Deductive Verification of Object-Oriented Software

Part B

Bernhard Beckert | VTSA, 24.–28.08.2015
Part III

Program Verification with Dynamic Logic

7. JAVA CARD DL

8. Sequent Calculus

9. Rules for Programs: Symbolic Execution

10. A Calculus for 100% JAVA CARD

11. Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
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, then $F$ holds in the final state
  (partial correctness)
- $\langle p \rangle F$: $p$ terminates 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, then $F$ holds in the final state (partial correctness)
- $\langle p \rangle \ F$: $p$ terminates 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, then $F$ holds in the final state (partial correctness)
- $\langle p \rangle F$: $p$ terminates 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, then $F$ holds in the final state  
  (partial correctness)
- $<p> F$: $p$ terminates 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

\[(\text{balance} \geq c & \text{ amount} > 0) \rightarrow \text{<charge(amount);> balance > c}\]

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

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

(balance >= c & amount > 0) -->
<charge(amount);> balance > c

<x = 1;> ([while (true) {}] false)

Program formulas can appear nested

\forall int val; ((<p> x = val) <-> (<q> x = val))

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

(balance \geq c \&\&\& amount > 0) \rightarrow \\
<\text{charge(amount)};> balance > c

<x = 1;>([\text{while (true) {}]} false)

- Program formulas can appear nested

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

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

(balance >= c & amount > 0) --> <charge(amount);> balance > c

<x = 1;> ([while (true) {}] false)

Program formulas can appear nested

\forall int val; ((<p>x \equiv val) \leftrightarrow (<q>x \equiv val))

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

(balance \geq c \land amount > 0) \rightarrow \langle 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 \stackrel{\text{def}}{=} val) \leftrightarrow (\langle q\rangle x \stackrel{\text{def}}{=} val))

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

\[ a \neq \text{null} \rightarrow \]
\[ \forall \text{int } j; \ (j \geq 0 \& j < a.length \rightarrow \ max \geq a[j]) \] &\n\[ \exists \text{int } j; \ (j \geq 0 \& j < a.length \& \ max = a[j]) \]
Variables

Logical variables disjoint from program variables

- No quantification over program variables
- Programs do not contain logical variables
- “Program variables” actually non-rigid functions
Rigid and Flexible Terms

Example

\begin{align*}
\langle \text{int } i; \rangle \forall \text{int } x; (i + 1 \equiv x \rightarrow \langle i++; \rangle (i \equiv x))
\end{align*}

- Interpretation of $i$ depends on computation state $\Rightarrow$ flexible
- Interpretation of $x$ and $+$ do not depend on state $\Rightarrow$ rigid

Locations are always flexible

Logical variables, standard functions are always rigid
Rigid and Flexible Terms

Example

\( \langle \text{int } i; \rangle \forall \text{int } x; (i + 1 \equiv x \rightarrow \langle i++; \rangle (i \equiv x)) \)

- Interpretation of \( i \) depends on computation state \( \Rightarrow \) flexible
- Interpretation of \( x \) and + do not depend on state \( \Rightarrow \) rigid

Locations are always flexible
Logical variables, standard functions are always rigid
Rigid and Flexible Terms

Example

\[ \text{<int } i; > \forall \text{int } x; (i + 1 \div x \rightarrow <i++;> (i \div x)) \]

- Interpretation of \( i \) depends on computation state \( \Rightarrow \) flexible
- Interpretation of \( x \) and \( + \) do not depend on state \( \Rightarrow \) rigid

Locations are always flexible
Logical variables, standard functions are always rigid
Rigid and Flexible Terms

Example

\[
\langle \text{int } i; \rangle \forall \text{int } x; (i + 1 \equiv x \rightarrow \langle \text{i++;} \rangle (i \equiv x))
\]

- Interpretation of \( i \) depends on computation state \( \Rightarrow \) flexible
- Interpretation of \( x \) and \( + \) do not depend on state \( \Rightarrow \) rigid

Locations are always flexible
Logical variables, standard functions are always rigid
Rigid and Flexible Terms

Example

\<int i;> \\forall int x; (i + 1 \div x \rightarrow <i++;>(i \div x))\)

- Interpretation of \(i\) depends on computation state \(\Rightarrow\) flexible
- Interpretation of \(x\) and \(+\) do not depend on state \(\Rightarrow\) rigid

Locations are always flexible
Logical variables, standard functions are always rigid
A 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
Part III

Program Verification with Dynamic Logic

7. JAVA CARD DL

8. Sequent Calculus

9. Rules for Programs: Symbolic Execution

10. A Calculus for 100% JAVA CARD

11. Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7. JAVA CARD DL

8. Sequent Calculus

9. Rules for Programs: Symbolic Execution

10. A Calculus for 100% JAVA CARD

11. Taclets – KeY’s Rule Description Language
Sequents and their Semantics

Syntax

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

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

Semantics

Same as the formula

\[
(\psi_1 \land \cdots \land \psi_m) \implies (\phi_1 \mid \cdots \mid \phi_n)
\]
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

\[
(\psi_1 \& \cdots \& \psi_m) \Rightarrow (\phi_1 \| \cdots \| \phi_n)
\]
Sequent Rules

