

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

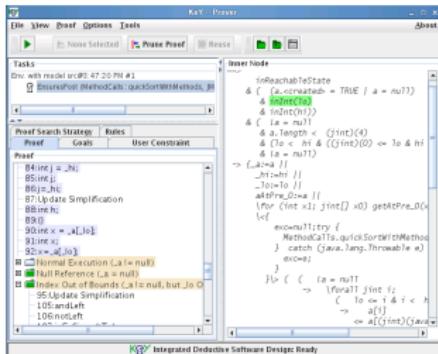


TECHNISCHE
UNIVERSITÄT
DARMSTADT



KeY-Projekt

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



KeY-Tool

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)

Ahrend, et al. The KeY Platform for Verification and Analysis of Java Programs.
VSTTE 2014.

The KeY-Platform

Basis of the platform

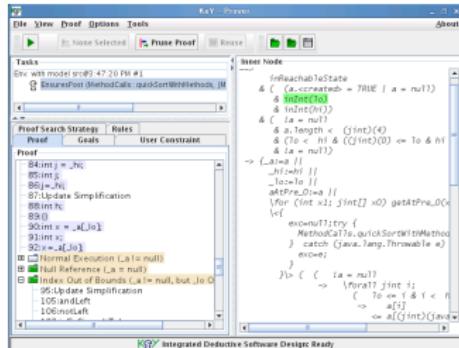
- dynamic logic
- symbolic execution (Java source code level)
- theorem proving
- explicit heap and dynamic frames

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)

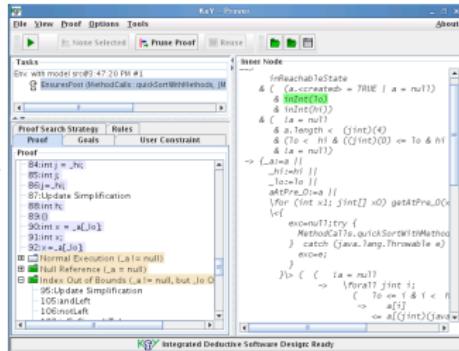
Ahrend, et al. The KeY Platform for Verification and Analysis of Java Programs.
VSTTE 2014.

Background – Deductive Software Verification



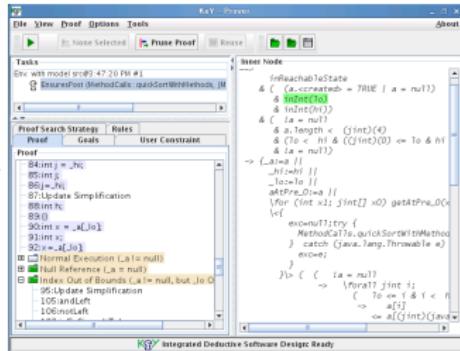
Background – Deductive Software Verification

Implementation
Specification →
Annotations



Background – Deductive Software Verification

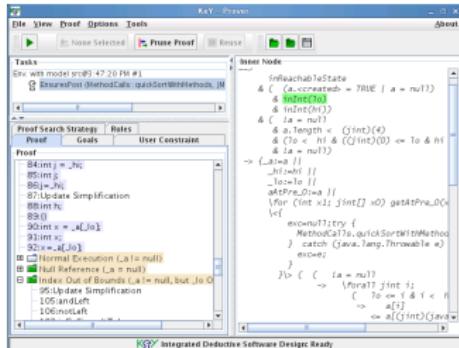
Implementation
Specification →
Annotations



→ ✓ Correct
? Unknown (evtl. Counterex.)

Motivation

Implementation
Specification →
Annotations



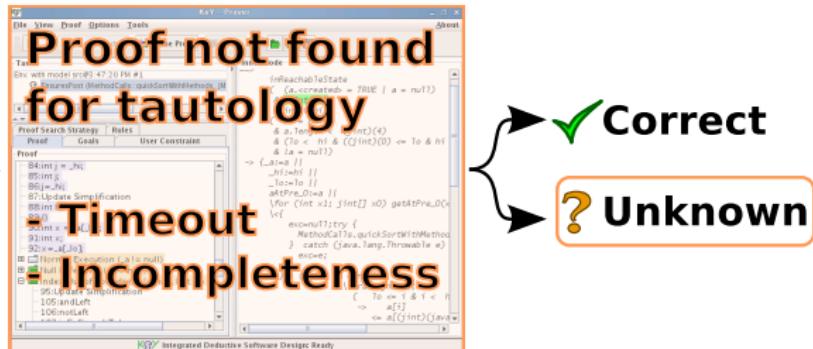
✓ Correct
? Unknown

Why is this important?

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

Motivation

Implementation
Specification →
Annotations

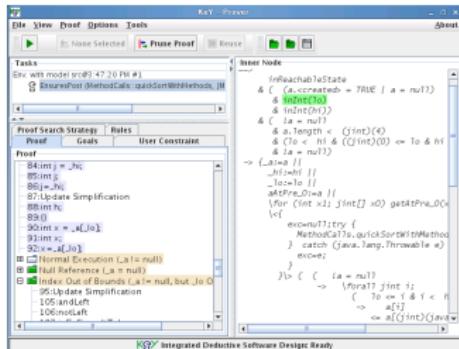


Why is this important?

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

Motivation

Implementation
Specification →
Annotations



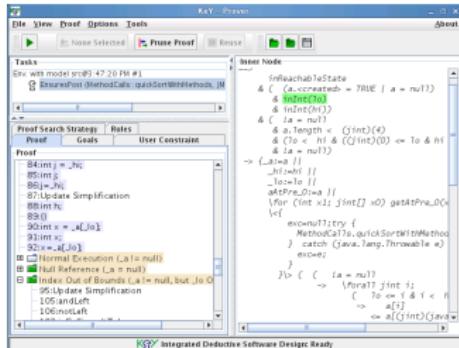
✓ Correct
? Unknown

Why is this important?

