Testing, Debugging, Program Verification Formal Verification, Part II

Wolfgang Ahrendt & Vladimir Klebanov & Moa Johansson

3 December 2012

- Symbolic initial values.
 - E.g. set variable $x := x_0$ for new symbol x_0 .
- Arbitrary but fixed values.
- Represents a set of possible concrete values.
- Execution Tree with branches annotated by Path Conditions
- Symbolic run says something about a set of concrete runs.
 E.g. all runs where array a is non-null and has length >0.

Recap: State, State Update

- ► The symbolic execution state, denoted U, records values of variables.
- Updated as execution progresses.

Let \mathcal{U} and \mathcal{V} be updates, with $\mathcal{U} = (x := x_0)$, $\mathcal{V} = (x := 1)$.

- Sequential update: \mathcal{U}, \mathcal{V}
 - ▶ First apply U to state s, obtaining state U^s.
 - ▶ Then apply \mathcal{V} to \mathcal{U}^{s} .
 - What is the final value of x?

Recap: State, State Update

- ► The symbolic execution state, denoted U, records values of variables.
- Updated as execution progresses.

Let \mathcal{U} and \mathcal{V} be updates, with $\mathcal{U} = (x := x_0)$, $\mathcal{V} = (x := 1)$.

- Sequential update: \mathcal{U}, \mathcal{V}
 - ▶ First apply U to state s, obtaining state U^s.
 - ▶ Then apply \mathcal{V} to \mathcal{U}^{s} .
 - What is the final value of x?
- ▶ Parallel update: $\mathcal{U} \parallel \mathcal{V} = (x := x_0 \parallel x := 1)$
 - If same location updated, rightmost update wins.
 - Apply $\mathcal{U} \parallel \mathcal{V}$ to an intial state *s*.
 - What is the value of x afterwards?

We can turn sequential updates into parallel ones.

Recap: Hoare Triples (with updates)

Definition (Hoare Triple with Update)

A Hoare Triple with Update is an expression of the form

 $\{P\}\left[\mathcal{U}\right]\pi\left\{Q\right\}$

where P and Q are first-order formulas over locations appearing in the WHILE program π and ${\cal U}$ is an update.

Recap: Hoare Triples (with updates)

Definition (Hoare Triple with Update)

A Hoare Triple with Update is an expression of the form

 $\{P\} [\mathcal{U}] \pi \{Q\}$

where P and Q are first-order formulas over locations appearing in the $\rm WHILE$ program π and ${\cal U}$ is an update.

Definition (Truth of a Hoare Triple with Update in a State)

A Hoare triple $\{P\} [\mathcal{U}] \pi \{Q\}$ is true in state *s* when:

- ▶ If *P* is true in *s*, and
- π terminates when started in \mathcal{U}^{s} , then
- Q is true in the final state reached by π .

Assignment

assignment
$$\frac{\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}}{\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}}$$

Assignment

assignment
$$\frac{\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}}{\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}}$$

Example

$${P} [x := x_0, x := x + 17] \pi {Q} {P} [x := x_0] x = x + 17; \pi {Q}$$

TDV: Verification II

CHALMERS/GU

Assignment

assignment
$$\frac{\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}}{\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}}$$

Turn assignment into update and append sequentially

- Important that e has no side effects
- e can be evaluated as FOL term

Schematic rule: match against concrete update, program, etc.

