# Formal Specification and Verification of Object-Oriented Programs

## Loop Invariants

TECHNISCHE UNIVERSITÄT DARMSTADT

## Program Logic Calculus: Recapitulation

TECHNISCHE UNIVERSITÄT DARMSTADT

Calculus realises symbolic interpreter

- work on first active statement
- decompose complex statements into simpler ones
- represent atomic assignments as symbolic updates
- accumulated parallel updates capture changed program state
- control flow branching induces proof splitting
- applying update on FOL postcondition gives weakest precondition

$$\dfrac{\dfrac{\dfrac{\psi' \Rightarrow \{\mathcal{U}'\}\phi \qquad \ldots}{\vdots \qquad \vdots}}{\begin{array}{c}\psi, \{\mathcal{U}\}(\mathtt{isValid} \doteq \mathtt{TRUE}) \Rightarrow \{\mathcal{U}\}\langle\{ok = \mathbf{true};\}\ldots\rangle\phi \\ \psi, !\{\mathcal{U}\}(\mathtt{isValid} \doteq \mathtt{FALSE}) \Rightarrow \{\mathcal{U}\}\langle\ldots\rangle\phi\end{array}}}{\psi \Rightarrow \{\mathtt{t} := \mathtt{j} \| \mathtt{j} := \mathtt{j} + 1 \| \mathtt{i} := \mathtt{j}\}\langle\mathbf{if}(isValid)\{ok = \mathbf{true};\}\ldots\rangle\phi}$$

$$\vdots$$

$$\dfrac{\dfrac{\psi \Rightarrow \{\mathtt{t} := \mathtt{j}\}\langle j = j + 1; i = t; \mathbf{if}(isValid)\{ok = \mathbf{true};\}\ldots\rangle\phi}{\psi \Rightarrow \langle t = j; j = j + 1; i = t; \mathbf{if}(isValid)\{ok = \mathbf{true};\}\ldots\rangle\phi}}{\psi \Rightarrow \langle i = j++; \mathbf{if}(isValid)\{ok = \mathbf{true};\}\ldots\rangle\phi}$$

## Loop Invariants

TECHNISCHE UNIVERSITÄT DARMSTADT

### Symbolic execution of loops: unwind

$$\text{unwindLoop}\ \frac{\Gamma \Rightarrow \mathcal{U}[\pi\ \mathbf{if}(b)\ \{p; \mathbf{while}(b)\ p\}\ \omega]\phi, \Delta}{\Gamma \Rightarrow \mathcal{U}[\pi\ \mathbf{while}(b)\ p\ \omega]\phi, \Delta}$$

How to handle a loop with...

- 0 iterations? Unwind 1 $\times$
- 10 iterations? Unwind 11 $\times$
- 10000 iterations? Unwind 10001 $\times$
  (and don't make any plans for the rest of the day)
- an unknown number of iterations?

We need an invariant rule (or some other form of induction)

# Loop Invariants Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

## Idea behind loop invariants

- A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Desired postcondition after loop *Inv* must be closely related

## Basic Invariant Rule

$$\text{loopInvariant}\quad \frac{\begin{array}{ll} \Gamma \Rightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta & \text{(initially valid)} \\ \mathit{Inv},\ b \doteq \mathtt{TRUE} \Rightarrow [p]\mathit{Inv} & \text{(preserved)} \\ \mathit{Inv},\ b \doteq \mathtt{FALSE} \Rightarrow [\pi\ \omega]\phi & \text{(use case)} \end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\pi\ \mathbf{while}(b)\ p\ \omega]\phi, \Delta}$$

# Why Does the Invariant Rule Work?

TECHNISCHE UNIVERSITÄT DARMSTADT

## Induction Argument

We prove by induction over the number $n$ of loop iterations that *Inv* holds in all loop iterations (used in third premiss)

**Hypothesis** *Inv* holds in the first $n$ loop iterations

**Base Case** *Inv* holds in the first 0 loop iterations
iff *Inv* holds in the state at the start of the loop
iff the first premiss of the invariant rule holds

**Step Case** If *Inv* holds in the first $n$ loop iterations, then *Inv* holds even in the first $n + 1$ loop iterations
follows from: in any[a] state where *Inv* holds and the guard is true *Inv* holds after one more iteration
iff the second premiss of the invariant rule holds

[a]For this reason we cannot use $\Gamma$, $\Delta$ or $\mathcal{U}$ in (preserved) and (use case)

## How to Derive Loop Invariants Systematically?

TECHNISCHE UNIVERSITÄT DARMSTADT

### Example (First active statement of symbolic execution is loop)

```
n >= 0  & wellFormed(heap) ==>
{i := 0} \[{
 while (i < n) {
      i = i + 1;
    }
 }\](i = n)
```

### Look at desired postcondition (`i = n`)

What, in addition to negated guard (`i >= n`), is needed? (`i <= n`)

### Is (`i <= n`) established at beginning and preserved?

Yes! We have found a suitable loop invariant!

