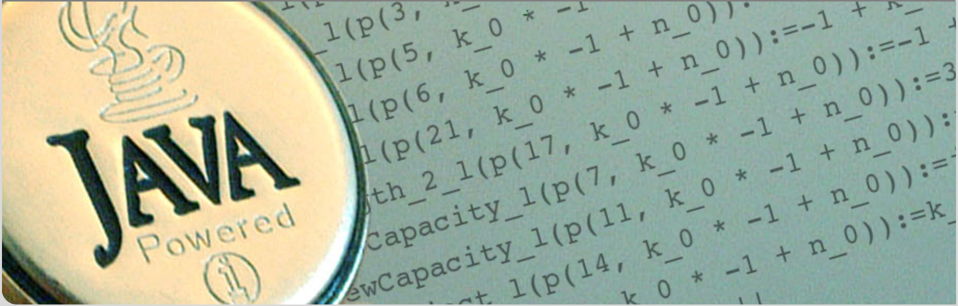
KIT
Karlsruhe Institute of Technology

# Applications of Formal Verification

## Functional Verification of Java Programs: Java Dynamic Logic

Dr. Vladimir Klebanov · Dr. Mattias Ulbrich · (Folien nach Prof. Dr. Bernhard Beckert) | SS 2015

KIT – Institut für Theoretische Informatik



1 JAVA CARD DL

2 Sequent Calculus

3 Rules for Programs: Symbolic Execution

4 A Calculus for 100% JAVA CARD

5 Loop Invariants

# Syntax and Semantics

## Syntax

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

## Semantics (Kripke)

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

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

# Why Dynamic Logic?

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

- Programs are "first-class citizens"
- Real Java syntax

# Why Dynamic Logic?

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

Hoare triple $\{\psi\}\ \alpha\ \{\phi\}$ equiv. to DL formula $\psi \rightarrow [\alpha]\phi$

# Why Dynamic Logic?

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

Not merely partial/total correctness:

- can employ programs for specification (e.g., verifying program transformations)
- can express security properties (two runs are indistinguishable)
- extension-friendly (e.g., temporal modalities)

# Dynamic Logic Example Formulas

$(\texttt{balance} >= c \wedge \texttt{amount} > 0) \rightarrow$
$\langle \texttt{charge(amount);} \rangle\, \texttt{balance} > c$

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

- Program formulas can appear nested

$\texttt{\textbackslash forall}\ \mathit{int}\ \mathit{val};\ ((\langle \texttt{p} \rangle \texttt{x} = \mathit{val}) \longleftrightarrow (\langle \texttt{q} \rangle \texttt{x} = \mathit{val}))$

- 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

Karlsruhe Institute of Technology

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

# Validity

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

We need a calculus for checking validity of formulas

# Teil


1. JAVA CARD DL
2. Sequent Calculus
3. Rules for Programs: Symbolic Execution
4. A Calculus for 100% JAVA CARD
5. Loop Invariants

# Sequents and their Semantics

## Syntax

$$\underbrace{\psi_1, \ldots, \psi_m}_{\textit{Antecedent}} \quad \Longrightarrow \quad \underbrace{\phi_1, \ldots, \phi_n}_{\textit{Succedent}}$$

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

## Semantics

Same as the formula

$$(\psi_1 \wedge \cdots \wedge \psi_m) \quad \rightarrow \quad (\phi_1 \vee \cdots \vee \phi_n)$$

# 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

# Some Simple Sequent Rules

NOT_LEFT $$\frac{\Gamma \Longrightarrow A, \Delta}{\Gamma, \neg A \Longrightarrow \Delta}$$

IMP_LEFT $$\frac{\Gamma \Longrightarrow A, \Delta \qquad \Gamma, B \Longrightarrow \Delta}{\Gamma, A \to B \Longrightarrow \Delta}$$

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

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

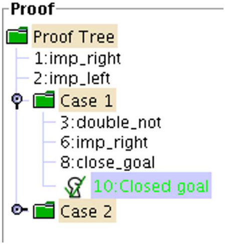
ALL_LEFT $$\frac{\Gamma, \backslash\texttt{forall}\ t\ x; \phi,\ \{x/e\}\phi \Longrightarrow \Delta}{\Gamma, \backslash\texttt{forall}\ t\ x; \phi \Longrightarrow \Delta}$$

