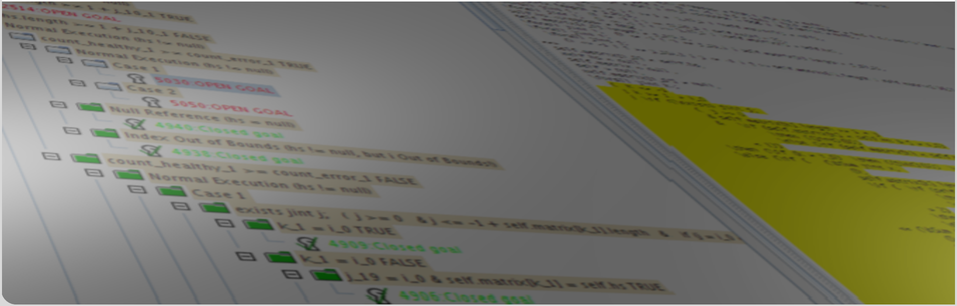
Karlsruhe Institute of Technology

Deductive Verification of Object-Oriented Software

Part B

Bernhard Beckert | VTSA, 24.–28.08.2015

KIT – Institute for Theoretical Computer Science



Part III

Program Verification with Dynamic Logic

Part III

Program Verification with Dynamic Logic

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)

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)

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 ψ `->` $[\alpha]\ \phi$

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

$(\texttt{balance} >= c \;\&\; \texttt{amount} > 0)$ ->
`<charge(amount);>` $\texttt{balance} > c$

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

- Program formulas can appear nested

$\backslash\texttt{forall}\; int\; val; ((\texttt{<p>}\, \texttt{x} \doteq val) \;\texttt{<->}\; (\texttt{<q>}\, \texttt{x} \doteq val))$

- p, q equivalent relative to computation state restricted to x

Dynamic Logic Example Formulas

$(\texttt{balance} >= c \ \& \ \texttt{amount} > 0) \texttt{->}$
$\texttt{<charge(amount);>} \ \texttt{balance} > c$

$\texttt{<x = 1;>}([\texttt{while (true) \{\}}]\, \mathit{false})$

- Program formulas can appear nested

$\backslash\texttt{forall}\ \mathit{int\ val}; ((\texttt{<p>}\,\texttt{x} \doteq \mathit{val}) \ \texttt{<->} \ (\texttt{<q>}\,\texttt{x} \doteq \mathit{val}))$

- p, q equivalent relative to computation state restricted to x

Dynamic Logic Example Formulas

$(\texttt{balance} >= c \;\&\; \texttt{amount} > 0) \;{-}{>}$
$\texttt{<charge(amount);>}\ \texttt{balance} > c$

$\texttt{<x = 1;>}([\texttt{while (true) \{\}}]\,\mathit{false})$

- Program formulas can appear nested

$\backslash\texttt{forall}\ \mathit{int\ val}; ((\texttt{<p>}\,\texttt{x} \doteq \mathit{val}) \;{<}{-}{>}\; (\texttt{<q>}\,\texttt{x} \doteq \mathit{val}))$

- 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 $\doteq$ *val*) $<->$ (`<q>` x $\doteq$ *val*))

- p, q equivalent relative to computation state restricted to x

Dynamic Logic Example Formulas

$$(\texttt{balance} >= c \ \& \ \texttt{amount} > 0) \ {-}{>}$$
$$\texttt{<charge(amount);>} \ \texttt{balance} > c$$

$$\texttt{<x = 1;>}([\texttt{while (true) \{\}}]\, \mathit{false})$$

- Program formulas can appear nested

$$\backslash\texttt{forall}\ \mathit{int}\ \mathit{val}; ((\texttt{<p>}\,\texttt{x} \doteq \mathit{val}) \ {<}{-}{>}\ (\texttt{<q>}\,\texttt{x} \doteq \mathit{val}))$$