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

Motivation

Implementation
Specification
Annotations

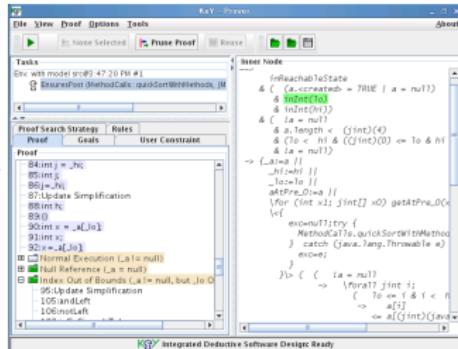


Why is this important?

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

Motivation

Implementation
Specification
Annotations



Correct
Unknown (evtl. Counterex.)
Faulty

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

$$pre \rightarrow [p]post$$

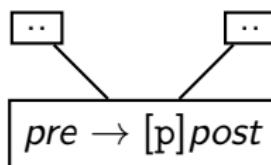
- $pre \rightarrow [p]post$ corresponds to the Hoare-Triple $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic

Step 1. Verification Attempt

$$pre \rightarrow [p]post$$

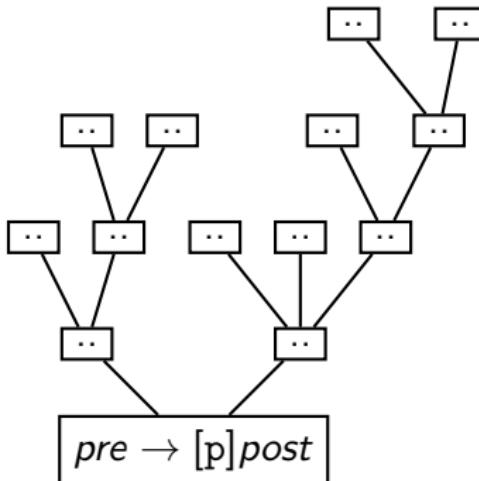
- $pre \rightarrow [p]post$ corresponds to the Hoare-Triple $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic

Step 1. Verification Attempt



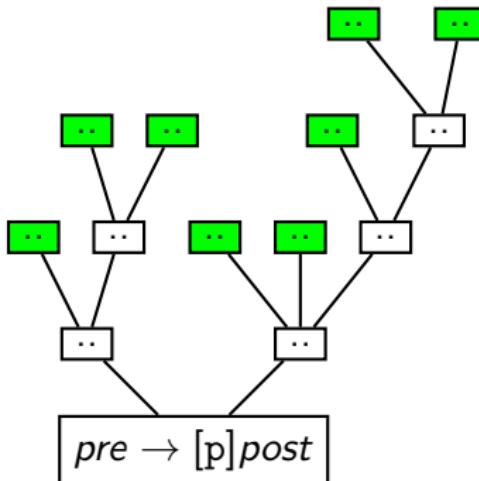
- $pre \rightarrow [p]post$ corresponds to the Hoare-Triple $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic

Step 1. Verification Attempt



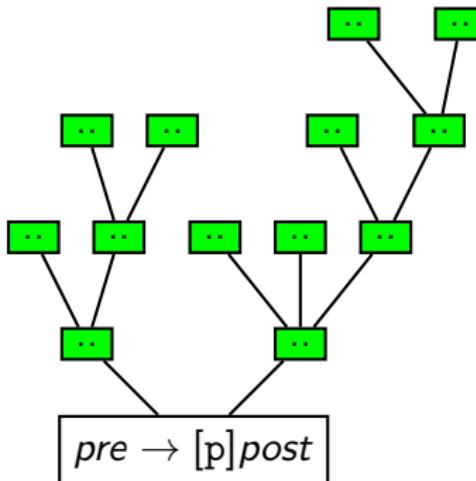
- $\text{pre} \rightarrow [p]\text{post}$ corresponds to the Hoare-Triple $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic

Step 1. Verification Attempt



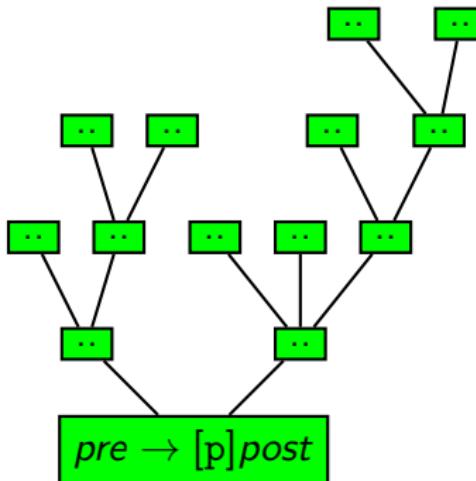
- $\text{pre} \rightarrow [p]\text{post}$ corresponds to the Hoare-Triple $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic

Step 1. Verification Attempt



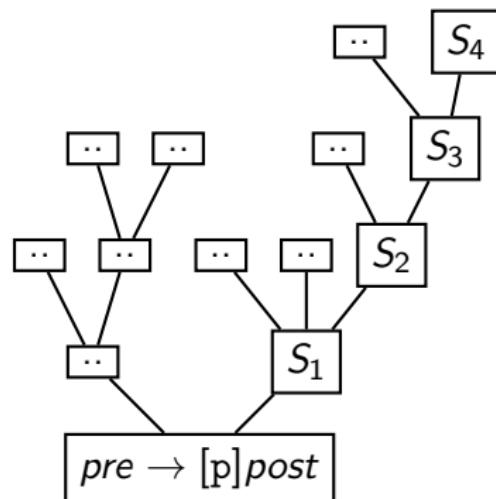
- $\text{pre} \rightarrow [p]\text{post}$ corresponds to the Hoare-Triple $\{pre\}p\{post\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic

Step 1. Verification Attempt

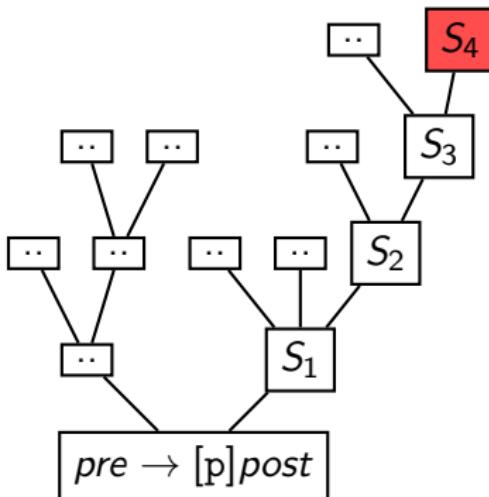


- $\text{pre} \rightarrow [\text{p}]\text{post}$ corresponds to the Hoare-Triple $\{\text{pre}\}\text{p}\{\text{post}\}$
- Symbolic execution and FOL-Theorem proving
- Tree structure through case-distinctions: in program and logic