Demo `loops/simple.key` (auto after inv)

# Obtaining Invariants by Strengthening

TECHNISCHE UNIVERSITÄT DARMSTADT

## Example (Slightly changed loop)

```
n >= 0 & n = m & wellFormed(heap) ==>
{i := 0}\[{
  while (i < n) {
    i = i + 1;
  }
}\] (i = m)
```

### Look at desired postcondition `(i = m)`

What, in addition to negated guard `(i >= n)`, is needed? `(i = m)`

### Is `(i = m)` established at beginning and preserved? Neither!

Can we use something from the precondition or the update?

- If we know that `(n = m)` then `(i <= n)` suffices
- Strengthen the invariant candidate to: `(i <= n & n = m)`

# Generalization



## Example (Addition: `x`,`y` program variables, `x0`,`y0` rigid constants)

```
x = x0 & y = y0 & y0 >= 0  & wellFormed(heap) ==>
\[{
 while (y > 0) {
  x = x + 1;
  y = y - 1;
} }\] (x = x0 + y0)
```

## Finding the invariant

First attempt: use postcondition `x = x0 + y0`

- Not true at start whenever `y0 > 0`
- Not preserved by loop, because `x` is increased

# Generalization

TECHNISCHE UNIVERSITÄT DARMSTADT

## Example (Addition: `x,y` program variables, `x0,y0` rigid constants)

```
x = x0 & y = y0 & y0 >= 0  & wellFormed(heap) ==>
\[{
 while (y > 0) {
  x = x + 1;
  y = y - 1;
} }\] (x = x0 + y0)
```

## Finding the invariant

What stays invariant?

- The sum of `x` and `y`: `x + y = x0 + y0` “Generalization”
- Can help to think of partial result: “$\delta$” between `x` and `x0 + y0`

# Generalization

## Example (Addition: x,y program variables, x0,y0 rigid constants)

```
x = x0 & y = y0 & y0 >= 0  & wellFormed(heap) ==>
\[{
 while (y > 0) {
  x = x + 1;
  y = y - 1;
} }\] (x = x0 + y0)
```

## Checking the invariant

Is `x + y = x0 + y0` a good invariant?

- Holds in the beginning and is preserved by loop
- But postcondition not achieved by `x + y = x0 + y0 & y <= 0`

## Generalization

TECHNISCHE UNIVERSITÄT DARMSTADT

### Example (Addition: `x`,`y` program variables, `x0`,`y0` rigid constants)

```
x = x0 & y = y0 & y0 >= 0  & wellFormed(heap) ==>
\[{
 while (y > 0) {
  x = x + 1;
  y = y - 1;
} }\] (x = x0 + y0)
```

### Strenghtening the invariant

Postcondition holds if `y = 0`

- Sufficient to add `y >= 0` to `x + y = x0 + y0`

Demo `loops/simple3.key`

# Basic Loop Invariant: Context Loss

TECHNISCHE UNIVERSITÄT DARMSTADT

## Basic Invariant Rule: a Problem

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Rightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta & \text{(initially valid)}\\ \mathit{Inv},\ b \doteq \texttt{TRUE} \Rightarrow [p]\mathit{Inv} & \text{(preserved)}\\ \mathit{Inv},\ b \doteq \texttt{FALSE} \Rightarrow [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\pi\ \mathbf{while}(b)\ p\ \omega]\phi, \Delta}$$

- Context $\Gamma, \Delta, \mathcal{U}$ must be omitted in 2nd and 3rd premise:

  $\Gamma, \Delta$ in general don't hold in state defined by $\mathcal{U}$

  2nd premise *Inv* must be invariant for any state, not only $\mathcal{U}$

  3rd premise We don't know the state after the loop exits
- But: context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant *Inv*