- p, q equivalent relative to computation state restricted to `x`

Dynamic Logic Example Formulas

```
  a != null
->
  <
    int max = 0;
    if ( a.length > 0 ) max = a[0];
    int i = 1;
    while ( i < a.length ) {
      if ( a[i] > max ) max = a[i];
      ++i;
    }
  >
    (
      \forall int j; (j >= 0 & j < a.length -> max >= a[j])
      &
      (a.length > 0 ->
        \exists int j; (j >= 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

$\texttt{<int i;>}\backslash\texttt{forall}\ \mathit{int}\ x; (\texttt{i} + 1 \doteq x \texttt{ -> } \texttt{<i++;>}(\texttt{i} \doteq x))$

- 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

$\texttt{<int i;>}\backslash\texttt{forall}\ int\ x; (\texttt{i} + 1 \doteq x \texttt{ -> } \texttt{<i++;>}(\texttt{i} \doteq x))$

- 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;>` $\backslash$`forall` $int\ x; (\mathtt{i} + 1 \doteq x$ `->` `<i++;>` $(\mathtt{i} \doteq x))$

- 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

$\texttt{<int i;>}\ \backslash\texttt{forall}\ int\ x; (\texttt{i} + 1 \doteq x \texttt{ -> } \texttt{<i++;>}(\texttt{i} \doteq x))$

- 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

$\texttt{<int i;>}\backslash\texttt{forall}\ int\ x; (\texttt{i} + 1 \doteq x \texttt{ -> } \texttt{<i++;>} (\texttt{i} \doteq x))$

- 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

Validity

A JAVA CARD DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas

Validity

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

Part III

Program Verification with Dynamic Logic

Sequents and their Semantics

Syntax

$$\underbrace{\psi_1, \ldots, \psi_m}_{Antecedent} \implies \underbrace{\phi_1, \ldots, \phi_n}_{Succedent}$$

where the ϕ_i, ψ_i are formulae (without free variables)

Semantics

Same as the formula

$$(\psi_1 \ \& \ \cdots \ \& \ \psi_m) \ \ {-}{>} \ \ (\phi_1 \ | \ \cdots \ | \ \phi_n)$$

Sequents and their Semantics

Syntax

$$\underbrace{\psi_1, \ldots, \psi_m}_{Antecedent} \implies \underbrace{\phi_1, \ldots, \phi_n}_{Succedent}$$

where the ϕ_i, ψ_i are formulae (without free variables)

Semantics

Same as the formula

$$(\psi_1 \ \& \ \cdots \ \& \ \psi_m) \ \ \texttt{->} \ \ (\phi_1 \ | \ \cdots \ | \ \phi_n)$$

Sequent Rules

General form

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ 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

$$\text{rule\_name}\ \frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}}}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}$$

($r = 0$ 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

$$\text{rule\_name} \quad \overbrace{\frac{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}{\underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}}}^{\text{Premisses}}$$

($r = 0$ 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

$$\text{rule\_name} \quad \overbrace{\Gamma_1 \Longrightarrow \Delta_1 \quad \cdots \quad \Gamma_r \Longrightarrow \Delta_r}^{\text{Premisses}} \over \underbrace{\Gamma \Longrightarrow \Delta}_{\text{Conclusion}}$$

($r = 0$ 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 $$\frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

imp_left $$\frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \;\text{->}\; B \Longrightarrow \Delta}$$

close_goal $$\frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

close_by_true $$\frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

all_left $$\frac{\Gamma, \backslash\mathtt{forall}\; t\; x; \phi,\; \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\mathtt{forall}\; t\; x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

Karlsruhe Institute of Technology

$$\text{not\_left}\ \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\,A \Longrightarrow \Delta}$$

$$\text{imp\_left}\ \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \;\text{->}\; B \Longrightarrow \Delta}$$

$$\text{close\_goal}\ \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true}\ \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left}\ \frac{\Gamma, \backslash\texttt{forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\texttt{forall}\ t\ x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\text{not\_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

$$\text{imp\_left} \quad \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \mathrel{->} B \Longrightarrow \Delta}$$

$$\text{close\_goal} \quad \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true} \quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left} \quad \frac{\Gamma, \texttt{\textbackslash forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \texttt{\textbackslash forall}\ t\ x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\text{not\_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\, A \Longrightarrow \Delta}$$

$$\text{imp\_left} \quad \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \text{ -> } B \Longrightarrow \Delta}$$

$$\text{close\_goal} \quad \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true} \quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

$$\text{all\_left} \quad \frac{\Gamma, \backslash\mathtt{forall}\; t\; x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\mathtt{forall}\; t\; x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Some Simple Sequent Rules

$$\text{not\_left} \quad \frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, !\,A \Longrightarrow \Delta}$$

$$\text{imp\_left} \quad \frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \;\text{->}\; B \Longrightarrow \Delta}$$

$$\text{close\_goal} \quad \frac{}{\Gamma, A \Longrightarrow A, \Delta}$$

$$\text{close\_by\_true} \quad \frac{}{\Gamma \Longrightarrow \mathrm{true}, \Delta}$$

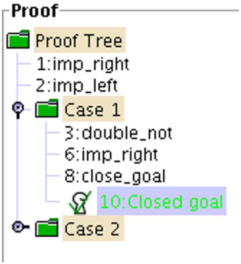
$$\text{all\_left} \quad \frac{\Gamma, \backslash\mathtt{forall}\; t\; x; \phi,\; \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\mathtt{forall}\; t\; x; \phi \Longrightarrow \Delta}$$

where e var-free term of type $t' \prec t$

Sequent Calculus Proofs

Proof tree

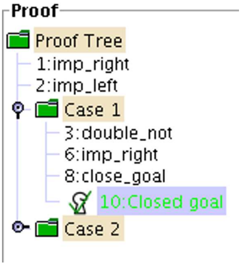
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed



Sequent Calculus Proofs

Proof tree

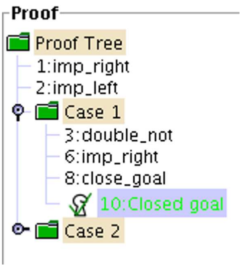
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed



Sequent Calculus Proofs

Proof tree

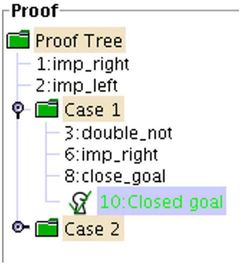
- Proof is tree structure with goal sequent as root
- Rules are applied from conclusion (old goal) to premisses (new goals)
- Rule with no premiss closes proof branch
- Proof is finished when all goals are closed



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

Part III

Program Verification with Dynamic Logic

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:{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

