

Verification Based Test Case Generation

Christian Engel

ITI, Universität Karlsruhe

June 8th, 2006

Outline

- 1 Motivation
- 2 Some Basics on Unit Tests
 - Quality Criteria
- 3 Verification Based Testing
 - How KeY comes into play
 - Examples
- 4 Conclusion

Formal Verification vs. Testing

Formal Verification

- ... can prove correctness of a program on the source code level.

Formal Verification vs. Testing

Formal Verification

- ... can prove correctness of a program on the source code level.

but

- For ensuring the correct behaviour of the entire system verified hardware, compilers and VMs would be required.

Formal Verification vs. Testing

Formal Verification

- ... can prove correctness of a program on the source code level.

but

- For ensuring the correct behaviour of the entire system verified hardware, compilers and VMs would be required.
- Some background knowledge in formal methods is required.

Formal Verification vs. Testing

Formal Verification

- ... can prove correctness of a program on the source code level.

but

- For ensuring the correct behaviour of the entire system verified hardware, compilers and VMs would be required.
- Some background knowledge in formal methods is required.

Testing

- *"Program testing can be used to show the presence of bugs, but never to show their absence!"* Edsger Wybe Dijkstra

Formal Verification vs. Testing

Formal Verification

- ... can prove correctness of a program on the source code level.

but

- For ensuring the correct behaviour of the entire system verified hardware, compilers and VMs would be required.
- Some background knowledge in formal methods is required.

Testing

- *"Program testing can be used to show the presence of bugs, but never to show their absence!"* Edsger Wybe Dijkstra

but

- Testing can discover bugs in the hardware, compilers or VMs.

Conclusion

Testing makes sense, even in cases when a formal proof exists.

Quality of Unit Tests

Code-Based Approach

Determining the quality of a test by measuring how much of the code is covered.

Quality of Unit Tests

Code-Based Approach

Determining the quality of a test by measuring how much of the code is covered.

Some ways to do this:

- Statement coverage

Quality of Unit Tests

Code-Based Approach

Determining the quality of a test by measuring how much of the code is covered.

Some ways to do this:

- Statement coverage
- Branch coverage

Quality of Unit Tests

Code-Based Approach

Determining the quality of a test by measuring how much of the code is covered.

Some ways to do this:

- Statement coverage
- Branch coverage
- Path coverage

Quality of Unit Tests

Code-Based Approach

Determining the quality of a test by measuring how much of the code is covered.

Some ways to do this:

- Statement coverage
- Branch coverage
- Path coverage

Defined as the ratio of executed statements/branches/paths :
feasible statements/branches/paths

Other Approaches

- Specification coverage
- Boundary coverage

Primarily applied to black box testing

Verification Based Testing

- White box testing

Verification Based Testing

- White box testing
- All information about the code and its specification is extracted from a KeY proof.

Verification Based Testing

- White box testing
- All information about the code and its specification is extracted from a KeY proof.

Goal

Use a formal proof to generate a test case that with high code coverage.

Test Case Ingredients

- The code fragment we want to test

Test Case Ingredients

- The code fragment we want to test
- A test oracle

Test Case Ingredients

- The code fragment we want to test
- A test oracle
- All feasible execution paths

Test Case Ingredients

- The code fragment we want to test
- A test oracle
- All feasible execution paths
- A test setup for each execution path

Test Case

```
public class Test<code> extends TestCase{

    ...

    test<code>_1(){
        <setup_1>
        <code>
        <oracle>
    }

    ...

    test<code>_n(){...}
}
```

How KeY comes into play

Problem: How to find feasible execution branches/paths and the corresponding path conditions?

How KeY comes into play

Problem: How to find feasible execution branches/paths and the corresponding path conditions?

Solution: Use symbolic execution.

How KeY comes into play

Problem: How to find feasible execution branches/paths and the corresponding path conditions?

Solution: Use symbolic execution.

Idea: Use KeY for this.

How KeY comes into play

Problem: How to find feasible execution branches/paths and the corresponding path conditions?

Solution: Use symbolic execution.

Idea: Use KeY for this.

Why is that a good idea?

How KeY comes into play

Problem: How to find feasible execution branches/paths and the corresponding path conditions?

Solution: Use symbolic execution.

Idea: Use KeY for this.

Why is that a good idea?

- KeY has full support of all JavaCard features.

How KeY comes into play

Problem: How to find feasible execution branches/paths and the corresponding path conditions?

Solution: Use symbolic execution.

Idea: Use KeY for this.

Why is that a good idea?

- KeY has full support of all JavaCard features.
- KeY can find every feasible execution path/branch.

How KeY comes into play

Problem: How to find feasible execution branches/paths and the corresponding path conditions?