Step 1. Verification Attempt



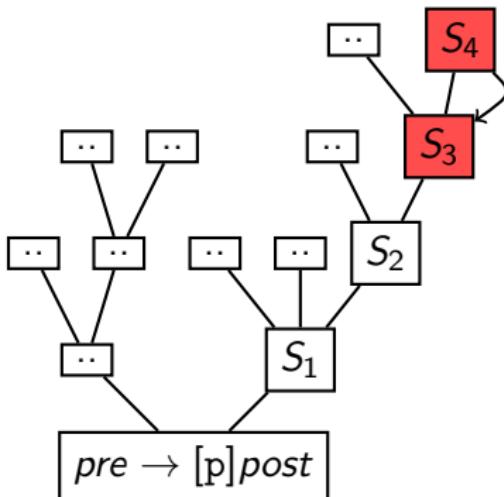
Step 2a. Finding a Counterexample



Principle

If $\neg S_4$ is satisfiable and $\neg S_4 \rightarrow \neg S_0$ holds, then $\neg S_0$ is satisfiable.
⇒ Implementation does not satisfy its spec.

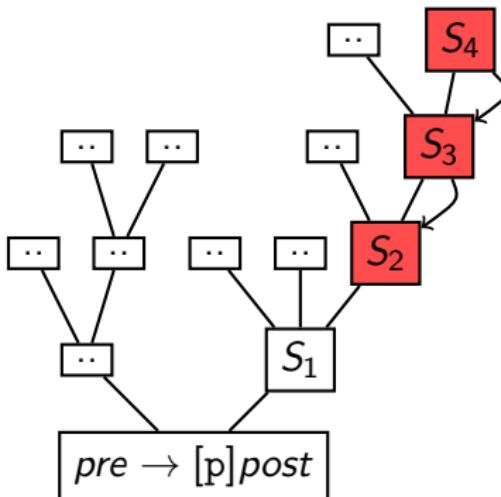
Step 2b. Validity Preservation



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.

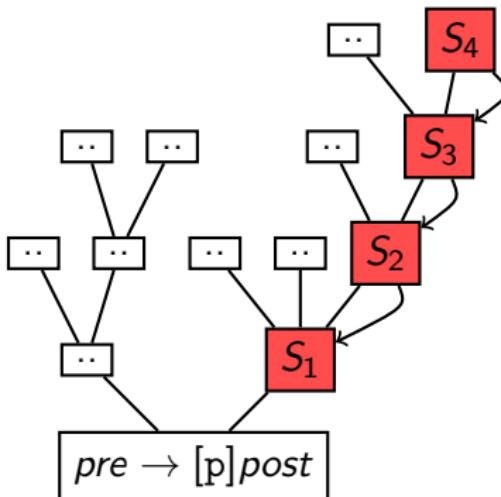
Step 2b. Validity Preservation



Principle

If $\neg S_4$ is satisfiable and $\neg S_4 \rightarrow \neg S_0$ holds, then $\neg S_0$ is satisfiable.
⇒ Implementation does not satisfy the spec.

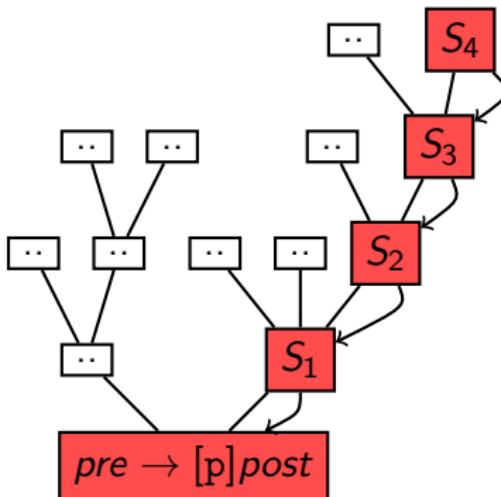
Step 2b. Validity Preservation



Principle

If $\neg S_4$ is satisfiable and $\neg S_4 \rightarrow \neg S_0$ holds, then $\neg S_0$ is satisfiable.
⇒ Implementation does not satisfy the spec.

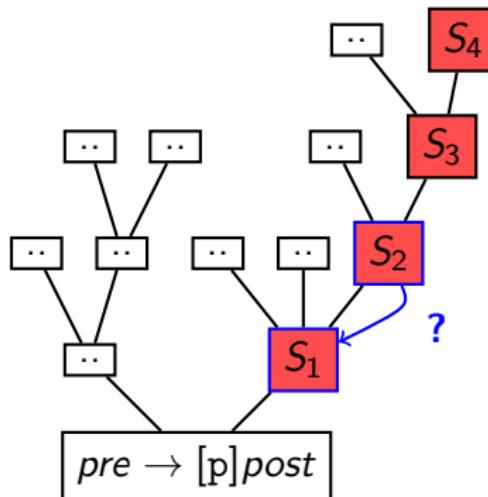
Step 2b. Validity Preservation



Principle

If $\neg S_4$ is satisfiable and $\neg S_4 \rightarrow \neg S_0$ holds, then $\neg S_0$ is satisfiable.
⇒ Implementation does not satisfy the spec.

