Applications of Formal Verification

Functional Verification of Java Programs: Java Dynamic Logic

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2012
1. **JAVA CARD DL**

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. **A Calculus for 100% JAVA CARD**

5. Loop Invariants
   - Basic Invariant Rule
1. **JAVA CARD DL**

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. A Calculus for 100% JAVA CARD

5. Loop Invariants
   - Basic Invariant Rule
# Syntax and Semantics

## Syntax
- **Basis:** Typed first-order predicate logic
- **Modal operators** $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program $p$
- **Class definitions in background** (not shown in formulas)

## Semantics (Kripke)
Modal operators allow referring to the final state of $p$:
- $[p]F$: If $p$ terminates normally, then $F$ holds in the final state ("partial correctness")
- $\langle p \rangle F$: $p$ terminates normally, and $F$ holds in the final state ("total correctness")
Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program $p$
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of $p$:

- $[p]F$: If $p$ terminates normally, then $F$ holds in the final state ("partial correctness")
- $\langle p \rangle F$: $p$ terminates normally, and $F$ holds in the final state ("total correctness")
Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program $p$
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of $p$:

- $[p]F$: If $p$ terminates normally, then $F$ holds in the final state (“partial correctness”)
- $\langle p \rangle F$: $p$ terminates normally, and $F$ holds in the final state (“total correctness”)

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov – Applications of Formal Verification
Syntax and Semantics

Syntax

- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (JAVA CARD) program $p$
- Class definitions in background (not shown in formulas)

Semantics (Kripke)

Modal operators allow referring to the final state of $p$:

- $[p]F$: If $p$ terminates normally, then $F$ holds in the final state ("partial correctness")
- $\langle p \rangle F$: $p$ terminates normally, and $F$ holds in the final state ("total correctness")
Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

- Programs are “first-class citizens”
- Real Java syntax
Why Dynamic Logic?

- Transparency wrt target programming language
- **Encompasses Hoare Logic**
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

Hoare triple \( \{ \psi \} \, \alpha \, \{ \phi \} \) equiv. to DL formula \( \psi \rightarrow [\alpha] \phi \)
Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm

Not merely partial/total correctness:
- can employ programs for specification (e.g., verifying program transformations)
- can express security properties (two runs are indistinguishable)
- extension-friendly (e.g., temporal modalities)
Why Dynamic Logic?

- Transparency wrt target programming language
- Encompasses Hoare Logic
- More expressive and flexible than Hoare logic
- Symbolic execution is a natural interactive proof paradigm
Dynamic Logic Example Formulas

\[(\text{balance} \geq c \land \text{amount} > 0) \rightarrow \langle \text{charge(amount)}; \rangle \text{balance} > c\]

\[\langle x = 1; \rangle \langle \text{while (true) } \{ \} \rangle false\]
- Program formulas can appear nested

\[\forall \text{int } val; ((\langle p \rangle x \triangleq val) \leftrightarrow (\langle q \rangle x \triangleq val))\]
- p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

\[(\text{balance} \geq c \land \text{amount} > 0) \rightarrow \langle \text{charge(amount)}; \rangle \text{balance} > c\]

\[\langle x = 1; \rangle([\text{while (true) \{} \{}]false)\]

Program formulas can appear nested

\[\forall \text{int val}; ((\langle p \rangle x \vdash val) \leftrightarrow (\langle q \rangle x \vdash val))\]

p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

\[(\text{balance} \geq c \land \text{amount} > 0) \Rightarrow \\
\langle \text{charge(amount)}; \rangle \text{balance} > c\]

\[\langle x = 1; \rangle (\text{while (true) {}}) \text{false}\]

- Program formulas can appear nested

\[\forall \text{int val}; (\langle p \rangle x \doteq val) \leftrightarrow (\langle q \rangle x \doteq val)\]

- p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

\[(balance \geq c \land amount > 0) \rightarrow \langle \text{charge(amount)}; \rangle balance > c\]

\[\langle x = 1; \rangle ([\text{while (true) } \{} \} false)\]
- Program formulas can appear nested

\[\forall int val; ((\langle p \rangle x = val) \leftrightarrow (\langle q \rangle x = val))\]
- p, q equivalent relative to computation state restricted to x
Dynamic Logic Example Formulas

(balance \geq c \land \text{amount} > 0) \rightarrow 
\langle\text{charge(amount)}\rangle \text{balance} > c

