Formal Specification and Verification
Reasoning about Programs with Loops

Bernhard Beckert

Based on a lecture by Wolfgang Ahrendt and Reiner Hähnle at Chalmers University, Göteborg
Symbolic execution of loops: unwind

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

Symbolic execution of loops: unwind

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

How to handle a loop with...

- 0 iterations?

- 1 iteration?

- 10 iterations?

- 10000 iterations?

- an unknown number of iterations?

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

Symbolic execution of loops: unwind

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

How to handle a loop with…

- 0 iterations? Unwind 1

- Unwind 1

- Unwind 11

- Unwind 10001

- (and don't make any plans for the rest of the day)

- Unwind an unknown number of iterations? We need an invariant rule (or some other form of induction)
Loop Invariants

Symbolic execution of loops: unwind

\[
\text{unwindLoop: } \begin{align*}
\Gamma &\Rightarrow U[\pi \text{ if } (b) \{ p; \text{ while } (b) p \} \omega] \phi, \Delta \\
\Gamma &\Rightarrow U[\pi \text{ while } (b) p \omega] \phi, \Delta
\end{align*}
\]

How to handle a loop with...

- 0 iterations? Unwind 1×
- 10 iterations?

(formal specification and verification: loops)
Symbolic execution of loops: unwind

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

How to handle a loop with…

- 0 iterations? Unwind 1×
- 10 iterations? Unwind 11×
Loop Invariants

Symbolic execution of loops: unwind

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

How to handle a loop with...

- 0 iterations? Unwind 1\times
- 10 iterations? Unwind 11\times
- 10000 iterations?

Formal Specification and Verification: Loops
Loop Invariants

Symbolic execution of loops: unwind

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

How to handle a loop with...

- 0 iterations? Unwind 1\times
- 10 iterations? Unwind 11\times
- 10000 iterations? Unwind 10001\times
  (and don’t make any plans for the rest of the day)
## Symbolic execution of loops: unwind

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

How to handle a loop with...

- 0 iterations? Unwind 1×
- 10 iterations? Unwind 11×
- 10000 iterations? Unwind 10001×
  (and don’t make any plans for the rest of the day)
- an **unknown** number of iterations?
Loop Invariants

Symbolic execution of loops: unwind

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

How to handle a loop with…

- 0 iterations? Unwind 1
- 10 iterations? Unwind 11
- 10000 iterations? Unwind 10001
  (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

**Idea behind loop invariants**

- A formula $\text{Inv}$ whose validity is *preserved* by loop guard and body
- **Consequence:** if $\text{Inv}$ was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then $\text{Inv}$ holds *afterwards*
- Encode the desired *postcondition* after loop into $\text{Inv}$
Loop Invariants Cont’d

Idea behind loop invariants

- A formula $\mathit{Inv}$ whose validity is preserved by loop guard and body
- **Consequence**: if $\mathit{Inv}$ was valid at start of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then $\mathit{Inv}$ holds afterwards
- Encode the desired **postcondition** after loop into $\mathit{Inv}$

Basic Invariant Rule

$$\text{loopInvariant} \quad \Gamma \implies U[\pi \text{ while (b)} \ p \omega] \phi, \Delta$$
Loop Invariants Cont’d

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

▶ Encode the desired postcondition after loop into $Inv$

Basic Invariant Rule

\[
\Gamma \Rightarrow UInv, \Delta \quad \text{(initially valid)}
\]

\[
\text{loopInvariant} \quad \Gamma \Rightarrow U[\pi \text{ while } (b) \ p \omega]\phi, \Delta
\]
Loop Invariants Cont’d

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
- Encode the desired postcondition after loop into $Inv$

Basic Invariant Rule

$$\Gamma \implies \mathcal{U}Inv, \Delta$$ (initially valid)

$$Inv, b \neq \text{TRUE} \implies [p]Inv$$ (preserved)

$$\text{loopInvariant} \quad \Gamma \implies \mathcal{U}[\pi \text{ while } (b) p \omega]\phi, \Delta$$
## Loop Invariants Cont’d

### 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**
- Encode the desired **postcondition** after loop into \( Inv \)

### Basic Invariant Rule

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

- \( Inv, b \models \text{TRUE} \Rightarrow [p] Inv \) (preserved)
- \( Inv, b \models \text{FALSE} \Rightarrow [\pi \omega] \phi \) (use case)
Basic Invariant Rule: Problem

\[
\begin{align*}
\text{loopInvariant} & \quad \Gamma \Rightarrow U \text{Inv}, \Delta \\
& \quad \text{(initially valid)} \\
\text{Inv}, \quad b \doteq \text{TRUE} & \quad \Rightarrow [p] \text{Inv} \\
& \quad \text{(preserved)} \\
\text{Inv}, \quad b \doteq \text{FALSE} & \quad \Rightarrow [\pi \omega] \phi \\
& \quad \text{(use case)} \\
\Gamma & \Rightarrow U [\pi \text{ while } (b) \ p \omega] \phi, \Delta
\end{align*}
\]
Loop Invariants Cont’d