Example

$$\{P\} [x := x_0, x := x + 17] \pi \{Q\}$$

$$\{P\} [x := x_0] x = x + 17; \pi \{Q\}$$

Assignment

assignment
$$\frac{\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}}{\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}}$$

Turn assignment into update and append sequentially

- Important that e has no side effects
- e can be evaluated as FOL term
- Schematic rule: match against concrete update, program, etc.
- Turn sequential into parallel update

Example

$$\{P\} [x := x_0 || x := x_0 + 17] \pi \{Q\}$$
$$\{P\} [x := x_0] x = x + 17; \pi \{Q\}$$

Assignment

assignment
$$\frac{\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}}{\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}}$$

Turn assignment into update and append sequentially

- Important that e has no side effects
- e can be evaluated as FOL term
- Schematic rule: match against concrete update, program, etc.
- Turn sequential into parallel update, then simplify

Example

$$\{P\} [x := x_0 + 17] \pi \{Q\}$$
$$\{P\} [x := x_0] x = x + 17; \pi \{Q\}$$

Rules of Calculus for Hoare Logic, Cont'd

Exit

exit
$$\frac{\vdash P \longrightarrow \mathcal{U}(Q)}{\{P\} [\mathcal{U}] \{Q\}}$$

- Applied when original program is fully symbolically executed
- "Precondition implies postcondition in final state of the original program, which is now summarized by U"
- ▶ The meaning of U(Q) is to apply U to Q:
 - ► If x := t is atomic update in U then replace each occurrence of x in Q with t
 - \blacktriangleright Assume that ${\cal U}$ is a parallel update

 \blacktriangleright Premiss is FOL formula, handed over to automated theorem prover \vdash

Rules assignment and exit suffice to do earlier example:

Example (swap)

{
$$x = x_0 \& y = y_0$$
}
[]
d = x; x = y; y = d;
{ $x = y_0 \& y = x_0$ }
Start with empty update []

Rules Used

assignment
$$\frac{\{P\}\left[\mathcal{U},\,\mathbf{x}:=\mathbf{e}\right]\pi\left\{Q\right\}}{\{P\}\left[\mathcal{U}\right]\mathbf{x}=\mathbf{e};\pi\left\{Q\right\}} \qquad \mathsf{exit} \ \frac{\vdash P \longrightarrow \mathcal{U}(Q)}{\{P\}\left[\mathcal{U}\right]\left\{Q\right\}}$$

Rules assignment and exit suffice to do earlier example:

Example (swap)

$$\{x = x_0 \& y = y_0\} \\ [d := x] \\ x = y; y = d; \\ \{x = y_0 \& y = x_0\} \end{cases}$$

Rules Used assignment $\frac{\{P\}\left[\mathcal{U}, \mathbf{x} := \mathbf{e}\right] \pi\{Q\}}{\{P\}\left[\mathcal{U}\right] \mathbf{x} = \mathbf{e}; \pi\{Q\}} \qquad \text{exit } \frac{\vdash P \longrightarrow \mathcal{U}(Q)}{\{P\}\left[\mathcal{U}\right] \{Q\}}$

Rules assignment and exit suffice to do earlier example:

Example (swap)

$$\{x = x_0 & y = y_0\} \\ [d := x || x := y] \\ y = d; \\ \{x = y_0 & y = x_0\}$$

Rules Used assignment $\frac{\{P\}\left[\mathcal{U}, \mathbf{x} := \mathbf{e}\right] \pi\{Q\}}{\{P\}\left[\mathcal{U}\right] \mathbf{x} = \mathbf{e}; \pi\{Q\}} \qquad \text{exit } \frac{\vdash P \longrightarrow \mathcal{U}(Q)}{\{P\}\left[\mathcal{U}\right] \{Q\}}$

Rules assignment and exit suffice to do earlier example:

Example (swap)

 $\{x = x_0 \& y = y_0\} \\ [(d := x || x := y), y := d]$

$$\{x = y_0 \& y = x_0\}$$

Empty program

Rules Usedassignment
$$\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}$$
exit $\vdash P \longrightarrow \mathcal{U}(Q)$ $\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}$ $exit \frac{\vdash P \longrightarrow \mathcal{U}(Q)}{\{P\} [\mathcal{U}] \{Q\}}$

Rules assignment and exit suffice to do earlier example:

Example (swap)

 $\{x = x_0 \& y = y_0\} \\ [d := x || x := y || y := x]$

$$\{x = y_0 \& y = x_0\}$$

Parallel update: use previous value of d!