# Example



Precondition: $\mathtt{a} \not\doteq \mathbf{null}\ \&\ \mathit{ClassInv}$

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathtt{int}\ x;\ (0 \leq x\ \&\ x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \ \&\ \ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \ \forall\, \mathbf{int}\ x;\ (0 \leq x\ \&\ x < \mathtt{i} \rightarrow \mathtt{a}[x] \doteq 1)$
$\&\ \ \mathtt{a} \not\doteq \mathbf{null}$
$\&\ \ \mathit{ClassInv}'$

# Keeping the Context (As In Method Contract Rule)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Want to keep part of the context that is unmodified by loop
- **assignable** clauses for loops tell what can possibly be modified

```
@ assignable i, a[*];
```

- How to erase all values of **assignable** locations in formula Γ ?

  Analogous situation: ∀-Right quantifier rule $\Rightarrow \forall\, x;\ \phi$
  Replace $x$ with a fresh constant $*$

  To change value of program location use update
- Anonymising updates $\mathcal{V}$ erase information about modified locations

$\mathcal{V} \ = \ \{\mathtt{i} := \mathtt{c} \parallel \mathtt{heap} := \mathtt{anon}(\mathtt{heap}, \mathtt{allFields}(\mathtt{this.a}), \mathtt{heap}')\}$
($\mathtt{c}, \mathtt{heap}'$ new constant symbols)

# Anonymising JAVA Locations



```
@ assignable i, a[*];
```

To erase all knowledge about the values of the locations of the assignable expression:

- introduce a new (not yet used) constant of type $\mathtt{int}$, e.g., $\mathtt{c}$
- introduce a new (not yet used) constant of type Heap, e.g., $\mathtt{heap}'$
  - anonymise the current heap: $\mathtt{anon}(\mathtt{heap}, \mathtt{allFields}(\mathtt{this.a}), \mathtt{heap}')$
- compute anonymizing update for assignable locations

$$\mathcal{V} = \mathtt{i} := \mathtt{c} \parallel \mathtt{heap} := \mathtt{anon}(\mathtt{heap}, \mathtt{allFields}(\mathtt{this.a}), \mathtt{heap}')$$

For local program variables (e.g., $\mathtt{i}$) KeY computes assignable clause automatically

## Loop Invariants Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

### Improved Invariant Rule

$$\frac{\begin{array}{ll}\Gamma \Rightarrow \{\mathcal{U}\}\mathit{Inv}, \Delta & \text{(initially valid)}\\ \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}\}(\mathit{Inv} \ \&\ b \doteq \mathtt{TRUE} \rightarrow [p]\mathit{Inv}), \Delta & \text{(preserved)}\\ \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}\}(\mathit{Inv} \ \&\ b \doteq \mathtt{FALSE} \rightarrow [\pi\ \omega]\phi), \Delta & \text{(use case)}\end{array}}{\Gamma \Rightarrow \{\mathcal{U}\}[\pi\ \mathbf{while}(b)\ p\ \omega]\phi, \Delta}$$

- Context is kept as far as possible:
  $\{\mathcal{V}$ wipes out only information in locations assignable in loop
- Invariant *Inv* does not need to include unmodified locations
- For `assignable \everything` (the default):
  - $\mathtt{heap} := \mathtt{anon}(\mathtt{heap}, \mathtt{allLocs}, \mathtt{heap}')$ wipes out **all** heap information
  - Equivalent to basic invariant rule
  - Avoid this! Always give a specific **assignable** clause

# Example with Improved Invariant Rule

TECHNISCHE UNIVERSITÄT DARMSTADT

Precondition: $\mathtt{a} \mathrel{\dot{\neq}} \mathbf{null}$ & *ClassInv*

```
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```

Postcondition: $\forall\, \mathtt{int}\ x;\ (0 \leq x\ \&\ x < \mathtt{a.length} \rightarrow \mathtt{a}[x] \doteq 1)$

Loop invariant: $0 \leq \mathtt{i}\ \&\ \mathtt{i} \leq \mathtt{a.length}$
$\&\ \forall\, \mathtt{int}\ x;\ (0 \leq x\ \&\ x < \mathtt{i} \rightarrow \mathtt{a}[x] \doteq 1)$

## Example in JML/JAVA – Loop.java

Demo

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public int[] a;
/*@ public normal_behavior
  @  ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
  @  diverges true;
  @*/
public void m() {
  int i = 0;
  /*@ loop_invariant
    @  0 <= i && i <= a.length &&
    @  (\forall int x; 0<=x && x<i; a[x]==1);
    @ assignable a[*];
    @*/
  while(i < a.length) {
    a[i] = 1;
    i++;
}
```

