# Disproving in Java Dynamic Logic

Philipp Rümmer
Chalmers University of Technology, Gothenburg
philipp@cs.chalmers.se

18th October 2006

# Program Analysis:

Setting: program and specification are given

# Program Analysis: Deductive Verification

Setting: program and specification are given

Deductive Verification

- Hypothesis: program is *correct*
- Search for logical argument why program is correct
- Incorrect program $\rightarrow$ non-termination of search

# Program Analysis: Deductive Verification and Testing

Setting: program and specification are given

Deductive Verification

- Hypothesis: program is *correct*
- Search for logical argument why program is correct
- Incorrect program $\rightarrow$ non-termination of search

Testing

- Hypothesis: program is *incorrect*
- Search for program inputs that witness incorrectness
- Correct program $\rightarrow$ non-termination of search

# Outline

In this talk:

- Proving program incorrectness using the JavaDL calculus
- Sound and complete (relatively to DL calculus):
  Solutions are guaranteed to reveal program incorrectness
  $\rightarrow$ Disproving
- Classes of program inputs that reveal bugs can be found
  $\rightarrow$ Symbolic reasoning
- Method is tailored to Java programs operating on heap

Context:

- Dynamic logic for Java, non-rigid functions, updates
- Implemented in KeY

Joint work with Muhammad Ali Shah

# Essential Problem: Construction of a Pre-State

Wanted: pre-states that ...

- satisfy pre-conditions
- make the program choose a certain path
- violate post-conditions for this path

# Masterplan for Constructing Pre-State

1. Verification condition characterising incorrectness
   - Negated correctness condition + quantifier over pre-states
2. Free-variable DL sequent calculus
   - Construction of models of pre-conditions
   - Construction of counter-models of post-conditions
   - Realises symbolic program execution
3. Pre-state: substitutions applied to close proof

# 1. Verification Condition Characterising Incorrectness

$$\exists\, \textit{pre-state}.\; \neg\big(\textit{pre-conditions} \rightarrow \langle\ \textit{program code}\ \rangle\ \ \textit{post-conditions}\big)$$

This formula holds if the program (for valid input)

- does not terminate, or
- terminates but violates the post-conditions.

# Quantification over Pre-States

$$\exists\, \textit{pre-state}.$$
$$\neg(\textit{pre-conditions} \rightarrow \langle\ \textit{program code}\ \rangle\ \ \textit{post-conditions})$$

JavaDL considers program symbols as first-order vocabulary (non-rigid functions)

| | | |
|---|---|---|
| Variables, class attributes | $\longrightarrow$ | Constants |
| Instance attributes | $\longrightarrow$ | Unary functions |
| Arrays | $\longrightarrow$ | Binary functions |

# Quantification over Pre-States

$$\exists\ \textit{pre-state}.\ \neg(\textit{pre-conditions} \rightarrow \langle\ \textit{program code}\ \rangle\ \ \textit{post-conditions})$$

JavaDL considers program symbols as first-order vocabulary (non-rigid functions)

| | | |
|---|---|---|
| Variables, class attributes | $\rightarrow$ | Constants |
| Instance attributes | $\rightarrow$ | Unary functions |
| Arrays | $\rightarrow$ | Binary functions |

For first-order quantification:

- 2nd encoding with algebraic datatypes (“store”) necessary

# Quantification over Pre-States

$$\exists\, \textit{pre-state}.\ \neg(\textit{pre-conditions} \rightarrow \langle\ \textit{program code}\ \rangle\ \ \textit{post-conditions})$$

JavaDL considers program symbols as first-order vocabulary (non-rigid functions)

| | | |
|---|---|---|
| Variables, class attributes | $\rightarrow$ | Constants |
| | $\rightarrow$ | scalar variables |
| Instance attributes | $\rightarrow$ | Unary functions |
| | $\rightarrow$ | lists/arrays |
| Arrays | $\rightarrow$ | Binary functions |
| | $\rightarrow$ | lists of lists |

For first-order quantification:

- 2nd encoding with algebraic datatypes (“store”) necessary

# Quantification over Pre-States

$$\exists \, \textit{pre-state}. \\ \neg(\textit{pre-conditions} \rightarrow \langle \; \textit{program code} \; \rangle \; \textit{post-conditions})$$

JavaDL considers program symbols as first-order vocabulary (non-rigid functions)

| | | |
|---|---|---|
| Variables, class attributes | $\rightarrow$ | Constants |
| | $\rightarrow$ | scalar variables |
| Instance attributes | $\rightarrow$ | Unary functions |
| | $\rightarrow$ | lists/arrays |
| Arrays | $\rightarrow$ | Binary functions |
| | $\rightarrow$ | lists of lists |

For first-order quantification:

- 2nd encoding with algebraic datatypes (“store”) necessary
- The two representations can be connected using updates

# Quantification over Pre-States

$$\exists\, \textit{pre-state}.\ \{\textit{pre-state}\}$$
$$\neg(\textit{pre-conditions} \rightarrow \langle\ \textit{program code}\ \rangle\ \ \textit{post-conditions})$$

JavaDL considers program symbols as first-order vocabulary (non-rigid functions)

| | | |
|---|---|---|
| Variables, class attributes | $\rightarrow$ | Constants |
| | $\rightarrow$ | scalar variables |
| Instance attributes | $\rightarrow$ | Unary functions |
| | $\rightarrow$ | lists/arrays |
| Arrays | $\rightarrow$ | Binary functions |
| | $\rightarrow$ | lists of lists |

For first-order quantification:

- 2nd encoding with algebraic datatypes (“store”) necessary
- The two representations can be connected using updates

# Example

- Variables, class attributes:

$$\exists\, p_V : \mathit{int}.\ \{p := p_V\}\ \ \neg(p \geq -5 \rightarrow \langle\, \alpha \,\rangle\; p > 3)$$

# Example

- Variables, class attributes:

$$\exists\, p_V : int.\ \{p := p_V\}\ \neg(p \geq -5 \rightarrow \langle\, \alpha\, \rangle\ p > 3)$$

- Instance attributes:

```
class C {
  int a;
}
```

$$\exists\, a_V : intList.\ \{\texttt{for}\ x : nat\ \{C.get(x).a := a_V \downarrow x\}\}\ \neg(\ \dots\ )$$

$$[a_0, \dots, a_n] \downarrow i\ := \begin{cases} a_i & \text{for } i \leq n \\ 0 & \text{otherwise} \end{cases} \qquad (i : nat)$$

# Example

- Variables, class attributes:

$$\exists\, p_V : \mathit{int}.\ \{p := p_V\}\ \neg(p \geq -5 \rightarrow \langle\, \alpha \,\rangle\, p > 3)$$

- Instance attributes:

```
class C {
  int a;
}
```

$$\exists\, a_V : \mathit{intList}.\ \{\mathtt{for}\ x : \mathit{nat}\ \{C.\mathit{get}(x).a := a_V \downarrow x\}\}\ \neg(\ \ldots\ )$$

- ...similarly for arrays, etc

$$[a_0, \ldots, a_n] \downarrow i\ := \begin{cases} a_i & \text{for } i \leq n \\ 0 & \text{otherwise} \end{cases} \qquad (i : \mathit{nat})$$

# 2. Reasoning about Incorrectness Conditions

$$\exists\, \textit{pre-state}.\ \{\textit{pre-state}\}$$
$$\neg(\textit{pre-conditions} \rightarrow \langle\ \textit{program code}\ \rangle\ \ \textit{post-conditions})$$

Construction of pre-state using DL sequent calculus:

- Ground reasoning
  $\rightarrow$ Standard testing technique
  $\rightarrow$ Choose pre-state upfront, actual program execution
- Metavariables, substitutions, backtracking
  $\rightarrow$ Standard proving technique
  $\rightarrow$ Symbolic execution
- Metavariables, constraints
  $\rightarrow$ Backtracking-free proving
  $\rightarrow$ Symbolic execution

# Example: Metavariables and Symbolic Execution

$$\exists\, p_V : int.\ \{p := p_V\} \quad \neg(p \geq -5 \rightarrow \langle\, \alpha \,\rangle\ p > 3)$$