```
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

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

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

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

passive prefix	π
active statement	`i=0;`
rest	ω

- Sequent rules execute symbolically the active statement

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = \mathit{true} \Longrightarrow \texttt{<}p \;\; \omega\texttt{>}\,\phi, \Delta \qquad \Gamma, B = \mathit{false} \Longrightarrow \texttt{<}q \;\; \omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<if (}B\texttt{) \{ }p\texttt{ \} else \{ }q\texttt{ \} }\omega\texttt{>}\,\phi, \Delta}$$

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

$$\frac{\Gamma \Longrightarrow \texttt{<v=y; y=y+1; x=v; }\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<x=y++; }\omega\texttt{>}\,\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{loc := val\}\texttt{<}\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<}loc\texttt{=}val\texttt{; }\omega\texttt{>}\,\phi, \Delta}$$

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = \textit{true} \Longrightarrow \texttt{<}p \;\; \omega\texttt{>}\,\phi, \Delta \qquad \Gamma, B = \textit{false} \Longrightarrow \texttt{<}q \;\; \omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<if (}B\texttt{) \{ }p\texttt{ \} else \{ }q\texttt{ \} }\omega\texttt{>}\,\phi, \Delta}$$

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

$$\frac{\Gamma \Longrightarrow \texttt{<v=y; y=y+1; x=v; }\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<x=y++; }\omega\texttt{>}\,\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{loc := val\}\texttt{<}\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<}loc\texttt{=}val\texttt{; }\omega\texttt{>}\,\phi, \Delta}$$

Rules for Symbolic Program Execution

If-then-else rule

$$\frac{\Gamma, B = \textit{true} \Longrightarrow \texttt{<}p\ \ \omega\texttt{>}\,\phi, \Delta \qquad \Gamma, B = \textit{false} \Longrightarrow \texttt{<}q\ \ \omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<if (}B\texttt{) \{ }p\texttt{ \} else \{ }q\texttt{ \} }\omega\texttt{>}\,\phi, \Delta}$$

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

$$\frac{\Gamma \Longrightarrow \texttt{<v=y; y=y+1; x=v; }\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<x=y++; }\omega\texttt{>}\,\phi, \Delta}$$

Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{loc := val\}\texttt{<}\omega\texttt{>}\,\phi, \Delta}{\Gamma \Longrightarrow \texttt{<}loc\texttt{=}val\texttt{; }\omega\texttt{>}\,\phi, \Delta}$$

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

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

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

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

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

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

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  ==>  x < y
              ⋮
x < y  ==>  {x:=y || y:=x}<> y < x
              ⋮
x < y  ==>  {t:=x || x:=y || y:=x}<> y < x
              ⋮
x < y  ==>  {t:=x || x:=y}{y:=t}<> y < x
              ⋮
x < y  ==>  {t:=x}{x:=y}<y=t;> y < x
              ⋮
x < y  ==>  {t:=x}<x=y; y=t;> y < x
              ⋮
       ==>  x < y -> <int t=x; x=y; y=t;> y < x
```

Symbolic Execution with Updates

(by Example)