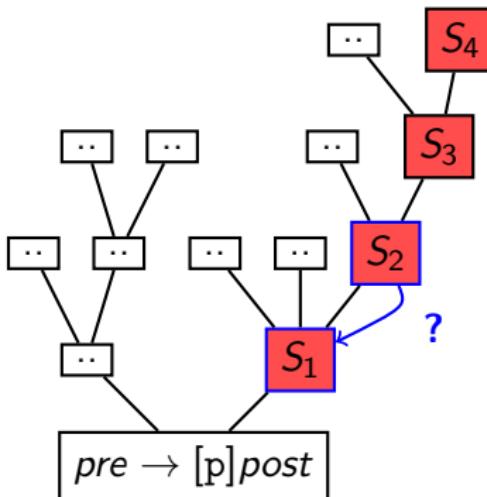
Validity Preservation



Problem: Contract rules

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

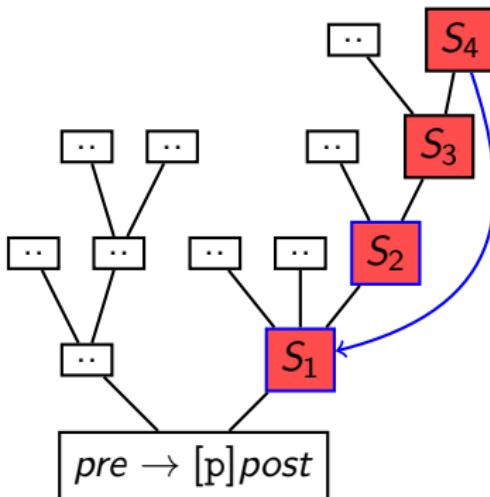
Validity Preservation



Validity Preservation Condition (Approach 1)

$$\neg S_2 \rightarrow \neg S_1$$

Validity Preservation



Validity Preservation Condition (Approach 2)

$$\neg S_4 \rightarrow \neg S_1$$

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

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

Acceleration is possible

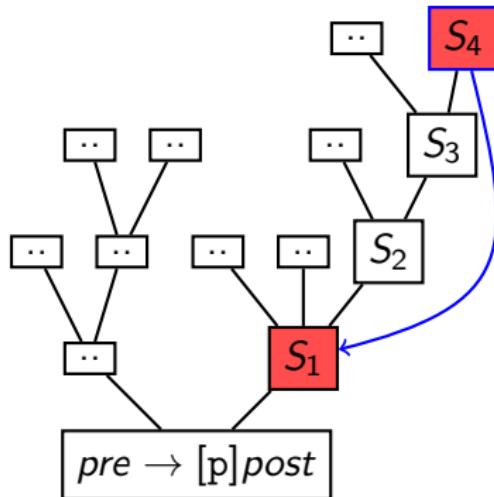
Validity Preservation

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

Acceleration is possible

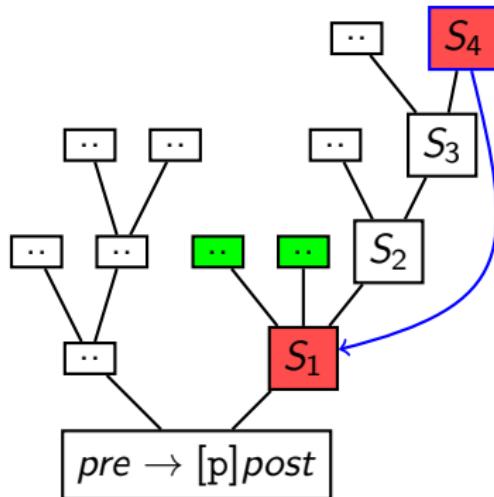
Special Validity Preservation (Acceleration)



Validity Preservation (Approach 2)

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

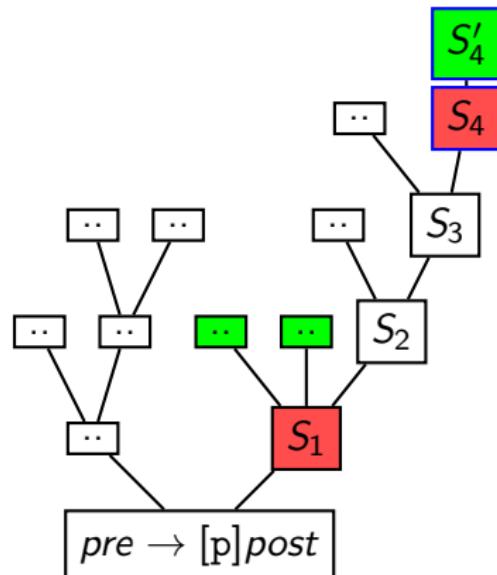
Special Validity Preservation (Acceleration)



Validity Preservation (Approach 2)

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

Special Validity Preservation (Acceleration)

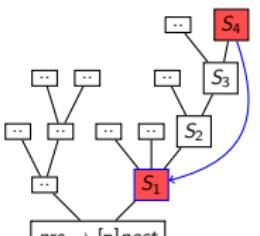


Special Validity Preservation (Approach 3)

$$((\{M^1 := M^2\} S_4) \wedge \{U\}\{M^2\} post_B) \rightarrow S_4$$

Validity Preservation – Evaluation

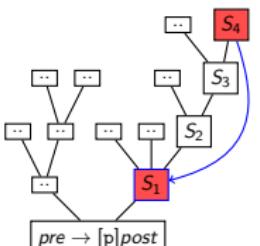
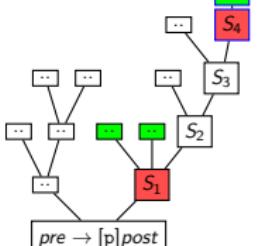
Computational overhead for validity preservation wrt. verification

| Approach 2 | Approach 3 |
|---|---|
|  <p>$pre \rightarrow [p]post$</p> |  <p>$pre \rightarrow [p]post$</p> |
| 45% | 10% |