General form

Premisses

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

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
Sequent Rules

General form

\[
\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r \\
\Downarrow \\
\Gamma \Rightarrow \Delta
\]

(rule_name)

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

\[ \frac{\Gamma_1 \Rightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Rightarrow \Delta_r}{\Gamma \Rightarrow \Delta} \]

(rule_name)

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

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

(rule_name)

(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  \[
\frac{\Gamma \Rightarrow A, \Delta}{\Gamma, !A \Rightarrow \Delta}
\]

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

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

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

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

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

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

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

**not_left**

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

**imp_left**

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

**close_goal**

\[
\Gamma, A \Rightarrow A, \Delta
\]

**close_by_true**

\[
\Gamma \Rightarrow true, \Delta
\]

**all_left**

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

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

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

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

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

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

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

where e var-free term of type $t' < t$
Some Simple Sequent Rules

\[
\text{not}_{-}\text{left} \quad \frac{\Gamma \Rightarrow A, \Delta}{\Gamma, \neg A \Rightarrow \Delta}
\]

\[
\text{imp}_{-}\text{left} \quad \frac{\Gamma \Rightarrow A, \Delta \quad \Gamma, B \Rightarrow \Delta}{\Gamma, A \rightarrow B \Rightarrow \Delta}
\]

\[
\text{close}_{-}\text{goal} \quad \frac{\Gamma, A \Rightarrow A, \Delta}{\Gamma, A \Rightarrow A, \Delta}
\]

\[
\text{close}_{-}\text{by}_{-}\text{true} \quad \frac{\Gamma \Rightarrow \text{true}, \Delta}{\Gamma \Rightarrow \text{true}, \Delta}
\]

\[
\text{all}_{-}\text{left} \quad \frac{\Gamma, \forall t \, x; \phi, \{x/e\} \phi \Rightarrow \Delta}{\Gamma, \forall t \, x; \phi \Rightarrow \Delta}
\]

where \( e \) var-free term of type \( t' \prec t \)
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
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
Part III

Program Verification with Dynamic Logic

7. JAVA CARD DL

8. Sequent Calculus

9. Rules for Programs: Symbolic Execution

10. A Calculus for 100% JAVA CARD

11. Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7 JAVA CARD DL

8 Sequent Calculus

9 Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
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

```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

```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

1:{try{ i=0; j=0; } finally{ k=0; }}

passive prefix π
active statement i=0;
rest ω

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

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

- 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

\[
\Gamma, B = \text{true} \implies <p \omega> \phi, \Delta \quad \Gamma, B = \text{false} \implies <q \omega> \phi, \Delta
\]

\[
\Gamma \implies <\text{if} (B) \{ p \} \text{ else } \{ q \} \omega> \phi, \Delta
\]

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

\[
\Gamma \implies <v=y; y=y+1; x=v; \omega> \phi, \Delta
\]

\[
\Gamma \implies <x=y++; \omega> \phi, \Delta
\]

Simple assignment rule

\[
\Gamma \implies \{ \text{loc} := \text{val} \} <\omega> \phi, \Delta
\]

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

If-then-else rule

\[ \Gamma, B = true \implies <p \omega > \phi, \Delta \]
\[ \Gamma, B = false \implies <q \omega > \phi, \Delta \]
\[ \Gamma \implies <\text{if} (B) \{ \ p \ \} \ \text{else} \{ \ q \ \} \omega > \phi, \Delta \]

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

\[ \Gamma \implies <v=y; \ y=y+1; \ x=v; \omega > \phi, \Delta \]
\[ \Gamma \implies <x=y++; \omega > \phi, \Delta \]

Simple assignment rule

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

If-then-else rule

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

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

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

\[ \Gamma \Rightarrow <\text{v}=\text{y}; \ \text{y}=\text{y}+1; \ \text{x}=\text{v}; \ \omega> \phi, \Delta \]

\[ \Gamma \Rightarrow <\text{x}=\text{y}++; \ \omega> \phi, \Delta \]

Simple assignment rule

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

\[ \Gamma \Rightarrow <\text{loc}=\text{val}; \ \omega> \phi, \Delta \]
Updates

explicit syntactic elements in the logic

Elementary Updates

\{loc := val\} \phi

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

Parallel Updates

\{loc_1 := t_1 \mid \cdots \mid loc_n := t_n\} \phi

no dependency between the \(n\) components (but ‘right wins’ semantics)
Extending DL by Explicit State Updates

Updates

explicit syntactic elements in the logic

Elementary Updates

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

where (roughly)

