

# Formal Methods for Software Development

## Reasoning about Programs with Loops and Method Calls

Wolfgang Ahrendt

17 October 2017

# Program Logic Calculus – Repetition

Calculus realises **symbolic interpreter**:

- ▶ works on **first active statement**
- ▶ **decomposition** of complex statements into simpler ones
- ▶ simple assignments to **updates**
- ▶ accumulated update captures changed program state (abbr. w.  $\mathcal{U}$ )
- ▶ **control flow branching** induces proof splitting
- ▶ application of update computes **weakest precondition** of  $\mathcal{U}'$  wrt.  $\phi$

$$\Gamma' \Rightarrow \{\mathcal{U}'\}\phi \quad \dots$$

...

$$\text{'branch1'} \quad \Gamma, \{\mathcal{U}\}(\text{isValid} = \text{TRUE}) \Rightarrow \{\mathcal{U}\}\langle\{\text{ok}=\text{true};\}\dots\rangle\phi$$

$$\text{'branch2'} \quad \Gamma, \{\mathcal{U}\}(\text{isValid} = \text{FALSE}) \Rightarrow \{\mathcal{U}\}\langle\dots\rangle\phi$$

$$\Gamma \Rightarrow \{\mathbf{t} := \mathbf{j} \parallel \mathbf{j} := \mathbf{j} + 1 \parallel \mathbf{i} := \mathbf{j}\}\{\mathcal{U}\}\{\mathcal{U}\}\langle\text{if}(\text{isValid})\{\text{ok}=\text{true};\}\dots\rangle\phi$$

...

$$\Gamma \Rightarrow \{\mathbf{t} := \mathbf{j}\}\langle\mathbf{j}=\mathbf{j}+1; \mathbf{i}=\mathbf{t}; \text{if}(\text{isValid})\{\text{ok}=\text{true};\}\dots\rangle\phi$$

$$\Gamma \Rightarrow \langle\mathbf{t}=\mathbf{j}; \mathbf{j}=\mathbf{j}+1; \mathbf{i}=\mathbf{t}; \text{if}(\text{isValid})\{\text{ok}=\text{true};\}\dots\rangle\phi$$

$$\Gamma \Rightarrow \langle\mathbf{i}=\mathbf{j}++; \text{if}(\text{isValid})\{\text{ok}=\text{true};\}\dots\rangle\phi$$

# An Example

```
\javaSource "src/";

\programVariables{
  Person p;
  int j;
}

\problem {
  (\forall int i;
    (!p=null ->
      ({j := i}\<\{p.setAge(j);} \>(p.age = i))))
}
```

# Method Calls

**Method Call** with actual parameters  $arg_0, \dots, arg_n$

$$\langle o.m(arg_0, \dots, arg_n); \omega \rangle \phi$$

assume  $m$  declared as  $\text{void } m(\tau_0 p_0, \dots, \tau_n p_n)$

## Actions of rule **methodCall**

1. Declare **new local variables**  $p\#i$ , initialize them with actual parameter:  $\tau_i p\#i = arg_i$ ;
2. **Look-up implementing class**  $C$  of  $m$ ;  
split proof if implementation cannot be uniquely determined.
3. Replace method call with **implementation invocation**  
 $o.m(p\#0, \dots, p\#n)@C$

# Method Calls Cont'd

After executing the initialisers:  $\tau_i \text{ p\#i} = \text{arg}_i$ ; apply:

## Method Body Expand

Call rule `methodBodyExpand`

$$\frac{\Gamma \Rightarrow \langle \text{method-frame}(\text{source}=\text{m}(\tau_0, \dots, \tau_n) @ \text{C}, \text{this}=\text{o}): \{\text{body}\} \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \text{o.m}(\text{p}\#0, \dots, \text{p}\#n) @ \text{C}; \omega \rangle \phi, \Delta}$$

1. Replaces method invocation by method frame with method body
2. Renames  $p_i$  in body to  $p\#i$

Method frames:

Required in proof to represent call stack

## Demo

```
methods/instanceMethodInlineSimple.key  
methods/inlineDynamicDispatch.key
```

## JAVA has complex rules for **localisation** of fields and method implementations

- ▶ Polymorphism
- ▶ Late binding (dynamic dispatch)
- ▶ Scoping (class vs. instance)
- ▶ Visibility (private, protected, public)

Proof split into cases if implementation not statically determined

# Object initialization

## JAVA has complex rules for object initialization

- ▶ Chain of constructor calls until **Object**
- ▶ Implicit calls to `super()`
- ▶ Visibility issues
- ▶ Initialization sequence

Coding of initialization rules in methods `<createObject>()`, `<init>()`, ... which are then symbolically executed

# Limitations of Method Inlining: `methodBodyExpand`

- ▶ Source code might be **unavailable**
  - ▶ API method implementation vendor-specific
  - ▶ Source code often unavailable for commercial APIs
- ▶ Method is invoked **multiple times** in a program
  - ▶ Avoid multiple symbolic execution of identical code
- ▶ Cannot handle **unbounded recursion**
- ▶ **Not modular**:  
Changing a method requires re-verification of all callers

Use **method contract** instead of method implementation:

1. Show that **requires** clause is satisfied
2. Continue after method call:
  - ▶ assume **ensures** clause
  - ▶ forget prestate values of **modifiable** locations



# Method Contract Rule: Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
   @ requires preNormal;
   @ ensures postNormal;
   @ assignable mod;
   @*/ // implementation contract of m()
```

$$\frac{\begin{array}{l} \Gamma \Rightarrow \mathcal{UF}(\text{preNormal}), \Delta \quad (\text{precondition}) \\ \Gamma \Rightarrow \mathcal{UV}_{\text{mod}}(\mathcal{F}(\text{postNormal}) \rightarrow \langle \pi \omega \rangle \phi), \Delta \quad (\text{normal}) \end{array}}{\Gamma \Rightarrow \mathcal{U}\langle \pi \text{ result} = m(a_1, \dots, a_n); \omega \rangle \phi, \Delta}$$

- ▶  $\pi$  are openings of try blocks and method frames
- ▶  $\mathcal{F}(\cdot)$ : translation from JML to Java DL
- ▶  $\mathcal{V}_{\text{mod}}$ : anonymising update,  
*forgetting prevalues of modifiable locations*

# JML Method Contracts Revisited

```
/*@ public normal_behavior
   @ requires preNormal;
   @ ensures postNormal;
   @ assignable mod;
   @*/
T m(T1 a1, ..., Tn an) { ... }
```

## Implicit Preconditions and Postconditions

- ▶ The object referenced by `this` is not null: `this!=null` (precondition only; `this` cannot be changed by method)
- ▶ The heap is wellformed: `wellFormed(heap)` (precondition only)
- ▶ Invariant for 'this': `\invariant_for(this)`

# Method Contract Rule: Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
   @ requires preNormal;
   @ ensures postNormal;
   @ assignable mod;
   @*/ // implementation contract of m()
```

$$\frac{\begin{array}{l} \Gamma \Rightarrow \mathcal{UF}(\text{preNormal}), \Delta \quad (\text{precondition}) \\ \Gamma \Rightarrow \mathcal{UV}_{\text{mod}}(\mathcal{F}(\text{postNormal}) \rightarrow \langle \pi \omega \rangle \phi), \Delta \quad (\text{normal}) \end{array}}{\Gamma \Rightarrow \mathcal{U}\langle \pi \text{ result} = m(a_1, \dots, a_n); \omega \rangle \phi, \Delta}$$

- ▶  $\pi$  are openings of try blocks and method frames
- ▶  $\mathcal{F}(\cdot)$ : translation from JML to Java DL
- ▶  $\mathcal{V}_{\text{mod}}$ : anonymising update,  
*forgetting prevalues of modifiable locations*

# Keeping the Context

- ▶ Want to keep part of prestate  $\mathcal{U}$  that is **unmodified** by called method
- ▶ **assignable clause** of contract tells what can possibly be modified

```
@ assignable mod;
```

- ▶ How to erase all values of **assignable** locations in state  $\mathcal{U}$  ?
- ▶ **Anonymising updates**  $\mathcal{V}$  erase information about modified locations

# Anonymising Heap Locations

**Define anonymising function**  $\text{anon}: \text{Heap} \times \text{LocSet} \times \text{Heap} \rightarrow \text{Heap}$

The resulting heap  $\text{anon}(\dots)$  coincides with the first heap on all locations except for those specified in the location set. Those locations attain the value specified by the second heap.

