Source code based formal techniques for fault-detection, testing, and specification of Java programs

Christoph Gladisch

Karlsruhe Institute of Technology (KIT)

Formal Methods and Tools, University of Twente, 09.04.2015
The KeY-Platform

Karlsruhe Institut of Technology (KIT) (Germany)
Darmstadt University of Technology (Germany)
Chalmers University (Gothenburg, Sweden)

KeY-Tool

Christoph Gladisch: Techniques for fault-detection, testing, and specification:
Univ. Twente, 9.4.2015
The KeY-Platform

Basis of the platform

- dynamic logic
- symbolic execution (Java source code level)
- theorem proving
- explicit heap and dynamic frames

Techniques beyond deductive verification

- deductive bug detection; test generation; visual debugging
- abstract interpretation; loop invariant generation
- analysis of non-functional properties (e.g. information flow)

The KeY-Platform

**Basis of the platform**
- dynamic logic
- symbolic execution (Java source code level)
- theorem proving
- explicit heap and dynamic frames

**Techniques beyond deductive verification**
- deductive bug detection; test generation; visual debugging
- abstract interpretation; loop invariant generation
- analysis of non-functional properties (e.g. information flow)

Background – Deductive Software Verification

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Implementation → Specification → Annotations
Background – Deductive Software Verification

Implementation → Specification → Annotations

Correct

Unknown (evtl. Counterex.)
Motivation

Why is this important?

- Verification attempts usually fail. The reason is often unclear.
- Software-fault detection is important to localise problems.
Motivation

Why is this important?

- Verification attempts usually fail. The reason is often unclear.
- Software-fault detection is important to localise problems.
Motivation

Why is this important?

- Verification attempts usually fail. The reason is often unclear.
- Software-fault detection is important to localise problems.

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Motivation

Why is this important?

- Verification attempts usually fail. The reason is often unclear.
- Software-fault detection is important to localise problems.
Motivation

Why is this important?

- Verification attempts usually fail. The reason is often unclear.
- Software-fault detection is important to localise problems.
Step 1. Verification Attempt

\[ pre \rightarrow [p]\textit{post} \]

- \( pre \rightarrow [p]\textit{post} \) corresponds to the Hoare-Tripel \( \{pre\} p\{post\} \)
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic
Step 1. Verification Attempt

\[ pre \rightarrow [p] post \]

- \( pre \rightarrow [p] post \) corresponds to the Hoare-Tripel \( \{pre\}p\{post\} \)
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic
Step 1. Verification Attempt

- $pre \rightarrow [p]post$ corresponds to the Hoare-Tripel $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic
Step 1. Verification Attempt

- $pre \rightarrow [p]post$ corresponds to the Hoare-Tripel $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic
Step 1. Verification Attempt

- \( \text{pre} \rightarrow [p]\text{post} \) corresponds to the Hoare-Tripel \( \{\text{pre}\}p\{\text{post}\} \)
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Step 1. Verification Attempt

- $pre \rightarrow [p]post$ corresponds to the Hoare-Tripel $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic
Step 1. Verification Attempt

- $pre \rightarrow [p]post$ corresponds to the Hoare-Tripel $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic
Step 1. Verification Attempt

\[ \text{pre} \rightarrow [p] \text{post} \]
Step 2a. Finding a Counterexample

Principle

If $\neg S_4$ is satisfiable and $\neg S_4 \rightarrow \neg S_0$ holds, then $\neg S_0$ is satisfiable.

$\implies$ Implementation does not satisfy its spec.
Step 2b. Validity Preservation

Principle

If $\neg S_4$ is satisfiable and $\neg S_4 \rightarrow \neg S_0$ holds, then $\neg S_0$ is satisfiable.

$\Rightarrow$ Implementation does not satisfy the spec.
Step 2b. Validity Preservation

Principle

If $\neg S_4$ is satisfiable and $\neg S_4 \rightarrow \neg S_0$ holds, then $\neg S_0$ is satisfiable.