```
x < y  ==>  x < y
               ⋮
x < y  ==>  {x:=y || y:=x}<> y < x
               ⋮
x < y  ==>  {t:=x || x:=y || y:=x}<> y < x
               ⋮
x < y  ==>  {t:=x || x:=y}{y:=t}<> y < x
               ⋮
x < y  ==>  {t:=x}{x:=y}<y=t;> y < x
               ⋮
x < y  ==>  {t:=x}<x=y; y=t;> y < x
               ⋮
       ==>  x < y -> <int t=x; x=y; y=t;> y < x
```

Symbolic Execution with Updates

(by Example)

$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \mathtt{x} < \mathtt{y}$

$\vdots$

$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$

$\vdots$

$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{t:=x} \parallel \mathtt{x:=y} \parallel \mathtt{y:=x}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$

$\vdots$

$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{t:=x} \parallel \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\ \mathtt{y} < \mathtt{x}$

$\vdots$

$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\ \mathtt{y} < \mathtt{x}$

$\vdots$

$\mathtt{x} < \mathtt{y} \quad \Longrightarrow \quad \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\ \mathtt{y} < \mathtt{x}$

$\vdots$

$\Longrightarrow \quad \mathtt{x} < \mathtt{y} \;\texttt{->}\; \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\ \mathtt{y} < \mathtt{x}$

Symbolic Execution with Updates
(by Example)

$$\texttt{x} < \texttt{y} \Longrightarrow \texttt{x} < \texttt{y}$$

$\vdots$

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$\vdots$

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{t:=x} \parallel \texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$\vdots$

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{t:=x} \parallel \texttt{x:=y}\}\{\texttt{y:=t}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$\vdots$

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{t:=x}\}\{\texttt{x:=y}\}\texttt{<y=t;>}\ \texttt{y} < \texttt{x}$$

$\vdots$

$$\texttt{x} < \texttt{y} \Longrightarrow \{\texttt{t:=x}\}\texttt{<x=y; y=t;>}\ \texttt{y} < \texttt{x}$$

$\vdots$

$$\Longrightarrow \texttt{x} < \texttt{y} \texttt{ -> <int t=x; x=y; y=t;>}\ \texttt{y} < \texttt{x}$$

Symbolic Execution with Updates
(by Example)

```
x < y  ==>  x < y
       ⋮
x < y  ==>  {x:=y || y:=x}<> y < x
       ⋮
x < y  ==>  {t:=x || x:=y || y:=x}<> y < x
       ⋮
x < y  ==>  {t:=x || x:=y}{y:=t}<> y < x
       ⋮
x < y  ==>  {t:=x}{x:=y}<y=t;> y < x
       ⋮
x < y  ==>  {t:=x}<x=y; y=t;> y < x
       ⋮
       ==>  x < y -> <int t=x; x=y; y=t;> y < x
```

Symbolic Execution with Updates

(by Example)

$$x < y \;\Longrightarrow\; x < y$$

$$\vdots$$

$$x < y \;\Longrightarrow\; \{\texttt{x:=y} \,\|\, \texttt{y:=x}\}\texttt{<>}\ y < x$$

$$\vdots$$

$$x < y \;\Longrightarrow\; \{\texttt{t:=x} \,\|\, \texttt{x:=y} \,\|\, \texttt{y:=x}\}\texttt{<>}\ y < x$$

$$\vdots$$

$$x < y \;\Longrightarrow\; \{\texttt{t:=x} \,\|\, \texttt{x:=y}\}\{\texttt{y:=t}\}\texttt{<>}\ y < x$$

$$\vdots$$

$$x < y \;\Longrightarrow\; \{\texttt{t:=x}\}\{\texttt{x:=y}\}\texttt{<y=t;>}\ y < x$$

$$\vdots$$

$$x < y \;\Longrightarrow\; \{\texttt{t:=x}\}\texttt{<x=y; y=t;>}\ y < x$$

$$\vdots$$

$$\Longrightarrow\; x < y \;\texttt{->}\; \texttt{<int t=x; x=y; y=t;>}\ y < x$$

Symbolic Execution with Updates
(by Example)

$$\texttt{x} < \texttt{y} \implies \texttt{x} < \texttt{y}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \implies \{\texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \implies \{\texttt{t:=x} \parallel \texttt{x:=y} \parallel \texttt{y:=x}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \implies \{\texttt{t:=x} \parallel \texttt{x:=y}\}\{\texttt{y:=t}\}\texttt{<>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \implies \{\texttt{t:=x}\}\{\texttt{x:=y}\}\texttt{<y=t;>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\texttt{x} < \texttt{y} \implies \{\texttt{t:=x}\}\texttt{<x=y; y=t;>}\ \texttt{y} < \texttt{x}$$

$$\vdots$$

$$\implies \texttt{x} < \texttt{y} \texttt{ -> <int t=x; x=y; y=t;>}\ \texttt{y} < \texttt{x}$$

Program State Representation

Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays:

$$
\begin{aligned}
&heap: && \to Heap \text{ (the heap in the current state)}\\
&select: && Heap \times Object \times Field \to Any\\
&store: && Heap \times Object \times Field \times Any \to Heap
\end{aligned}
$$

Heap axioms (excerpt)

$$select(store(h, o, f, x), o, f) = x$$

$$select(store(h, o, f, x), u, f) = select(h, u, f) \text{ if } o \neq u$$

Program State Representation

Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays:

$$
\begin{array}{ll}
\mathit{heap}: & \to \mathit{Heap} \text{ (the heap in the current state)} \\
\mathit{select}: & \mathit{Heap} \times \mathit{Object} \times \mathit{Field} \to \mathit{Any} \\
\mathit{store}: & \mathit{Heap} \times \mathit{Object} \times \mathit{Field} \times \mathit{Any} \to \mathit{Heap}
\end{array}
$$

Heap axioms (excerpt)

$$\mathit{select}(\mathit{store}(h, o, f, x), o, f) = x$$

$$\mathit{select}(\mathit{store}(h, o, f, x), u, f) = \mathit{select}(h, u, f) \text{ if } o \neq u$$

Program State Representation

Local program variables

Modeled as non-rigid constants

Heap

Modeled with theory of arrays:

$$\begin{array}{ll} heap: & \rightarrow Heap \text{ (the heap in the current state)} \\ select: & Heap \times Object \times Field \rightarrow Any \\ store: & Heap \times Object \times Field \times Any \rightarrow Heap \end{array}$$

Heap axioms (excerpt)

$$select(store(h, o, f, x), o, f) = x$$
$$select(store(h, o, f, x), u, f) = select(h, u, f) \text{ if } o \neq u$$

Handling Abrupt Termination

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

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l}\texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \quad \texttt{else \{s throw exc;\}} \quad \omega \end{array} \right\rangle \phi, \Delta}{\Gamma \Longrightarrow \texttt{<try\{throw exc; q\} catch(T e)\{r\} finally\{s\}} \ \omega\texttt{>} \phi, \Delta}$$

Handling Abrupt Termination

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

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l}\texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \quad \texttt{else \{s throw exc;\}} \quad \omega \end{array} \right\rangle \phi, \Delta}{\Gamma \Longrightarrow \texttt{<try\{throw exc; q\} catch(T e)\{r\} finally\{s\}} \ \omega\texttt{>} \phi, \Delta}$$

Handling Abrupt Termination

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

Example

TRY-THROW

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l}\pi\ \texttt{if (exc instanceof T)} \\ \quad \texttt{\{try \{e=exc; r\} finally \{s\}\}} \\ \quad \texttt{else \{s throw exc;\}}\ \ \omega \end{array}\right\rangle \phi, \Delta}{\Gamma \Longrightarrow \texttt{<}\pi\ \texttt{try\{throw exc; q\} catch(T e)\{r\} finally\{s\}}\ \omega\texttt{>}\phi, \Delta}$$

Part III

Program Verification with Dynamic Logic

Part III

Program Verification with Dynamic Logic

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

Java—A Language of Many Features

Karlsruhe Institute of Technology

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

Karlsruhe Institute of Technology

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

Karlsruhe Institute of Technology

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

Karlsruhe Institute of Technology

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

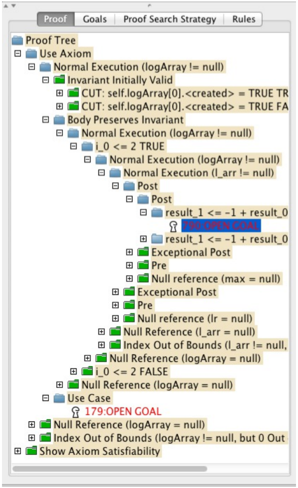
Part III

Program Verification with Dynamic Logic

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

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \,\&\, B \Longrightarrow \Delta}$$

Taclet