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

Parallel Updates

\[ \{ loc_1 := t_1 \mid \cdots \mid loc_n := t_n \} \phi \]

no dependency between the \( n \) components (but ‘right wins’ semantics)
Extending DL by Explicit State Updates

**Updates**
explicit syntactic elements in the logic

**Elementary Updates**

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

where (roughly)
- \text{loc} 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)
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)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\[ x < y \Rightarrow x < y \]

\[ x < y \Rightarrow \{x := y \parallel y := x\} \not\Rightarrow y < x \]

\[ x < y \Rightarrow \{t := x \parallel x := y \parallel y := x\} \not\Rightarrow y < x \]

\[ x < y \Rightarrow \{t := x \parallel x := y\}\{y := t\} \not\Rightarrow y < x \]

\[ x < y \Rightarrow \{t := x\}\{x := y\} < y = t; > y < x \]

\[ x < y \Rightarrow \{t := x\} y = t; > y < x \]

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

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

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

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

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

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

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

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

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

\[
\vdots
\]

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

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

\[ \vdots \]

\[ x < y \implies \{x:=y \parallel y:=x\} \not\sim y < x \]

\[ \vdots \]

\[ x < y \implies \{t:=x \parallel x:=y \parallel y:=x\} \not\sim y < x \]

\[ \vdots \]

\[ x < y \implies \{t:=x \parallel x:=y\}\{y:=t\} \not\sim y < x \]

\[ \vdots \]

\[ x < y \implies \{t:=x\}\{x:=y\}\langle y=t;\rangle \not\sim y < x \]

\[ \vdots \]

\[ x < y \implies \{t:=x\}\langle x=y; y=t;\rangle \not\sim y < x \]

\[ \implies x < y \rightarrow \langle \text{int } t=x; x=y; y=t;\rangle \not\sim y < x \]
Local program variables
Modeled as non-rigid constants

Heap
Modeled with theory of arrays:

\[
\text{heap} : \rightarrow \text{Heap} \quad \text{(the heap in the current state)} \\
\text{select} : \text{Heap} \times \text{Object} \times \text{Field} \rightarrow \text{Any} \\
\text{store} : \text{Heap} \times \text{Object} \times \text{Field} \times \text{Any} \rightarrow \text{Heap}
\]

Heap axioms (excerpt)

\[
\begin{align*}
\text{select}(\text{store}(h, o, f, x), o, f) &= x \\
\text{select}(\text{store}(h, o, f, x), u, f) &= \text{select}(h, u, f) \text{ if } o \neq u
\end{align*}
\]
Program State Representation

Local program variables
Modeled as non-rigid constants

Heap
Modeled with theory of arrays:

\[
\begin{align*}
heap & : \rightarrow \text{Heap} \text{ (the heap in the current state)} \\
select & : \text{Heap} \times \text{Object} \times \text{Field} \rightarrow \text{Any} \\
store & : \text{Heap} \times \text{Object} \times \text{Field} \times \text{Any} \rightarrow \text{Heap}
\end{align*}
\]

Heap axioms (excerpt)

\[
\begin{align*}
select (store (h, o, f, x), o, f) &= x \\
select (store (h, o, f, x), u, f) &= select (h, u, f) \text{ if } o \neq u
\end{align*}
\]
Program State Representation

Local program variables
Modeled as non-rigid constants

Heap
Modeled with theory of arrays:

\[ \text{heap} : \rightarrow \text{Heap} \text{ (the heap in the current state)} \]
\[ \text{select} : \text{Heap} \times \text{Object} \times \text{Field} \rightarrow \text{Any} \]
\[ \text{store} : \text{Heap} \times \text{Object} \times \text{Field} \times \text{Any} \rightarrow \text{Heap} \]

Heap axioms (excerpt)

\[ \text{select} (\text{store}(h, o, f, x), o, f) = x \]
\[ \text{select} (\text{store}(h, o, f, x), u, f) = \text{select}(h, u, f) \text{ if } o \neq u \]
Handling Abrupt Termination

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

Example

TRY-THROW

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

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

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

Example

TRY-THROW

\[
\Gamma \Rightarrow \left\langle \begin{array}{l}
\text{if (exc instanceof T)} \\
\{\text{try } e = \text{exc}; r \} \text{ finally } \{s\}
\end{array} \right\rangle \phi, \Delta
\]

\[
\text{else } \{s \text{ throw exc;}\} \quad \omega
\]

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

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

Example

TRY-THROW

\[ \Gamma \Rightarrow \left\langle \pi \text{ if (exc instanceof T)} \right. \right. \]

\[ \{ \text{try } \{ \text{e=exc; r} \} \text{ finally } \{s\} \} \phi, \Delta \]

\[ \text{else } \{ s \text{ throw exc; } \} \ \omega \]