$\implies$ Implementation does not satisfy the spec.
Step 2b. Validity Preservation

Principle

If \( \neg S_4 \) is satisfiable and \( \neg S_4 \rightarrow \neg S_0 \) holds, then \( \neg S_0 \) is satisfiable.

\[ \Rightarrow \text{Implementation does not satisfy the spec.} \]
Step 2b. Validity Preservation

Principle

If \( \neg S_4 \) is satisfiable and \( \neg S_4 \rightarrow \neg S_0 \) holds, then \( \neg S_0 \) is satisfiable.

\[ \Rightarrow \text{Implementation does not satisfy the spec.} \]
Validity Preservation

Problem: Contract rules

- Loop invariant and method contract rules
- Validity preservation depends on the used contract
Validity Preservation

Validity Preservation Condition (Approach 1)

\[ \neg S_2 \rightarrow \neg S_1 \]
Validity Preservation

Validity Preservation Condition (Approach 2)

\[ \neg S_4 \rightarrow \neg S_1 \]
### Properties of the approach (Approach 2)
- Easy to implement
- Correct for fault-detection; no “false positives”
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification
- Problem $\models \neg S_4 \rightarrow \neg S_1$
- Acceleration is possible
Validity Preservation

Properties of the approach (Approach 2)

- Easy to implement
- Correct for fault-detection; no “false positives”
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification

Problem $\models \neg S_4 \rightarrow \neg S_1$

Acceleration is possible
## Validity Preservation

### Properties of the approach (Approach 2)

- Easy to implement
- Correct for fault-detection; no "false positives"
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification
- Problem $\models \neg S_4 \rightarrow \neg S_1$
- Acceleration is possible

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Validity Preservation

Properties of the approach (Approach 2)

- Easy to implement
- Correct for fault-detection; no “false positives”
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification
  
Problem $\models \neg S_4 \rightarrow \neg S_1$

Acceleration is possible
Validity Preservation

Properties of the approach (Approach 2)

- Easy to implement
- Correct for fault-detection; no “false positives”
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification
- Problem $\models \neg S_4 \rightarrow \neg S_1$
- Acceleration is possible
Validity Preservation

Properties of the approach (Approach 2)

- Easy to implement
- Correct for fault-detection; no “false positives”
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification

Problem $\models \neg S_4 \rightarrow \neg S_1$

Acceleration is possible
Validity Preservation

Properties of the approach (Approach 2)

- Easy to implement
- Correct for fault-detection; no “false positives”
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification

Problem $\models \neg S_4 \rightarrow \neg S_1$

Acceleration is possible

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Validity Preservation

Properties of the approach (Approach 2)

- Easy to implement
- Correct for fault-detection; no “false positives”
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification

Problem $\models \neg S_4 \rightarrow \neg (pre \rightarrow \{U\}[\text{while}(c)\{b\};]post)$

Acceleration is possible
Validity Preservation

Properties of the approach (Approach 2)

- Easy to implement
- Correct for fault-detection; no “false positives”
- Relatively complete for fault-detection
- Search space for faults is pruned significantly
- Starts with the actual goal: Verification

Predicate: $\lnot S_4 \rightarrow \lnot (pre \rightarrow \{U\}[while(c){b};]post)$

Acceleration is possible
Special Validity Preservation (Acceleration)

Validity Preservation (Approach 2)

\[ \neg S_4 \rightarrow \neg (pre \rightarrow \{U\}[\text{while}(c)\{b\};]post) \]
Special Validity Preservation (Acceleration)

Validity Preservation (Approach 2)

\( \neg S_4 \rightarrow \neg (pre \rightarrow \{U\}[while(c){b};]post) \)

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Special Validity Preservation (Approach 3)

\[((\{M^1 := M^2\} S_4) \land \{U\} M^2) \text{post}_B\) → S_4
**Validity Preservation – Evaluation**

**Computational overhead for validity preservation wrt. verification**

<table>
<thead>
<tr>
<th>Approach 2</th>
<th>Approach 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Diagram" /></td>
<td><img src="#" alt="Diagram" /></td>
</tr>
<tr>
<td>45%</td>
<td>10%</td>
</tr>
</tbody>
</table>

