Software Engineering using Formal Methods Reasoning about Programs with Loops and Method Calls

Wolfgang Ahrendt

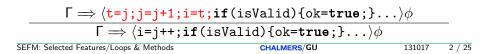
17 October 2013

Calculus realises symbolic interpreter:

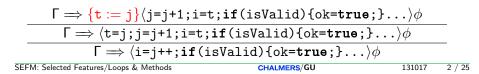
works on first active statement

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

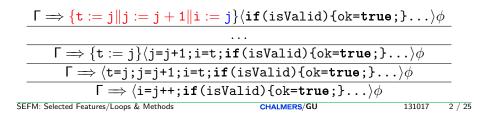
- works on first active statement
- decomposition of complex statements into simpler ones



- works on first active statement
- decomposition of complex statements into simpler ones
- simple assignments to updates

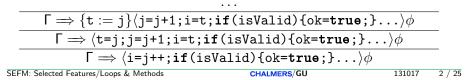


- works on first active statement
- decomposition of complex statements into simpler ones
- simple assignments to updates
- accumulated update captures changed program state

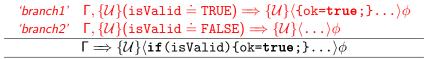


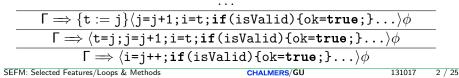
- works on first active statement
- decomposition of complex statements into simpler ones
- simple assignments to updates
- accumulated update captures changed program state (abbr. w. U)

$$\mathsf{\Gamma} \Longrightarrow \{\mathcal{U}\} \langle \texttt{if(isValid)}\{\texttt{ok=true;}\} \dots \rangle \phi$$



- works on first active statement
- decomposition of complex statements into simpler ones
- simple assignments to updates
- accumulated update captures changed program state
- control flow branching induces proof splitting





Calculus realises symbolic interpreter:

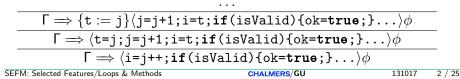
- works on first active statement
- decomposition of complex statements into simpler ones
- simple assignments to updates
- accumulated update captures changed program state
- control flow branching induces proof splitting
- \blacktriangleright application of update computes weakest precondition of \mathcal{U}'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

. . .

. . .

$$\Gamma' \Longrightarrow \{\mathcal{U}'\}\phi$$

 $\begin{array}{ll} \text{`branch1'} & \Gamma, \{\mathcal{U}\}(\texttt{isValid} \doteq \texttt{TRUE}) \Longrightarrow \{\mathcal{U}\}\langle\{\texttt{ok=true};\}\dots\rangle\phi \\ \text{`branch2'} & \Gamma, \{\mathcal{U}\}(\texttt{isValid} \doteq \texttt{FALSE}) \Longrightarrow \{\mathcal{U}\}\langle\dots\rangle\phi \\ & \Gamma \Longrightarrow \{\mathcal{U}\}\langle\texttt{if}(\texttt{isValid})\{\texttt{ok=true};\}\dots\rangle\phi \end{array}$



How to express using updates that a formula $\boldsymbol{\phi}$ is evaluated in a state where

program variable i has been set to 5?

How to express using updates that a formula $\boldsymbol{\phi}$ is evaluated in a state where

- program variable i has been set to 5? $\{i := 5\}\phi$
- program variable i has been increased by 1?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable i has been set to 5? $\{i := 5\}\phi$
- ▶ program variable i has been increased by 1? {i := i+1}φ
- program variables i and j swapped values?

How to express using updates that a formula $\boldsymbol{\phi}$ is evaluated in a state where

- program variable i has been set to 5? $\{i := 5\}\phi$
- program variable i has been increased by 1? ${i := i+1}\phi$
- ▶ program variables i and j swapped values? ${i := j || j := i}\phi$
- all components of an array arr of length 2 have value 0?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable i has been set to 5? $\{i := 5\}\phi$
- program variable i has been increased by 1? ${i := i+1}\phi$
- ▶ program variables i and j swapped values? ${i := j || j := i}\phi$
- ▶ all components of an array arr of length 2 have value 0? {arr[0] := 0 || arr[1] := 0}φ
- all components of an array arr of length n have value 0?

How to express using updates that a formula ϕ is evaluated in a state where

- program variable i has been set to 5? $\{i := 5\}\phi$
- program variable i has been increased by 1? ${i := i+1}\phi$
- ▶ program variables i and j swapped values? ${i := j \parallel j := i}\phi$
- ▶ all components of an array arr of length 2 have value 0? $\{ \arg[0] := 0 \parallel \arg[1] := 0 \} \phi$
- all components of an array arr of length n have value 0?