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

Program Verification with Dynamic Logic

7. JAVA CARD DL

8. Sequent Calculus

9. Rules for Programs: Symbolic Execution

10. A Calculus for 100% JAVA CARD

11. Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

1. JAVA CARD DL
2. Sequent Calculus
3. Rules for Programs: Symbolic Execution
4. A Calculus for 100% JAVA CARD
5. Taclets – KeY’s Rule Description Language
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
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:** Flexible, easy to implement, usable
**Contra:** Not expressive enough for all features
**Example in KeY:** Complex expression eval, method inlining, etc., etc.
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
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:** 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’s invocation 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’s invocation 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’s invocation 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’s invocation 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’s invocation by the method’s contract

5. Update simplification
Part III

Program Verification with Dynamic Logic

7. JAVA CARD DL

8. Sequent Calculus

9. Rules for Programs: Symbolic Execution

10. A Calculus for 100% JAVA CARD

11. Taclets – KeY’s Rule Description Language
Part III

Program Verification with Dynamic Logic

7  JAVA CARD DL

8  Sequent Calculus

9  Rules for Programs: Symbolic Execution

10 A Calculus for 100% JAVA CARD

11 Taclets – KeY’s Rule Description Language
Taclets:
KeY’s Rule Description Language

Taclets ...

- represent sequent calculus rules in KeY
- use a simple text-based format
- are descriptive, but with operational flavor
- are *not* a tactic metalanguage
Taclet Syntax

\[
\text{andLeft} \quad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta}
\]

Taclet

\[
\text{andLeft} \{ \\
\quad \text{\find ( A & B ==> )} \\
\quad \text{\replacewith ( A, B ==>)}
\};
\]

- **Unique name**
- **Find expression:**
  - Formula (Term) to be modified
  - Sequent arrow ==> formula must occur top-level and on the corresponding side of the sequent
- **Goal Description:** describes new sequent
Taclet Syntax

\[
\begin{align*}
\text{andLeft} & \quad \Gamma, A, B \Rightarrow \Delta \\
\beta & \quad \Gamma, A \& B \Rightarrow \Delta
\end{align*}
\]

### Taclet

```latex
\text{andLeft} \{
\begin{align*}
\text{\textbackslash find} \ (A \& B ==>) \\
\text{\textbackslash replacewith} \ (A, B ==>)
\end{align*}
\}
```

- **Unique name**
- **Find expression:**
  - **Formula (Term) to be modified**
  - **Sequent arrow ==>** formula must occur top level and on the corresponding side of the sequent.
- **Goal Description:** describes new sequent
Taclet Syntax

\[
\text{andLeft} \quad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta}
\]

Taclet

\[
\text{andLeft} \{ \\
\text{\ find \ ( A \ & \ B \ ==>) } \\
\text{\ replacewith \ ( A, B ==>) }
\}
\]

- Unique name
- Find expression:
  - Formula (Term) to be modified
  - Sequent arrow ==> formula must occur top level and on the corresponding side of the sequent.
- Goal Description: describes new sequent
Taclet Syntax

\[
\text{andLeft} \quad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta}
\]

Taclet

\begin{verbatim}
andLeft {
    \find ( A & B ==> )
    \replacewith ( A, B ==>)
};
\end{verbatim}

- Unique name
- Find expression:
  - Formula (Term) to be modified
  - Sequent arrow ==> formula must occur top level \textit{and} on the corresponding side of the sequent.

Goal Description: describes new sequent
Taclet Syntax

\[
\text{andLeft} \quad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \& B \Rightarrow \Delta}
\]

Taclet

\[
\text{andLeft} \{ \\
\quad \text{\textbackslash find ( A \& B ==> )} \\
\quad \text{\textbackslash replacewith ( A, B ==>)}
\}
\]

- Unique name
- Find expression:
  - Formula (Term) to be modified
  - Sequent arrow ==> formula must occur top level and on the corresponding side of the sequent.
- Goal Description: describes new sequent
Some rules are only sound in a certain context

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

Taclet Syntax

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

Taclet

\[
\text{modusPonens} \{
\text{\texttt{\textbackslash \texttt{assumes ( A ==> )}}}
\text{\texttt{\textbackslash \texttt{find ( A -> B ==> )}}}
\text{\texttt{\textbackslash \texttt{replacewith( B ==> ))}}}
\};
\]
Taclet Syntax

Some rules are only sound in a certain context

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

Taclet

\[
\text{modusPonens} \{ \\
\text{\textbackslash{}assumes ( A ==> )} \\
\text{\textbackslash{}find ( A \rightarrow B ==> )} \\
\text{\textbackslash{}replacewith( B ==> )} \\
\}
\]
Taclet Syntax

Some rules are only sound in a certain context

modusPonens

\[ \Gamma, A, B \implies \Delta \]
\[ \Gamma, A, A \rightarrow B \implies \Delta \]

Taclet

modusPonens {
\textbf{\textbackslash assumes} ( A ==> ) \\
\textbf{\textbackslash find} ( A -> B ==> ) \\
\textbf{\textbackslash replacewith} ( B ==> )
}