```
andLeft {
    \find ( A & B ==> )
    \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

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \,\&\, B \Longrightarrow \Delta}$$

Taclet

```
andLeft {
    \find ( A & B ==> )
    \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

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, {\color{red}A \ \& \ B} \Longrightarrow \Delta}$$

Taclet

```
andLeft {
    \find ( A & B ==> )
    \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

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \,\&\, B \Longrightarrow \Delta}$$

Taclet

```
andLeft {
    \find ( A & B ==> )
    \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

KIT — Karlsruhe Institute of Technology

$$\text{andLeft}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A \,\&\, B \Longrightarrow \Delta}$$

Taclet

```
andLeft {
    \find ( A & B ==> )
    \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

Some rules are only sound in a certain context

$$\text{modusPonens}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A, A \;\texttt{->}\; B \Longrightarrow \Delta}$$

Taclet

```
modusPonens {
  \assumes ( A ==> )
  \find ( A -> B ==> )
  \replacewith( B ==> )
};
```

Taclet Syntax

Karlsruhe Institute of Technology

Some rules are only sound in a certain context

$$\text{modusPonens}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A, A \mathbin{->} B \Longrightarrow \Delta}$$

Taclet

```
modusPonens {
  \assumes ( A ==> )
  \find ( A -> B ==> )
  \replacewith( B ==> )
};
```

Taclet Syntax

Some rules are only sound in a certain context

$$\text{modusPonens}\ \frac{\Gamma, A, B \Longrightarrow \Delta}{\Gamma, A, A \mathbin{-\!\!>} B \Longrightarrow \Delta}$$

Taclet

```
modusPonens {
  \assumes ( A ==> )
  \find ( A -> B ==> )
  \replacewith( B ==> )
};
```

Taclet Syntax

Proof Splitting: andRight

$$\frac{\Gamma \Longrightarrow A, \Delta \quad \Gamma \Longrightarrow B, \Delta}{\Gamma \Longrightarrow A \;\&\; B, \Delta}$$

```
andRight {
  \find ( ==> A & B )
  \replacewith (==> A );
  \replacewith (==> B )
};
```