Basic Invariant Rule: Problem

\[
\begin{align*}
\text{loopInvariant:} & \quad \Gamma \Rightarrow U \text{Inv}, \Delta \quad \text{(initially valid)} \\
& \quad \text{Inv, } b \doteq \text{TRUE} \Rightarrow [p] \text{Inv} \quad \text{(preserved)} \\
& \quad \text{Inv, } b \doteq \text{FALSE} \Rightarrow [\pi \omega] \phi \quad \text{(use case)} \\
\hline
& \Gamma \Rightarrow U[\pi \text{ while (b) p } \omega] \phi, \Delta
\end{align*}
\]

- Context \( \Gamma, \Delta, U \) must be omitted in 2nd and 3rd premise:
  - \( \Gamma, \Delta \) in general don’t hold in state defined by \( U \)
  - **2nd premise** \( \text{Inv} \) must be invariant for any state, not only \( U \)
  - **3rd premise** We don’t know the state after the loop exits
Loop Invariants Cont’d

Basic Invariant Rule: Problem

\[ \Gamma \Rightarrow \mathcal{U} Inv, \Delta \]  
\[ \text{(initially valid)} \]

\[ Inv, b \upmodels \text{TRUE} \Rightarrow [p] Inv \]  
\[ \text{(preserved)} \]

\[ Inv, b \upmodels \text{FALSE} \Rightarrow [\pi \omega] \phi \]  
\[ \text{(use case)} \]

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

- 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
### Basic Invariant Rule: Problem

<table>
<thead>
<tr>
<th>LoopInvariant</th>
<th>[ \Gamma \Rightarrow \mathcal{U} Inv, \Delta ] (initially valid)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[ Inv, b \doteq \text{TRUE} \Rightarrow [p]Inv ] (preserved)</td>
</tr>
<tr>
<td></td>
<td>[ Inv, b \doteq \text{FALSE} \Rightarrow [\pi \omega]\phi ] (use case)</td>
</tr>
<tr>
<td></td>
<td>[ \Gamma \Rightarrow \mathcal{U}[\pi \text{while (b) p \omega}]\phi, \Delta ]</td>
</tr>
</tbody>
</table>

- 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 $Inv$
int i = 0;
while (i < a.length) {
    a[i] = 1;
    i++;
}
Example

Precondition: !a ∉ null

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

Precondition: \( \lnot a \Rightarrow \text{null} \)

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a\.\text{length} \Rightarrow a[x] \Rightarrow 1) \)
Example

Precondition: \( \neg a \equiv \text{null} \)

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

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

Loop invariant: \( 0 \leq i \land i \leq a\text{.length} \)
Example

Precondition: \( !a \neq \text{null} \)

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

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

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

Precondition: !a != null

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

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

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

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

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

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

Loop invariant: \( 0 \leq i \& i \leq a.length \)
\& \( \forall \text{int } x; (0 \leq x < i \rightarrow a[x] \neq 1) \)
\& \!a \neq \text{null}
\& ClassInv'
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 $\Gamma$?
Keeping the Context

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

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

- How to erase all values of **assignable** locations in formula $\Gamma$?

  Analogous situation: $\forall$-Right quantifier rule $\implies \forall x; \phi$

  Replace $x$ with a fresh constant $*$

  To change value of program location use **update**, not substitution
Keeping the Context

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

\[ \text{@ assignable } i, a[*]; \]

- How to erase all values of *assignable* locations in formula \( \Gamma \)?
  - Analogous situation: \( \forall \)-Right quantifier rule \( \Rightarrow \forall x; \phi \)
  - Replace \( x \) with a fresh constant \( * \)
  - To change value of program location use *update*, not substitution
- *Anonymising updates* \( \forall \) erase information about modified locations

\[ \forall = \{ i := * || \text{for } x; a[x] := * \} \]
Loop Invariants Cont’d

Improved Invariant Rule

\[ \Gamma \Rightarrow U[\pi \text{ while } (b) \ p \ w] \phi, \Delta \]
Loop Invariants Cont’d