;
**Taclet Syntax**

**Proof Splitting: andRight**

\[
\Gamma \Rightarrow A, \Delta \quad \Gamma \Rightarrow B, \Delta
\]

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

\[
\text{andRight} \{
\text{\find ( ==> A \& B )} \\
\text{\replacewith (==> A );} \\
\text{\replacewith (==> B )}
\}
\]

**Variable Conditions: allRight**

\[
\Gamma \Rightarrow \{x/c}\Phi, \Delta
\]

\[
\frac{}{\Gamma \Rightarrow \forall T \ x; \Phi, \Delta}
\]

\[
\text{allRight} \{
\text{\find ( ==> \forall x;phi )} \\
\text{\varcond(\new(c,\dependingOn(phi)))} \\
\text{\replacewith ( ==> \{\subst x;c}phi )}
\}
\]
Taclet Syntax

Proof Splitting: andRight

\[ \Gamma \Rightarrow A, \Delta \quad \Gamma \Rightarrow B, \Delta \]

\[ \Rightarrow A \land B, \Delta \]

andRight {
\find ( \Rightarrow A \land B )
\replacewith ( \Rightarrow A )
\replacewith ( \Rightarrow B )
};

Variable Conditions: allRight

\[ \Gamma \Rightarrow \{ x/c \} \Phi, \Delta \]

\[ \Rightarrow \forall T \ x ; \Phi, \Delta \]

\[ , c \ \text{new} \]

allRight {
\find ( \Rightarrow \forall x ; \phi )
\varcond(\textbf{new}(c,\text{dependingOn}(\phi)))
\replacewith ( \Rightarrow \{ \text{subst} x ; c \} \phi )
};
Taclets for Program Transformations

\[ \Gamma \Rightarrow \begin{cases} \pi \text{ if } (\text{exc} == \text{null}) \{ \\
\text{try}\{ \text{throw new NPE()}; \text{catch}(\text{T e}) \{\text{r}\}; \} \text{else if } (\text{exc instanceof T}) \{\text{e=exc; r}\} \\
\text{else throw exc; } \omega \\
\end{cases} \phi \]

\[ \Gamma \Rightarrow \langle \pi \text{ try}\{\text{throw exc; q}\} \text{ catch}(\text{T e})\{\text{r}\}; \omega \rangle \phi \]

\text{\textbackslash{}find ( } <.. \text{ try } \{ \text{throw } \#\text{se}; \#\text{slist} \} \\
\text{catch } ( \#t \#v0 ) \{ \#\text{slist1} \} \ldots \text{ post } ) \text{\textbackslash{}replacewith ( } \\
<.. \text{ if } (\#\text{se} == \text{null}) \{ \\
\text{try } \{ \text{throw new NullPointerException(); } \} \\
\text{catch } (\#t \#v0) \{ \#\text{slist1} \} \\
\text{else if } (\#\text{se} \text{ instanceof } \#t) \{ \\
\#t \#v0 = (\#t) \#\text{se}; \\
\#\text{slist1} \\
\text{else throw } \#\text{se}; \ldots \text{ post } ) \} \}

Bernhard Beckert – Deductive Verification of Object-Oriented Software

VTSA, 24.–28.08.2015
Part IV

Verifying Information-Flow Properties

12 Information Flow

13 Formalisation in DL

14 Objects and Information Flow
Part IV

Verifying Information-Flow Properties

12 Information Flow

13 Formalisation in DL

14 Objects and Information Flow
Secret and public information

Partitioning of the set of program variables into

- variables which contain confidential information ("high variables")
  - NOT observable by the attacker –
- variables which contain non-confidential information ("low variables")
  - observable by the attacker –

Informal definition of non-interference

A program is secure, if the initial values of the high variables do not interfere with the final values of the low variables.
Secret and public information

Partitioning of the set of program variables into

- variables which contain confidential information ("high variables")
  - NOT observable by the attacker –
- variables which contain non-confidential information ("low variables")
  - observable by the attacker –

Informal definition of non-interference

A program is secure, if the initial values of the high variables do not interfere with the final values of the low variables.
Examples

Note

Sequential Java programs
Termination not considered

Which methods are secure?

```java
void m_1() {
    low = high;
}

void m_3() {
    if (high > 0) {low = 1;}
    else {low = 2;};
}
```
Examples

Note

Sequential Java programs
Termination not considered

Which methods are secure?

```java
void m_1() {
    low = high;
}
```

```java
void m_3() {
    if (high > 0) {low = 1;}
    else            {low = 2;};
}
```
Examples

Note

Sequential Java programs
Termination not considered

Which methods are secure?

```java
void m_1() {
    low = high;
}

void m_3() {
    if (high > 0) {low = 1;}
    else {low = 2;}
}
```

NOT SECURE

Information Flow

Formalisation in DL

Objects and Information Flow

Bernhard Beckert – Deductive Verification of Object-Oriented Software
Examples

Note
Sequential Java programs
Termination not considered

Which methods are secure?

```java
void m_1() {
    low = high;
}
```

```java
void m_3() {
    if (high > 0) {low = 1;}
    else {low = 2;};
}
```