# Example from A Previous Lecture

TECHNISCHE UNIVERSITÄT DARMSTADT

$\forall$ **int** $x$;
$(x \doteq \mathtt{n} \wedge x >= 0 \rightarrow$
$[$ `i = 0; r = 0;`
**while** `(i<n){ i = i + 1; r = r + i;}`
`r=r+r-n;`
$]\,(\mathtt{r} \doteq x * x)$

How can we prove that the above formula is valid
(i.e., satisfied in all states)?

Solution:

```
@ loop_invariant
@     i>=0 && 2*r == i*(i + 1) && i <= n;
@ assignable \nothing; // no heap locations changed
```

Demo `Loop2.java`

# Hints

TECHNISCHE UNIVERSITÄT DARMSTADT

## Proving `assignable`

- Invariant rule above assumes that `assignable` is correct
  E.g., with `assignable \nothing;` one can prove nonsense
- Invariant rule of KeY generates proof obligation that ensures correctness of `assignable`
  This proof obligation is part of (Body preserves invariant) branch

## Setting in the KeY Prover when proving loops

- Loop treatment: Invariant
- Quantifier treatment: No Splits with Progs
- If program contains `*, /`: Arithmetic treatment: DefOps
- Is search limit high enough (time out, rule apps.)?
- When proving partial correctness, add `diverges true;`

## What is still missing?

TECHNISCHE UNIVERSITÄT DARMSTADT

Is the sequent

$$\Rightarrow [\texttt{i = -1;}\ \textbf{while}\ \texttt{(}\textbf{true}\texttt{)\{\}}]\texttt{i} \doteq 4711$$

provable?

Yes, e.g.,

```
@ loop_invariant true;
@ assignable \nothing;
```

Possible to prove correctness of non-terminating loop

- Invariant trivially initially valid and preserved $\Rightarrow$
  Initial Case and Preserved Case immediately closable
- Loop condition never false: Use case immediately closable

But need a method to prove termination of loops

# Mapping Loop Execution into Well-Founded Order

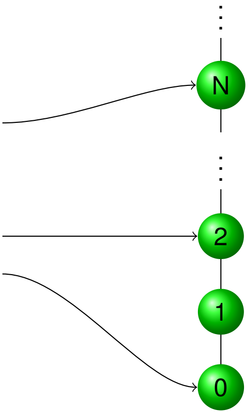
```
while (b) {
  body
}
```

`if (b) { body }`$_1$

⋮

`if (b) { body }`$_{17}$

`if (b) { body }`$_{42}$




**Need to find expression getting smaller wrt $\mathbb{N}$ in each iteration**

Such an expression is called a decreasing term or variant

# Total Correctness: Decreasing Term (Variant)

TECHNISCHE UNIVERSITÄT DARMSTADT

## Find a decreasing integer term $v$ (called variant)

Add the following premises to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- $v$ is strictly decreased by the loop body

## Proving termination in JML/JAVA

- Remove directive `diverges true;` from contract
- Add directive `decreasing v;` to loop invariant
- KeY creates suitable invariant rule and PO (with $\langle ... \rangle \phi$)

### Example (The `array` loop)

```
@ decreasing  a.length - i;
```

Files:

- `LoopT.java`, `Loop2T.java`

# Example: Computing the GCD

TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
public class Gcd {
 /*@ public normal_behavior
   @ requires _small>=0 && _big>=_small;
   @ ensures _big!=0 ==>
   @  (_big % \result == 0 && _small % \result == 0 &&
   @    (\forall int x; x>0 && _big % x == 0
   @       && _small % x == 0; \result % x == 0));
   @ assignable \nothing; @*/
 private static int gcdHelp(int _big, int _small) {
   int big = _big; int small = _small;
   while (small != 0) {
     final int t = big % small;
     big = small;
     small = t;
   }
   return big;
 } }
```

# Computing the GCD: Method Specification

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Gcd {
 /*@ public normal_behavior
   @ requires _small>=0 && _big>=_small;
   @ ensures _big!=0 ==>
   @ (_big % \result == 0 && _small % \result == 0 &&
   @   (\forall int x; x>0 && _big % x == 0
   @     && _small % x == 0; \result % x == 0));
   @ assignable \nothing;
   @*/
 private static int gcdHelp(int _big, int _small) {...}
```