\langle x = 1; \rangle ([\text{while (true) {} }] false)
- Program formulas can appear nested

\forall \text{int val}; ((\langle p \rangle x = val) \leftrightarrow (\langle q \rangle x = val))
- \( p, q \) equivalent relative to computation state restricted to \( x \)
Dynamic Logic Example Formulas

\[ a \neq \text{null} \rightarrow \]

\[ \forall \text{int} \ j; \ (j \geq 0 \land j < a.\text{length} \rightarrow \max \geq a[j]) \land \]

\[ (a.\text{length} > 0 \rightarrow \exists \text{int} \ j; \ (j \geq 0 \land j < a.\text{length} \land \max = a[j])) \]
Variables

- Logical variables disjoint from program variables
  - No quantification over program variables
  - Programs do not contain logical variables
  - “Program variables” actually non-rigid functions
Validity

A \text{JAVA CARD} DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas.
A JAVA CARD DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas.
1. JAVA CARD DL

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. A Calculus for 100% JAVA CARD

5. Loop Invariants
   - Basic Invariant Rule
Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% JAVA CARD

Loop Invariants
  • Basic Invariant Rule
Sequents and their Semantics

Syntax

\[ \psi_1, \ldots, \psi_m \Rightarrow \phi_1, \ldots, \phi_n \]

Antecedent

Succedent

where the \( \phi_i, \psi_i \) are formulae (without free variables)

Semantics

Same as the formula

\[ (\psi_1 \ & \ \cdots \ & \ \psi_m) \ \Rightarrow \ (\phi_1 \ | \ \cdots \ | \ \phi_n) \]
Sequents and their Semantics

Syntax

\[ \psi_1, \ldots, \psi_m \Rightarrow \phi_1, \ldots, \phi_n \]

where the \( \phi_i, \psi_i \) are formulae (without free variables)

Semantics

Same as the formula

\[ (\psi_1 \& \cdots \& \psi_m) \rightarrow (\phi_1 \mid \cdots \mid \phi_n) \]
Sequent Rules

General form

\[
\frac{Γ_1 \Rightarrow Δ_1 \quad \cdots \quad Γ_r \Rightarrow Δ_r}{Γ \Rightarrow Δ}
\]

Premisses

Conclusion

(rule_name)

\((r = 0 \text{ possible: closing rules})\)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Sequent Rules

General form

Premisses

Conclusion

\((r = 0 \text{ possible: closing rules})\)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Sequent Rules

General form

Premisses

\[ \Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r \]

Conclusion

\[ \Gamma \Rightarrow \Delta \]

\( r = 0 \) possible: closing rules

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Sequent Rules

General form

\[
\begin{array}{c}
\text{rule name} \\
\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r \\
\Gamma \Rightarrow \Delta \\
\end{array}
\]

Premisses

Conclusion

(\(r = 0\) possible: closing rules)

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Some Simple Sequent Rules

not_left
\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

imp_left
\[
\frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\]

close_goal
\[
\frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}
\]

close_by_true
\[
\frac{\Gamma \Rightarrow \text{true}, \Delta}{\Gamma \Rightarrow \text{true}, \Delta}
\]

close_by_true
\[
\frac{\Gamma \Rightarrow \text{true}, \Delta}{\Gamma \Rightarrow \text{true}, \Delta}
\]

all_left
\[
\frac{\Gamma, \forall t x; \phi, \{x/e\}\phi \Rightarrow \Delta}{\Gamma, \forall t x; \phi \Rightarrow \Delta}
\]

where \(e\) var-free term of type \(t' < t\)
Some Simple Sequent Rules

\[
\text{not_left} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

\[
\text{imp_left} \quad \frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\]

\[
\text{close_goal} \quad \frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}
\]

\[
\text{close_by_true} \quad \frac{\Gamma \Rightarrow \text{true}, \Delta}{\Gamma \Rightarrow \text{true}, \Delta}
\]

\[
\text{all_left} \quad \frac{\Gamma, \forall t x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t x; \phi \Rightarrow \Delta}
\]

where \( e \) is a var-free term of type \( t' < t \)
Some Simple Sequent Rules

not_left
\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

imp_left
\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\]

\[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, B \Rightarrow \Delta}
\]

close_goal
\[
\frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, \forall t x; \phi, \{x/e\} \phi \Rightarrow \Delta}
\]

where \(e\) var-free term of type \(t' \prec t\)
Some Simple Sequent Rules

- **not_left**: \[ \Gamma \Rightarrow A, \Delta \quad \Gamma, ! A \Rightarrow \Delta \]

