ i \geq 0 \land k+2 = 2+2 \cdot i+1 \land \sum = (i+1) \cdot (i+1) \land \sum \leq a \}
k = k+2
\{ i \geq 0 \land k = 2+2 \cdot i+1 \land \sum = (i+1) \cdot (i+1) \land \sum \leq a \}
\{ i+1 \geq 1 \land k = 2 \cdot (i+1) + 1 \land \sum = (i+1) \cdot (i+1) \land \sum \leq a \}
i = i+1
\{ i \geq 1 \land k = 2 \cdot i + 1 \land \sum = i \cdot i \land \sum \leq a \}
\{ i \cdot i \leq a \land i \geq 1 \land k = 2 \cdot i + 1 \land \sum + k = i \cdot i + k \land \sum + k \leq a + k \}
sum = \sum + k
\{ i \cdot i \leq a \land i \geq 1 \land k = 2 \cdot i + 1 \land \sum = i \cdot i + k \land \sum \leq a + k \}
\{ i \cdot i \leq a \land i \geq 1 \land k = 2 \cdot i + 1 \land \sum = i \cdot i + 2 \cdot i + 1 \land \sum \leq a + k \}
\{ i \cdot i \leq a \land i \geq 1 \land k = 2 \cdot i + 1 \land \sum = (i+1) \cdot (i+1) \land \sum \leq a + k \}
\{ i \cdot i \leq a \land i \geq 0 \land k = 2 \cdot i + 1 \land \sum = (i+1) \cdot (i+1) \}
Step 2: Apply the while rule

\[
\{ P \land \text{sum} \leq a \} \quad k = k + 2; \ i = i + 1; \ \text{sum} = \text{sum} + k \quad \{ P \} \\
\{ P \} \ \text{while} \ \text{sum} \leq a \ \text{do} \ k = k + 2; \ i = i + 1; \ \text{sum} = \text{sum} + k \quad \{ P \land \text{sum} \geq a \}
\]

Now, \( P \land \text{sum} > a \) is

\[
\{ i \cdot i \leq a \ \land \ i \geq 0 \ \land \ k = 2 \cdot i + 1 \ \land \ \text{sum} = (i + 1) \cdot (i + 1) \ \land \ \text{sum} > a \}
\]
implies

\[
\{ i \cdot i \leq a \ \land \ a < (i + 1) \cdot (i + 1) \}
\]
Correctness of While-Rule

\[
\{P \land B\} \quad C \quad \{P\}
\]
\[
\{P\} \quad \text{while} \quad B \quad \text{do} \quad C \quad \{P \land \neg B\}
\]

- Consider \(S[\text{while } B \text{ do } C]\sigma = F(\sigma)\)
  where \(F(\sigma) = B[B] \sigma = \text{true} \rightarrow F(S[C] \sigma), \sigma\)
- Case \(\forall n \in \mathbb{N}, B[B](S[C]^{(n)} \sigma) = \text{true}: \text{ set } F(\sigma) = \bot\).
- Case \(\exists n \in \mathbb{N}, B[B](S[C]^{(n)} \sigma) = \text{false}: \text{ let } n_0 \text{ be minimal}\)
- Let \(\sigma \in P = (P \land B) \cup (P \land \neg B)\)
- Case \(n_0 = 0: \sigma \in P \land \neg B, \text{ then } F(\sigma) = \sigma \in P \land \neg B. \text{ OK.}\)
- Case \(n_0 > 0: \sigma \in P \land B, \text{ then } \sigma' = S[C] \sigma \in P \cup \{\bot\} \text{ by assumption. By induction, } F(\sigma') = S[C]^{(n_0-1)} \sigma' \in P \land \neg B \cup \{\bot\}\)
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
  - real programs use arrays and dynamic data structures (pointers, objects)