NOT SECURE
Examples

Note

Sequential Java programs
Termination not considered

Which methods are secure?

```java
void m_1() {
    low = high;
}
```

NOT SECURE

```java
void m_3() {
    if (high > 0) {low = 1;}
    else {low = 2;};
}
```

NOT SECURE
Examples

Which methods are secure?

```c
void m_4() {
    high = 0;
    low = high;
}
```

```c
void m_5() {
    low = high;
    low = low - high;
}
```

```c
void m_6() {
    if (false) low = high;
}
```
Examples

Which methods are secure?

```c
void m_4() {
    high = 0;
    low = high;
}
```

```c
SECURE
void m_5() {
    low = high;
    low = low - high;
}
```

```c
void m_6() {
    if (false) low = high;
}
```
Examples

Which methods are secure?

```c
void m_4() {
    high = 0;
    low = high;
}
```

SECURE

```c
void m_5() {
    low = high;
    low = low - high;
}
```

```c
void m_6() {
    if (false) low = high;
}
```
Examples

Which methods are secure?

```c
void m_4() {
    high = 0;
    low = high;
}
```

SECURE

```c
void m_5() {
    low = high;
    low = low - high;
}
```

SECURE

```c
void m_6() {
    if (false) low = high;
}
```
Examples

Which methods are secure?

```c
void m_4() {
    high = 0;
    low = high;  // SECURE
}

void m_5() {
    low = high;
    low = low - high;  // SECURE
}

void m_6() {
    if (false) low = high;
}
```
Examples

Which methods are secure?

```c
void m_4() {
    high = 0;
    low = high;
}
```

```c
void m_5() {
    low = high;
    low = low - high;
}
```

```c
void m_6() {
    if (false) low = high;
}
```
Information-flow Analysis Approaches

- performance

- security type systems
  - graph-theoretical reachability
    - on dependence graph
    - explicit dependency
      - tracking by ghost code
  - approximative calculi
    - for Hoare style logics
- formalization in
  - Hoare style logics
    - by self-composition
- formalization in
  - higher order logic

- precision / expressiveness
Information-flow Analysis Approaches

- Performance
- Precision / Expressiveness

Formalization in Hoare style logics by self-composition
**Definition (Low-equivalence on states)**

Two states are low-equivalent if they assign the same values to low variables.

**Definition (Non-interference)**

Starting $P$ in two arbitrary low-equivalent states results in two final states that are also low-equivalent.
Non-Interference

Definition (Low-equivalence on states)

Two states are low-equivalent if they assign the same values to low variables.

Definition (Non-interference)

Starting $P$ in two arbitrary low-equivalent states results in two final states that are also low-equivalent.
Non-interference

- $P$ a program
- $L$ the set of low variables

where

$$s_i \simeq_L s'_i \iff \forall v \in L \ (v^{s_i} = v^{s'_i})$$
Non-interference

- $P$ a program
- $L_1, L_2$ sets of low variables

\[ s_2 \sim_{L_2} s'_2 \]
\[ s_1 \sim_{L_1} s'_1 \]

where

\[ s_i \sim_{L_i} s'_i \iff \forall v \in L_i \ (v^{s_i} = v^{s'_i}) \]
Non-Interference in JavaDL (Version 1)

Encoding with alternating quantifiers

For all low input values \( in_l \), there exist low output values \( r \) such that for all high input values \( in_h \), if we assign the values \( in_l \) to the program variables \( low \) and \( in_h \) to the program variables \( high \), then after execution of \( P \) the values of \( low \) are \( r \).

\[
\forall in_l \exists r \forall in_h \{ \{ low := in_l \ |\ |\ high := in_h \} [P] low = r \}
\]

Problem

Not suitable for automatic verification

\( \iff \) instantiation of existential quantifier difficult.
Non-Interference in JavaDL (Version 1)

Encoding with alternating quantifiers

For all low input values \( in_l \), there exist low output values \( r \) such that for all high input values \( in_h \), if we assign the values \( in_l \) to the program variables \( low \) and \( in_h \) to the program variables \( high \), then after execution of \( P \) the values of \( low \) are \( r \).

\[
\forall in_l \exists r \forall in_h (\{ low := in_l \mid high := in_h \}[P] low = r)
\]

Problem

Not suitable for automatic verification

\( \leadsto \) instantiation of existential quantifier difficult.
Encoding with self-composition

Running two instances of $P$ on the same low values but on arbitrary high values results in low variables which have the same values.

$$\forall \text{in}_l \forall \text{in}^1_h \forall \text{in}^2_h \forall \text{out}^1_l \forall \text{out}^2_l \{\text{low} := \text{in}_l\} (\{\text{high} := \text{in}^1_h\}[P]\text{out}^1_l = \text{low} \land \{\text{high} := \text{in}^2_h\}[P]\text{out}^2_l = \text{low} \rightarrow \text{out}^1_l = \text{out}^2_l$$
Declassification

**Intuition**

Let \( T(\text{high}, \text{low}) \) be a term.