- **imp_left**: \[ \Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta \quad \Gamma, A \rightarrow B \Rightarrow \Delta \]

- **close_goal**: \[ \Gamma, A \Rightarrow A, \Delta \]

- **close_by_true**: \[ \Gamma \Rightarrow true, \Delta \]

- **all_left**: \[ \Gamma, \forall t x; \phi, \{x/e\}\phi \Rightarrow \Delta \]

  where \( e \) var-free term of type \( t' < t \)
Some Simple Sequent Rules

\[
\text{not_left} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \lnot A \Rightarrow \Delta}
\]

\[
\text{imp_left} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta} \quad \frac{\Gamma, B \Rightarrow \Delta}{\Gamma, \Delta}
\]

\[
\text{close_goal} \quad \frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow \Delta}
\]

\[
\text{close_by_true} \quad \frac{\Gamma \Rightarrow \text{true}, \Delta}{\Gamma}
\]

\[
\text{all_left} \quad \frac{\Gamma, \forall t x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t x; \phi \Rightarrow \Delta}
\]

where \(e\) var-free term of type \(t' \prec t\)
Sequent Calculus Proofs

Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed
Sequent Calculus Proofs

Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed
Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed
Sequent Calculus Proofs

Proof tree

- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed
1. JAVA CARD DL

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. A Calculus for 100% JAVA CARD

5. Loop Invariants
   - Basic Invariant Rule
Teil

1. Java Card DL

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. A Calculus for 100% Java Card

5. Loop Invariants
   - Basic Invariant Rule
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

- Sequent rules execute symbolically the active statement
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

\[ l: \{ \text{try} \{ \ i=0; \ j=0; \ \} \ \text{finally} \{ \ k=0; \ \} \} \]

- Sequent rules execute symbolically the active statement
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

```java
l:{try{ i=0; j=0; } finally{ k=0; }}
```

- Sequent rules execute symbolically the active statement
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

\[
\begin{align*}
1: & \{ \text{try} \{ & i=0; \quad j=0; \} \quad \text{finally} \{ & k=0; \} \} \\
\pi & \quad \omega
\end{align*}
\]

- Passive prefix: \( \pi \)
- Active statement: \( i=0; \)
- Rest: \( \omega \)

Sequent rules execute symbolically the active statement
Proof by Symbolic Program Execution

- Sequent rules for program formulas?
- What corresponds to top-level connective in a program?

The Active Statement in a Program

```
l:{try{ i=0; j=0; } finally{ k=0; }}
```

- Passive prefix: \( \pi \)
- Active statement: \( i=0; \)
- Rest: \( \omega \)

- Sequent rules execute symbolically the active statement
Rules for Symbolic Program Execution

If-then-else rule
\[
\begin{align*}
\Gamma, B = true & \implies \langle p \; \omega \rangle \phi, \Delta \\
\Gamma, B = false & \implies \langle q \; \omega \rangle \phi, \Delta
\end{align*}
\]
\[\Gamma \implies \langle if \; (B) \; \{ \; p \; \} \; else \; \{ \; q \; \} \; \omega \rangle \phi, \Delta\]

Complicated statements/expressions are simplified first, e.g.
\[\Gamma \implies \langle v=y; \; y=y+1; \; x=v; \; \omega \rangle \phi, \Delta\]
\[\Gamma \implies \langle x=y++; \; \omega \rangle \phi, \Delta\]

Simple assignment rule
\[\Gamma \implies \{ loc := val \} \langle \omega \rangle \phi, \Delta\]
\[\Gamma \implies \langle loc=val; \; \omega \rangle \phi, \Delta\]
Rules for Symbolic Program Execution

If-then-else rule

\[ \Gamma, B = \text{true} \implies \langle p \hspace{1em} \omega \rangle \phi, \Delta \]
\[ \Gamma, B = \text{false} \implies \langle q \hspace{1em} \omega \rangle \phi, \Delta \]
\[ \Gamma \implies \langle \text{if} \ (B) \ { p } \ \text{else} \ { q } \ \omega \rangle \phi, \Delta \]