C. Gladisch. Could we have chosen a better loop invariant or method contract? TAP 2009.
Validity Preservation – Evaluation

Computational overhead for validity preservation wrt. verification

<table>
<thead>
<tr>
<th>Approach 2</th>
<th>Approach 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
<td><img src="https://via.placeholder.com/150" alt="Diagram" /></td>
</tr>
</tbody>
</table>

45% | 10%

C. Gladisch. Could we have chosen a better loop invariant or method contract? TAP 2009.
Problem

Satisfiability modulo theory (SMT) Solvers, e.g.: Z3, CVC3, Yices

Quantified formulas in combination with theories sometimes/often not solvable with SMT

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Problem

- Satisfiability modulo theory (SMT) Solvers, e.g.: Z3, CVC3, Yices
- Quantified formulas in combination with theories sometimes/often not solvable with SMT
Counterexample generation $\approx$ Model generation

$\forall x. (x \geq 0 \rightarrow \text{prev}(\text{next}(x)) = x) \land \ldots \land \Phi$

Problem

- Satisfiability modulo theory (SMT) Solvers, e.g.: Z3, CVC3, Yices
- Quantified formulas in combination with theories sometimes/often not solvable with SMT
Verification-based Satisfiability Proving (VSP)

\[\forall x. (x \geq 0 \rightarrow prev(next(x)) = x) \land \ldots \land \Phi\]

1. Generate program \( p \) from the selected quantified formula
2. Prove that \( \vdash \langle p \rangle \forall x.\phi \)
3. Eliminate \( \forall x.\phi \) and apply \( p \) on \( \Phi \)
∀x. (x ≥ 0 → prev(next(x)) = x) ∧ . . . ∧ Φ

1. Generate program $p$ from the selected quantified formula
2. Prove that $\models \langle p \rangle \forall x. \phi$
3. Eliminate $\forall x. \phi$ and apply $p$ on $\Phi$
Verification-based Satisfiability Proving (VSP)

\[
\langle p \rangle \forall x. (x \geq 0 \rightarrow prev(\text{next}(x)) = x) \land \ldots \land \Phi
\]

1. Generate program \( p \) from the selected quantified formula
2. Prove that \( \models \langle p \rangle \forall x. \phi \)
3. Eliminate \( \forall x. \phi \) and apply \( p \) on \( \Phi \)
Verification-based Satisfiability Proving (VSP)

\[ \langle p \rangle \forall x. (x \geq 0 \rightarrow \text{prev}(\text{next}(x)) = x) \land \ldots \land \Phi \]

1. Generate program \( p \) from the selected quantified formula
2. Prove that \( \models \langle p \rangle \forall x. \phi \)
3. Eliminate \( \forall x. \phi \) and apply \( p \) on \( \Phi \)
Verification-based Satisfiability Proving (VSP)

\[
\begin{align*}
&\text{for}(i=0;\text{true};i++)\{ \text{next}[i]=i; \} \\
&\text{for}(i=0;\text{true};i++)\{ \text{prev}[\text{next}[i]]=i; \}
\end{align*}
\]

\[
\langle p \rangle \forall x. (x \geq 0 \rightarrow \text{prev}(\text{next}(x)) = x) \quad \text{true}
\]

1. Generate program \( p \) from the selected quantified formula
2. Prove that \( \models \langle p \rangle \forall x.\phi \)
3. Eliminate \( \forall x.\phi \) and apply \( p \) on \( \Phi \)

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Verification-based Satisfiability Proving (VSP)

for(i=0;true;i++) { next[i]=i; }
for(i=0;true;i++) { prev[next[i]]=i; }

\[ true \land \ldots \land \Phi \]

1. Generate program \( p \) from the selected quantified formula
2. Prove that \( \models \langle p \rangle \forall x.\phi \)
3. Eliminate \( \forall x.\phi \) and apply \( p \) on \( \Phi \)
for(i=0;true;i++){ next[i]=i; }
for(i=0;true;i++){ prev[next[i]]=i; }

\[ \text{true} \land \ldots \land \langle p \rangle \Phi \]

1. Generate program \( p \) from the selected quantified formula
2. Prove that \( \Vdash \langle p \rangle \forall x.\phi \)
3. Eliminate \( \forall x.\phi \) and apply \( p \) on \( \Phi \)
Evaluation Result

VSP + SMT allows counterexample generation, that is not possible with SMT solvers alone.