The only thing the attacker is allowed to learn about the secret inputs is the value of \( T \) in the initial state.

**Definition (Non-interference with declassification)**

Starting \( P \) in two arbitrary low-equivalent states coinciding in the value of \( T \) results in two final states that are also low-equivalent.
Declassification

Intuition

Let $T(\text{high, low})$ be a term.

The only thing the attacker is allowed to learn about the secret inputs is the value of $T$ in the initial state.

Definition (Non-interference with declassification)

Starting $P$ in two arbitrary low-equivalent states coinciding in the value of $T$ results in two final states that are also low-equivalent.
Declassification in JavaDL

Encoding non-interference with declassification

Running two instances of $P$ on the same low values and arbitrary high values coinciding on $T$ results in low variables which have the same values.

\[
\forall \text{in}_l \forall \text{in}_h^1 \forall \text{in}_h^2 \forall \text{out}_l^1 \forall \text{out}_l^2 \{ \text{low} := \text{in}_l \} ( \\
\{ \text{high} := \text{in}_h^1 \} T = \{ \text{high} := \text{in}_h^2 \} T \\
\land \{ \text{high} := \text{in}_h^1 \} [P] \text{out}_l^1 = \text{low} \\
\land \{ \text{high} := \text{in}_h^2 \} [P] \text{out}_l^2 = \text{low} \\
\rightarrow \text{out}_l^1 = \text{out}_l^2 
) 
\]
DEMO

Verifying Information-flow Properties with the KeY Tool
Object-Sensitive Non-interference

Leakage by aliasing

```c
void m() {
    C c1 = new C(); // new obj
    C c2 = c1;       // alias
    c2.x = high;
    low  = c1.x;
}
```
Object-Sensitive Non-interference

Leakage by aliasing

```c
void m() {
    C c1 = new C(); // new obj
    C c2 = c1;       // alias
    c2.x = high;
    low = c1.x;
}
```

NOT SECURE
Object-Sensitive Non-interference

Object creation and object identity

```java
if (high > 0) {
    low1 = new C();
    low2 = new C();
} else {
    low2 = new C();
    low1 = new C();
}
```

Assumption

- References are opaque
- Only comparison of objects by `==` is observable
Object-Sensitive Non-interference

Object creation and object identity

```java
if (high > 0) {
    low1 = new C();
    low2 = new C();
} else {
    low2 = new C();
    low1 = new C();
}
```

Assumption

- References are opaque
- Only comparison of objects by `==` is observable
Object-Sensitive Non-interference

Object creation and object identity

```java
if (high > 0) {
    low1 = new C();
    low2 = new C();
} else {
    low2 = new C();
    low1 = new C();
}
```

Assumption

- References are opaque
- Only comparison of objects by == is observable
Object-Sensitive Non-interference

Object creation and object identity

low1 = new C();
low2 = new C();
if (high > 0) { low1 = low2; }

if (high > 0) { low = new C(); }
Object-Sensitive Non-interference

Object creation and object identity

low1 = new C();
low2 = new C();
if (high > 0) { low1 = low2; }

if (high > 0) { low = new C(); }

NOT SECURE
Object-Sensitive Non-interference

Object creation and object identity

```java
low1 = new C();
low2 = new C();
if (high > 0) { low1 = low2; }
```

```java
if (high > 0) { low = new C(); }
```

NOT SECURE
Object-Sensitive Non-interference

Object creation and object identity

low1 = \texttt{new} C();
low2 = \texttt{new} C();
if (high > 0) { low1 = low2; }

if (high > 0) { low = \texttt{new} C() } \quad \text{NOT SECURE}
Object-Sensitive Non-interference

Idea

ISOMORPHIC object structures in low variables
IDENTITY NOT required

Instead of

\[ s_i \simeq_{L_i} s'_i \iff \forall v \in L_i \ (v^{s_i} = v^{s'_i}) \]

use

\[ s_i \simeq_{L_i} \pi_i \ s'_i \iff \forall v \in L_i \ (\pi(v^{s_i}) = v^{s'_i}) \]

where

\[ \pi_1, \pi_2 \text{ are compatible} \]

i.e.