Complicated statements/expressions are simplified first, e.g.

\[ \Gamma \implies \langle v:y; y=y+1; x=v; \omega \rangle \phi, \Delta \]
\[ \Gamma \implies \langle x=y++; \ \omega \rangle \phi, \Delta \]

Simple assignment rule

\[ \Gamma \implies \{ \text{loc} := \text{val} \} \langle \omega \rangle \phi, \Delta \]
\[ \Gamma \implies \langle \text{loc}=\text{val}; \ \omega \rangle \phi, \Delta \]
Rules for Symbolic Program Execution

If-then-else rule

\[ \Gamma, B = \text{true} \Rightarrow \langle p \ \omega \rangle \phi, \Delta \quad \Gamma, B = \text{false} \Rightarrow \langle q \ \omega \rangle \phi, \Delta \]

\[ \Gamma \Rightarrow \langle \text{if } (B) \ { p } \ \text{else} \ { q } \ \omega \rangle \phi, \Delta \]

Complicated statements/expressions are simplified first, e.g.

\[ \Gamma \Rightarrow \langle v=y; \ y=y+1; \ x=v; \ \omega \rangle \phi, \Delta \]

\[ \Gamma \Rightarrow \langle x=y++; \ \omega \rangle \phi, \Delta \]

Simple assignment rule

\[ \Gamma \Rightarrow \{ \text{loc := val} \} \langle \omega \rangle \phi, \Delta \]

\[ \Gamma \Rightarrow \langle \text{loc=val; } \ \omega \rangle \phi, \Delta \]
Treating Assignment with “Updates”

Updates

explicit syntactic elements in the logic

Elementary Updates

\[ \{ \text{loc} := \text{val} \} \phi \]

where (roughly)

- \text{loc} is a program variable \( x \), an attribute access \( o.\text{attr} \), or an array access \( a[i] \)
- \text{val} is same as \text{loc}, or a literal, or a logical variable

Parallel Updates

\[ \{ \text{loc}_1 := t_1 \; || \; \cdots \; || \; \text{loc}_n := t_n \} \phi \]

no dependency between the \( n \) components (but ‘right wins’ semantics)
Treating Assignment with “Updates”

Updates

explicit syntactic elements in the logic

Elementary Updates

\{ \text{loc} := \text{val} \} \, \phi

where (roughly)

- \text{loc} is a program variable \(x\), an attribute access \(o.at\)tr, or an array access \(a[i]\)
- \text{val} is same as \text{loc}, or a literal, or a logical variable

Parallel Updates

\{ \text{loc}_1 := t_1 \mid \cdots \mid \text{loc}_n := t_n \} \, \phi

no dependency between the \(n\) components (but ‘right wins’ semantics)
Treating Assignment with “Updates”

**Updates**

explicit syntactic elements in the logic

**Elementary Updates**

\[
\{ \text{loc} := \text{val} \} \phi
\]

where (roughly)

- \( \text{loc} \) is a program variable \( x \), an attribute access \( o.\text{attr} \), or an array access \( a[i] \)
- \( \text{val} \) is same as \( \text{loc} \), or a literal, or a logical variable

**Parallel Updates**

\[
\{ \text{loc}_1 := t_1 \parallel \cdots \parallel \text{loc}_n := t_n \} \phi
\]

no dependency between the \( n \) components (but ‘right wins’ semantics)
Why Updates?

Updates are:

- *lazily applied* (i.e., substituted into postcondition)
- *eagerly parallelised* + simplified

Advantages

- no renaming required
- delayed/minimized proof branching (efficient aliasing treatment)
Why Updates?

Updates are:
- *lazily applied* (i.e., substituted into postcondition)
- *eagerly parallelised* + simplified

Advantages
- no renaming required
- delayed/minimized proof branching (efficient aliasing treatment)
Symbolic Execution with Updates
(by Example)

\[
\begin{align*}
x < y & \implies x < y \\
\vdots \\
x < y & \implies \{x := y \parallel y := x\} \langle \rangle y < x \\
\vdots \\
x < y & \implies \{t := x \parallel x := y \parallel y := x\} \langle \rangle y < x \\
\vdots \\
x < y & \implies \{t := x \parallel x := y\} \{y := t\} \langle \rangle y < x \\
\vdots \\
x < y & \implies \{t := x\} \{x := y\} \langle y = t; \rangle y < x \\
\vdots \\
x < y & \implies \{t := x\} \langle x = y; y = t; \rangle y < x \\
\vdots \\
\implies x < y & \rightarrow \langle \text{int } t = x; x = y; y = t; \rangle y < x
\end{align*}
\]
Symbolic Execution with Updates
(by Example)

