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
1. **JAVA CARD DL**

2. Sequent Calculus

3. Rules for Programs: Symbolic Execution

4. A Calculus for 100% JAVA CARD

5. Loop Invariants
### 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

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

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

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

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

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

\[
\begin{align*}
\forall \text{int} & \quad j; \quad (j \geq 0 \land j < a.\text{length} \rightarrow \text{max} \geq a[j]) \\
\& \\
(a.\text{length} > 0 \rightarrow \exists \text{int} \quad j; \quad (j \geq 0 \land j < a.\text{length} \land \text{max} = a[j]))
\end{align*}
\]
Variables

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

A Java Card DL formula is valid iff it is true in all states.

We need a calculus for checking validity of formulas.
Sequent Calculus

Rules for Programs: Symbolic Execution

A Calculus for 100% Java Card

Loop Invariants
Sequents and their Semantics

Syntax

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

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

Semantics

Same as the formula

\[ (\psi_1 \land \cdots \land \psi_m) \quad \Rightarrow \quad (\phi_1 \lor \cdots \lor \phi_n) \]
Sequent Rules

General form

RULE_NAME

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

\[ \Gamma \implies \Delta \]

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

Soundness

If all premisses are valid, then the conclusion is valid

Use in practice

Goal is matched to conclusion
Some Simple Sequent Rules

### NOT_LEFT

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

### IMP_LEFT

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

### CLOSE_GOAL

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

### CLOSE_BY_TRUE

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

### ALL_LEFT

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

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

Proof tree

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

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

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

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

If-then-else rule

\[
\Gamma, B = true \implies \langle p \ \omega \rangle \phi, \Delta \quad \Gamma, B = false \implies \langle q \ \omega \rangle \phi, \Delta
\]

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

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

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

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

Simple assignment rule

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

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

Updates

syntactic elements in the logic – (explicit substitutions)

Elementary Updates

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

where
- \text{loc} is a program variable
- \text{val} is an expression type-compatible with \text{loc}

Parallel Updates

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

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)

\[\begin{align*}
x < y & \implies x < y \\
\vdots \\
x < y & \implies \{x := y \mid y := x\}\langle \rangle y < x \\
\vdots \\
x < y & \implies \{t := x || x := y \mid y := x\}\langle \rangle y < x \\
\vdots \\
x < y & \implies \{t := x \mid x := y\}\{y := t\}\langle \rangle y < x \\
\vdots \\
x < y & \implies \{t := x\}\{x := y\}\langle y = t; \rangle y < x \\
\vdots \\
x < y & \implies \{t := x\}\langle x = y; \ y = t; \rangle y < x \\
\vdots \\
\implies x < y & \implies \langle \text{int } t = x; \ x = y; \ y = t; \rangle y < x
\end{align*}\]
The theory of arrays

An abstract datatype

Types: Indices \( \mathbb{I} \), Values \( \mathbb{V} \)

Function symbols:
- select : Array(\( \mathbb{I} \), \( \mathbb{V} \)) \( \times \mathbb{I} \rightarrow \mathbb{V} \)
- store : Array(\( \mathbb{I} \), \( \mathbb{V} \)) \( \times \mathbb{I} \times \mathbb{V} \rightarrow \) Array(\( \mathbb{I} \), \( \mathbb{V} \))

Axioms
\[
\forall a, i, v. \quad \text{select}(\text{store}(a, i, v), i) = v
\]
\[
\forall a, i, j, v. \quad i \neq j \rightarrow \text{select}(\text{store}(a, i, v), j) = \text{select}(a, j)
\]

Intuition
\( D(\text{Array}(\mathbb{I}, \mathbb{V})) \) represents the set of functions \( D(\mathbb{I}) \rightarrow D(\mathbb{V}) \)

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

Local program variables
Modeled as non-rigid constants

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

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

Some special program variables
- $\text{self}$: the current receiver object ($\text{this}$ in Java)
- $\text{exc}$: the currently active exception ($\text{null}$ if none thrown)
- $\text{result}$: 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

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.
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

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

How to handle a loop with…

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

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

Idea behind loop invariants

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

Basic Invariant Rule

$$
\begin{align*}
\Gamma &\Rightarrow U Inv, \Delta & \text{(initially valid)} \\
Inv, b \Downarrow \text{TRUE} &\Rightarrow [p] Inv & \text{(preserved)} \\
Inv, b \Downarrow \text{FALSE} &\Rightarrow [\pi \omega] \phi & \text{(use case)} \\
\Gamma &\Rightarrow U [\pi \text{while}(b) p \omega] \phi, \Delta
\end{align*}
$$
Loop Invariants Cont’d

Basic Invariant Rule: Problem

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

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

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

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

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

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

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

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

Loop invariant: \( 0 \leq i \land i \leq a.\text{length} \land \forall \text{int } x; (0 \leq x < i \rightarrow a[x] = 1) \land a \neq \text{null} \land \text{ClassInv}' \)
Want to keep part of the context that is *unmodified* by loop

**assignable** clauses for loops can tell what might be modified

```plaintext
@ assignable i, a[ ];
```
Example with Improved Invariant Rule

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

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

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

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

```java
public int[] a;

/*@ public normal_behavior
@ ensures (\forall int x; 0<=x \&\& x<a.length; a[x]==1);
@ diverges true;
@*/

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

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

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

Solution:

@ loop_invariant
@ i >= 0 \&\& 2 \ast r == i \ast (i + 1) \&\& i \leq n;
@ assignable i, r;

File: Loop2.java
Hints

Proving assignable

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

Setting in the KeY Prover when proving loops

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

Find a decreasing integer term \( v \) (called variant)

Add the following premisses to the invariant rule:
- \( v \geq 0 \) is initially valid
- \( v \geq 0 \) is preserved by the loop body
- \( v \) is strictly decreased by the loop body

Proving termination in JML/Java

- Remove directive \texttt{diverges true;}
- Add directive \texttt{decreasing v;} to loop invariant
- KeY creates suitable invariant rule and PO (with \( \langle . . . \rangle \phi \))

Example: The array loop

\@ \texttt{decreasing a.length - i;}

Files:
- LoopT.java
- Loop2T.java