**requires** normalization assumptions on method parameters (both non-negative and `_big` $\geq$ `_small`)

**ensures** if `_big` positive, then

- the return value `\result` is a divider of both arguments
- all other dividers `x` of the arguments are also dividers of `\result` and thus smaller or equal to `\result`

# Computing the GCD: Specify the Loop Body

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int big = _big; int small = _small;
while (small != 0) {
  final int t = big % small;
  big = small;
  small = t;
}
return big;
```

Which locations are changed (at most)?

```
@ assignable \nothing; // no heap locations changed
```

What is the variant?

```
@ decreases small;
```

# Computing the GCD: Specify the Loop Body Cont’d

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int big = _big; int small = _small;
while (small != 0) {
  final int t = big % small;
  big = small;
  small = t;
}
return big;
```

## Loop Invariant

- Order between `small` and `big` preserved by loop: `big>=small`
- Possible for `big` to become 0 in a loop iteration? No.
- Adding `big>0` to loop invariant? No. Not initially valid.
- Weaker condition necessary: `big==0 ==> _big==0`

# Computing the GCD: Specify the Loop Body Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

```
int big = _big; int small = _small;
while (small != 0) {
  final int t = big % small;
  big = small;
  small = t;
}
return big;
```

## Loop Invariant

- Order between `small` and `big` preserved by loop: `big>=small`
- Weaker condition necessary: `big==0 ==> _big==0`
- What does the loop preserve? The set of dividers!
  All common dividers of `_big, _small` are also dividers of `big, small`

```
(\forall int x; x > 0;
  (_big%x == 0 && _small%x == 0) <==>
            (big%x == 0 && small%x == 0));
```

# Computing the GCD: Final Specification

```
int big = _big; int small = _small;
/*@ loop_invariant small >= 0 && big >= small &&
  @   (big == 0 ==> _big == 0) &&
  @   (\forall int x; x > 0; (_big % x == 0 && _small % x == 0)
  @          <==>
  @   (big % x == 0 && small % x == 0));
  @ decreases small;
  @ assignable \nothing;
  @*/
  while (small != 0) {
    final int t = big % small;
    big = small;
    small = t;
  }
  return big; // assigned to \result
```

Why does `big` divides `_small` and `_big` follow from the loop invariant?

If `big` is positive, one can instantiate `x` with it, and use `small == 0`

# Computing the GCD: Demo



Demo `loops/Gcd.java`

1. Show `Gcd.java` and `gcd(a,b)`
2. Ensure that “DefOps” and “Contracts” is selected, $\geq$ 10,000 steps
3. Proof contract of `gcd()`, using contract of `gcdHelp()`
4. Note KeY check sign in parentheses:
   4.1 Click “Proof Management”
   4.2 Choose tab “By Proof”
   4.3 Select proof of `gcd()`
   4.4 Select used method contract of `gcdHelp()`
   4.5 Click “Start Proof”
5. After finishing proof obligations of `gcdHelp()` parentheses are gone

# Some Tips On Finding Invariants

TECHNISCHE UNIVERSITÄT DARMSTADT

## General Advice

- Invariants must be developed, they don't come out of thin air!
- Be as systematic in deriving invariants as when debugging a program
- Don't forget: the program or contract (more likely) can be buggy
  - In this case, you won't find an invariant!

# Some Tips On Finding Invariants, Cont'd



## Technical Tips

- The desired postcondition is a good starting point
  - What, in addition to negated loop guard, is needed for it to hold?
- If the invariant candidate is not preserved by the loop body:
  - Can you add stuff from the precondition?
  - Does it need strengthening?
  - Try to express the relation between partial and final result
- Simulate a few loop body executions to discover invariant patterns
- If the invariant is not initially valid:
  - Can it be weakened such that the postcondition still follows?
  - Did you forget an assumption in the requires clause?
- Several "rounds" of weakening/strengthening might be required
- Use the KeY tool for each premiss of invariant rule
  - After each change of the invariant make sure all cases are ok
  - Interactive dialogue: previous invariants available in "Alt" tabs
  - Look at open first-order goals: use model search!