\[ x < y \implies x < y \]

\[ x < y \implies \{ x := y \parallel y := x \} \langle \rangle \ y < x \]

\[ x < y \implies \{ t := x \parallel x := y \parallel y := x \} \langle \rangle \ y < x \]

\[ x < y \implies \{ t := x \parallel x := y \} \{ y := t \} \langle \rangle \ y < x \]

\[ x < y \implies \{ t := x \} \{ x := y \} \{ y := t \} \langle y = t; \rangle \ y < x \]

\[ x < y \implies \{ t := x \} \langle x = y; \ y = t; \rangle \ y < x \]

\[ \implies x < y \rightarrow \langle \text{int} \ t = x; \ x = y; \ y = t; \rangle \ y < x \]
Symbolic Execution with Updates (by Example)

\[
x < y \implies x < y
\]
\[
\vdots
\]
\[
x < y \implies \{x:=y \parallel y:=x\}\{y < x \}
\]
\[
\vdots
\]
\[
x < y \implies \{t:=x \parallel x:=y \parallel y:=x\}\{y < x \}
\]
\[
\vdots
\]
\[
x < y \implies \{t:=x \parallel x:=y\}\{y:=t\}\{y < x \}
\]
\[
\vdots
\]
\[
x < y \implies \{t:=x\}\{x:=y\}\{y=t; \}\{y < x \}
\]
\[
\vdots
\]
\[
x < y \implies \{t:=x\}\{x:=y; y=t; \}\{y < x \}
\]
\[
\vdots
\]
\[
\implies x < y \rightarrow \langle \text{int } t=x; x=y; y=t; \} \{y < x \}
\]
Symbolic Execution with Updates
(by Example)

\[ x < y \implies x < y \]

\[ \vdots \]

\[ x < y \implies \{ x:=y \parallel y:=x \} y < x \]

\[ \vdots \]

\[ x < y \implies \{ t:=x \parallel x:=y \parallel y:=x \} y < x \]

\[ \vdots \]

\[ x < y \implies \{ t:=x \parallel x:=y \} \{ y:=t \} y < x \]

\[ \vdots \]

\[ x < y \implies \{ t:=x \} \{ x:=y \} \{ y=t \} y < x \]

\[ \vdots \]

\[ x < y \implies \{ t:=x \} \{ x:=y \} \{ y=t \} y < x \]

\[ \implies x < y \implies \{ int \ t=x; x=y; y=t; \} y < x \]
Symbolic Execution with Updates (by Example)

\[
x < y \implies x < y
\]

\[
x < y \implies \{ x := y \mid y := x \} \langle \rangle \ y < x
\]

\[
x < y \implies \{ t := x \mid x := y \mid y := x \} \langle \rangle \ y < x
\]

\[
x < y \implies \{ t := x \mid x := y \} \{ y := t \} \langle \rangle \ y < x
\]

\[
x < y \implies \{ t := x \} \{ x := y \} \langle y = t; \rangle \ y < x
\]

\[
x < y \implies \{ t := x \} \langle x = y; \ y = t; \rangle \ y < x
\]

\[
\implies x < y \rightarrow \langle \text{int } t = x; \ x = y; \ y = t; \rangle \ y < x
\]
Symbolic Execution with Updates
(by Example)

\[ x < y \implies x < y \]

\[ \vdots \]

\[ x < y \implies \{ x := y \parallel y := x \}\langle \rangle y < x \]

\[ \vdots \]

\[ x < y \implies \{ t := x \parallel x := y \parallel y := x \}\langle \rangle y < x \]

\[ \vdots \]

\[ x < y \implies \{ t := x \parallel x := y \}\{ y := t \}\langle \rangle y < x \]

\[ \vdots \]

\[ x < y \implies \{ t := x \}\{ x := y \}\langle y = t; \rangle y < x \]

\[ \vdots \]

\[ x < y \implies \{ t := x \}\langle x = y; \ y = t; \rangle y < x \]

\[ \implies x < y \implies \langle \text{int} \ t = x; \ x = y; \ y = t; \rangle y < x \]
Symbolic Execution with Updates (by Example)