Variable Conditions: allRight

$$\frac{\Gamma \Longrightarrow \{x/c\}\Phi, \Delta}{\Gamma \Longrightarrow \forall T\; x; \Phi, \Delta}, \text{c new}$$

```
allRight {
  \find ( ==> \forall x;phi )
  \varcond(\new(c,\dependingOn(phi)))
  \replacewith ( ==> {\subst x;c}phi )
};
```

Taclet Syntax

Proof Splitting: andRight

$$\frac{\Gamma \Longrightarrow A, \Delta \quad \Gamma \Longrightarrow B, \Delta}{\Gamma \Longrightarrow A \mathbin{\&} B, \Delta}$$

```
andRight {
  \find ( ==> A & B )
  \replacewith (==> A );
  \replacewith (==> B )
};
```

Variable Conditions: allRight

$$\frac{\Gamma \Longrightarrow \{x/c\}\Phi, \Delta}{\Gamma \Longrightarrow \forall T\ x; \Phi, \Delta}, \text{c new}$$

```
allRight {
  \find ( ==> \forall x;phi )
  \varcond(\new(c,\dependingOn(phi)))
  \replacewith ( ==> {\subst x;c}phi )
};
```

Taclets for Program Transformations

$$\frac{\Gamma \Longrightarrow \left\langle \begin{array}{l} \pi\ \texttt{if (exc == null) \{} \\ \quad \texttt{try\{ throw new NPE(); catch(T e) \{r\};} \\ \texttt{\} else if (exc instanceof T) \{e=exc; r\}} \\ \texttt{else throw exc;}\ \omega \end{array} \right\rangle \phi}{\Gamma \Longrightarrow \texttt{<}\pi\ \texttt{try\{throw exc; q\} catch(T e)\{r\};}\ \omega\texttt{>}\phi}$$

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

Part IV

Verifying Information-Flow Properties

Part IV

Verifying Information-Flow Properties

12 Information Flow

13 Formalisation in DL

14 Objects and Information Flow

Language-based Information Security

Karlsruhe Institute of Technology

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.

Language-based Information Security

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?

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

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

Note

Sequential Java programs
Termination not considered

Which methods are secure?

```
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?

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

NOT SECURE

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

Examples

Note

Sequential Java programs
Termination not considered

Which methods are secure?

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

NOT SECURE

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

Examples

Note

Sequential Java programs
Termination not considered

Which methods are secure?

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

NOT SECURE

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

NOT SECURE

Examples

Which methods are secure?

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

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

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

Examples

Which methods are secure?

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

SECURE

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

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

Examples

Which methods are secure?

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

SECURE

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

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

Examples

Which methods are secure?