For example to deal with things like

Quantified Updates

Definition (Quantified Update)

For T well-ordered type (no ∞ descending chains): quantified update:

$$\{ \mathsf{Vfor } T x; \mathsf{Vif } \phi(x); l(x) := r(x) \}$$

- ► For all objects d in T such that φ(d) perform the updates {I(d) := r(d)} in parallel
- ▶ If there are several *I* with conflicting *d* then choose *T*-minimal one
- The conditional expression is optional
- Typically, x occurs in ϕ , *I*, and *r* (but doesn't need to)
- There is a normal form for updates computed efficiently by KeY

Quantified Updates Cont'd

Example (Initialization of field a for all objects in class C)

 $\{ \texttt{lfor } C \ o; o.a := 0 \}$

Quantified Updates Cont'd

Example (Initialization of field a for all objects in class C)

 $\{ \texttt{lfor } C \ o; o.a := 0 \}$

Example (Initialization of components of array a)

 $\{ \texttt{\for int } i; a[i] := 0 \}$

Quantified Updates Cont'd

Example (Initialization of field a for all objects in class C)

 $\{ \texttt{lfor } C \ o; o.a := 0 \}$

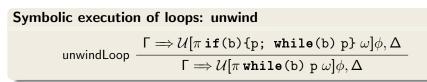
Example (Initialization of components of array a)

 $\{\texttt{\for int } i; a[i] := 0\}$

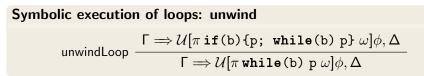
Example (Integer types are well-ordered in KeY)

$$\{\texttt{\for int } i; a[0] := i\}(a[0] \doteq 0)$$

- Non-standard order for Z (with 0 smallest and preserving < for arguments of same sign)
- Proven automatically by update simplifier



(We omitted \mathcal{U} previous lectures, for simplicity.)



(We omitted \mathcal{U} previous lectures, for simplicity.)

How to handle a loop with...

0 iterations?

Symbolic execution of loops: unwind $\frac{\Gamma \Longrightarrow \mathcal{U}[\pi \, if(b) \{p; \, while(b) \, p\} \, \omega] \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi \, while(b) \, p \, \omega] \phi, \Delta}$

(We omitted $\mathcal U$ previous lectures, for simplicity.)

How to handle a loop with...

► 0 iterations? Unwind 1×

Symbolic execution of loops: unwind $\frac{\Gamma \Longrightarrow \mathcal{U}[\pi \, if(b) \{p; \, while(b) \, p\} \, \omega] \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi \, while(b) \, p \, \omega] \phi, \Delta}$

(We omitted $\mathcal U$ previous lectures, for simplicity.)

- ► 0 iterations? Unwind 1×
- 10 iterations?

Symbolic execution of loops: unwind unwindLoop $\frac{\Gamma \Longrightarrow \mathcal{U}[\pi if(b) \{p; while(b) p\} \omega] \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi while(b) p \omega] \phi, \Delta}$

(We omitted \mathcal{U} previous lectures, for simplicity.)

- ► 0 iterations? Unwind 1×
- ▶ 10 iterations? Unwind $11 \times$

Symbolic execution of loops: unwind $\frac{\Gamma \Longrightarrow \mathcal{U}[\pi \, if(b) \{p; \, while(b) \, p\} \, \omega] \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi \, while(b) \, p \, \omega] \phi, \Delta}$

(We omitted $\mathcal U$ previous lectures, for simplicity.)

- ► 0 iterations? Unwind 1×
- ▶ 10 iterations? Unwind $11 \times$
- 10000 iterations?

Symbolic execution of loops: unwind unwindLoop $\frac{\Gamma \Longrightarrow \mathcal{U}[\pi \texttt{if}(\texttt{b})\{\texttt{p}; \texttt{while}(\texttt{b}) \texttt{p}\} \omega] \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi \texttt{while}(\texttt{b}) \texttt{p} \omega] \phi, \Delta}$

(We omitted $\mathcal U$ previous lectures, for simplicity.)

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

Symbolic execution of loops: unwind

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

(We omitted \mathcal{U} previous lectures, for simplicity.)

How to handle a loop with...