- C. Gladisch. Model Generation for Quantified Formulas with Application to Test Data Generation. STTT 2012, Vol. 14, Nr. 4
- C. Gladisch. Satisfiability Solving and Model Generation for Quantified First-order Logic Formulas. FoVeOOS 2010
Fault Detection Approaches

Deductive Fault-detection
- Unified deductive verification and fault-detection
- Sound and complete modulo arithmetics/recursive enumeration
- Automation: path unwinding, invariant generation
Fault Detection Approaches

Deductive Fault-detection

- Unified deductive verification and fault-detection
- Sound and complete modulo arithmetics/recursive enumeration
- Automation: path unwinding, invariant generation

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Fault Detection Approaches

Deductive Fault-detection

- Unified deductive verification and fault-detection
- Sound and complete modulo arithmetics/recursive enumeration
- Automation: path unwinding, invariant generation

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Fault Detection Approaches

- **Test-based Fault-detection**
  - Localisation of faults using a debugger
  - Regression testing (tests can be reused)
  - Detection of errors in the runtime environment

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Fault Detection Approaches

- Test-based Fault-detection
  - Localisation of faults using a debugger
  - Regression testing (tests can be reused)
  - Detection of errors in the runtime environment
Fault Detection Approaches

- Test-based Fault-detection
  - Localisation of faults using a debugger
  - Regression testing (tests can be reused)
  - Detection of errors in the runtime environment

Christoph Gladisch: Techniques for fault-detection, testing, and specification: Univ. Twente, 9.4.2015
Fault Detection Approaches

Test-based Fault-detection

- Localisation of faults using a debugger
- Regression testing (tests can be reused)
- Detection of errors in the runtime environment
Why Test Generation

Implementation

Execution Paths
Why Test Generation

Verification

Execution Paths

100% coverage of program behavior described by source code
Why Test Generation

Verification

Execution Paths

Compiler
Libraries
Operating System
Hardware
Why Test Generation

Testing

Execution Paths

Compiler
Libraries
Operating System
Hardware
Combination

Execution Paths

Compiler
Libraries
Operating System
Hardware
Various Works on Test Generation

- Gladisch. Test Data Generation For Programs with Quantified First-order Logic Specifications. ICTSS 2010
- Gladisch. Generating Regression Unit Tests using a Combination of Verification and Capture & Replay. TAP 2010

...
Java Modeling Language (JML) is a formal specification language designed for both: verification and testing

- JML specifications written for verification often cannot be used for testing and vice versa
 Specification Problem

- **Java Modeling Language (JML)** is a formal specification language designed for both: verification and testing
- JML specifications written for verification **often cannot be used for testing and vice versa**
Recursive Observer Method (query) get

--- JAVA + JML ---------------------------------------------

/*@ public normal_behavior

requires n>=0;
assignable \nothing;
accessible Node.footprint;
ensures (o==null || n==0) ==> \result == o;
ensures n>0 ==> \result==(get(o,n-1)!=null?
get(o,n-1).next : null);
measured_by n;
@*/
/*@nullable pure*/ Node get(/*@nullable*/Node o, int n)

----------------------------------------------- JAVA + JML ---

- C. Gladisch, et al. ... SBMF 2013.
Example: `remove` operations on a List

Pre: A → B → C → D → E → F
Post: A → C → D → E → F
Operations on a list: remove

--- JAVA + JML ---

/*@ public normal_behavior
requires 0<i && i<size(o) && acyclic(o);
assignable Node.footprint;
accessible Node.footprint;
ensures (\forall int j;0<=j && j<i; get(o,j)==\old(get(o,j)));
ensures (\forall int k;i<k; get(o,k)==\old(get(o,k+1)));
void remove(Node o, int i){
    Node n=get(o,i-1);
    n.next=n.next.next;
}
--- JAVA + JML ---
Problems with the Specification

