Integrated and Tool-Supported Teaching of Testing, Debugging, and Verification

Wolfgang Ahrendt, Richard Bubel, Reiner Hähnle

Chalmers University
Department of Computer Science and Engineering
Gothenburg
Sweden

November 16, 2009

Background

- authors:
  ‘Software Engineering with Formal Methods’ group at Chalmers

Background

- authors:
  'Software Engineering with Formal Methods' group at Chalmers
- co-developed KeY tool for deductive JAVA source code verification

Background

- authors:
  ‘Software Engineering with Formal Methods’ group at Chalmers
- co-developed KeY tool for deductive JAVA source code verification
- we run Master level course:
  ‘Software Engineering using Formal Methods’
  using Spin and KeY

Background

- authors:
  ‘Software Engineering with Formal Methods’ group at Chalmers
- co-developed KeY tool for deductive JAVA source code verification
- we run Master level course:
  ‘Software Engineering using Formal Methods’
  using Spin and KeY
- we run 3rd year Bachelor level course:
  ‘Testing, Debugging, and Verification’

Background

- authors:
  'Software Engineering with Formal Methods' group at Chalmers
- co-developed KeY tool for deductive JAVA source code verification
- we run Master level course:
  'Software Engineering using Formal Methods'
  using Spin and KeY
- we run 3rd year Bachelor level course:
  'Testing, Debugging, and Verification'

this talk: conception of the latter

Computing Education at Chalmers

Chalmers University of Technology

- Strong engineering tradition, most Swedish engineers from Chalmers
- Emphasis on traditional math courses: calculus, algebra, statistics
- Computing courses focus on programming
- on Bachelor level:
  No dedicated courses on theoretical computer science topics

Course Goals

Integration
: FMs as integrated aspect of quality code construction

Course Goals

Integration
: FMs as integrated aspect of quality code construction

Diversity
: We present spectrum of quality ensuring activities: error detection, error elimination, ensuring error freedom

Course Goals

Integration

FMs as integrated aspect of quality code construction

Diversity

We present spectrum of quality ensuring activities:
error detection, error elimination, ensuring error freedom

Applicability

All methods in action with executable programs

Course Goals

Integration

FMs as integrated aspect of quality code construction

Diversity

We present spectrum of quality ensuring activities:
error detection, error elimination, ensuring error freedom

Applicability

All methods in action with executable programs

Formalisation $\Rightarrow$ Tool Support

Formalisation prerequisite for far-reaching analysis tools

Course Goals

Integration

FMs as integrated aspect of quality code construction

Diversity

We present spectrum of quality ensuring activities:
error detection, error elimination, ensuring error freedom

Applicability

All methods in action with executable programs

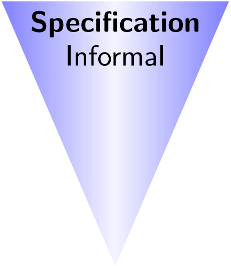
Formalisation ⇒ Tool Support

Formalisation prerequisite for far-reaching analysis tools

Tools are essential

Without tools potential of formalisation not fully realised

The TDV Course Structure



Specification
Informal

Informal Specification

- Exercise informal, but precise specification

Informal Specification

- Exercise informal, but precise specification
- Allegory of specification as contract

Informal Specification

- Exercise informal, but precise specification
- Allegory of specification as contract
- Raise awareness:

Informal Specification

- Exercise informal, but precise specification
- Allegory of specification as contract
- Raise awareness:
  - if you write specs or not, you always program towards contracts

Informal Specification

- Exercise informal, but precise specification
- Allegory of specification as contract
- Raise awareness:
  - if you write specs or not, you always program towards contracts
- Example from JAVA programming:

Informal Specification

- Exercise informal, but precise specification
- Allegory of specification as contract
- Raise awareness:
    - if you write specs or not, you always program towards contracts
- Example from JAVA programming:
    - All classes inherit contract from `Object`

Informal Specification

- Exercise informal, but precise specification
- Allegory of specification as contract
- Raise awareness:
  - if you write specs or not, you always program towards contracts
- Example from JAVA programming:
  - All classes inherit contract from `Object`
  - `Object` contract requires:
    `hashcode()` consistent with `equals()`

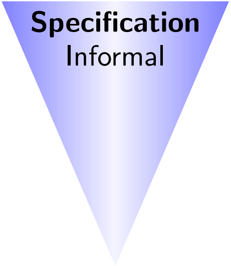
Informal Specification

- Exercise informal, but precise specification
- Allegory of specification as contract
- Raise awareness:
  - if you write specs or not, you always program towards contracts
- Example from JAVA programming:
  - All classes inherit contract from `Object`
  - `Object` contract requires:
    `hashcode()` consistent with `equals()`
  - Programmers/students typically break that contract