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

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

Idea behind loop invariants

► A formula *Inv* whose validity is preserved by loop guard and body

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

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, then *Inv* holds afterwards

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, then *Inv* holds afterwards
- Construct *Inv* such that, together with loop exit condition, it implies postcondition of loop

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, then *Inv* holds afterwards
- Construct *Inv* such that, together with loop exit condition, it implies postcondition of loop

Basic Invariant Rule

loopInvariant

$$\Longrightarrow \mathcal{U}[\pi \, \texttt{while(b)} \, \texttt{p} \, \omega] \phi, \Delta$$

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, then *Inv* holds afterwards
- Construct *Inv* such that, together with loop exit condition, it implies postcondition of loop

Basic Invariant Rule

 $\Gamma \Longrightarrow \mathcal{U} \operatorname{Inv}, \Delta$

(valid when entering loop)

loopInvariant

$$\Longrightarrow \mathcal{U}[\pi \, \texttt{while(b)} \, \texttt{p} \, \omega] \phi, \Delta$$

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, then *Inv* holds afterwards
- Construct *Inv* such that, together with loop exit condition, it implies postcondition of loop

Basic Invariant Rule

$$\Gamma \Longrightarrow \mathcal{U} Inv, \Delta$$
$$Inv, b \doteq \text{TRUE} \Longrightarrow [p] Inv$$

(valid when entering loop) (preserved by p)

loopInvariant

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

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, then *Inv* holds afterwards
- Construct *Inv* such that, together with loop exit condition, it implies postcondition of loop

Basic Invariant Rule

$$\begin{split} & \Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & (\text{valid when entering loop}) \\ & \mathit{Inv}, \ b \doteq \mathsf{TRUE} \Longrightarrow [p]\mathit{Inv} & (\text{preserved by p}) \\ & \mathsf{loopInvariant} & \frac{\mathit{Inv}, \ b \doteq \mathsf{FALSE} \Longrightarrow [\pi \ \omega]\phi}{\Gamma \Longrightarrow \mathcal{U}[\pi \ \texttt{while}(b) \ p \ \omega]\phi, \Delta} & (\text{assumed after exit}) \end{split}$$

Basic Invariant Rule: Problem

$$\begin{split} & \Gamma \Longrightarrow \mathcal{U} \textit{Inv}, \Delta & \text{(valid when entering loop)} \\ & \textit{Inv}, b \doteq \text{TRUE} \Longrightarrow [p]\textit{Inv} & \text{(preserved by p)} \\ & \text{loopInvariant} & \frac{\textit{Inv}, b \doteq \text{FALSE} \Longrightarrow [\pi \ \omega]\phi}{\Gamma \Longrightarrow \mathcal{U}[\pi \text{ while (b) p } \omega]\phi, \Delta} & \text{(assumed after exit)} \end{split}$$

Basic Invariant Rule: Problem

$$\begin{split} & \Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta & (\text{valid when entering loop}) \\ & \mathit{Inv}, b \doteq \text{TRUE} \Longrightarrow [p]\mathit{Inv} & (\text{preserved by p}) \\ & \textit{loopInvariant} & \frac{\mathit{Inv}, b \doteq \text{FALSE} \Longrightarrow [\pi \, \omega]\phi}{\Gamma \Longrightarrow \mathcal{U}[\pi \, \texttt{while}(b) \, p \, \omega]\phi, \Delta} & (\text{assumed after exit}) \end{split}$$

- Context Γ , Δ , \mathcal{U} must be omitted in 2nd and 3rd premise:
 - \blacktriangleright ${\cal U}$ represents state when entering loop, not after some loop iterations
 - \blacktriangleright keeping $\Gamma,\,\Delta$ without ${\cal U}$ meant executing p in prestate of program

Basic Invariant Rule: Problem

$$\begin{split} & \Gamma \Longrightarrow \mathcal{U} \textit{Inv}, \Delta & (\text{valid when entering loop}) \\ & \textit{Inv}, b \doteq \text{TRUE} \Longrightarrow [p]\textit{Inv} & (\text{preserved by p}) \\ & \text{loopInvariant} & \frac{\textit{Inv}, b \doteq \text{FALSE} \Longrightarrow [\pi \ \omega]\phi}{\Gamma \Longrightarrow \mathcal{U}[\pi \text{ while}(b) \ p \ \omega]\phi, \Delta} & (\text{assumed after exit}) \end{split}$$

- Context Γ , Δ , \mathcal{U} must be omitted in 2nd and 3rd premise:
 - \blacktriangleright ${\cal U}$ represents state when entering loop, not after some loop iterations
 - ▶ keeping Γ , Δ without \mathcal{U} meant executing p in prestate of program
- **But**: context contains important preconditions and class invariants