Rules Used

$$\begin{array}{l} \text{assignment} \quad \displaystyle \frac{\{P\}\left[\mathcal{U},\,\mathbf{x}:=\mathbf{e}\right]\pi\left\{Q\right\}}{\{P\}\left[\mathcal{U}\right]\mathbf{x}\,=\,\mathbf{e}\,;\pi\left\{Q\right\}} \qquad \text{exit} \quad \displaystyle \frac{\vdash \ P \rightarrow \mathcal{U}(Q)}{\{P\}\left[\mathcal{U}\right]\left\{Q\right\}} \end{array} \end{array}$$

Rules assignment and exit suffice to do earlier example:

Example (swap)

$$\vdash (x = x_0 \& y = y_0) \rightarrow [d := x || x := y || y := x](x = y_0 \& y = x_0)$$
Exit

Rules Used
assignment
$$\frac{\{P\}\left[\mathcal{U}, \mathbf{x} := \mathbf{e}\right]\pi\{Q\}}{\{P\}\left[\mathcal{U}\right]\mathbf{x} = \mathbf{e};\pi\{Q\}} \qquad \text{exit } \frac{\vdash P \longrightarrow \mathcal{U}(Q)}{\{P\}\left[\mathcal{U}\right]\{Q\}}$$

Rules assignment and exit suffice to do earlier example:

Example (swap)

 $\vdash (x = x_0 \& y = y_0) \rightarrow (y = y_0 \& x = x_0)$

Apply update to postcondition — valid FOL formula!

Rules Usedassignment
$${\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}}{\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}}$$
exit $\vdash P \longrightarrow \mathcal{U}(Q)$ $\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}$ exit $\{P\} [\mathcal{U}] \{Q\}$

What is KeY-Hoare?

- ► Interactive software verification system for WHILE programs
- Uses Hoare Calculus with Updates
- Derived from KeY system for (almost) full JAVA and JML
- Symbolic execution rules must be applied "by hand"
- Display, navigate, and pretty-print formulas, proof trees
- Validity of FOL formulas, update simplification/application: automatic!
- System takes care of correctness: can't prove invalid Hoare triple!

KeY-Hoare: Input File Syntax

```
\functions {
  FirstOrderFunctionDeclaration*
    // initial values, user-defined functions
}
\programVariables {
  ProgramLocationDeclaration*
   // all locations appearing in Update, Program below
}
\hoare {
  { Precondition }
  [ Update ]
  \[{ // funny brackets needed for KeY-compatibility
      Program
 }\]
  { Postcondition }
}
```

KeY-Hoare: Demo

```
\functions {
  int x0; // one per line
  int y0;
}
\programVariables {
  int x, y, d;
}
\hoare {
  \{x = x0 \& y = y0 \} // can skip empty initial update
  \Γ{
      d = x; x = y; y = d;
 3/1
  \{ x = y0 \& y = x0 \}
}
```

Rules of Calculus for Hoare Logic, Cont'd

Conditional

 $\frac{\{P \& \mathcal{U}(b=TRUE)\}[\mathcal{U}] \pi_1 \rho \{Q\}}{\{P\} [\mathcal{U}] \text{ if } (b) \{\pi_1\} \text{ else} \{\pi_2\} \rho \{Q\}}$

Case distinction necessary, because value of b symbolic

- In general, Hoare calculus proofs are trees
- Important that b has no side effects
 - Can treat b as FOL Boolean term
- In premisses b must be evaluated in state \mathcal{U}
- U(b=TRUE) and U(b=FALSE) extend existing path condition P
 - Can simplify b=TRUE to b and b=FALSE to !b

```
Example (max)
{true}
[]
if (x > y) { res = x; } else { res = y; }
{???}
Postcondition?
```

Example (max)

{true}
[]
if (x > y) { res = x; } else { res = y; }
{(res = x | res = y) & res >= x & res >= y}