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

# 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



# Teil

1. JAVA CARD DL
2. Sequent Calculus
3. **Rules for Programs: Symbolic Execution**
4. A Calculus for 100% JAVA CARD
5. Loop Invariants

# 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 $\pi$
active statement `i=0;`
rest $\omega$

- Sequent rules execute symbolically the active statement

# Rules for Symbolic Program Execution

Karlsruhe Institute of Technology

## If-then-else rule

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

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

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

## Simple assignment rule

$$\frac{\Gamma \Longrightarrow \{\mathit{loc} := \mathit{val}\}\langle\omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \langle \mathit{loc}\texttt{=}\mathit{val}\texttt{;} \;\; \omega\rangle\phi, \Delta}$$

# Treating Assignment with “Updates”

## Updates

syntactic elements in the logic – (explicit substitutions)

## Elementary Updates

$$\{\mathit{loc} := \mathit{val}\}\,\phi$$

where

- *loc* is a program variable
- *val* is an expression type-compatible with *loc*

## Parallel Updates

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

no dependency between the *n* components (but ‘last wins’ semantics)

# Why Updates?

## Updates are

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

## Advantages

- no renaming required
  (compared to another forward proof technique:
  strongest-postcondition calculus)
- delayed/minimised proof branching
  efficient aliasing treatment)

# Symbolic Execution with Updates (by Example)

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

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{x:=y} \,\|\, \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \,\|\, \mathtt{x:=y} \,\|\, \mathtt{y:=x}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x} \,\|\, \mathtt{x:=y}\}\{\mathtt{y:=t}\}\langle\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\{\mathtt{x:=y}\}\langle\mathtt{y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\mathtt{x} < \mathtt{y} \implies \{\mathtt{t:=x}\}\langle\mathtt{x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

$$\vdots$$

$$\implies \mathtt{x} < \mathtt{y} \to \langle\mathtt{int\ t=x;\ x=y;\ y=t;}\rangle\, \mathtt{y} < \mathtt{x}$$

# The theory of arrays

An abstract data

**Types:** Indices $\mathbb{I}$,

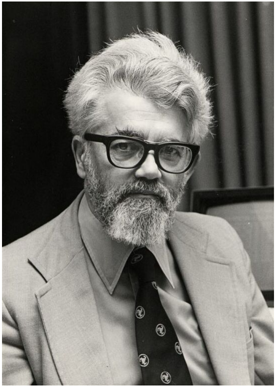
**Function symbol**

- *select* : *Array*
- *store* : *Array*(

Axioms

$\forall a, i, v.$

$\forall a, i, j, v.\ i \neq$ ... $t(a, j)$



John McCarthy (1927–2011):
Theory of arrays is decidable

Intuition

$\mathcal{D}(Array(\mathbb{I}, \mathbb{V}))$ represents the set of functions $\mathcal{D}(\mathbb{I}) \to \mathcal{D}(\mathbb{V})$

# Program State Representation

## Local program variables

Modeled as non-rigid constants

## Heap

Modeled with theory of arrays: $\mathbb{I} = \mathit{Object} \times \mathit{Field}$, $\mathbb{V} = \mathit{Any}$

| | |
|---|---|
| *heap* : | *Heap* (the heap in the current state) |
| *select* : | $\mathit{Heap} \times \mathit{Object} \times \mathit{Field} \to \mathit{Any}$ |
| *store* : | $\mathit{Heap} \times \mathit{Object} \times \mathit{Field} \times \mathit{Any} \to \mathit{Heap}$ |

## Some special program variables

| | |
|---|---|
| `self` | the current receiver object (`this` in Java) |
| `exc` | the currently active exception (`null` if none thrown) |
| `result` | the result of the method invocation |

# Teil


1. JAVA CARD DL
2. Sequent Calculus
3. Rules for Programs: Symbolic Execution
4. A Calculus for 100% JAVA CARD
5. Loop Invariants

# 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

## 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 invocations by the method's contract
5. Update simplification

# Loop Invariants

## Symbolic execution of loops: unwind

$$\textsc{unwindLoop}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{\textbf{if}(b) \{ p; \textbf{while}(b) p\}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \texttt{\textbf{while}(b) p}\ \omega]\phi, \Delta}$$