Informal Specification

- Exercise informal, but precise specification
- Allegory of specification as contract
- Raise awareness:
  - if you write specs or not, you always program towards contracts
- Example from JAVA programming:
  - All classes inherit contract from `Object`
  - `Object` contract requires:
    `hashcode()` consistent with `equals()`
  - Programmers/students typically break that contract
  - Consequence: collection classes malfunction

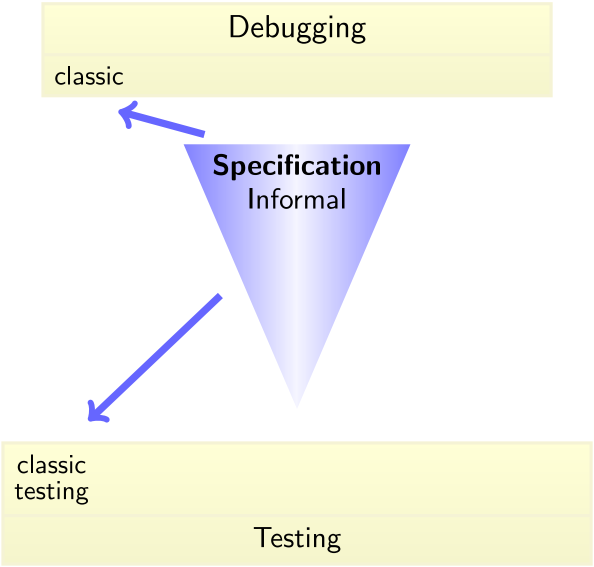
The TDV Course Structure



Specification

Informal

The TDV Course Structure



Debugging

classic

Specification
Informal

classic
testing

Testing

Classic Testing Theory

test inputs and test oracles based on informal specification

Classic Testing Theory

test inputs and test oracles based on informal specification

Contents

- White-/Blackbox testing

Classic Testing Theory

test inputs and test oracles based on informal specification

Contents

- White-/Blackbox testing
- Coverage criteria:

Classic Testing Theory

test inputs and test oracles based on informal specification

Contents

- White-/Blackbox testing
- Coverage criteria:
    - control flow graph coverage

Classic Testing Theory

test inputs and test oracles based on informal specification

Contents

- White-/Blackbox testing
- Coverage criteria:
    - control flow graph coverage
    - logic coverage

Classic Testing Theory

test inputs and test oracles based on informal specification

Contents

- White-/Blackbox testing
- Coverage criteria:
  - control flow graph coverage
  - logic coverage
  - input space partitioning

Classic Testing Theory

test inputs and test oracles based on informal specification

Contents

- White-/Blackbox testing
- Coverage criteria:
  - control flow graph coverage
  - logic coverage
  - input space partitioning
- Writing hand-crafted test cases

Classic Testing Theory

test inputs and test oracles based on informal specification

Contents

- White-/Blackbox testing
- Coverage criteria:
  - control flow graph coverage
  - logic coverage
  - input space partitioning
- Writing hand-crafted test cases

Tool: `JUnit`

Debugging

Disregarded by most software engineering lectures, in contrast to development time actually spent on debugging.

A. Zeller *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, Oct. 2005.

Classic Debugging Techniques

- Logging of events
- Controlled execution: Step into/over, breakpoints
- Inspection: variable values, heap inspection