Solution: Use symbolic execution.

Idea: Use KeY for this.

Why is that a good idea?

- KeY has full support of all JavaCard features.
- KeY can find every feasible execution path/branch.
- KeY provides the path condition for each execution path in the proof tree.

Information contained in a Proof Tree

- Execution paths/traces correspond to branches in the proof tree.
- Path conditions can be extracted from nodes.

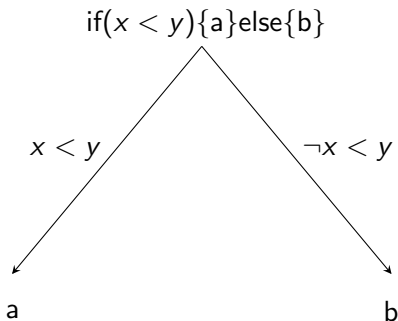
Information contained in a Proof Tree

- Execution paths/traces correspond to branches in the proof tree.
- Path conditions can be extracted from nodes.

`if(x < y){a}else{b}`

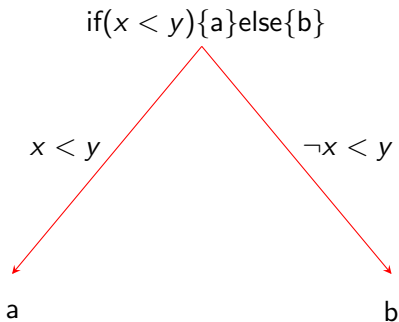
Information contained in a Proof Tree

- Execution paths/traces correspond to branches in the proof tree.
- Path conditions can be extracted from nodes.



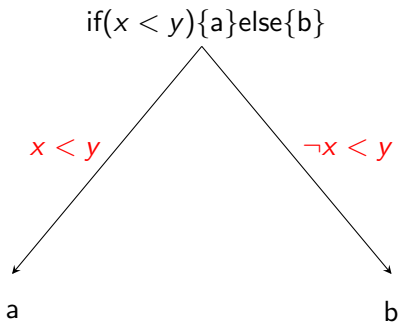
Information contained in a Proof Tree

- Execution paths/traces correspond to branches in the proof tree.
- Path conditions can be extracted from nodes.



Information contained in a Proof Tree

- Execution paths/traces correspond to branches in the proof tree.
- Path conditions can be extracted from nodes.



Information contained in a Proof Tree

- Execution paths/traces correspond to branches in the proof tree.
- Path conditions can be extracted from nodes.

$$\frac{\Gamma, x < y \Rightarrow \langle \pi a \omega \rangle \psi, \Delta \quad \Gamma \Rightarrow x < y, \langle \pi b \omega \rangle \psi, \Delta}{\Gamma \Rightarrow \langle \pi \text{if}(x < y)\{a\}\text{else}\{b\} \omega \rangle \psi, \Delta}$$

Information contained in a Proof Tree

- Execution paths/traces correspond to branches in the proof tree.
- Path conditions can be extracted from nodes.

$$\frac{\Gamma, x < y \Rightarrow \langle \pi a \omega \rangle \psi, \Delta \quad \Gamma \Rightarrow x < y, \langle \pi b \omega \rangle \psi, \Delta}{\Gamma \Rightarrow \langle \pi \text{if}(x < y)\{a\}\text{else}\{b\} \omega \rangle \psi, \Delta}$$

Building a Test from a Proof – Tested Code and Test Oracle

From a proof obligation

$$\Phi \rightarrow \langle c \rangle \Psi$$

Building a Test from a Proof – Tested Code and Test Oracle

From a proof obligation

$$\Phi \rightarrow \langle c \rangle \Psi$$

we can get ...

- the code fragment c we want to test.

Building a Test from a Proof – Tested Code and Test Oracle

From a proof obligation

$$\Phi \rightarrow \langle c \rangle \Psi$$

we can get ...

- the code fragment c we want to test.
- the postcondition Ψ .

Building a Test from a Proof – Tested Code and Test Oracle

From a proof obligation

$$\Phi \rightarrow \langle c \rangle \Psi$$

we can get ...

- the code fragment c we want to test.
- the postcondition Ψ .

The formula Ψ will serve as basis for a test oracle.

Building a Test from a Proof - Test Data

Idea: Search for branches in a closed proof tree, on which the code has been completely executed.

Building a Test from a Proof - Test Data

Idea: Search for branches in a closed proof tree, on which the code has been completely executed.

These branches contain a node

$$\Gamma \Rightarrow \langle \rangle \Psi, \Delta$$

Building a Test from a Proof - Test Data

Idea: Search for branches in a closed proof tree, on which the code has been completely executed.