After introducing metavariables, simplification:

$$P \geq -5\ \wedge\ \neg\{p := P\}\ \langle\, \alpha \,\rangle\ p > 3$$

# Example: Metavariables and Symbolic Execution (2)

$$\dfrac{\vdash\; P \geq -5 \qquad \begin{matrix} P+1>3, P\geq 0 \;\vdash \\ \vdots \end{matrix} \qquad \begin{matrix} -P>3 \;\vdash\; P\geq 0 \\ \vdots \end{matrix}}{\vdash\; P \geq -5 \land \neg\{p := P\}\,\langle\, \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p;\,\rangle\ p > 3}$$

- In general: symbolic execution does not terminate
- Requirements on an instantiation of metavariables:
  All branches have to be closed

# Metavariables and Search

With backtracking:

- Try to close one branch, then
- check whether the solution also suits the other branches
- Depth-first search
- Connections to logical programming

With constraints:

- Solutions of all branches are collected in a fair manner (as enumeration of constraints)
- Goal: find compatible solutions
- Breadth-first search

# Constraint Language to Describe Solutions?

Unification constraints:

- Conjunction of unification conditions $s \equiv t$
- Work well for algebraic datatypes
- Not very efficient for the integer-centric Java
  $\rightarrow$ Constraints like $X + 1 \equiv 2$ are inconsistent
  $\rightarrow$ Conditions like $P \geq -5$ cannot be expressed

# “Linear Unification Constraints” (LUC)

- Unification modulo theory
  (here: linear/Presburger Arithmetic)
- Constraints with further predicates ($\neq$, $<$, $\leq$)

# “Linear Unification Constraints” (LUC)

- Unification modulo theory
  (here: linear/Presburger Arithmetic)
- Constraints with further predicates ($\neq$, $<$, $\leq$)

$$
\begin{aligned}
\langle\text{LUC}\rangle \quad &::= \quad \langle\text{atom}\rangle \Big( \wedge \langle\text{atom}\rangle \Big) * \\
\langle\text{atom}\rangle \quad &::= \quad \langle\text{term}\rangle = \langle\text{term}\rangle \mid \\
& \quad\quad\; \langle\text{intTerm}\rangle < \langle\text{intTerm}\rangle \mid \\
& \quad\quad\; \langle\text{intTerm}\rangle \leq \langle\text{intTerm}\rangle \mid \\
& \quad\quad\; \langle\text{intTerm}\rangle \neq \langle\text{intTerm}\rangle
\end{aligned}
$$

# “Linear Unification Constraints” (LUC)

- Unification modulo theory
  (here: linear/Presburger Arithmetic)
- Constraints with further predicates ($\neq$, $<$, $\leq$)

$$
\begin{array}{rcl}
\langle \text{LUC} \rangle & ::= & \langle \text{atom} \rangle \Big( \wedge \langle \text{atom} \rangle \Big) * \\
\langle \text{atom} \rangle & ::= & \langle \text{term} \rangle = \langle \text{term} \rangle \mid \\
 & & \langle \text{intTerm} \rangle < \langle \text{intTerm} \rangle \mid \\
 & & \langle \text{intTerm} \rangle \leq \langle \text{intTerm} \rangle \mid \\
 & & \langle \text{intTerm} \rangle \neq \langle \text{intTerm} \rangle
\end{array}
$$

- So far unknown (work in progress):
  Decidability of consistency

# In the Example

$$\dfrac{\vdash\ P \geq -5 \qquad \dfrac{P+1>3,\ P \geq 0\ \vdash}{\vdots} \qquad \dfrac{-P>3\ \vdash\ P \geq 0}{\vdots}}{\vdash\ P \geq -5 \wedge \neg\{p := P\}\ \langle\, \texttt{if}\ (p \geq 0)\ p = p+1;\ \texttt{else}\ p = -p;\ \rangle\ p > 3}$$

- Simultaneous solution of all branches:

$$P \geq -5\ \wedge\ P+1 \not> 3\ \wedge\ -P \not> 3$$

$$\Leftrightarrow$$

$$P \in [-3, 2]$$

(post-conditions can be violated on both branches)