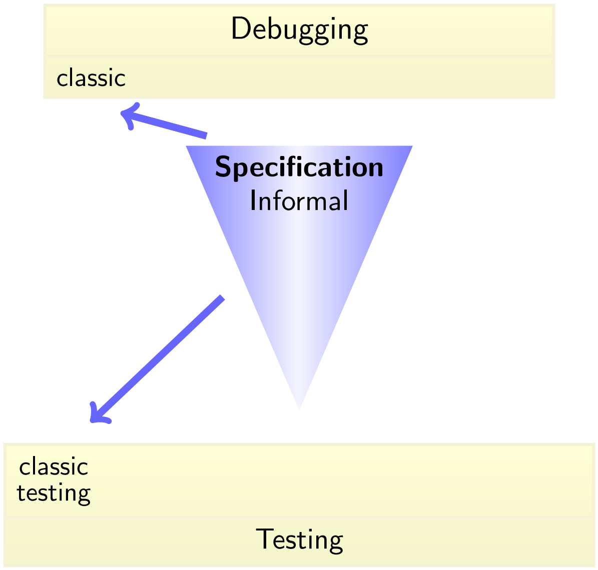
Tools: `log4j`, `eclipse debugger`

Delta Debugging

- Automatic retrieval of minimal input triggering the bug

Tool: `DDinput`

The TDV Course Structure



Debugging

classic

Specification
Informal

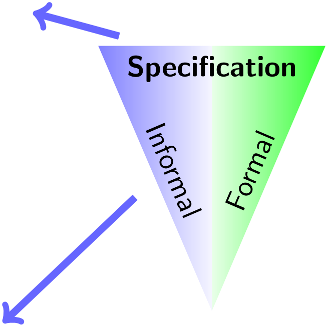
classic
testing

Testing

The TDV Course Structure

Debugging

classic



Specification

Informal

Formal

classic
testing

Testing

Formal Specification

Students learn

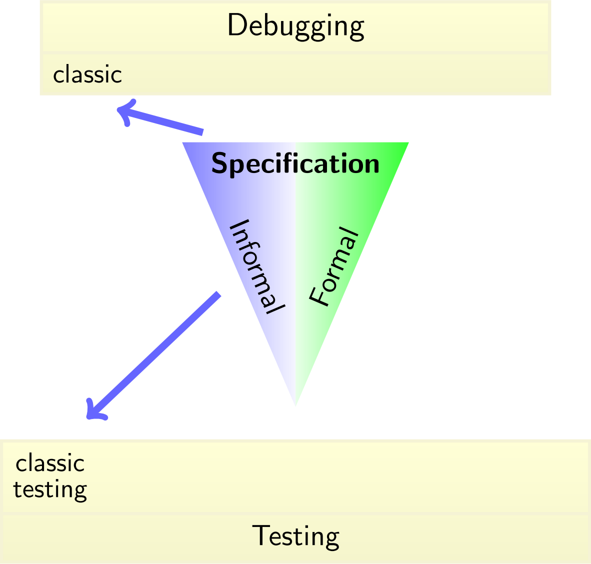
- Formalisation of real-world problems,
- Basics of first-order logic and
- Java Modelling Language (JML) as specification language

Tools: `jml` (syntax and type checker, COMMON JML TOOLS)

Formal specification prerequisite for automation of

- Test generation
- Symbolic debugging
- Formal verification

The TDV Course Structure



Debugging

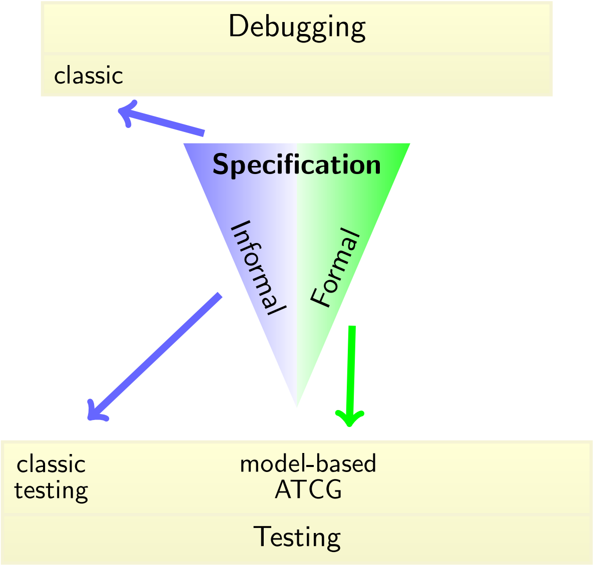
classic

Specification

classic
testing

Testing

The TDV Course Structure



Debugging

classic

Specification

Informal

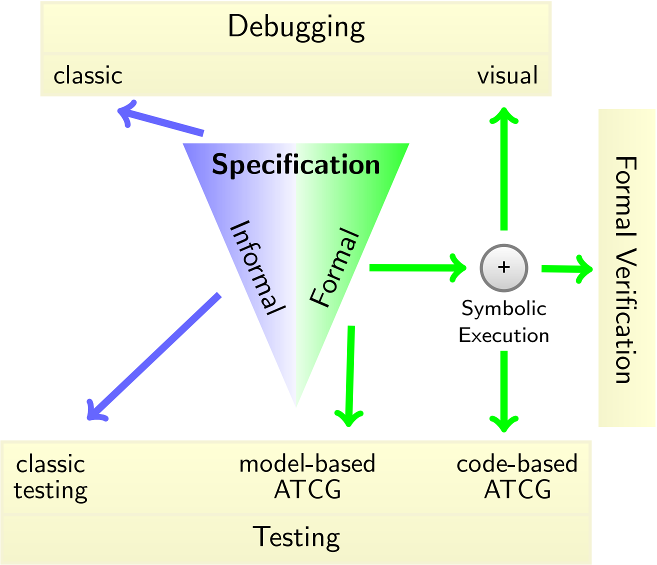
Formal

classic testing

model-based ATCG

Testing

The TDV Course Structure



Automatic Test Generation

Blackbox Testing: Model-based testing

Based on formal specification

- Coverage criteria incl. specifications
- Derivation of test scenarios/cases by disjunction analysis
- Deriving test cases from JML specifications