Basic Invariant Rule: Problem

$$\begin{split} & \Gamma \Longrightarrow \mathcal{U} \textit{Inv}, \Delta & \text{(valid when entering loop)} \\ & \textit{Inv}, b \doteq \text{TRUE} \Longrightarrow [p]\textit{Inv} & \text{(preserved by p)} \\ & \text{loopInvariant} & \frac{\textit{Inv}, b \doteq \text{FALSE} \Longrightarrow [\pi \ \omega]\phi}{\Gamma \Longrightarrow \mathcal{U}[\pi \text{ while}(b) \ p \ \omega]\phi, \Delta} & \text{(ssumed after exit)} \end{split}$$

- Context Γ , Δ , \mathcal{U} must be omitted in 2nd and 3rd premise:
 - \blacktriangleright ${\cal U}$ represents state when entering loop, not after some loop iterations
 - ▶ keeping Γ, Δ without U meant executing p in prestate of program
- But: context contains important preconditions and class invariants
- Relevant context information must be added to Inv ③

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

```
(Implicit) Class Invariant: a \neq null
```

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

```
(Implicit) Class Invariant: a \neq null
```

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

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

(Implicit) Class Invariant: $a \neq null$

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

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

Loop invariant: $0 \le i \& i \le a.length$

```
(Implicit) Class Invariant: a \neq null
```

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

Postcondition: $\forall \texttt{int } x; (0 \le x < \texttt{a.length} \rightarrow \texttt{a}[x] \doteq 1)$

(Implicit) Class Invariant: $a \neq null$

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

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

Want to keep part of the context that is unmodified by loop

Want to keep part of the context that is unmodified by loop

assignable clauses for loops can tell what might be modified

@ assignable i, a[*];

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

@ assignable i, a[*];

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

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

@ assignable i, a[*];

- How to erase all values of assignable locations in formula Γ ?
- Anonymising update \mathcal{V} erases information about assignable locations
- $\{i := c\}$ (c fresh constant symbol)

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

@ assignable i, a[*];

- How to erase all values of assignable locations in formula Γ ?
- Anonymising update \mathcal{V} erases information about assignable locations

 $\{i := c\} \qquad (c \text{ fresh constant symbol}) \\ \{ for x; a[x] := f(x) \} \qquad (f \text{ fresh function symbol})$

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

@ assignable i, a[*];

- How to erase all values of assignable locations in formula Γ ?
- Anonymising update \mathcal{V} erases information about assignable locations

 $\{i := c\} \qquad (c \text{ fresh constant symbol}) \\ \{ for x; a[x] := f(x) \} \qquad (f \text{ fresh function symbol})$

$$\mathcal{V} = \{ i := c \mid | \text{for } x; a[x] := f(x) \}$$

(c, f fresh constant resp. function symbol)

Improved Invariant Rule

$$\mathsf{\Gamma} \Longrightarrow \mathcal{U}[\pi \, \texttt{while(b)} \, \operatorname{p} \omega] \phi, \Delta$$



 $\Gamma \Longrightarrow \mathcal{U}$ Inv, Δ

(valid when entering loop)

 $\Gamma \Longrightarrow \mathcal{U}[\pi \text{ while (b) } p \ \omega]\phi, \Delta$

Improved Invariant Rule

$$\begin{split} & \Gamma \Longrightarrow \mathcal{U}\textit{Inv}, \Delta & (\text{valid when entering loop}) \\ & \Gamma \Longrightarrow \mathcal{U}\mathcal{V}(\textit{Inv \& } b \doteq \texttt{TRUE} \rightarrow [p]\textit{Inv}), \Delta & (\text{preserved by } p) \end{split}$$

 $\Gamma \Longrightarrow \mathcal{U}[\pi \text{ while (b) } p \ \omega]\phi, \Delta$

Improved Invariant Rule

 $\Gamma \Longrightarrow \mathcal{U} lnv, \Delta$ $\Gamma \Longrightarrow \mathcal{UV}(lnv \& b \doteq \text{TRUE} \rightarrow [p] lnv), \Delta$ $\Gamma \Longrightarrow \mathcal{UV}(Inv \& b \doteq FALSE \to [\pi \ \omega]\phi), \Delta$ (assumed after exit) $\Gamma \Longrightarrow \mathcal{U}[\pi \text{ while (b) } p \ \omega]\phi, \Delta$

(valid when entering loop) (preserved by p)