How to handle a loop with…

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

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

# Loop Invariants Cont'd

## Idea behind loop invariants

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

## Basic Invariant Rule

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}\, Inv, \Delta & \text{(initially valid)} \\ Inv,\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathtt{p}]\, Inv & \text{(preserved)} \\ Inv,\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathbf{while}\,\mathtt{(b)\ p}\ \omega]\phi, \Delta}$$

# Loop Invariants Cont'd

## Basic Invariant Rule: Problem

$$\text{loopInvariant}\ \frac{\Gamma \Longrightarrow \mathcal{U}\mathit{Inv}, \Delta \qquad \mathit{Inv},\ b \doteq \mathtt{TRUE} \Longrightarrow [\mathtt{p}]\mathit{Inv} \qquad \mathit{Inv},\ b \doteq \mathtt{FALSE} \Longrightarrow [\pi\ \omega]\phi}{\Gamma \Longrightarrow \mathcal{U}[\pi\, \mathbf{while}\, (\mathtt{b})\ \mathtt{p}\ \omega]\phi, \Delta}$$

(initially valid)
(preserved)
(use case)

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

# Example

Precondition: $a \neq \texttt{null}$ & *ClassInv*

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

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

Loop invariant: $0 \leq \texttt{i} \land \texttt{i} \leq \texttt{a.length}$
$\land \forall\, \texttt{int}\ x;\ (0 \leq x < \texttt{i} \rightarrow \texttt{a}[x] \doteq 1)$
$\land\ \texttt{a} \neq \texttt{null}$
$\land\ \mathit{ClassInv}'$

# Keeping the Context

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

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

# Example with Improved Invariant Rule

Precondition: $\mathtt{a} \doteq\!\!\!\!/\ \mathbf{null}\ \&\ \mathit{ClassInv}$

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

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

Loop invariant: $0 \leq \mathtt{i}\ \wedge\ \mathtt{i} \leq \mathtt{a.length}$
$\wedge\ \forall\,\mathbf{int}\ x;\ (0 \leq x < \mathtt{i} \rightarrow \mathtt{a[}x\mathtt{]} \doteq 1)$

# Example in JML/Java – Loop.java

```
public int[] a;
/*@ public normal_behavior
  @  ensures (\forall int x; 0<=x && x<a.length; a[x]==1);
  @  diverges true;
  @*/
public void m() {
  int i = 0;
  /*@ loop_invariant
    @  (0 <= i && i <= a.length &&
    @   (\forall int x; 0<=x && x<i; a[x]==1));
    @ assignable i, a[*];
    @*/
  while(i < a.length) {
    a[i] = 1;
    i++;
  }
}
```

# Example

$\forall$ `int` $x$;
 (n $\doteq x \land x >= 0 \rightarrow$
  [`i = 0; r = 0;`
   `while (i<n) { i = i + 1; r = r + i;}`
   `r=r+r-n;`
  ]r $\doteq ?x * x$)

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

Solution:

```
@ loop_invariant
@     i>=0 && 2*r == i*(i + 1) && i <= n;
@ assignable i, r;
```

File: `Loop2.java`

# Hints

## Proving **`assignable`**

- The invariant rule *assumes* that **`assignable`** is correct
  E.g., with **`assignable \nothing;`** one can prove nonsense
- Invariant rule of KeY generates *proof obligation* that ensures correctness of **`assignable`**

## Setting in the KeY Prover when proving loops

- Loop treatment: *Invariant*
- Quantifier treatment: *No Splits with Progs*
- If program contains `*, /`:
  Arithmetic treatment: *DefOps*
- Is search limit high enough (time out, rule apps.)?
- When proving partial correctness, add **`diverges true;`**

# Total Correctness

Karlsruhe Institute of Technology

## Find a decreasing integer term $v$ (called *variant*)

Add the following premises to the invariant rule:

- $v \geq 0$ is initially valid
- $v \geq 0$ is preserved by the loop body
- $v$ is strictly decreased by the loop body

## Proving termination in JML/Java

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

## Example: The `array` loop

```
@ decreasing a.length - i;
```

Files:

- `LoopT.java`
- `Loop2T.java`