- Problem when calling `remove(o,x); remove(o,y);`
- Not suitable for testing due to unbounded quantification
Operations on a List: remove

```java
/*@
public normal_behavior
requires 0<i && i<size(o) && acyclic(o);
assignable Node.footprint;  //for KeY: get(o,i-1).next;
accessible Node.footprint;
ensures (∀ int \ j;0<=j && j<i; get(o,j)==\old(get(o,j)));
ensures (∀ int \ k;i<k && k<=\old(size(o));
        get(o,k)==\old(get(o,k+1)));
ensures size(o) == \old(size(o))-1 && acyclic(o); @*/

void remove(Node o, int i){
    Node n=get(o,i-1);
    n.next=n.next.next;
}
```

--- JAVA + JML ---
Improved **query handling strategy** options in the KeY-tool:

**Auto Induction Strategy Option**
Tested Successfully with the Testing Tool JET
Specification using Observers (Queries)

- Compatibility with deductive verification and testing
- Good readability for linked data structures
- More difficult readability for tree data structures. Easy to make mistakes.
JML

(\forall Data x;
 (\exists Entry a, \textbf{int} i; i>0 && \text{hasNext(this.head},i,a) && a.data==x)
 <=>
 (x==d || (\exists Entry b, \textbf{int} j;
 j>0 && \text{\texttt{old hasNext(this.head},j,b)) && \text{\texttt{old b.data==x}}))))

Alloy

\texttt{this.head'.^next'.data' = this.head.^next.data + d;
Bringing together

Java + Alloy

- Classes
- Fields
- Program states

- Sets
- Relations
- No states
Relational Specifications are Concise

```java
class Tree {
    Tree left, right;
    Data data;
}
```


\[
\begin{align*}
\text{Tree, Data} & \subseteq \text{Object} \\
\sim & \quad \text{left, right} \subseteq \text{Tree} \times \text{Tree} \cup \text{Null} \\
& \quad \text{data} \subseteq \text{Tree} \times \text{Data} \cup \text{Null}
\end{align*}
\]

- A tree \( t \) is a subtree of the tree \( \text{root} \):

\[
t \in \text{root} \cdot *(\text{left} + \text{right})
\]

- All instances of \( \text{Tree} \) are acyclic:

\[
\forall t : \text{Tree} \mid t \notin t \cdot ^*(\text{left} + \text{right})
\]
Relational Specifications are Concise

```
class Tree {
  Tree left, right;
  Data data;
}
```

\[
\begin{align*}
\text{Tree, } \text{Data} & \subseteq \text{Object} \\
\text{left, right} & \subseteq \text{Tree} \times \text{Tree} \cup \text{Null} \\
\text{data} & \subseteq \text{Tree} \times \text{Data} \cup \text{Null}
\end{align*}
\]

- A tree \( t \) is a subtree of the tree \( \text{root} \):
  
  \[
t \in \text{root} \cdot (\text{left} + \text{right})
\]

- All instances of \text{Tree} are acyclic:
  
  \[
  \forall t : \text{Tree} \mid t \notin t \cdot ^{\ast} (\text{left} + \text{right})
  \]
A tree $t$ is a subtree of the tree $\text{root}$:
\[
t \in \text{root} \ast (\text{left} + \text{right})
\]

All instances of Tree are acyclic:
\[
\forall t : \text{Tree} \mid t \notin t \ast (\text{left} + \text{right})
\]
Alloy2JML

Alloy

\texttt{this.head'.^next'.data' = this.head.^next.data + d;}

is translated by Alloy2JML into

JML

\[
\forall \text{Data } x; \\
\quad \exists \text{Entry } a, \text{int } i; i>0 \land \text{hasNext} (\text{this.head},i,a) \land a.data=x \land \exists \text{Entry } b, \text{int } j\land j>0 \land \text{old(hasNext} (\text{this.head},j,b)) \land \text{old}(b.data=x))
\]
Conclusion

Presented Techniques

- Unified deductive verification and fault-detection
- Model generation (using programs as models)
- Verification-based test generation
- Query-based specification of linked data structures for V & T
- Alloy2JML: Translation from Alloy to JML