Next Rule Used

 $\frac{\{P \& \mathcal{U}(b=TRUE)\}[\mathcal{U}] \pi_1 \rho \{Q\} \qquad \{P \& \mathcal{U}(b=FALSE)\}[\mathcal{U}] \pi_2 \rho \{Q\}}{\{P\}[\mathcal{U}] \text{ if } (b) \{\pi_1\} \text{else} \{\pi_2\} \rho \{Q\}}$

Example (max)

{x > y}
[]
res = x;
{(res = x | res = y) & res >= x & res >= y}
Left premiss

Next Rules Used assignment $\frac{\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}}{\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}} \qquad \text{exit } \frac{\vdash P \longrightarrow \mathcal{U}(Q)}{\{P\} [\mathcal{U}] \{Q\}}$

Example (max)

{x > y} [res := x]

{(res = x | res = y) & res >= x & res >= y}

Next Rules Used assignment $\frac{\{P\} [\mathcal{U}, \mathbf{x} := \mathbf{e}] \pi \{Q\}}{\{P\} [\mathcal{U}] \mathbf{x} = \mathbf{e}; \pi \{Q\}} \qquad \text{exit } \frac{\vdash P \longrightarrow \mathcal{U}(Q)}{\{P\} [\mathcal{U}] \{Q\}}$

Example (max)

Example (max)

Example (max)

```
Example (max)
```

```
{!(x > y)}
[]
res = y;
{(res = x | res = y) & res >= x & res >= y}
Right premiss, similar as before
```

Demo: max.key

Difficulties of While Loops

- Need to "unwind" loop body one by one
- In general, no fixed loop bound known (depends on input)
- New mathematical principle needed! Can you guess which?

A Program With Loops

Example (Loop with fixed bound)

{true}
[]
i = 0;
n = 2;
while (i < n) {
 i = i + 1;
}
{i = n}</pre>

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
while (i < n) {
    i = i + 1;
}
{i = n}</pre>
```

A Program With Loops

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
while (i < n) {
    i = i + 1;
}
{i = n}</pre>
```

Possible Rule: Unwind $\frac{\{P\} [\mathcal{U}] \text{ if (b) } \{\pi \text{ while (b) } \{\pi\}\} \rho \{Q\}}{\{P\} [\mathcal{U}] \text{ while (b) } \{\pi\} \rho \{Q\}}$

TDV: Verification II

CHALMERS/GU
Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
if (i < n) {
    i = i + 1;
    while (i < n) {
        i = i + 1;
    }
}
{i = n}</pre>
```

Possible Rule: Unwind

$$\{P\} [\mathcal{U}] \text{ if (b) } \{\pi \text{ while (b) } \{\pi\}\} \rho \{Q\}$$
$$\{P\} [\mathcal{U}] \text{ while (b) } \{\pi\} \rho \{Q\}$$

TDV: Verification II

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
if (i < n) {
    i = i + 1;
    while (i < n) {
        i = i + 1;
    }
{    i = n}</pre>
```

Example (Loop with fixed bound)

```
{true}
[i := 0 || n := 2]
i = i + 1;
while (i < n) {
    i = i + 1;
}
{i = n}</pre>
```

Example (Loop with fixed bound)

```
{true}
[i := 1 || n := 2]
while (i < n) {
    i = i + 1;
}
{i = n}</pre>
```

Example (Loop with fixed bound)

```
{true}
[i := 1 || n := 2]
if (i < n) {
    i = i + 1;
    while (i < n) {
        i = i + 1;
}
{i = i + 1;
}
{i = n}</pre>
```

Unwind second time

$$\{P\} [\mathcal{U}] \text{ if (b) } \{\pi \text{ while (b) } \{\pi\}\} \rho \{Q\}$$
$$\{P\} [\mathcal{U}] \text{ while (b) } \{\pi\} \rho \{Q\}$$