\[ \pi_1(o) = \pi_2(o) \quad \text{if } o \text{ observable in both } s_1 \text{ and } s_2 \]
Object-Sensitive Non-interference

\begin{align*}
\text{low1} & \mapsto \pi_2(o_1) = o_2, \\
o_1 & \mapsto \pi_2(o_2) = o_1 \\
s_2: \text{low2} & \mapsto \pi_2(o_1) = o_2, \\
o_2 & \mapsto \pi_2(o_2) = o_1 \\
\text{high} & \mapsto 1 \\
P & \mapsto P \\
s_1: \text{high} & \mapsto \pi_1 = \text{id} \\
\text{high} & \mapsto -1 \\

s'_2: \text{low2} & \mapsto \pi_2(o_1) = o_2, \\
o_1 & \mapsto \pi_2(o_2) = o_1 \\
\text{high} & \mapsto -1 \\
P & \mapsto P \\
s'_1: \text{high} & \mapsto \pi_1 = \text{id} \\
\text{high} & \mapsto -1
\end{align*}

Secure because $o_1, o_2$ not observable in $s_1$. 
Object-Sensitive Non-interference

\[
\begin{align*}
\text{low1} & \iff \pi_2(o_1) = o_2, \\
\text{high} & \iff \pi_1 = \text{id}
\end{align*}
\]

Secure because \(o_1, o_2\) not observable in \(s_1\)
Object-Sensitive Non-interference

\[
\begin{align*}
\text{low1} & \mapsto o_1 \\
\text{low2} & \mapsto ? \\
\text{high} & \mapsto 1 \\
\end{align*}
\]

\[
\begin{align*}
\pi_1 &= id \\
\pi_2 &= ? \\
\end{align*}
\]

Not secure because no suitable \( \pi_2 \) exists.
Object-Sensitive Non-interference

Not secure because no suitable $\pi_2$ exists
Object-Sensitive Non-interference

$s_2: \begin{align*}
\text{low} & \mapsto o_2 \\
\text{high} & \mapsto 1
\end{align*}$

$s_2': \begin{align*}
\text{low} & \mapsto o_1 \\
\text{high} & \mapsto -1
\end{align*}$

$P$

$s_1: \begin{align*}
\text{low} & \mapsto o_1 \\
\text{high} & \mapsto 1
\end{align*}$

$s_1': \begin{align*}
\text{low} & \mapsto o_1 \\
\text{high} & \mapsto -1
\end{align*}$

$\pi_2(o_2) = o_1$

$\pi_1 = id$

$\pi_1(o_1) \neq \pi_2(o_1)$

Not secure because $o_1$ observable in $s_1$ and $\pi_1(o_1) \neq \pi_2(o_1)$
Object-Sensitive Non-interference

Not secure because $o_1$ observable in $s_1$ and $\pi_1(o_1) \neq \pi_2(o_1)$
Optimisations

- $L_1, L_2$ sequences of low terms (instead of sets of variables)
- $\pi_1$ can be fixed to be $id$
DEMO

Objects and Information-flow with the KeY Tool
Part V

Wrap Up

15 Further Usage of Verification Technology

16 Directions of Current Research in KeY
Part V

Wrap Up

15 Further Usage of Verification Technology

16 Directions of Current Research in KeY
Further Usage of Verification Technology

- Verification performs deep *Program Analysis*
- Information in (partial) proofs usable for other purposes
Further Usage: Verification-Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence:
  proof branches $\leftrightarrow$ program execution paths
- Each leaf of (partial) proof branch contains constraint on inputs resulting in corresponding path condition
Further Usage: Verification-Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence:
  proof branches ↔ program execution paths
- Each leaf of (partial) proof branch contains constraint on inputs resulting in corresponding path condition
Further Usage:
Verification-Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence:
  proof branches $\leftrightarrow$ program execution paths
- Each leaf of (partial) proof branch contains
  constraint on inputs
  resulting in
  corresponding path condition
Further Usage: Verification-Driven Test Generation

- Specification- and code-based approach
- Achieve strong hybrid coverage criteria
- Exploit strong correspondence: proof branches $\leftrightarrow$ program execution paths
- Each leaf of (partial) proof branch contains constraint on inputs resulting in corresponding path condition
Part V

Wrap Up

15 Further Usage of Verification Technology

16 Directions of Current Research in KeY
Part V

Wrap Up

15 Further Usage of Verification Technology

16 Directions of Current Research in KeY
Topics

- Scalability (combine with light-weight techniques)
- Usability (support user in understanding proof state)
- Concurrency and distribution
- Information-flow / security properties
- Application: eVoting
Directions of Current Research with KeY

Topics

- Scalability (combine with light-weight techniques)
- Usability (support user in understanding proof state)
- Concurrency and distribution
- Information-flow / security properties
- Application: eVoting
Directions of Current Research with KeY

Topics

- Scalability (combine with light-weight techniques)
- Usability (support user in understanding proof state)
- Concurrency and distribution
- Information-flow / security properties
- Application: eVoting
Directions of Current Research with KeY

Topics

- Scalability (combine with light-weight techniques)
- Usability (support user in understanding proof state)
- Concurrency and distribution
- Information-flow / security properties
- Application: eVoting
Topics

- Scalability (combine with light-weight techniques)
- Usability (support user in understanding proof state)
- Concurrency and distribution
- Information-flow / security properties
- Application: eVoting
THE END

(for now)