```
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?

```
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?

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

SECURE

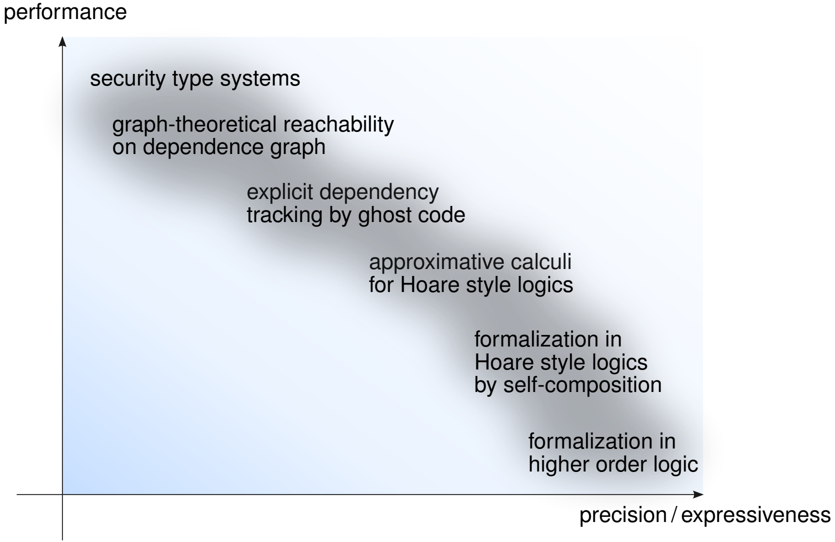
```
void m_5() {
  low = high;
  low = low-high;
}
```

SECURE

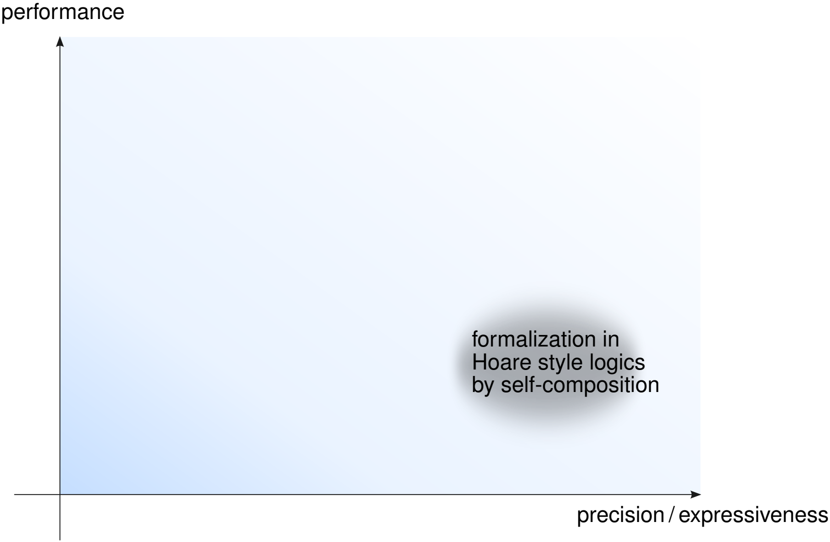
```
void m_6() {
  if (false) low = high;
}
```

SECURE

Information-flow Analysis Approaches



Information-flow Analysis Approaches



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

Karlsruhe Institute of Technology

Definition (Low-equivalence on states)

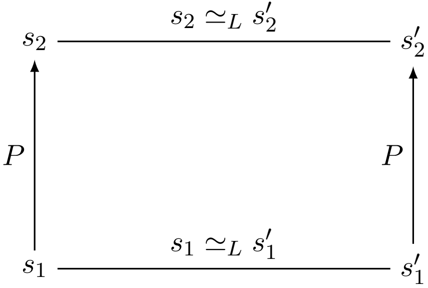
Two states are low-equivalent if they assign the same values to low variables.

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



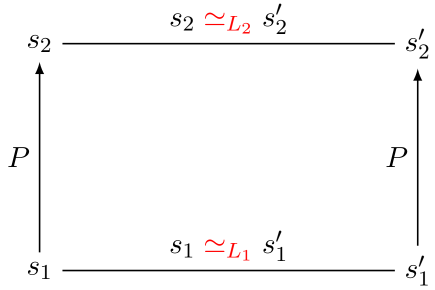
where

$$s_i \simeq_L s'_i \quad \Leftrightarrow \quad \forall\, v \in L \; (v^{s_i} = v^{s'_i})$$

Non-interference

Karlsruhe Institute of Technology

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



where

$$s_i \simeq_{L_i} s'_i \quad \Leftrightarrow \quad \forall\, v \in L_i \; (v^{s_i} = v^{s'_i})$$

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.

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

Problem

Not suitable for automatic verification
$\rightsquigarrow$ 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.

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

Problem

Not suitable for automatic verification
$\leadsto$ instantiation of existential quantifier difficult.

Non-Interference in JavaDL (Version 2)

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.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \; \{low := in_l\}(\\
&\quad \{high := in_h^1\}[P]out_l^1 = low \\
&\quad \land \{high := in_h^2\}[P]out_l^2 = low \\
&\quad \rightarrow out_l^1 = out_l^2 \\
&)
\end{aligned}
$$

Declassification

Intuition

Let $T(high, low)$ 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

Intuition

Let $T(high, low)$ 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.

$$
\begin{aligned}
&\forall in_l \forall in_h^1 \forall in_h^2 \forall out_l^1 \forall out_l^2 \ \{low := in_l\}(\\
&\qquad \{high := in_h^1\}T = \{high := in_h^2\}T \\
&\quad \land \{high := in_h^1\}[P]out_l^1 = low \\
&\quad \land \{high := in_h^2\}[P]out_l^2 = low \\
&\quad \to out_l^1 = out_l^2 \\
&)
\end{aligned}
$$

DEMO

Verifying Information-flow Properties with the KeY Tool

Object-Sensitive Non-interference

Leakage by aliasing

```
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

```
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

```
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 creation and object identity

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

SECURE

Assumption

- References are opaque
- Only comparison of objects by == is observable

Object creation and object identity

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

SECURE

Assumption

- References are opaque
- Only comparison of objects by == is observable

Object creation and object identity

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

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

Object creation and object identity

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

NOT SECURE

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

Object creation and object identity

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

NOT SECURE

```
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; }
```

NOT SECURE

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

NOT SECURE

Object-Sensitive Non-interference

Idea

ISOMORPHIC object structures in low variables
IDENTITY NOT required

Instead of

$$s_i \simeq_{L_i} s_i' \quad \Leftrightarrow \quad \forall\, v \in L_i\ (v^{s_i} = v^{s_i'})$$

use

$$s_i \simeq_{L_i}^{\pi_i} s_i' \quad \Leftrightarrow \quad \forall\, v \in L_i\ (\pi_i(v^{s_i}) = v^{s_i'})$$

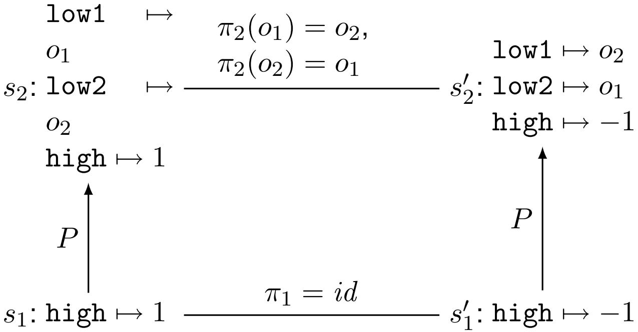
where

π_1, π_2 are compatible

i.e.