TDV: Verification II

CHALMERS/GU

Example (Loop with fixed bound)

```
{true}
[i := 1 || n := 2]
if (i < n) {
    i = i + 1;
    while (i < n) {
        i = i + 1;
    }}
{i = n}</pre>
```

Example (Loop with fixed bound)

```
{true}
[i := 1 || n := 2]
i = i + 1;
while (i < n) {
    i = i + 1;
}
{i = n}</pre>
```

Example (Loop with fixed bound)

```
{true}
[i := 2 || n := 2]
while (i < n) {
    i = i + 1;
}
{i = n}</pre>
```

Example (Loop with fixed bound)

```
{true}
[i := 2 || n := 2]
if (i < n) {
    i = i + 1;
    while (i < n) {
        i = i + 1;
}
{
    i = i + 1;
}
{
    i = n}</pre>
```

Unwind third time

$$\{P\} [\mathcal{U}] \text{ if (b) } \{\pi \text{ while (b) } \{\pi\}\} \rho \{Q\}$$
$$\{P\} [\mathcal{U}] \text{ while (b) } \{\pi\} \rho \{Q\}$$

TDV: Verification II

CHALMERS/GU

Example (Loop with fixed bound)

```
{true}
[i := 2 || n := 2]
```

 $\{i = n\}$

Guard of conditional is false, else branch is empty

How to handle a loop with...

0 iterations?

How to handle a loop with...

▶ 0 iterations? Unwind 1×

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

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

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

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

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

How to handle a loop with...

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

Idea behind loop invariants

- ► A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was true at start state of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Make sure to include the desired postcondition after loop into Inv

Idea behind loop invariants

- ► A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was true at start state of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Make sure to include the desired postcondition after loop into Inv

Invariant Rule

$$\vdash P \longrightarrow \mathcal{U}(Inv) \qquad (initially valid)$$

loopInvariant
$$P$$
 [\mathcal{U}] while (b) $\{\pi\} \rho \{Q\}$

Idea behind loop invariants

- ► A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was true at start state of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Make sure to include the desired postcondition after loop into Inv

Invariant Rule $\begin{array}{c} \vdash P \longrightarrow \mathcal{U}(Inv) & \text{(initially valid)} \\ \{Inv \& b\} [] \pi \{Inv\} & \text{(preserved)} \\ \hline \\ IoopInvariant & \hline \\ \hline \{P\} [\mathcal{U}] \text{ while (b) } \{\pi\} \rho \{Q\} \end{array}$

Idea behind loop invariants

- ► A formula *Inv* whose validity is preserved by loop guard and body
- Consequence: if *Inv* was true at start state of the loop, then it still holds after arbitrarily many loop iterations
- If the loop terminates at all, then *Inv* holds afterwards
- Make sure to include the desired postcondition after loop into Inv

Invariant Rule $\begin{array}{c} \vdash P \rightarrow \mathcal{U}(Inv) & \text{(initially valid)} \\ \{Inv \& b\} [] \pi \{Inv\} & \text{(preserved)} \\ \frac{\{Inv \& !b\} [] \rho \{Q\}}{\{P\} [\mathcal{U}] \text{ while (b) } \{\pi\} \rho \{Q\}} & \text{(use case)} \end{array}$

Example (Unbounded Loop)

```
{n >= 0}
[]
i = 0;
while (i < n) {
    i = i + 1;
}
{i = n}</pre>
```

Example (Unbounded Loop)

```
{n >= 0}
[i := 0]
while (i < n) {
    i = i + 1;
}
{i = n}</pre>
```



Example (Unbounded Loop)

```
{n >= 0}
[i := 0]
while (i < n) {
    i = i + 1;
}
{i = n}
What is a suitable invariant?</pre>
```



TDV: Verification II

Example (Unbounded Loop)

Invariant Rule

$$\begin{array}{c} \vdash P \longrightarrow \mathcal{U}(Inv) & \text{(initially valid)} \\ \{Inv \& b\} [] \pi \{Inv\} & \text{(preserved)} \\ \{Inv \& !b\} [] \rho \{Q\} & \text{(use case)} \\ \hline \{P\} [\mathcal{U}] \text{ while (b) } \{\pi\} \rho \{Q\} \end{array}$$

TDV: Verification II

Example (Unbounded Loop)

Tryi<=n

Note that precondition P and U are missing in preserves/use case! Why?