Improved Invariant Rule

\[ \Gamma \Rightarrow \mathcal{U} Inv, \Delta \quad \text{(initially valid)} \]

\[ \Gamma \Rightarrow \mathcal{U}[\pi \text{ while } (b) \ p \ \omega] \phi, \Delta \]
Improved Invariant Rule

\[ \Gamma \Rightarrow U Inv, \Delta \quad \text{(initially valid)} \]
\[ \Gamma \Rightarrow U (Inv \land b \equiv \text{TRUE} \Rightarrow [p] Inv), \Delta \quad \text{(preserved)} \]
\[ \Gamma \Rightarrow U [\pi \text{ while } (b) p \omega] \phi, \Delta \]
Improved Invariant Rule

\[
\begin{align*}
\Gamma &\Rightarrow \mathcal{U} \text{Inv}, \Delta \quad \text{(initially valid)} \\
\Gamma &\Rightarrow \mathcal{U} \text{V}(\text{Inv} \& b \doteq \text{TRUE} \rightarrow [p]\text{Inv}), \Delta \quad \text{(preserved)} \\
\Gamma &\Rightarrow \mathcal{U} \text{V}(\text{Inv} \& b \doteq \text{FALSE} \rightarrow [\pi \omega]\phi), \Delta \quad \text{(use case)} \\
\Gamma &\Rightarrow \mathcal{U}[\pi \text{while } (b) p \omega]\phi, \Delta
\end{align*}
\]
Loop Invariants Cont’d

**Improved Invariant Rule**

\[
\begin{align*}
\Gamma &\Rightarrow \mathcal{U} \text{Inv}, \Delta & \text{(initially valid)} \\
\Gamma &\Rightarrow \mathcal{U} \text{V}(\text{Inv} & b \equiv \text{TRUE} \rightarrow [p] \text{Inv}), \Delta & \text{(preserved)} \\
\Gamma &\Rightarrow \mathcal{U} \text{V}(\text{Inv} & b \equiv \text{FALSE} \rightarrow [\pi \omega] \phi), \Delta & \text{(use case)} \\
\Gamma &\Rightarrow \mathcal{U}[\pi \text{ while } (b) p \omega] \phi, \Delta
\end{align*}
\]

- Context is kept as far as possible
- Invariant does not need to include unmodified locations
- For *assignable* everything (the default):
  - \( \mathcal{V} = \{ * := * \} \) wipes out all information
  - Equivalent to basic invariant rule
  - Avoid this! Always give a specific assignable clause
Example with Improved Invariant Rule

```java
int i = 0;
while (i < a.length) {
    a[i] = 1;
    i++;
}
```
Example with Improved Invariant Rule

Precondition: !a ∉ null

```java
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
```
Example with Improved Invariant Rule

Precondition: !a \neq \text{null}

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a.length \rightarrow a[x] = 1) \)
Example with Improved Invariant Rule

Precondition: !a \neq \text{null}

\begin{verbatim}
int i = 0;
while(i < a.length) {
    a[i] = 1;
    i++;
}
\end{verbatim}

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

Loop invariant: \( 0 \leq i \land i \leq a.length \)
Example with Improved Invariant Rule

Precondition: \(!a \equiv \text{null}\)

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

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

Loop invariant: \(0 \leq i \& i \leq a\text{.length}\)

\(\& \forall \text{int } x; (0 \leq x < i \rightarrow a[x] \equiv 1)\)
Example with Improved Invariant Rule

Precondition: !a = null

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a\cdot\text{length} \rightarrow a[x] \equiv 1) \)

Loop invariant: \( 0 \leq i \ & \ i \leq a\cdot\text{length} \)
\& \( \forall \text{int } x; (0 \leq x < i \rightarrow a[x] \equiv 1) \)
Precondition: \( !a \neq \text{null} \& \text{ClassInv} \)

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

Postcondition: \( \forall \text{int } x; (0 \leq x < a\cdot \text{length} \rightarrow a[x] \neq 1) \)

Loop invariant: \( 0 \leq i \& i \leq a\cdot \text{length} \)
\& \( \forall \text{int } x; (0 \leq x < i \rightarrow a[x] \neq 1) \)
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 i, a[ ];
    @*/

  while(i < a.length) {
    a[i] = 1;
    i++;
  }
}
### 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;
Find a decreasing integer term $\nu$ (called variant)

Add the following premisses to the invariant rule:

- $\nu \geq 0$ is initially valid
- $\nu \geq 0$ is preserved by the loop body
- $\nu$ 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:

- $v \geq 0$ 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;`
- Add directive `decreasing v;` to loop invariant
- KeY creates suitable invariant rule and PO (with $\langle \ldots \rangle \phi$)
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 directive \texttt{diverges true;}
- Add directive \texttt{decreasing v;} to loop invariant
- KeY creates suitable invariant rule and PO (with \( \langle \ldots \rangle \phi \))

Example (Same loop as above)

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

Example (Same loop as above)

```
@ decreasing a.length - i;
```
Literature for this Lecture

Essential

**KeY Book** Verification of Object-Oriented Software (see course webpage), Chapter 3: Dynamic Logic (Section 3.7)