Improved Invariant Rule

 $\Gamma \Longrightarrow \mathcal{U} lnv, \Delta$ $\Gamma \Longrightarrow \mathcal{UV}(Inv \& b \doteq TRUE \rightarrow [p]Inv), \Delta$ $\Gamma \Longrightarrow \mathcal{UV}(Inv \& b \doteq FALSE \rightarrow [\pi \ \omega]\phi), \Delta$ (assumed after exit) $\Gamma \Longrightarrow \mathcal{U}[\pi \text{ while (b) } p \ \omega]\phi, \Delta$

(valid when entering loop) (preserved by p)

- Context is kept as far as possible
- Invariant not 'responsible' for un-assignable locations
- Missing assignable clause (equiv. to assignable \everything):
 - $\mathcal{V} = \{* := *\}$ wipes out **all** information
 - Equivalent to basic invariant rule
 - Avoid this! Always give a specific assignable clause

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

```
(Implicit) Class Invariant: a \neq null
```

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

```
(Implicit) Class Invariant: a \neq null
```

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

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

```
(Implicit) Class Invariant: a ≠ null
```

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

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

Loop invariant: $0 \le i \& i \le a.length$

```
(Implicit) Class Invariant: a \neq null
```

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

Postcondition: $\forall int x$; $(0 \le x < a.length \rightarrow a[x] \doteq 1)$

```
(Implicit) Class Invariant: a \neq null
```

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

Postcondition: $\forall \texttt{int } x; (0 \le x < \texttt{a.length} \rightarrow \texttt{a}[x] \doteq 1)$

(Implicit) Class Invariant: $a \neq \textbf{null}$ not needed for loop invariant

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

Postcondition: $\forall \texttt{int } x; (0 \le x < \texttt{a.length} \rightarrow \texttt{a}[x] \doteq 1)$

Example in JML/JAVA - Loop.java

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

SEFM: Selected Features/Loops & Methods

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

∀ int x;

$$(x \doteq n \land x >= 0 \rightarrow$$

[i = 0; r = 0;
while (i
r=r+r-n;
]r ≟ x * x)

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

∀ int x;

$$(x \doteq n \land x >= 0 \rightarrow$$

[i = 0; r = 0;
while (i
r=r+r-n;
]r ≟ 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 i, r;</pre>
```

∀ int x;

$$(x \doteq n \land x >= 0 \rightarrow$$

[i = 0; r = 0;
while (i
r=r+r-n;
]r ≟ 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 i, r;</pre>
```

File: Loop2.java

Hints

Proving assignable

- The invariant rule 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

Hints

Proving assignable

- The invariant rule 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

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;

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

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

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

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

Proving termination in JML/JAVA

- Remove directive diverges true;
- Add directive decreasing v; to loop invariant
- KeY uses suitable invariant rule and PO (with $\langle \ldots
 angle \phi$)

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

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

Proving termination in JML/JAVA

- Remove directive diverges true;
- Add directive decreasing v; to loop invariant
- KeY uses suitable invariant rule and PO (with $\langle \ldots
 angle \phi$)

Example (The array loop)

@ decreasing

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

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

Proving termination in JML/JAVA

- Remove directive diverges true;
- Add directive decreasing v; to loop invariant
- KeY uses suitable invariant rule and PO (with $\langle \ldots
 angle \phi$)

Example (The array loop)

@ decreasing a.length - i;

Total Correctness

Find a decreasing integer term v (called variant)

Add the following premisses to the invariant rule:

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

Proving termination in JML/JAVA

- Remove directive diverges true;
- Add directive decreasing v; to loop invariant
- KeY uses suitable invariant rule and PO (with $\langle \ldots
 angle \phi$)

Example (The array loop)

@ decreasing a.length - i;

Files:

- LoopT.java
- Loop2T.java

SEFM: Selected Features/Loops & Methods

CHALMERS/GU

Method Calls – Repetition

Method Call with actual parameters *arg*₀,..., *arg*_n

$$\{\operatorname{arg}_0 := t_0 || \dots || \operatorname{arg}_n := t_n || c := t_c\} \langle c.m(\operatorname{arg}_0, \dots, \operatorname{arg}_n); \rangle \phi$$

where m declared as void $m(T_0 p_0, \ldots, T_n p_n)$

Actions of rule methodCall

 for each formal parameter pi of m: declare and initialize new local variable Ti p#i = argi;