How to Derive Loop Invariants without Magic?

Example (when Symbolic Execution at start of loop)

```
{n >= 0}
[i := 0]
while (i < n) { i = i + 1; }
{i = n}</pre>
```

Look at desired postcondition (i = n) What, in addition to negated guard (i >= n), is needed?

How to Derive Loop Invariants without Magic?

Example (when Symbolic Execution at start of loop)

```
{n >= 0}
[i := 0]
while (i < n) { i = i + 1; }
{i = n}</pre>
```

Look at desired postcondition (i = n) What, in addition to negated guard (i >= n), is needed? (i <= n)

How to Derive Loop Invariants without Magic?

Example (when Symbolic Execution at start of loop)

```
{n >= 0}
[i := 0]
while (i < n) { i = i + 1; }
{i = n}</pre>
```

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!</pre>

Obtaining Invariants by Strengthening

Example (Slightly changed spec)

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



Obtaining Invariants by Strengthening

Example (Slightly changed spec)

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

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? No!

(i = m) is neither preserved nor true at the start! Can we use something from the precondition or U?

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

Obtaining Invariants by Strengthening

Example (Slightly changed spec—Demo count.key)

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

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? No!

(i = m) is neither preserved nor true at the start! Can we use something from the precondition or U?

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

Why Does the Invariant Rule Work?

Induction Argument

We prove by induction over the number n of loop iterations that lnv 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

^aFor this reason we cannot use P or U in (preserved) and (use case)

Generalization

Example (Addition)

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

Generalization

Example (Addition)

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

Finding the invariant

First attempt: use postcondition x = x0 + y0
Example (Addition)

```
{ x = x0 & y = y0 & y0 >= 0 }
[]
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

Example (Addition)

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

Finding the invariant

What stays invariant?

Example (Addition)

```
{ x = x0 & y = y0 & y0 >= 0 }
[]
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: " δ " between x and x0 + y0

Example (Addition)

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

Checking the invariant

ls x + y = x0 + y0 a good invariant?

Example (Addition)

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

Checking the invariant

ls x + y = x0 + y0 a good invariant?

Holds in the beginning and is preserved by loop

Example (Addition)

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

Checking the invariant

ls 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

Example (Addition)

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

Strenghtening the invariant

Postcondition holds if y = 0

• Sufficient to add $y \ge 0$ to x + y = x0 + y0 & $y \le 0$

Example (Addition)

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

Demo addition3.key

Example (Fibonacci)

```
{n = n0 \& n > 0}
٢٦
x1 = 1;
x2 = 1;
while (n > 2) {
  x^2 = x^1 + x^2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = ?? }
```

How do we specify the result?

Example (Fibonacci)

```
\{n = n0 \& n > 0 \& fib(1) = 1 \& fib(2) = 1 \&
\forall int m; (m > 2 \rightarrow fib(m) = fib(m-1) + fib(m-2))}
٢٦
x1 = 1;
x^2 = 1;
while (n > 2) {
  x^2 = x^1 + x^2;
  x1 = x2 - x1;
  n = n - 1;
}
\{ x2 = fib(n0) \}
```

Introduce \function int fib(int);

Example (Fibonacci)

```
\{n = n0 \& n > 0 \& fib(1) = 1 \& fib(2) = 1 \&
 \forall int m; (m > 2 \rightarrow fib(m) = fib(m-1) + fib(m-2))}
٢٦
x1 = 1:
x^2 = 1:
while (n > 2) {
  x^2 = x^1 + x^2;
  x1 = x2 - x1;
  n = n - 1;
}
\{ x2 = fib(n0) \}
```

Loop invariant must express complex relation between loop and fib()!