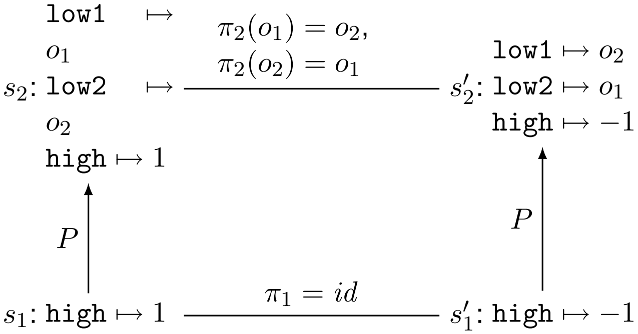
$\pi_1(o) = \pi_2(o)$ if o observable in both s_1 and s_2

Object-Sensitive Non-interference



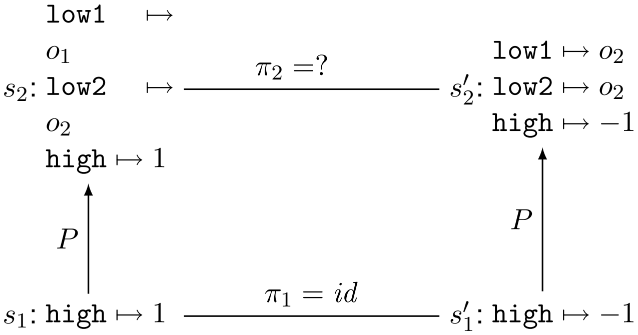
Secure because o_1, o_2 not observable in s_1

Object-Sensitive Non-interference



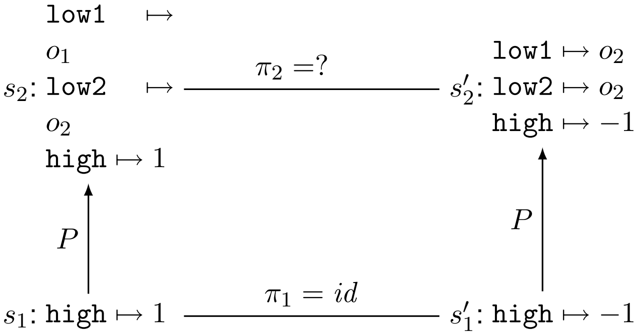
Secure because o_1, o_2 not observable in s_1

Object-Sensitive Non-interference



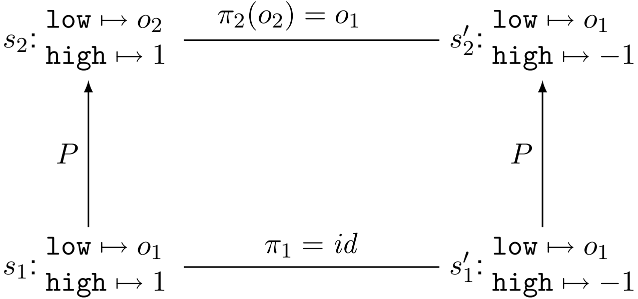
Not secure because no suitable π_2 exists

Object-Sensitive Non-interference



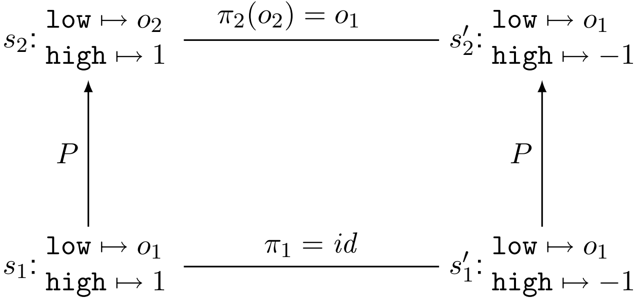
Not secure because no suitable π_2 exists

Object-Sensitive Non-interference



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

Object-Sensitive Non-interference

Optimisations

- L_1, L_2 sequences of low terms (instead of sets of variables)
- π_1 can be fixed to be id

DEMO

Objects and Information-flow with the KeY Tool

Part V

Wrap Up

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 $\leftrightarrow$ 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

Part V

Wrap Up

Part V

Wrap Up

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

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

THE END

(for now)