Method Calls – Repetition

Method Call with actual parameters arg_0, \ldots, arg_n

$$\{\operatorname{arg}_0 := t_0 || \dots || \operatorname{arg}_n := t_n || c := t_c\} \langle c.m(\operatorname{arg}_0, \dots, \operatorname{arg}_n); \rangle \phi$$

where m declared as void $m(T_0 p_0, \ldots, T_n p_n)$

Actions of rule methodCall

- for each formal parameter pi of m: declare and initialize new local variable Ti p#i = argi;
- look up implementation class C of m, or split proof if implementation cannot be uniquely determined

Method Calls – Repetition

Method Call with actual parameters *arg*₀,..., *arg*_n

$$\{arg_0 := t_0 || \dots || arg_n := t_n || c := t_c\} \langle c.m(arg_0, \dots, arg_n); \rangle \phi$$

where m declared as **void** m(T₀ p₀, ..., T_n p_n)

Actions of rule methodCall

- for each formal parameter p_i of m: declare and initialize new local variable T_i p#i =arg_i;
- look up implementation class C of m, or split proof if implementation cannot be uniquely determined
- ► create method invocation c.m(p#0,...,p#n)@C

Method Body Expand

- 1. Execute code that binds actual to formal parameters $T_i p \# i = arg_i$;
- 2. Call rule methodBodyExpand

$$\begin{split} \Gamma &\Rightarrow \langle \pi \text{ method-frame(source=C, this=c)} \{ \text{ body } \} \omega \rangle \phi, \Delta \\ \\ \Gamma &\Rightarrow \langle \pi \text{ c.m}(p \# 0, \dots, p \# n) @C; \omega \rangle \phi, \Delta \end{split}$$

Method Body Expand

- 1. Execute code that binds actual to formal parameters $T_i p \# i = arg_i$;
- 2. Call rule methodBodyExpand

$$\begin{split} \Gamma &\Rightarrow \langle \pi \text{ method-frame(source=C, this=c)} \{ \text{ body } \} \omega \rangle \phi, \Delta \\ \\ \Gamma &\Rightarrow \langle \pi \text{ c.m}(p \# 0, \dots, p \# n) @C; \omega \rangle \phi, \Delta \end{split}$$

Symbolic Execution

Only static information available, proof splitting;

Method Body Expand

- 1. Execute code that binds actual to formal parameters $T_i p \# i = arg_i$;
- 2. Call rule methodBodyExpand

$$\begin{split} \Gamma &\Rightarrow \langle \pi \text{ method-frame(source=C, this=c)} \{ \text{ body } \} \omega \rangle \phi, \Delta \\ \Gamma &\Rightarrow \langle \pi \text{ c.m}(p \# 0, \dots, p \# n) @C; \omega \rangle \phi, \Delta \end{split}$$

Symbolic Execution

Only static information available, proof splitting; Runtime infrastructure required in calculus

Method Body Expand

- 1. Execute code that binds actual to formal parameters $T_i p \# i = arg_i$;
- 2. Call rule methodBodyExpand

$$\begin{split} \Gamma &\Rightarrow \langle \pi \text{ method-frame(source=C, this=c)} \{ \text{ body } \} \omega \rangle \phi, \Delta \\ \\ \Gamma &\Rightarrow \langle \pi \text{ c.m}(p \# 0, \dots, p \# n) @C; \omega \rangle \phi, \Delta \end{split}$$

Symbolic Execution

Only static information available, proof splitting; Runtime infrastructure required in calculus

File: inlineDynamicDispatch.key

How to perform symbolic execution when JAVA API method is called?

1. Method has reference implementation in JAVA Inline method body and execute symbolically

How to perform symbolic execution when JAVA API method is called?

 Method has reference implementation in JAVA Inline method body and execute symbolically Problems Reference implementation not always available

How to perform symbolic execution when JAVA API method is called?

 Method has reference implementation in JAVA Inline method body and execute symbolically Problems Reference implementation not always available Too expensive

How to perform symbolic execution when JAVA API method is called?

 Method has reference implementation in JAVA Inline method body and execute symbolically
 Problems Reference implementation not always available Too expensive Impossible to deal with recursion

How to perform symbolic execution when JAVA API method is called?