\[
\begin{aligned}
x < y & \implies x < y \\
& \quad \vdots \\
x < y & \implies \{x := y \parallel y := x\} y < x \\
& \quad \vdots \\
x < y & \implies \{t := x \parallel x := y \parallel y := x\} y < x \\
& \quad \vdots \\
x < y & \implies \{t := x \parallel x := y\}\{y := t\} y < x \\
& \quad \vdots \\
x < y & \implies \{t := x\}\{x := y\}\{y=t;\} y < x \\
& \quad \vdots \\
x < y & \implies \{t := x\}\{x=y; \ y=t;\} y < x \\
& \quad \vdots \\
\end{aligned}
\]

\[
\quad \implies x < y \rightarrow \langle \text{int} \ t=x; \ x=y; \ y=t; \rangle y < x
\]
Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example

TRY-THROW

\[
\Gamma \Rightarrow \begin{cases}
\text{if (exc instanceof T)} & \{\text{try } \{e=\text{exc}; r\} \text{ finally } \{s\}\} \phi, \Delta \\
\text{else} & \{s \text{ throw exc;}\} \omega
\end{cases}
\]

\[
\Gamma \Rightarrow \langle \text{try}\{\text{throw exc}; q\} \text{ catch}(T\ e)\{r\} \text{ finally}\{s\} \omega \rangle \phi, \Delta
\]
Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example

TRY-THROW

\[
\Gamma \Rightarrow \begin{cases} 
\text{if (exc instanceof T)} & \{ \text{try } \{ e=exc; \ r \} \ \text{finally } \{ s \} \} \phi, \Delta \\
\text{else } \{ s \ \text{throw exc;} \} & \omega
\end{cases}
\]

\[
\Gamma \Rightarrow \langle \text{try} \{ \text{throw exc;} \ q \} \ \text{catch} (T \ e) \{ r \} \ \text{finally} \{ s \} \ \omega \rangle \phi, \Delta
\]
Handling Abrupt Termination

- Abrupt termination handled by program transformations
- Changing control flow = rearranging program parts

Example

TRY-THROW

$$
\Gamma \Rightarrow (\pi \text{ if (exc instanceof T)}$$

{try {e=exc; r} finally {s}}$$

\phi, \Delta$$

else {s throw exc; }$$

\omega$$

$$
\Gamma \Rightarrow (\pi \text{ try{throw exc; q} catch(T e){r} finally{s}} \omega$$

$$
\phi, \Delta$$

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov – Applications of Formal Verification

SS 2012 23/38
1. JAVA CARD DL
2. Sequent Calculus
3. Rules for Programs: Symbolic Execution
4. A Calculus for 100% JAVA CARD
5. Loop Invariants
   - Basic Invariant Rule
Teil

1. JAVA CARD DL
2. Sequent Calculus
3. Rules for Programs: Symbolic Execution
4. A Calculus for 100% JAVA CARD
5. Loop Invariants
   - Basic Invariant Rule
Supported Java Features

- method invocation with polymorphism/dynamic binding
- object creation and initialisation
- arrays
- abrupt termination
- throwing of NullPointerExceptions, etc.
- bounded integer data types
- transactions

All JAVA CARD language features are fully addressed in KeY
Supported Java Features

- method invocation with polymorphism/dynamic binding
- object creation and initialisation
- arrays
- abrupt termination
- throwing of NullPointerExceptions, etc.
- bounded integer data types
- transactions

All JAVA CARD language features are fully addressed in KeY
Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

Pro: Feature needs not be handled in calculus
Contra: Modified source code
Example in KeY: Very rare: treating inner classes
Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

**Pro:** Flexible, easy to implement, usable  
**Contra:** Not expressive enough for all features  
**Example in KeY:** Complex expression eval, method inlining, etc., etc.
Java—A Language of Many Features

Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

Pro: No logic extensions required, enough to express most features
Contra: Creates difficult first-order POs, unreadable antecedents
Example in KeY: Dynamic types and branch predicates
Ways to deal with Java features

- Program transformation, up-front
- Local program transformation, done by a rule on-the-fly
- Modeling with first-order formulas
- Special-purpose extensions of program logic

**Pro:** Arbitrarily expressive extensions possible

**Contra:** Increases complexity of all rules

**Example in KeY:** Method frames, updates
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Components of the Calculus