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

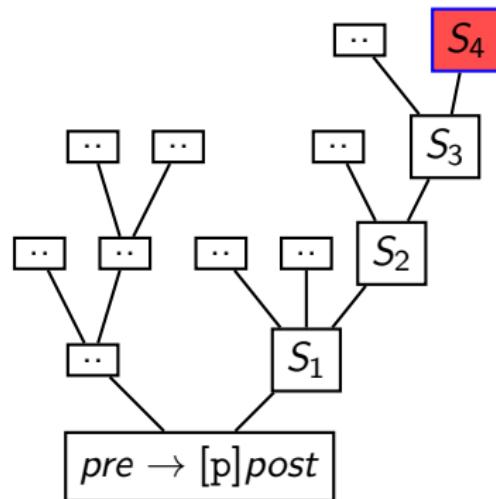
Validity Preservation – Evaluation

Computational overhead for validity preservation wrt. verification

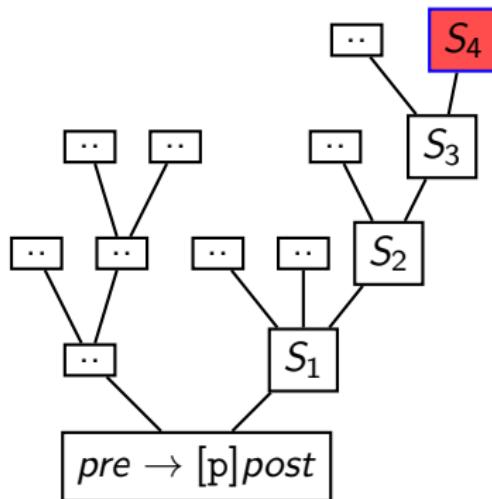
| Approach 2 | Approach 3 |
|--|--|
|  $pre \rightarrow [p]post$ |  $pre \rightarrow [p]post$ |
| 45% | 10% |

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

Counterexample generation \approx Model generation



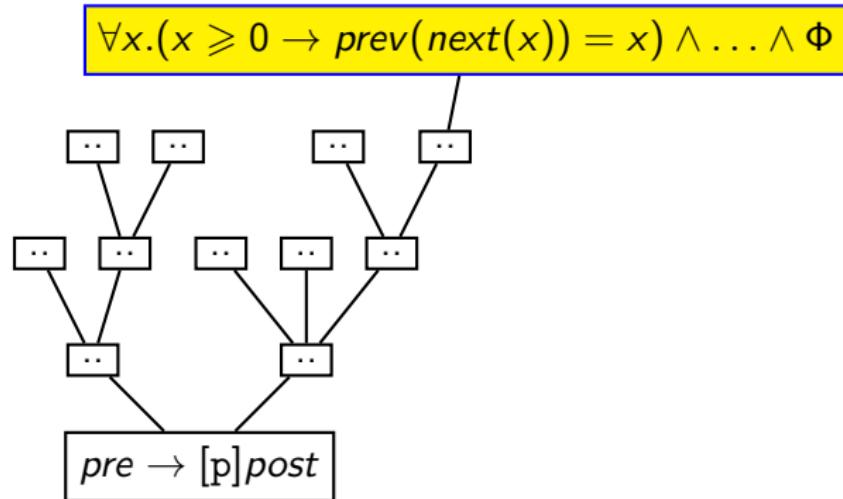
Counterexample generation \approx Model generation



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



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)

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

- 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 Φ

Verification-based Satisfiability Proving (VSP)

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

- 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 Φ

Verification-based Satisfiability Proving (VSP)

$$\langle p \rangle \forall x. (x \geq 0 \rightarrow \text{prev}(\text{next}(x)) = x) \wedge \dots \wedge \Phi$$

- 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 Φ

Verification-based Satisfiability Proving (VSP)

$$\underbrace{\langle p \rangle \forall x. (x \geq 0 \rightarrow \text{prev}(\text{next}(x)) = x)}_{\text{true}} \wedge \dots \wedge \Phi$$

- 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 Φ

Verification-based Satisfiability Proving (VSP)

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

$$\underbrace{\langle p \rangle \forall x. (x \geq 0 \rightarrow \text{prev}(\text{next}(x)) = x)}_{\text{true}} \wedge \dots \wedge \Phi$$

- 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 Φ

Verification-based Satisfiability Proving (VSP)

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

$true \wedge \dots \wedge \Phi$

- 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 Φ

Verification-based Satisfiability Proving (VSP)

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

$$true \wedge \dots \wedge \langle p \rangle \Phi$$

- 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 Φ

Verification-based Satisfiability Proving (VSP)

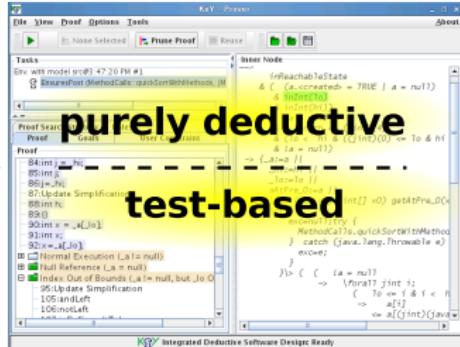
Evaluation Result

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

Implementation
Specification →
Annotations



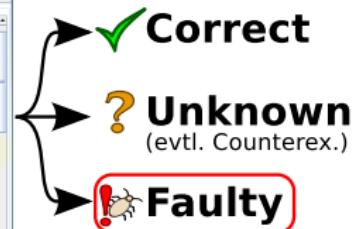
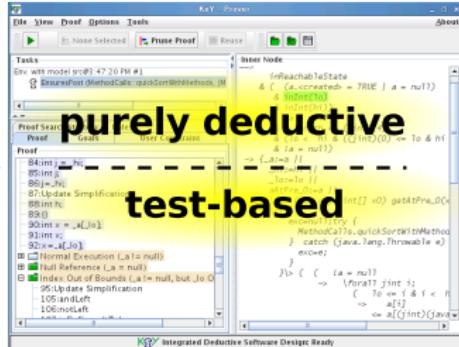
Correct ✓
Unknown ?
Faulty !