*Definition:*

$$\text{select}(\text{anon}(h1, locs, h2), o, f) = \begin{cases} \text{select}(h2, o, f) & \text{if } (o, f) \in locs \\ \text{select}(h1, o, f) & \text{otherwise} \end{cases}$$

*Usage:*

$$\mathcal{V}_{mod} = \{\text{heap} := \text{anon}(\text{heap}, locs_{mod}, h_{an})\}$$

where  $h_{an}$  a new (not yet used) constant of type Heap

*Effect:* After  $\mathcal{V}_{mod}$ , modified locations have unknown values

# Anonymising Heap Locations: Example

```
@ assignable o.a, this.*;
```

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

- ▶ Anonymise the current heap on the designated locations:

$$\text{anon}(\text{heap}, \{(o, a)\} \cup \text{allFields}(\text{this}), h_{an})$$

- ▶ Make that anonymised current heap the new current heap.

$$\mathcal{V}_{mod} = \{\text{heap} := \text{anon}(\text{heap}, \{(o, a)\} \cup \text{allFields}(\text{this}), h_{an})\}$$

# Method Contract Rule: Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
   @ requires preExc;
   @ signals (Exception exc) postExc;
   @ assignable mod;
   @*/
```

$$\frac{\begin{array}{l} \Gamma \Rightarrow \mathcal{UF}(\text{preExc}), \Delta \quad (\text{precondition}) \\ \Gamma \Rightarrow \mathcal{UV}_{\text{mod}}((\mathcal{F}(\text{postExc}) \wedge \text{exc} \neq \text{null}) \\ \qquad \qquad \qquad \rightarrow \langle \pi \text{ throw exc; } \omega \rangle \phi), \Delta \quad (\text{exceptional}) \end{array}}{\Gamma \Rightarrow \mathcal{U}\langle \pi \text{ result} = \text{m}(\mathbf{a}_1, \dots, \mathbf{a}_n); \omega \rangle \phi, \Delta}$$

- ▶  $\pi$  are openings of try blocks and method frames
- ▶  $\mathcal{F}(\cdot)$ : translation from JML to Java DL
- ▶  $\mathcal{V}_{\text{mod}}$ : anonymising update

# Method Contract Rule – Combined

(background only, no need to remember)

KeY uses actually **one rule for both** kinds of cases.

Therefore translation of postcondition  $\phi_{post}$  as follows (simplified):

$$\begin{aligned}\phi_{post\_n} &\equiv \mathcal{F}(\backslash\mathbf{old}(\mathbf{normalPre})) \wedge \mathcal{F}(\mathbf{normalPost}) \\ \phi_{post\_e} &\equiv \mathcal{F}(\backslash\mathbf{old}(\mathbf{excPre})) \wedge \mathcal{F}(\mathbf{excPost})\end{aligned}$$

$$\begin{array}{l} \Gamma \Rightarrow \mathcal{U}(\mathcal{F}(\mathbf{normalPre}) \vee \mathcal{F}(\mathbf{excPre})), \Delta \quad (\text{precondition}) \\ \Gamma \Rightarrow \mathcal{U}\mathcal{V}_{\mathbf{mod}_{\mathbf{normal}}}(\phi_{post\_n} \rightarrow \langle \pi \omega \rangle \phi), \Delta \quad (\text{normal}) \\ \Gamma \Rightarrow \mathcal{U}\mathcal{V}_{\mathbf{mod}_{\mathbf{exc}}}((\phi_{post\_e} \wedge \mathbf{exc} \neq \mathbf{null}) \\ \quad \rightarrow \langle \pi \mathbf{throw} \ \mathbf{exc}; \omega \rangle \phi), \Delta \quad (\text{exceptional}) \\ \hline \Gamma \Rightarrow \mathcal{U}\langle \pi \mathbf{result} = \mathbf{m}(\mathbf{a}_1, \dots, \mathbf{a}_n); \omega \rangle \phi, \Delta \end{array}$$

- ▶  $\mathcal{F}(\cdot)$ : translation to Java DL
- ▶  $\mathcal{V}_{\mathbf{mod}}$ : anonymising update



## Method Contract Rule: Example