 Method has reference implementation in JAVA Inline method body and execute symbolically
 Problems Reference implementation not always available Too expensive Impossible to deal with recursion

2. Use method contract instead of method implementation

Method Contract Rule – Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires normalPre;
  @ ensures normalPost;
  @ assignable mod;
  @*/
```

Method Contract Rule – Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires normalPre;
  @ ensures normalPost;
  @ assignable mod;
  @*/
```

 $\Gamma \Longrightarrow \mathcal{UF}(\texttt{normalPre}), \Delta \quad (\text{precondition})$

 $\Gamma \Longrightarrow \mathcal{U}\langle \pi \texttt{result} = \texttt{m}(\texttt{a}_1, \dots, \texttt{a}_n) \; \omega \rangle \phi, \Delta$

• $\mathcal{F}(\cdot)$: translation to Java DL (see last lecture)

Method Contract Rule – Normal Behavior Case

Warning: Simplified version

```
/*@ public normal_behavior
  @ requires normalPre;
  @ ensures normalPost;
  @ assignable mod;
  @*/
```

```
 \begin{array}{l} \Gamma \Longrightarrow \mathcal{UF}(\texttt{normalPre}), \Delta \quad (\texttt{precondition}) \\ \Gamma \Longrightarrow \mathcal{UV}_{mod}(\mathcal{F}(\texttt{normalPost}) \rightarrow \langle \pi \; \omega \rangle \phi), \Delta \quad (\texttt{normal}) \\ \Gamma \Longrightarrow \mathcal{U} \langle \pi \, \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_n) \; \omega \rangle \phi, \Delta \end{array}
```

- $\mathcal{F}(\cdot)$: translation to Java DL (see last lecture)
- ► *V_{mod}*: anonymising update (similar to loops)

Method Contract Rule – Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
  @ requires excPre;
  @ signals (Exception exc) excPost;
  @ assignable mod;
  @*/
```

Method Contract Rule – Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
    @ requires excPre;
    @ signals (Exception exc) excPost;
    @ assignable mod;
    @*/
```

 $\Gamma \Longrightarrow \mathcal{UF}(\texttt{excPre}), \Delta$ (precondition)

$$\mathsf{\Gamma} \Longrightarrow \mathcal{U} \langle \pi \, \texttt{result} = \mathtt{m}(\mathtt{a_1}, \dots, \mathtt{a_n}) \; \omega
angle \phi, \Delta$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- *V_{mod}*: anonymising update (similar to loops)

Method Contract Rule – Exceptional Behavior Case

Warning: Simplified version

```
/*@ public exceptional_behavior
  @ requires excPre;
  @ signals (Exception exc) excPost;
  @ assignable mod;
  @*/
```

$$\begin{split} \Gamma &\Rightarrow \mathcal{UF}(\texttt{excPre}), \Delta \quad (\texttt{precondition}) \\ \Gamma &\Rightarrow \mathcal{UV}_{mod}((\texttt{exc} \neq \texttt{null} \land \mathcal{F}(\texttt{excPost}))) \\ & \rightarrow \langle \pi \texttt{throw exc}; \omega \rangle \phi \rangle, \Delta \quad (\texttt{exceptional}) \\ \hline \Gamma &\Rightarrow \mathcal{U} \langle \pi \texttt{result} = \texttt{m}(\texttt{a}_1, \dots, \texttt{a}_n) \, \omega \rangle \phi, \Delta \end{split}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- ► *V_{mod}*: anonymising update (similar to loops)

Warning: Simplified version

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

Warning: Simplified version

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

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$egin{array}{rcl} \phi_{\textit{post_n}} &\equiv & \mathcal{F}(\texttt{\old}(\texttt{normalPre})) \wedge \mathcal{F}(\texttt{normalPost}) \ \phi_{\textit{post_e}} &\equiv & \mathcal{F}(\texttt{\old}(\texttt{excPre})) \wedge \mathcal{F}(\texttt{excPost}) \end{array}$$

Warning: Simplified version

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

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$egin{array}{rcl} \phi_{\textit{post_n}} &\equiv & \mathcal{F}(\texttt{\old}(\texttt{normalPre})) \wedge \mathcal{F}(\texttt{normalPost}) \ \phi_{\textit{post_e}} &\equiv & \mathcal{F}(\texttt{\old}(\texttt{excPre})) \wedge \mathcal{F}(\texttt{excPost}) \end{array}$$

 $\Gamma \Longrightarrow \mathcal{U}(\mathcal{F}(\texttt{normalPre}) \lor \mathcal{F}(\texttt{excPre})), \Delta \quad (\texttt{precondition})$

 $\mathsf{\Gamma} \Longrightarrow \mathcal{U}\langle \pi \, \texttt{result} = \mathtt{m}(\mathtt{a_1}, \dots, \mathtt{a_n}) \, \omega \rangle \phi, \Delta$

- $\mathcal{F}(\cdot)$: translation to Java DL
- *V_{mod}*: anonymising update (similar to loops)