These branches contain a node

$$\Gamma \Rightarrow \langle \rangle \Psi, \Delta$$

The corresponding path condition is implied by

$$\Phi := \bigwedge_{\gamma \in \Gamma} \gamma \wedge \bigwedge_{\delta \in \Delta} \neg \delta$$

Building a Test from a Proof - Test Data

Idea: Search for branches in a closed proof tree, on which the code has been completely executed.

These branches contain a node

$$\Gamma \Rightarrow \langle \rangle \Psi, \Delta$$

The corresponding path condition is implied by

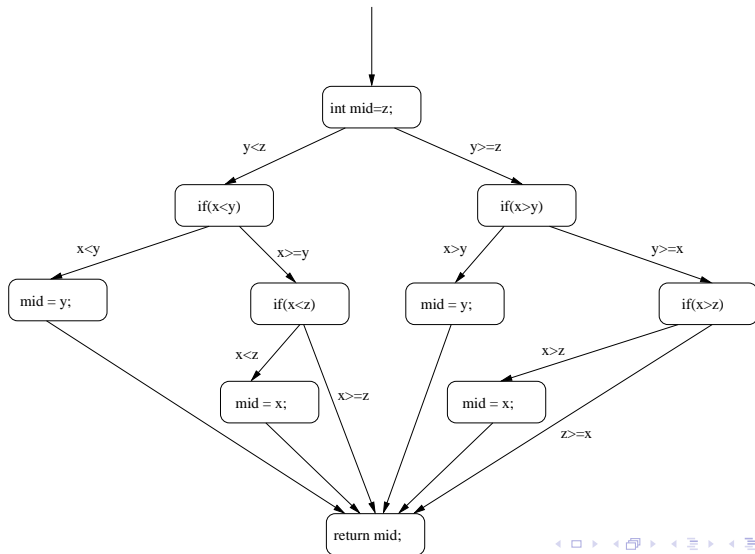
$$\Phi := \bigwedge_{\gamma \in \Gamma} \gamma \wedge \bigwedge_{\delta \in \Delta} \neg \delta$$

Find a model for Φ .

Example – Finite Number of Execution Paths

```
public static int middle(int x, int y, int z){
    int mid = z;
    if(y<z){
        if(x<y){
            mid = y;
        }else if(x<z){
            mid = x;
        }
    }else{
        if(x>y){
            mid = y;
        }else if(x>z){
            mid = x;
        }
    }
    return mid;
}
```

Example - Control Flow Graph



Achievable Test Coverage

A proof tree for a PO $\Phi \rightarrow \langle c \rangle \Psi$, that

- has no open leaf containing a modality.
- contains no application of a loop invariant or a method contract.

contains every feasible execution path in c .

Code with an unbounded Number of Execution Paths

The number of paths for code containing loops or recursion can be potentially infinite.

Code with an unbounded Number of Execution Paths

The number of paths for code containing loops or recursion can be potentially infinite.

- Path coverage is impossible in this case.

Code with an unbounded Number of Execution Paths

The number of paths for code containing loops or recursion can be potentially infinite.

- Path coverage is impossible in this case.

but:

- Branch or statement coverage can still be achieved.

Strategy for finding finite Execution Paths

Idea

Create a partial proof and extract the finite execution paths found in the proof tree.

Strategy for finding finite Execution Paths

Idea

Create a partial proof and extract the finite execution paths found in the proof tree.

For creating this proof tree, we ...

- unwind loops

Strategy for finding finite Execution Paths

Idea

Create a partial proof and extract the finite execution paths found in the proof tree.

For creating this proof tree, we ...

- unwind loops
- execute method bodies

Example - Binary Search Tree

```
public void insert(int value){
    Node current = root;
    if(root == null){
        root = new Node(value);
    }else{
        while(current!=null && current.value != value){
            if(current.value > value){
                if(current.left == null){
                    current.setLeft(new Node(value));
                }
                current = current.left;
            }else{
                if(current.right == null){
                    current.setRight(new Node(value));
                }
                current = current.right;
            }
        }
    }
}
```

Testing aimed symbolic Execution of Loops

$$\frac{\Gamma \Rightarrow \mathcal{U} [\pi \text{ while}(c)\{q\} \omega] \Phi, \Delta \quad \Gamma, \mathcal{UV}(post_{while} \wedge \neg c) \Rightarrow \mathcal{UV} [\pi \ \omega] \Phi, \Delta}{\Gamma \Rightarrow \mathcal{U} [\pi \text{ while}(c)\{q\} \omega] \Phi, \Delta}$$

Requirements on "verification aimed" and "testing aimed" taclets may differ.

Conclusion

- Verification based testing satisfies strong code coverage criteria.
- Largely automatic.

Questions

Questions?