Deductive Fault-detection

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

Fault Detection Approaches

Implementation
Specification →
Annotations

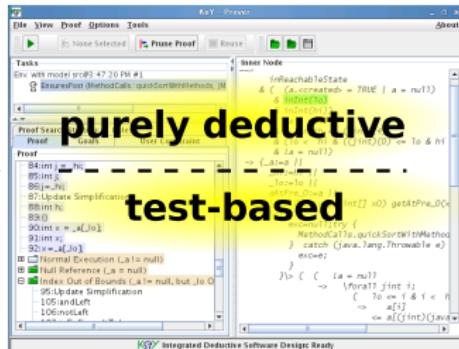


Deductive Fault-detection

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

Fault Detection Approaches

Implementation
Specification →
Annotations



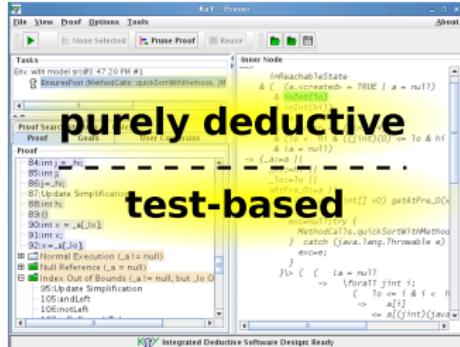
→ **Correct**
→ **Unknown**
(evtl. Counterex.)
→ **Faulty**

Deductive Fault-detection

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

Fault Detection Approaches

Implementation
Specification →
Annotations



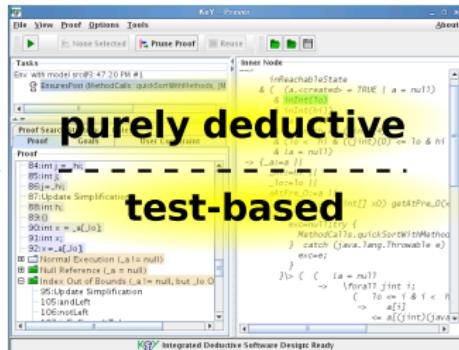
→ **Correct**
→ **Unknown**
(evtl. Counterex.)
→ **Faulty**

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

Implementation
Specification →
Annotations



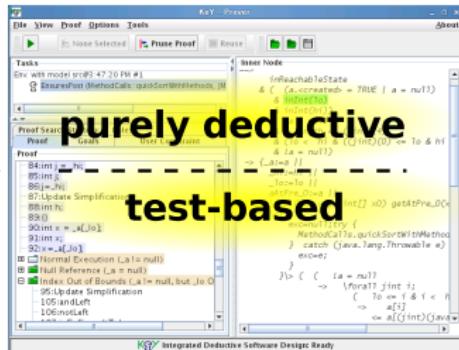
Correct
Unknown (evtl. Counterex.)
Faulty
+ JUnit Tests

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

Implementation
Specification →
Annotations



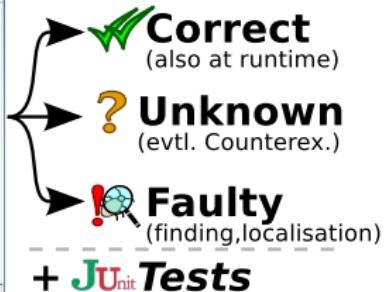
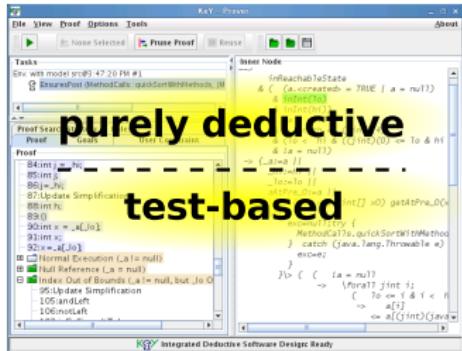
Correct
Unknown (evtl. Counterex.)
Faulty (finding,localisation)
+ JUnit Tests