Example (Fibonacci)

```
x1 = fib(1);
x2 = fib(2);
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
```

Simulate loop to discover pattern

x1 x2 n

Example (Fibonacci)

```
x1 = fib(1);
x2 = fib(2);
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
```

Simulate loop to discover pattern

#	x1	x2	n	
0	fib(1)	fib(2)	n0	

Example (Fibonacci)

```
x1 = fib(1);
x2 = fib(2);
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
```

Simulate loop to discover pattern

#	x1	x2	n
0	fib(1)	fib(2)	n0
1	fib(2)	fib(3)	n0 -

Example (Fibonacci)

```
x1 = fib(1);
x2 = fib(2);
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
```

Simulate loop to discover pattern

#	x1	x2	n
0	fib(1)	fib(2)	n0
1	fib(2)	fib(3)	n0 - 2
2	fib(3)	fib(4)	n0 - 2

Example (Fibonacci)

```
x1 = fib(1);
x2 = fib(2);
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
```

Simulate loop to discover pattern

#	x1	x2	n
0	fib(1)	fib(2)	n0
1	fib(2)	fib(3)	n0 -
2	fib(3)	fib(4)	n0 – 2

Partial result:

express argument of fib() as relation between n and n0

Example (Fibonacci)

```
x1 = fib(1);
x2 = fib(2);
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
```

Simulate loop to discover pattern

#	x1	x2	n
0	fib(1)	fib(2)	nO
1	fib(2)	fib(3)	n0 - 1
2	fib(3)	fib(4)	n0 - 2

Partial result: express argument of fib() as relation between n and n0 Conjecture: x1 = fib(n0-n+1) x2 = fib(n0-n+2)

```
{n = n0 & n > 0 & fib(1) = 1 & fib(2) = 1 &
  \forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2))}
[x1 := 1 || x2 := 1]
while (n > 2) {
  x2 = x1 + x2;
  x1 = x2 - x1;
  n = n - 1;
}
{ x2 = fib(n0) }
```

Definition of fib() not available in preserves case!

$$\frac{\vdash P \longrightarrow \mathcal{U}(\mathsf{Inv}) \quad \{\mathsf{Inv} \& b\} [] \pi \{\mathsf{Inv}\} \quad \{\mathsf{Inv} \& !b\} [] \rho \{Q\}}{\{P\} [\mathcal{U}] \text{ while } (b) \{\pi\} \rho \{Q\}}$$

TDV: Verification II

```
{n = n0 & n > 0 & F}
[x1 := 1 || x2 := 1]
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
{ x2 = fib(n0) }
```

Add definition F of fib() to invariant

lnv = (x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F)

```
{n = n0 & n > 0 & F}
[x1 := 1 || x2 := 1]
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
{ x2 = fib(n0) }
```

Does postcondition follow from *lnv* & n <= 2 ?

lnv = (x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F)

```
{n = n0 & n > 0 & F}
[x1 := 1 || x2 := 1]
while (n > 2) {
    x2 = x1 + x2;
    x1 = x2 - x1;
    n = n - 1;
}
{ x2 = fib(n0) }
```

Does postcondition follow from $lnv \& n \le 2$? lnv = (x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F)Yes, provided that $n \ge 2!$ Add this to lnv — now use case ok!

```
Example (Fibonacci, the "preserves" case)
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2 & n > 2}
[]
x2 = x1 + x2;
x1 = x2 - x1;
n = n - 1;
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2}
```

Perform symbolic execution, exit, and update simplification ... Five times andRight, all but 2nd case close After several andLeft, equations become applicable

Demo: fib.key

```
Example (Fibonacci, the "preserves" case)
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2 & n > 2}
[]
x2 = x1 + x2;
x1 = x2 - x1;
n = n - 1;
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n >= 2}
```

Perform symbolic execution, exit, and update simplification ... Five times andRight, all but 2nd case close After several andLeft, equations become applicable

Demo: fib.key

Example (Fibonacci, the "preserves" case, open subgoal)
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 2 ->
fib(n0-n+3) = fib(n0-n+1) + fib(n0-n+2) }

Look into definition of F

 $forall int m; (m > 2 \rightarrow fib(m) = fib(m-1) + fib(m-2))$

Example (Fibonacci, the "preserves" case, open subgoal)
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 2 ->
fib(n0-n+3) = fib(n0-n+1) + fib(n0-n+2) }

Look into definition of F — this looks ok, after all!

\forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2))
Instantiate m with n0-n+3

Example (Fibonacci, the "preserves" case, open subgoal)
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 2 ->
fib(n0-n+3) = fib(n0-n+1) + fib(n0-n+2) }

But need to prove that n0-n+3 > 2, so we add n0 >= n \forall int m; (m > 2 -> fib(m) = fib(m-1) + fib(m-2)) Instantiate m with n0-n+3

Example (Fibonacci, the "preserves" case, open subgoal)
{x1 = fib(n0-n+1) & x2 = fib(n0-n+2) & F & n > 2 ->
fib(n0-n+3) = fib(n0-n+1) + fib(n0-n+2) }

(Almost) final loop invariant

 $x1 = fib(n0-n+1) \& x2 = fib(n0-n+2) \& F \& n \ge 2 \& n0 \ge n$ Is preserved, but also initially valid?

Example (Fibonacci, the "initially valid" case)

Example (Fibonacci, the "initially valid" case)

n = n0 & n > 0 & F -> (1 = fib(1) & 1 = fib(2) & F & n >= 2 & n0 >= n}

After update and other simplification ...

Example (Fibonacci, the "initially valid" case, open subgoal)

|- n = n0 & n > 0 & ... -> n >= 2

Cannot be shown! $n \ge 2$ is too strong We get $n \ge 0$ from precondition and $n \le 2$ from negated guard. The critical case seems to be n = 1

Original invariant with $n \ge 2$

Weaken to n > 0. Too weak!

Works if we know that n = 1 can only occur if n0 = 1

Final Invariant

Some Tips 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
- 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?
- 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:
 - Does it need strengthening?
 - Can you add stuff from the precondition?
 - Try to express the relation between partial and final result

- 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:
 - Does it need strengthening?
 - Can you add stuff from the precondition?
 - Try to express the relation between partial and final result
- Simulate a few loop body executions to discover invariant patterns

- 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:
 - Does it need strengthening?
 - Can you add stuff from the precondition?
 - 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 precondition?

- 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:
 - Does it need strengthening?
 - Can you add stuff from the precondition?
 - 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 precondition?
- Several "rounds" of weakening/strengthening might be required

- 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:
 - Does it need strengthening?
 - Can you add stuff from the precondition?
 - 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 precondition?
- Several "rounds" of weakening/strengthening might be required
- Use the KeY-Hoare tool
 - Symbolic execution (of body), exit, update simplification, andRight
 - Look at open first-order goals: what is needed to make them closed?
 - After each change of the invariant make sure all cases are ok
 - Use the "pruning" mechanism to supply a new invariant

Summary

- Symbolic execution of loops by unwinding can only deal with fixed loop bounds
- In general some variant of induction is required to prove properties of programs with loops
- Invariant rule encodes induction over # of executed loop bodies
- Invariant rule has three parts:
 - The invariant must hold at the beginning of the loop
 - The invariant must be preserved by an arbitrary execution of the loop body provided that the guard is true
 - ► The negated guard plus the invariant imply the desired postcondition
- Loop invariants can be developed systematically
 - Start with the desired postcondition
 - Discover patterns through execution of a few loop bodies
 - Use strengthening, generalization, weakening
 - Use guidance by open first-order goals
- If you can't find a proof your program or contract might be wrong!

- 1. Proving termination of programs
- 2. Proving correctness of programs with arrays