1. **Non-program rules**
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. **Rules for reducing/simplifying the program (symbolic execution)**
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. **Rules for handling loops**
   - using loop invariants
   - using induction

4. **Rules for replacing a method invocations by the method’s contract**

5. **Update simplification**
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Components of the Calculus

1. Non-program rules
   - first-order rules
   - rules for data-types
   - first-order modal rules
   - induction rules

2. Rules for reducing/simplifying the program (symbolic execution)
   Replace the program by
   - case distinctions (proof branches) and
   - sequences of updates

3. Rules for handling loops
   - using loop invariants
   - using induction

4. Rules for replacing a method invocations by the method’s contract

5. Update simplification
Loop Invariants

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?

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

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?

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

Symbolic execution of loops: unwind

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

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

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

### Symbolic execution of loops: unwind

<table>
<thead>
<tr>
<th>Symbolic Execution</th>
<th>Unwind Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \Gamma \Rightarrow U[\pi \text{if}(b) { p; \text{while}(b) p } \omega] \phi, \Delta ]</td>
<td>[ \Gamma \Rightarrow U[\pi \text{while}(b) p \omega] \phi, \Delta ]</td>
</tr>
</tbody>
</table>

### How to handle a loop with...

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

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

Symbolic execution of loops: unwind

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

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

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

Symbolic execution of loops: unwind

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

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

Symbolic execution of loops: unwind

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

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

Symbolic execution of loops: unwind

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

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

Symbolic execution of loops: unwind

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

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 \( Inv \) 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

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

\[\text{loopInvariant}\]

\[
\begin{align*}
\Gamma \Rightarrow U \, Inv, \Delta \\
Inv, b \neq \text{TRUE} \Rightarrow [p] Inv \\
Inv, b \neq \text{FALSE} \Rightarrow [\pi \omega] \phi
\end{align*}
\]

(initially valid)

(preserved)

(use case)
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

\[
\begin{align*}
\Gamma & \Rightarrow U Inv, \Delta \\
Inv, b \models \text{TRUE} & \Rightarrow [p] Inv \\
Inv, b \models \text{FALSE} & \Rightarrow [\pi \omega] \phi \\
\Gamma & \Rightarrow U[\pi \text{while}(b) \ p \ \omega] \phi, \Delta
\end{align*}
\]

(initially valid)  
(preserved)  
(use case)
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

\[
\Gamma \iff U \quad Inv, \quad \Delta \\
\begin{align*}
Inv, \ b &\vdash TRUE \iff [p]Inv \\
Inv, \ b &\vdash FALSE \iff [\pi \omega]\phi
\end{align*}
\]  

(Initially valid)  
(Preserved)  
(Use Case)
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 \( \text{Inv} \)

Basic Invariant Rule

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

(initially valid) (preserved) (use case)
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 \( \text{Inv} \)

Basic Invariant Rule

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

\[
\text{Inv}, \; b \doteq \text{TRUE} \Rightarrow [p] \text{Inv} \quad \text{(preserved)}
\]

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

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

Basic Invariant Rule: Problem

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

\[
\text{Inv}, \ b \models \text{TRUE} \implies [p]\text{Inv} \quad \text{(preserved)}
\]

\[
\text{Inv}, \ b \models \text{FALSE} \implies [\pi \omega]\phi \quad \text{(use case)}
\]

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

- Context \( \Gamma, \Delta, \mathcal{U} \) must be omitted in 2nd and 3rd premise
- \textbf{But:} context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant \( \text{Inv} \)
### Basic Invariant Rule: Problem

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

- Context \( \Gamma, \Delta, \mathcal{U} \) must be omitted in 2nd and 3rd premise
- *But:* context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant \( Inv \)
Basic Invariant Rule: Problem

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

- Context \( \Gamma, \Delta, U \) must be omitted in 2nd and 3rd premise
- \textit{But}: context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant \text{Inv}
Basic Invariant Rule: Problem

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

\[
\text{Inv}, b = \text{TRUE} \Rightarrow \lbrack p \rbrack \text{Inv} \quad \text{(preserved)}
\]

\[
\text{Inv}, b = \text{FALSE} \Rightarrow \lbrack \pi \omega \rbrack \phi \quad \text{(use case)}
\]