Warning: Simplified version

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

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$egin{array}{rcl} \phi_{\textit{post_n}} &\equiv & \mathcal{F}(\texttt{\old}(\texttt{normalPre})) \wedge \mathcal{F}(\texttt{normalPost}) \ \phi_{\textit{post_e}} &\equiv & \mathcal{F}(\texttt{\old}(\texttt{excPre})) \wedge \mathcal{F}(\texttt{excPost}) \end{array}$$

$$\begin{split} &\Gamma \Longrightarrow \mathcal{U}(\mathcal{F}(\texttt{normalPre}) \lor \mathcal{F}(\texttt{excPre})), \Delta \quad (\texttt{precondition}) \\ &\Gamma \Longrightarrow \mathcal{U}_{\textit{mod}_{\textit{normal}}}(\phi_{\textit{post_n}} \to \langle \pi \: \omega \rangle \phi), \Delta \quad (\texttt{normal}) \end{split}$$

$$\mathsf{F} \Longrightarrow \mathcal{U}\langle \pi \, \texttt{result} = \mathtt{m}(\mathtt{a}_1, \dots, \mathtt{a}_n) \, \omega
angle \phi, \Delta$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- ► *V_{mod}*: anonymising update (similar to loops)

Warning: Simplified version

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

Therefore translation of postcondition ϕ_{post} as follows (simplified):

$$egin{array}{rcl} \phi_{\textit{post_n}} &\equiv & \mathcal{F}(\texttt{\old}(\texttt{normalPre})) \wedge \mathcal{F}(\texttt{normalPost}) \ \phi_{\textit{post_e}} &\equiv & \mathcal{F}(\texttt{\old}(\texttt{excPre})) \wedge \mathcal{F}(\texttt{excPost}) \end{array}$$

$$\begin{split} & \Gamma \Longrightarrow \mathcal{U}(\mathcal{F}(\texttt{normalPre}) \lor \mathcal{F}(\texttt{excPre})), \Delta \quad (\texttt{precondition}) \\ & \Gamma \Longrightarrow \mathcal{U}_{\textit{mod}_{\textit{normal}}}(\phi_{\textit{post_n}} \to \langle \pi \, \omega \rangle \phi), \Delta \quad (\texttt{normal}) \\ & \Gamma \Longrightarrow \mathcal{U}_{\textit{mod}_{\textit{exc}}}((\texttt{exc} \neq \texttt{null} \land \phi_{\textit{post_e}}) \\ & \to \langle \pi \, \texttt{throw} \, \texttt{exc}; \, \omega \rangle \phi), \Delta \quad (\texttt{exceptional}) \\ & \Gamma \Longrightarrow \mathcal{U} \langle \pi \, \texttt{result} = \texttt{m}(\texttt{a}_1, \dots, \texttt{a}_n) \, \omega \rangle \phi, \Delta \end{split}$$

- $\mathcal{F}(\cdot)$: translation to Java DL
- ► *V_{mod}*: anonymising update (similar to loops)

Understanding Proof Situations

Reasons why a proof may not close

- bug or incomplete specification
- bug in program
- ▶ maximal number of steps reached: restart or increase # of steps
- automatic proof search fails and manual rule applications necessary

Understanding Proof Situations

Reasons why a proof may not close

- bug or incomplete specification
- bug in program
- ▶ maximal number of steps reached: restart or increase # of steps
- automatic proof search fails and manual rule applications necessary

Understanding open proof goals

- follow the taken control-flow from the root to the open goal
- branch labels may give useful hints
- identify (part of) the post-condition or invariant that cannot be proven
- sequent remains always in "pre-state". I.e., constraints like i ≥ 0 refer to the value of i before executing the program (exception: sub-formulae prefixed by update or modality)
- ► remember: $\Gamma \Longrightarrow o \doteq \texttt{null}, \Delta$ is equivalent to $\Gamma, o \neq \texttt{null} \Longrightarrow \Delta$

- Most JAVA features covered in KeY
- Several of remaining features available in experimental version
 - Simplified multi-threaded JMM
 - Floats
- Degree of automation for loop-free programs is high
- Proving loops requires user to provide invariant
 - Automatic invariant generation sometimes possible
- Symbolic execution paradigm lets you use KeY w/o understanding details of logic

Essential

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 10: Using KeY

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 3: Dynamic Logic, Sections 3.1, 3.2, 3.4, 3.5, 3.6.1, 3.6.2, 3.6.3, 3.6.4, 3.6.5, 3.6.7, 3.7