Tool: `jmlunit` (Common JML Tools)

Automatic Test Generation

Blackbox Testing: Model-based testing

Based on formal specification

- Coverage criteria incl. specifications
- Derivation of test scenarios/cases by disjunction analysis
- Deriving test cases from JML specifications

Tool: `jmlunit` (Common JML Tools)

White-box testing

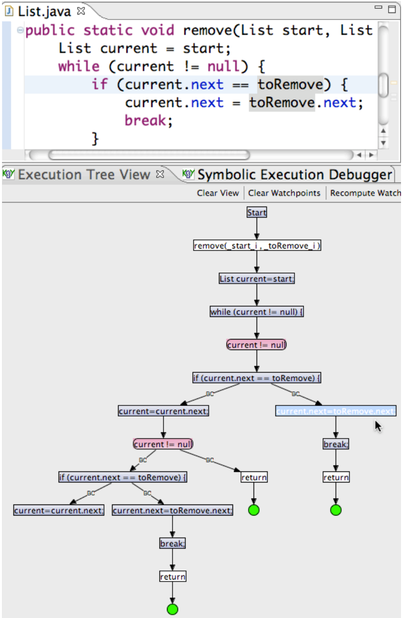
Test cases derived from

- Formal specification and
- Source code

Introducing symbolic execution as basic technology.

Tool: `KeY-VBT`

Symbolic Debugging



```
public static void remove(List start, List
    List current = start;
    while (current != null) {
        if (current.next == toRemove) {
            current.next = toRemove.next;
            break;
        }
```

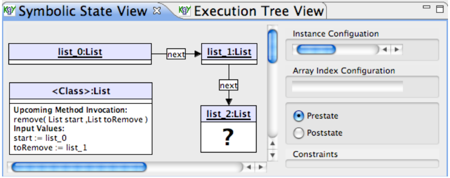
Start
remove(_start_i , _toRemove_i)
List current=start;
while (current != null) {
current != null
if (current.next == toRemove) {
BC
BC
current=current.next;
current.next=toRemove.next;
current != null
BC
BC
break;
if (current.next == toRemove) {
return
return
BC
BC
current=current.next;
current.next=toRemove.next;
break;
return

Based on symbolic execution

- Covers all possible execution paths
- No initialisation necessary
- Efficient omniscient debugging

Symbolic Debugging

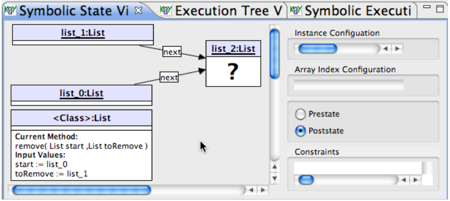
Before removal:



Upcoming Method Invocation:
remove(List start ,List toRemove)
Input Values:
start := list_0
toRemove := list_1

● Prestate
○ Poststate

After removal:



Current Method:
remove(List start ,List toRemove)
Input Values:
start := list_0
toRemove := list_1

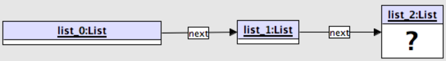
○ Prestate
● Poststate

Based on symbolic execution

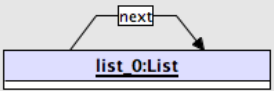
- Covers all possible execution paths
- No initialisation necessary
- Efficient omniscient debugging
- Symbolic heap inspection

Symbolic Debugging

Intended configuration:



Unintended configuration:



Based on symbolic execution

- Covers all possible execution paths
- No initialisation necessary
- Efficient omniscient debugging
- Symbolic heap inspection
  - Specification constrains valid heap configuration

Verification

Most formal approach taught in TDV.

Verification

Most formal approach taught in TDV.

Calculus and tool developed specifically for that course.

Hoare calculus with explicit substitutions

Hoare calculus variant based on symbolic execution

- forward reasoning
- elimination of most non-deterministic rules
- first-order reasoning as black-box

Verification

Most formal approach taught in TDV.

Calculus and tool developed specifically for that course.