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

Implementation
Specification →
Annotations



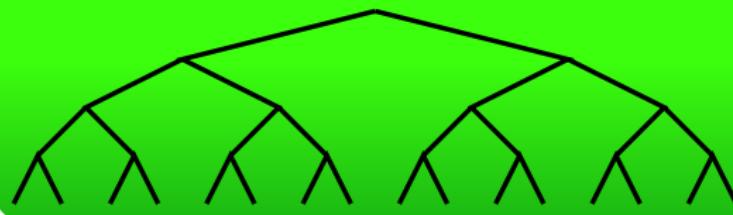
Test-based Fault-detection

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

Why Test Generation

Implementation

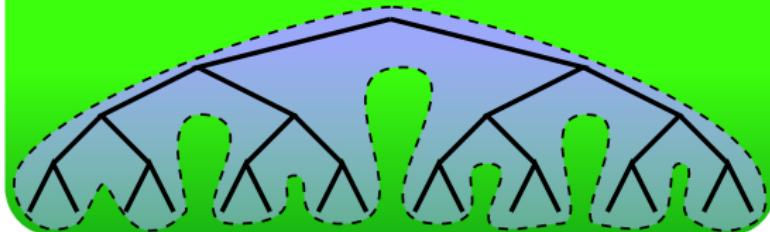
Execution Paths



Why Test Generation

Verification

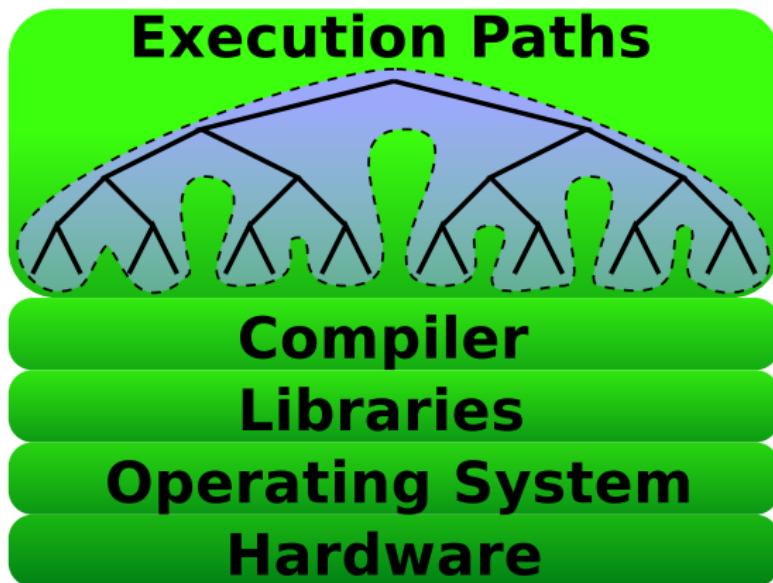
Execution Paths



*100% coverage of program
behavior described by
source code*

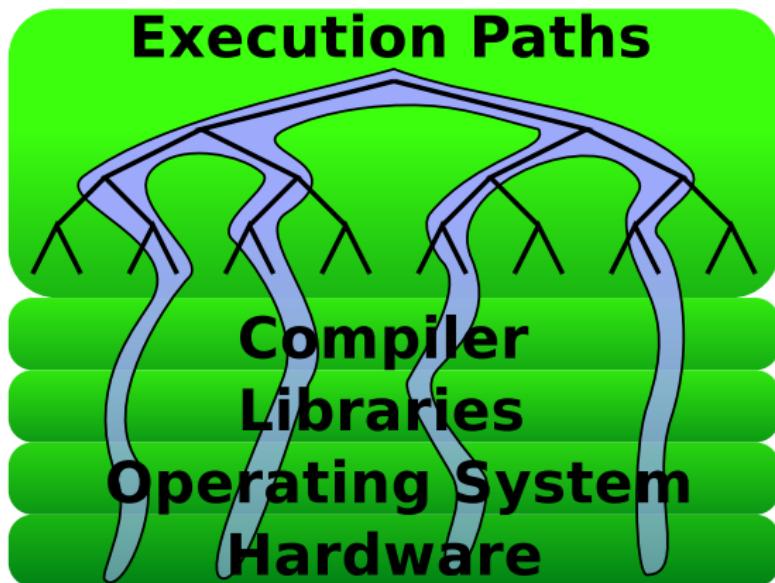
Why Test Generation

Verification



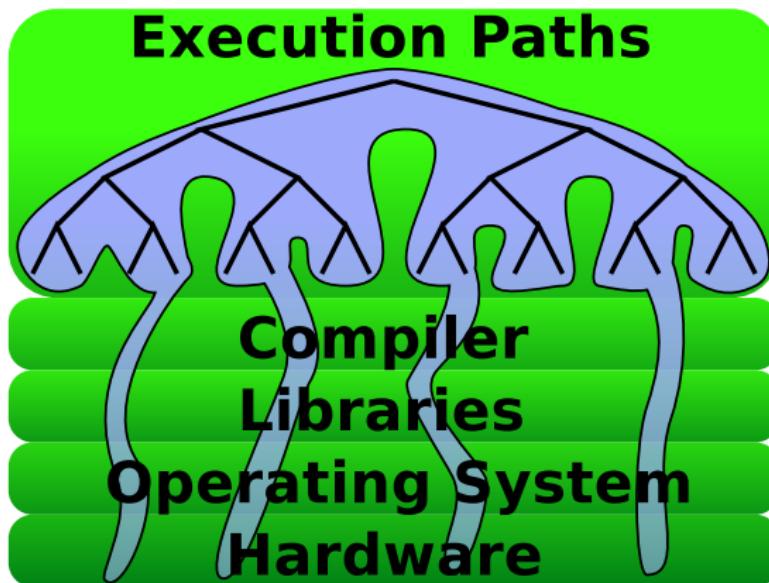
Why Test Generation

Testing

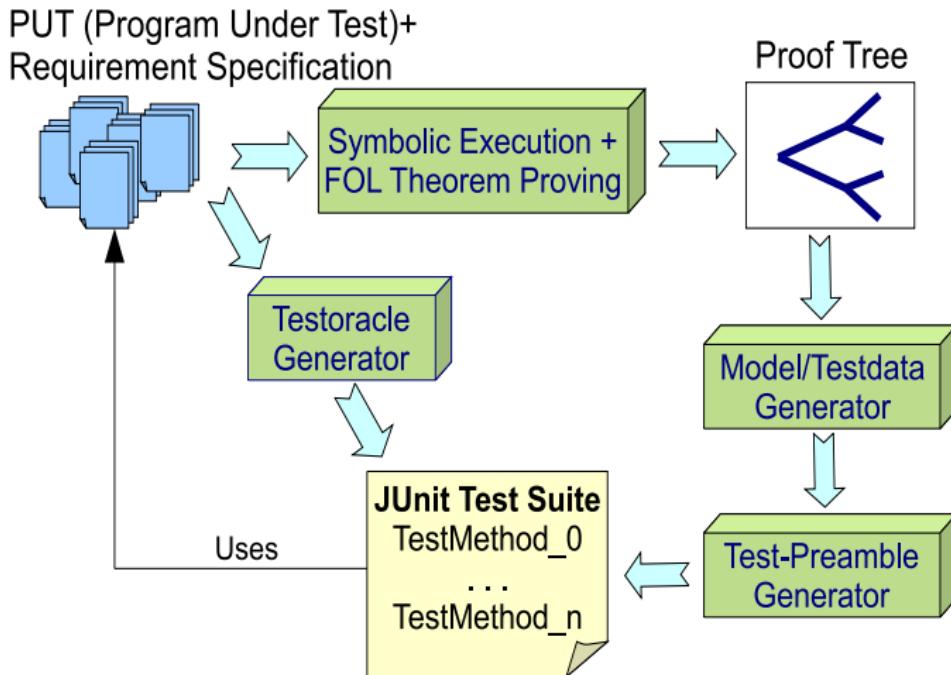


Why Test Generation

Combination



Test Generation in KeY



Various Works on Test Generation

- Gladisch, et al. Using Formal Verification and Statistical Testing Techniques for Reliability Estimation (submitted)
- Ahrend, et al. The KeY Platform for Verification and Analysis of Java Programs. VSTTE 2014
- Beckert, et al. KeYGenU: Combining Verification-Based and Capture and Replay Techniques for Regression Unit Testing. IJSAEM 2011
- 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
- Gladisch. Verification-based Testing for Full Feasible Branch Coverage. SEFM 2008
- ...

Specification using JML

- **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**
 - *L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.L. Lanet. Reusing a JML specication dedicated to verication for testing, and vice-versa: Case studies, 2010.*
 - S. K. Rajamani. Verication, testing and statistics. 2009.

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**
 - *L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.L. Lanet. Reusing a JML specication dedicated to verication for testing, and vice-versa: Case studies, 2010.*
 - S. K. Rajamani. Verication, testing and statistics. 2009.