```
class Person {
  private /*@ spec_public @*/ int age;
  /*@ public normal_behavior
    @ requires age < 29;
    @ ensures age == \old(age) + 1;
    @ assignable age;
    @ also
    @ public exceptional_behavior
    @ requires age >= 29;
    @ signals_only ForeverYoungException;
    @ assignable \nothing;
    @//allows object creation (not \strictly_nothing)
  @*/
  public void birthday() {
    if (age >= 29) throw new ForeverYoungException();
    age++;
  }
}
```

# Method Contract Rule: Example Cont'd

## Demo

`methods/useContractForBirthday.key`

- ▶ Prove without contracts
  - ▶ Method treatment: Expand
- ▶ Prove with contracts (until method contract application)
  - ▶ Method treatment: Contract
- ▶ Prove used contracts
  - ▶ Method treatment: Expand
  - ▶ Select contracts for `birthday()` in `src/Person.java`
  - ▶ Prove both specification cases

# Verification of Loops

## Symbolic execution of loops: unwind

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

How to handle a loop with...

- ▶ 0 iterations? Unwind 1×
- ▶ 10 iterations? Unwind 11×
- ▶ 10000 iterations? Unwind 10001×
- ▶ an **unknown** number of iterations?

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

# Loop Invariants

## Idea behind loop invariants

- ▶ A formula  $Inv$  whose validity is **preserved by loop body** whenever the loop guard is true
- ▶ **Consequence**: if  $Inv$  was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- ▶ In particular, if the loop terminates, then  $Inv$  holds **afterwards**
- ▶ Challenge: construct  $Inv$  such that, *together with loop exit condition*, it implies **postcondition** of loop

## Basic Invariant Rule

$$\text{loopInvariant} \frac{\begin{array}{l} \Gamma \Rightarrow \mathcal{U}Inv, \Delta \quad \text{(valid when entering loop)} \\ Inv, b = \text{TRUE} \Rightarrow [p]Inv \quad \text{(preserved by p)} \\ Inv, b = \text{FALSE} \Rightarrow [\pi \omega]\phi \quad \text{(assumed after exit)} \end{array}}{\Gamma \Rightarrow \mathcal{U}[\pi \text{ while}(b) p \omega]\phi, \Delta}$$

# How to Derive Loop Invariants Systematically?

## Example (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  $i <= n$  preserved by loop body?

Does it hold when entering loop?

Yes! We have found a suitable loop invariant!

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

# Obtaining Invariants by Strengthening

## Example (Slightly changed problem)

```
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?

$i <= n \ \& \ n = m$

Is  $i <= n \ \& \ n = m$  preserved by loop body?

Does it hold when entering loop?

Yes! We have found a suitable loop invariant!

# Generalization

## Example (Addition: $x, y$ program variables, $x_0, y_0$ 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  $x = x_0 + y_0$

- ▶ Not true at start whenever  $y_0 > 0$
- ▶ Not preserved by loop, because  $x$  is increased

# Generalization

## Example (Addition: $x, y$ program variables, $x_0, y_0$ 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  $y$ :  $x + y = x_0 + y_0$  “Generalization”
- ▶ Can help to think of “ $\delta$ ” between  $x$  and  $x_0 + y_0$



# Generalization

## Example (Addition: $x, y$ program variables, $x_0, y_0$ 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  $x + y = x_0 + y_0$  a good invariant?