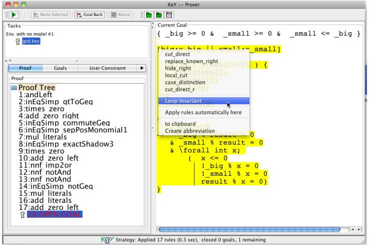
Hoare calculus with explicit substitutions

Hoare calculus variant based on symbolic execution

- forward reasoning
- elimination of most non-deterministic rules
- first-order reasoning as black-box

Tool: KeY-Hoare

- Interactive and automatic verification system (based on KeY)
- Powerful first-order/arithmetic proving capabilities
- Supports partial, total, and execution time aware correctness



```
Proof Tree
  1:andLeft
  2:inEqSimp_gtToGeq
  3:times_zero
  4:add_zero_right
  5:inEqSimp_commuteGeq
  6:inEqSimp_sepPosMonomial1
  7:mul_literals
  8:inEqSimp_exactShadow3
  9:times_zero
  10:add_zero_left
  11:nnf_imp2or
  12:nnf_notAnd
  13:nnf_notAnd
  14:inEqSimp_notGeq
  15:mul_literals
  16:add_literals
  17:add_zero_left
  18:OPEN GOAL
```

```
{ _big >= 0 &  _small >= 0 &  _small <= _big }

[big:=_big || small:=_small]
           ) {
       & _small % result = 0
       & \forall int x;
           (  x <= 0
            | !_big % x = 0
            | !_small % x = 0
            | result % x = 0)
}
```

Strategy: Applied 17 rules (0.3 sec), closed 0 goals, 1 remaining

Experiences and Discussion

Course given first time: Summer 2007
Course Name: Program verification
Participants: 15 students

Experiences and Discussion

Course given first time: Summer 2007
Course Name: Program verification
Participants: 15 students

Renamed to Testing, Debugging and Verification in 2008
Participants: 80 students

Experiences and Discussion

Course given first time: Summer 2007
Course Name: Program verification
Participants: 15 students

Renamed to Testing, Debugging and Verification in 2008
Participants: 80 students

Evaluation
Course has been rated high by participating students.
90% of students completed course (high rate for non-compulsory course)

Experiences and Discussion

Course given first time: Summer 2007
Course Name: Program verification
Participants: 15 students

Renamed to Testing, Debugging and Verification in 2008
Participants: 80 students

Evaluation

Course has been rated high by participating students.
90% of students completed course (high rate for non-compulsory course)

Limitations

- Compromise between available time, topic and depth
- Missing:
  - Software certification and code reviews
  - Integration into software development process (planned!)

Experiences and Discussion

Course given first time: Summer 2007
Course Name: Program verification
Participants: 15 students

Renamed to Testing, Debugging and Verification in 2008
Participants: 80 students

Evaluation

Course has been rated high by participating students.
90% of students completed course (high rate for non-compulsory course)

Limitations

- Compromise between available time, topic and depth
- Missing:
    - Software certification and code reviews
    - Integration into software development process (planned!)

Course adapted by U. of Innsbruck, TU of Madrid and U. of Freiburg

Research-Driven Course Development

Research-Driven Course Development

Can a Bachelor level course be research driven?

Research-Driven Course Development

Can a Bachelor level course be research driven?

- our research objective is precisely increased accessibility of FMs

Research-Driven Course Development

Can a Bachelor level course be research driven?

- our research objective is precisely increased accessibility of FMs
- students profit from this objective

Research-Driven Course Development

Can a Bachelor level course be research driven?

- our research objective is precisely increased accessibility of FMs
- students profit from this objective
- we profit from increased pressure on usability

Adaption

in spite of the close connection to our own research:
course can be run in any other context

- All tools freely available, mostly open source
- We actively support adaptation of course (units)
- Course parts adapted at:
  - Technical University of Madrid
  - University of Innsbruck
  - University of Freiburg
- Feedback from adaptions improved our course
  (e.g. worst-case execution time in KeY-Hoare
  suggested by Joanna Chimiak-Opoka, Innsbruck)

Overview: Tool-Based Teaching

Teaching Unit	Content	Formal	Tools
Testing	Systematic testing, specification, assertions, black/white box, path/code coverage	no	JUnit
Debugging	Bug tracking, execution control, failure input minimisation, logging, slicing	no	`DDinput`, Eclipse, `log4j`
Formal Specification	Design-by-contract, formalisation, first-order logic, JML	yes	`jml` (type checker)
Automated Test Case Generation	Model-based TC generation, Symbolic execution, Code-based TC generation	yes	`jmlunit`, KeY VSD, KeY VBT
Formal Verification	Hoare triple, weakest precondition, formal verification, loop invariant	yes	KeY-Hoare

All tools freely available software and most open source.