Recursive Observer Method (query) get

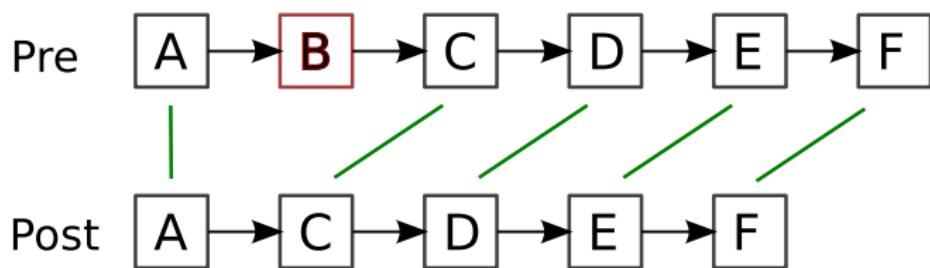
— JAVA + JML —

```
/*@ 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, S. Tyszberowicz. Specifying Linked Data Structures in JML for Formal Verification and Testing. In Sci. of Comp. Program., Elsevier, 2015.
- C. Gladisch, et al. ... SBMF 2013.

Example: remove operations on a List



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 + JML —

```
/*@ 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 (\forall int j; 0<=j && j<i; get(o,j)==\old(get(o,j)));
 ensures (\forall 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 —

Techniques for Automating the Proof

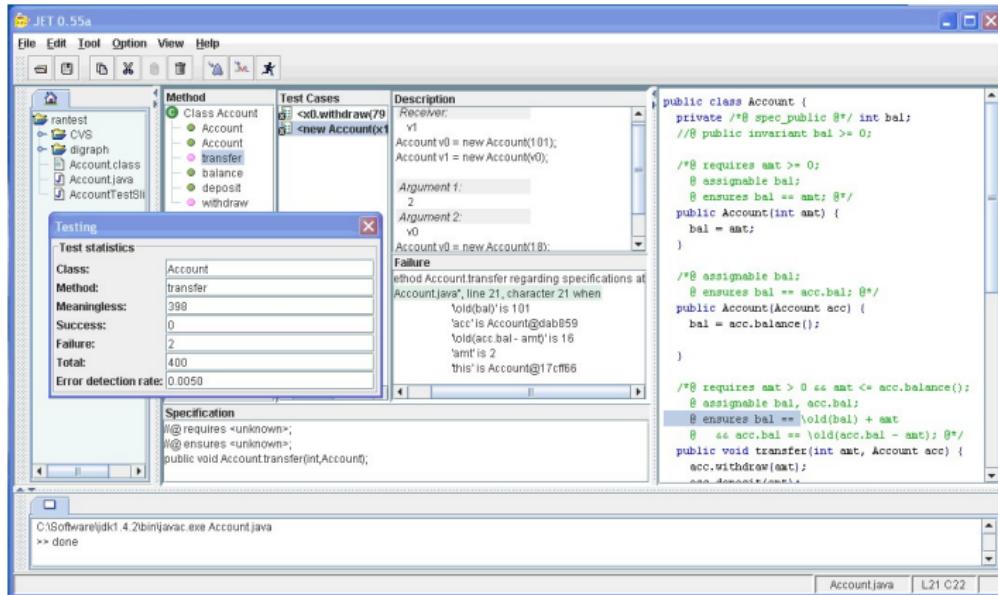
Improved **query handling strategy** options in the KeY-tool:



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.

Example JML vs. Alloy

JML

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

Alloy

```
this.head'.^next'.data' = this.head.^next.data + d;
```

Java +



- Classes
- Fields
- Program states
- Sets
- Relations
- No states

Relational Specifications are Concise

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

$$\approx \begin{array}{l} \text{Tree}, \text{Data} \subseteq \text{Object} \\ \text{left}, \text{right} \subseteq \text{Tree} \times \text{Tree} \cup \text{Null} \\ \text{data} \subseteq \text{Tree} \times \text{Data} \cup \text{Null} \end{array}$$

- A tree t is a subtree of the tree root:

$$t \in \text{root} . *(\text{left} + \text{right})$$

- All instances of Tree are acyclic:

$$\forall t : \text{Tree} \mid t \notin t . ^\wedge(\text{left} + \text{right})$$

Relational Specifications are Concise

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

$$\approx \begin{array}{l} \text{Tree}, \text{Data} \subseteq \text{Object} \\ \text{left}, \text{right} \subseteq \text{Tree} \times \text{Tree} \cup \text{Null} \\ \text{data} \subseteq \text{Tree} \times \text{Data} \cup \text{Null} \end{array}$$

- A tree t is a subtree of the tree root :

$$t \in \text{root} . *(\text{left} + \text{right})$$

- All instances of Tree are acyclic:

$$\forall t : \text{Tree} \mid t \notin t . ^*(\text{left} + \text{right})$$

Relational Specifications are Concise

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

\approx $\begin{aligned}Tree, Data &\subseteq Object \\ left, right &\subseteq Tree \times Tree \cup Null \\ data &\subseteq Tree \times Data \cup Null\end{aligned}$

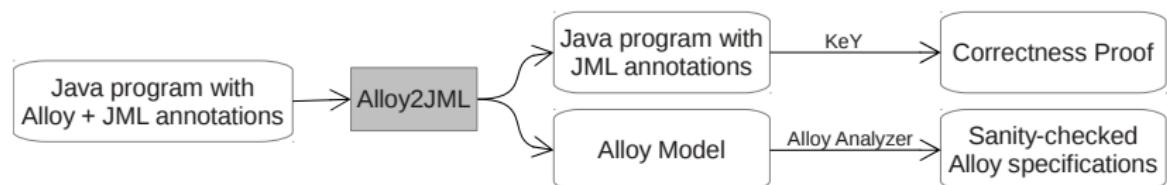
- A tree t is a subtree of the tree root :

$$t \in \text{root} . *(\text{left} + \text{right})$$

- All instances of Tree are acyclic:

$$\forall t : Tree \mid t \notin t . ^*(\text{left} + \text{right})$$

Alloy2JML



D. Grunwald, C. Gladisch, T. Liu, M. Taghdiri, S. Tyszberowicz. Generating JML Specifications from Alloy Expressions. HVC 2014.

Alloy2JML

Alloy

```
this.head'.^next'.data' = this.head.^next.data + d;
```

is translated by Alloy2JML into

JML

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