- ▶ Holds in the beginning and is preserved by loop
- ▶ But postcondition not achieved by  $x + y = x_0 + y_0$  and exit condition  $y \leq 0$

# Generalization

## Example (Addition: $x, y$ program variables, $x_0, y_0$ rigid constants)

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

## Strengthening the invariant

Postcondition holds if  $y = 0$

- ▶ Add  $y \geq 0$  to invariant:  $x + y = x_0 + y_0 \ \& \ y \geq 0$

Demo loops/simple3.key

# Basic Loop Invariant: Context Loss

## Problems with the Basic Invariant Rule

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

- ▶ Context  $\Gamma, \Delta, \mathcal{U}$  must be omitted in 2nd and 3rd premise:
  - $\Gamma, \neg\Delta$  cannot be assumed for arbitrary iterations or at loop exit
  - 2nd premise** State after some loop iterations is not  $\mathcal{U}$
  - 3rd premise** State at loop exit is not  $\mathcal{U}$
- ▶ Context contains preconditions and class invariants
- ▶ Only way to propagate context: add to loop invariant  $Inv$

# Example

Precondition:  $a \neq \text{null} \ \& \ \text{ClassInv}$

---

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

---

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

Loop invariant:  $0 \leq i \ \& \ i \leq a.length$   
 $\ \& \ \forall \text{int } x; (0 \leq x \ \& \ x < i \rightarrow a[x] = 1)$   
 $\ \& \ a \neq \text{null}$   
 $\ \& \ \text{ClassInv}$

# Keeping the Context (As In Method Contract Rule)

- ▶ Want to keep part of the context that is **not modified** by loop
- ▶ **assignable clauses** for loops tell what can possibly be modified

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

- ▶ How to erase all values of **assignable** locations?
- ▶ **Anonymising updates**  $\forall$  erase information about modified locations

# Anonymising JAVA Locations

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

To erase all knowledge about these assignable locations:

- ▶ introduce a new (not yet used) constant of type `int`, e.g., `c`
- ▶ introduce a new (not yet used) constant of type `Heap`, e.g., `han`
  - ▶ anonymise the current heap: `anon(heap, allFields(a), han)`
- ▶ compute anonymizing update for assignable locations

$$\mathcal{V} = \{i := c \parallel \text{heap} := \text{anon}(\text{heap}, \text{allFields}(a), h_{an})\}$$

For local program variables (e.g., `i`) KeY computes assignable clause automatically

# Loop Invariants Cont'd

## Improved Invariant Rule

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

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

## Example with Improved Invariant Rule

Precondition:  $a \neq \text{null} \ \& \ \text{ClassInv}$

---

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

---

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

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



```
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 an earlier Lecture

```
∀ int x;  
  (x = n ∧ x ≥ 0 →  
    [ i = 0; r = 0;  
      while (i < n) { i = i + 1; r = r + i; }  
      r = r + r - n;  
    ] (r = x * x))
```

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

Needed Invariant:

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

Demo [Loop2.java](#)

## Proving assignable

- ▶ Invariant rule above **assumes** that **assignable** is correct  
E.g., possible to prove nonsense with incorrect **assignable \nothing;**
- ▶ 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 w. given invariant

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

# Total Correctness

Is the sequent

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

provable?

Yes, e.g.,

```
@ loop_invariant true;
```

```
@ assignable \nothing;
```

With this, correctness of **non-terminating** loop is provable:

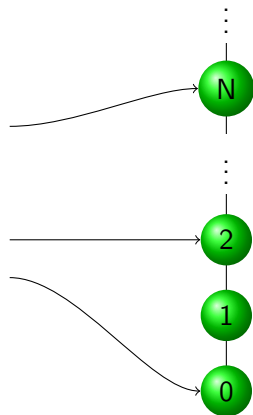
- ▶ Invariant trivially initially valid and preserved:  
Initial Case and Preserved Case close immediately
- ▶ Negated loop condition is false: Use case close immediately

But need a method to prove **termination** of loops

# Mapping Loop Execution to Well-Founded Order

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