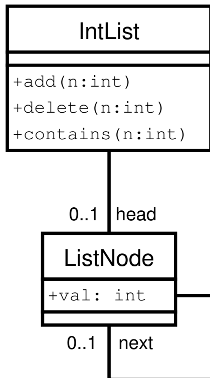
# (Slightly) Less Artificial Example . . .

```
public class IntList {
  private ListNode head;
  public void add (int n) { ... }

  /*@
    @ public normal_behavior
    @ ensures !contains(n);
    @*/
  public void delete(int n) {
    ListNode cur = head, prev = head;
    while (cur != null) {
      if (cur.val == n) prev.next = cur.next;
      else              prev = cur;
      cur = cur.next;
    }
  }

  public /*@ pure @*/ boolean contains(int n) {
    ListNode temp = head;
    while (temp != null) {
      if (temp.val == n) return true;
      temp = temp.next;
    }
    return false;
  }
}
```

```
class ListNode {
  public int      val;
  public ListNode next;
}
```



# Proving Method `IntList.delete` Incorrect . . .

$$\exists\, k_{IL}, k_{LN}, o_V : nat.\ \exists\, n_V : int.\ \exists\, head_V, next_V : natList.\ \exists\, val_V : intList.$$

$$\{IntList.nextToCreate := k_{IL} \mid ListNode.nextToCreate := k_{LN}\}$$

$$\{\texttt{for}\ x : nat\ \{IntList.get(x).head := ListNode.get(0, head_V \downarrow x)\} \mid$$

$$\texttt{for}\ x : nat\ \{ListNode.get(x).next := ListNode.get(0, next_V \downarrow x)\} \mid$$

$$\texttt{for}\ x : nat\ \{ListNode.get(x).val := val_V \downarrow x\} \mid$$

$$o := IntList.get(0, o_V) \mid n := n_V\}$$

$$\neg\big(\ o \neq null \rightarrow \langle\ o.\texttt{delete}(n)\ \rangle\ \langle\ b = o.\texttt{contains}(n)\ \rangle\ b = FALSE\ \big)$$

# Proving Method `IntList.delete` Incorrect . . .

$$\exists\, k_{IL}, k_{LN}, o_V : nat.\ \exists\, n_V : int.\ \exists\, head_V, next_V : natList.\ \exists\, val_V : intList.$$

$$\{IntList.nextToCreate := k_{IL} \mid ListNode.nextToCreate := k_{LN}\}$$

$$\{\texttt{for}\ x : nat\ \{IntList.get(x).head := ListNode.get(0, head_V \downarrow x)\} \mid$$

$$\texttt{for}\ x : nat\ \{ListNode.get(x).next := ListNode.get(0, next_V \downarrow x)\} \mid$$

$$\texttt{for}\ x : nat\ \{ListNode.get(x).val := val_V \downarrow x\} \mid$$

$$o := IntList.get(0, o_V) \mid n := n_V\}$$

$$\neg(\ o \neq null \rightarrow \langle\ o.\texttt{delete}(n)\ \rangle\ \langle\ b = o.\texttt{contains}(n)\ \rangle\ b = FALSE\ )$$

Solution:

| $k_{IL}$ | $k_{LN}$ | $o_V$ | $n_V$ | $head_V$ | $next_V$ | $val_V$ |
|---|---|---|---|---|---|---|
| $K_{IL}$ | $K_{LN}$ | $0$ | $N_V$ | $[0, \ldots]$ | $[E, \ldots]$ | $[N_V, \ldots]$ |

$$K_{IL} > 0\ \wedge$$
$$K_{LN} > 0\ \wedge$$
$$E \geq K_{LN}$$

# Conclusions, Future Work

- JavaDL can be used as framework for showing incorrectness of programs
- Testing techniques $\rightarrow$ proof strategies
- More general counterexamples than using testing
- Relatively complete
  (provided that pre/post-conditions do not contain loose symbols)

- Properties of linear unification constraints?
- Combination of depth-first and breadth-first search?
- Non-termination of programs?