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

- Context \( \Gamma, \Delta, \mathcal{U} \) must be omitted in 2nd and 3rd premise
- \textit{But:} context contains (part of) precondition and class invariants
- Required context information must be added to loop invariant \text{Inv}
Example

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Precondition: \( a \neq \text{null} \& \text{ClassInv} \)

\[
\text{int } i = 0; \\
\text{while}(i < a.\text{length}) \{ \\
\hspace{1em} a[i] = 1; \\
\hspace{1em} i++; \\
\}
\]

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

Loop invariant: \( 0 \leq i \& i \leq a.\text{length} \) \\
\& \( \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \) \\
\& \( a \neq \text{null} \) \\
\& ClassInv'
Want to keep part of the context that is *unmodified* by loop
assignable clauses for loops can tell what might be modified

```plaintext
@ 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[*];
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[*];
Example with Improved Invariant Rule

```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++;
}
```
Example with Improved Invariant Rule

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

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

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

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

```
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: $0 \leq i \& i \leq a.length$
Example with Improved Invariant Rule

Precondition: $a \neq \texttt{null}$

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

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

Loop invariant: $0 \leq i \land i \leq a.\text{length}$

$\land \forall \texttt{int} \; x; \; (0 \leq x < i \rightarrow a[x] = 1)$
Example with Improved Invariant Rule

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

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

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

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

Precondition: \( a \neq null \) & \( ClassInv \)

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

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

Loop invariant: \( 0 \leq i \) & \( i \leq a.length \)
& \( \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \)
Example in JML/Java – Loop.java

```java
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++;
    }
}
```
Example

∀ int x;
(n \div x \land x \geq 0 \rightarrow
[i = 0; r = 0;
   \textbf{while} (i < n) \{ i = i + 1; r = r + i;\}
   r = r + r - n;
] r \div ?)

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

Solution:

@ \texttt{loop\_invariant}
@ i \geq 0 \&\& 2 \times r == i \times (i + 1) \&\& i \leq n;
@ \texttt{assignable} i, r;

File: \texttt{Loop2.java}
Example

\[ \forall \text{int } x; \]
\[ (n \div x \land x \geq 0 \rightarrow \]
\[ [i = 0; \ r = 0; \]
\[ \text{while } (i < n) \{ i = i + 1; \ r = r + i; \} \]
\[ r = r + r - n; \]
\[ ]r \div x \times x) \]

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

Solution:

@ loop_invariant
@ i\geq0 \&\& 2\times r == i \times (i + 1) \&\& i \leq n;
@ assignable i, r;

File: Loop2.java
Example

\[ \forall \text{int } x; \]
\[ (n \div x \land x \geq 0 \rightarrow \]
\[ [i = 0; \ r = 0; \]
\[ \text{while } (i < n) \{ i = i + 1; \ r = r + i; \} \]
\[ r = r + r - n; \]
\[ ]r \div x \times x) \]

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

Solution:

@ \textit{loop\_invariant}
@ i \geq 0 \&\& 2 \times r == i \times (i + 1) \&\& i \leq n;
@ \textit{assignable} i, r;

File: Loop2.java
Example

\forall \text{int } x;
(n \div x \land x \geq 0 \rightarrow
\begin{align*}
i &= 0; \
r &= 0; \\
\text{while} (i < n) \{ i = i + 1; \
r &= r + i; \\
r &= r + r - n;
\}
\] 
[r \div x \times x)
\]

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

Solution:

@ loop_invariant
@ i \geq 0 \land 2 \times r = i \times (i + 1) \land i \leq n;
@ assignable i, r;

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`.

**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;`
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 \textit{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: The \texttt{array} loop

\( @ \texttt{decreasing} \)
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 \( \langle \ldots \rangle \phi \))

Example: The *array loop*

\[ \texttt{decreasing} \]
Total Correctness

Find a decreasing integer term $v$ (called \textit{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: The \texttt{array} loop
\begin{verbatim}
@ decreasing
\end{verbatim}
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 \texttt{diverges true;}
- Add directive \texttt{decreasing v;} to loop invariant
- KeY creates suitable invariant rule and PO (with \( \langle \ldots \rangle \phi \))

Example: The \texttt{array} loop

\texttt{@ decreasing a.length - i;}
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 \( \langle . . . \rangle \phi \))

Example: The array loop

@ `decreasing a.length - i;`

Files:
- `LoopT.java`
- `Loop2T.java`