```
if (b) { body }1  
⋮  
if (b) { body }17  
if (b) { body }18
```



**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)

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

Add the following premisses 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 **diverges true;** from contract
- ▶ Add **decreasing v;** to loop invariant
- ▶ KeY creates suitable invariant rule and PO (with  $\langle \dots \rangle \phi$ )

### Example (The array loop)

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

Files:

- ▶ LoopT.java
- ▶ Loop2T.java

## Final Example: Computing the GCD(see 16.3.8 [KeYbook])

```
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

```
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  $\backslashresult$  is a divisor of both arguments
- ▶ all other divisors  $x$  of the arguments are also dividers of  $\backslashresult$  and thus smaller or equal to  $\backslashresult$



## Computing the GCD: Specify the Loop Body

```
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

```
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

```
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:  $big \geq small$
- ▶ Weaker condition necessary:  $big == 0 \implies \_big == 0$
- ▶ 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 “Contract” 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 Hints On Finding Invariants

## 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

# Some Hints On Finding Invariants, Cont'd

## Technical Hints

- ▶ Good starting point: desired **postcondition** (of the loop!)
  - ▶ 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 to iteratively try invariants:
  - ▶ Loop treatment: **None**
  - ▶ apply **Loop Invariant** → **Enter Loop Specification**
  - ▶ After each change of invariant make sure all cases are ok
  - ▶ If not, prue and retry

# Understanding Unclosed Proofs (see also p.528ff [KeYbook])

## Reasons why a proof may not close

- ▶ Buggy or incomplete specification
- ▶ Bug in program
- ▶ Maximal number of steps reached: restart or increase # of steps
- ▶ Automatic proof search fails: manual rule applications necessary

## Understanding open proof goals

- ▶ Follow the control flow from the proof root to the open goal
- ▶ Branch labels give useful hints
- ▶ Identify unprovable part of post condition or invariant
- ▶ Sequent remains always in “pre-state”  
Constraints on program variables refer to value at start of program  
(exception: formula is behind update or modality)
- ▶ NB:  $\Gamma \Rightarrow o = \mathbf{null}, \Delta$  is equivalent to  $\Gamma, o \neq \mathbf{null} \Rightarrow \Delta$



# Literature for this Lecture

**KeYbook** *W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. Schmitt, M. Ulbrich, editors.*

*Deductive Software Verification - The KeY Book*

Vol 10001 of *LNCS*, Springer, 2016

(E-book at [link.springer.com](http://link.springer.com))

- ▶ *W. Ahrendt, S. Grebing, Using the KeY Prover*  
Chapter 15 in [KeYbook], p.528ff + Section 15.3 (also for Lab2)
- ▶ *B. Beckert, R. Hähnle, M. Hentschel, P.H. Schmitt, Formal Verification with KeY: A Tutorial*  
Chapter 16 in [KeYbook], except Section 16.6

further reading:

- ▶ *B. Beckert, V. Klebanov, B. Weiß, Dynamic Logic for Java*  
Chapter 3 in [KeYbook], Section 3.7

# Master's Thesis Projects in Formal Methods

## Examples:

- ▶ Extracting tests from runtime traces
- ▶ JM<sup>2</sup>L: a library of specification models for Java
- ▶ Tests from failed proofs
- ▶ Dynamic invariants for runtime verification
- ▶ Runtime verification of reactive systems
- ▶ Compositional runtime enforcement of reactive systems
- ▶ Learning automata (properties) from example traces
- ▶ Application of runtime monitoring for automotive systems
- ▶ Malware on Chrome: Modifying Secure Preferences
- ▶ Browser Extensions Metadata Correlation
- ▶ Chromium Modification to Parallelise Browser Extensions
- ▶ Various master theses on Blockchain, Bitcoin, and Smart Contracts

# Master's Thesis Projects in Formal Methods

see Formal Methods Master Theses on the [web \(click here\)](#).  
(and come back, many edits on that page these days)