Testing, Debugging, Program Verification

Automated Test Case Generation, Part I

Wolfgang Ahrendt & Vladimir Klebanov & Moa Johansson

10 December 2012

Introduction

With JML we can formally specify program behavior.
How to make (further) use of it?

Automated Test Case Generation (ATCG)

- ✔ Tool support for creating test cases
- ✔ Ensuring test case coverage methodically and reliably

Introduction

With JML we can formally specify program behavior.
How to make (further) use of it?

Automated Test Case Generation (ATCG)

- ✔ Tool support for creating test cases
- ✔ Ensuring test case coverage methodically and reliably

View JML-annotated code as formal description of all anticipated runs

General ATCG Approach

1. Specialize contract/code to representative selection of concrete runs
2. Turn these program runs into executable test cases

Ideas Behind Automated Test Generation

Ideas common to systematic (automated) test generation

- Formal analysis of specification and/or code yields enough information to produce test cases
- Systematic algorithms give certain coverage guarantees
- Post conditions can be turned readily into test oracles
- Mechanic reasoning technologies achieve automation:
  - constraint solving
  - logic-based deduction
  - symbolic execution
  - model finding

Automated Test Generation Framework: Unit Tests

Most ATCG methods focus on Unit Testing (and so do we):
Test a single method or function, the implementation under test (IUT)

Create test cases for most popular JAVA unit test framework: JUNIT

Components of Test Cases in Unit Testing

Automated Test Generation Framework: Unit Tests

Most ATCG methods focus on Unit Testing (and so do we):
Test a single method or function, the implementation under test (IUT)

Create test cases for most popular JAVA unit test framework: JUNIT

Components of Test Cases in Unit Testing

1. Setup test data: test fixture/preamble creates program state from which IUT is started
2. Invoke IUT
3. Inspect result: test oracle issues verdict whether test succeeded: PASS or FAIL

Reminder: Black box vs White Box Testing

Black box testing

Implementation is unknown, test data generated from spec, randomly, etc.

White box (aka glass box) testing

Implementation is analyzed to generate test data for it.

Reminder: Black box vs White Box Testing

Black box testing

Implementation is unknown, test data generated from spec, randomly, etc.

White box (aka glass box) testing

Implementation is analyzed to generate test data for it.

Specific Pros and Cons

- ✔ Black box testing does not require source code
- ✘ Black box testing can be irrelevant/insufficient for IUT

Reminder: Black box vs White Box Testing

Black box testing

Implementation is unknown, test data generated from spec, randomly, etc.

White box (aka glass box) testing

Implementation is analyzed to generate test data for it.

Specific Pros and Cons

- ✔ Black box testing does not require source code
- ✘ Black box testing can be irrelevant/insufficient for IUT

- ✔ White box testing can use full information from code
- ✘ White box testing does require source or byte code

Program States and JML Expressions

Reminder

A given program state *s* makes a boolean JML expression true or false (assuming `\old` is not used)

Example

Assume that `int[] ar` has value {1,2} in state *s*

Then "`ar.length == 2 && search(ar, 1)== 0`" is true in state *s*

(`search` is binary search, has signature `int search(int[] ar, int elem)`)

Program States and Test Cases

Basic Assumption:

A desired program state can be constructed by suitable fixture/preamble

Program States and Test Cases

Basic Assumption:

A desired program state can be constructed by suitable fixture/preamble

Example

Assume that `int[] ar` has value $\{1,2\}$ in s

This state can be constructed by the following fixture:

```
int[] ar = {1,2};
```

Further Assumption:

Initialization code (fixture) can be computed automatically from s

(not trivial: might need to add constructors, getter/setter methods, etc.)

Automated Test Generation Methods

Methods derived from black box testing

The implementation is unknown, test data generated from spec, randomly, etc.

Generate test cases from analysing
formal specification or formal model of IUT

Methods derived from white box testing

The implementation is analyzed to generate test data for it

Code-based test generation that uses symbolic execution of IUT
(next lecture)

Specification-/Model-Based Test Generation

Generate test cases from analysing
formal specification or formal model of IUT

- Black box technology with according pros and cons
- Many tools, commercial as well as academic:
  JMLUnit, JMLUnitNG, BZ-TT, JML-TT, UniTesK, JTest, TestEra, Cow_Suite, UTJML, ...
- Various specification languages: B, Z, Statecharts, JML, ...
- Detailed formal specification/system model required (here: JML)

Specification-Based Test Generation Cont'd

We use design-by-contract and JML as formal specification methodology:

View JML method contract as formal description of **all** anticipated runs

Specification-Based Test Generation Cont'd

We use design-by-contract and JML as formal specification methodology:

View JML method contract as formal description of all anticipated runs

Specification-Based Test Generation Approach

Look at one method and its JML contract at a time (unit testing)

1. Specialize JML contract to representative selection of concrete runs
   - concentrate on precondition (requires clause)
   - assumption: precondition of contract specifies all anticipated input
   - analysis of implicit and explicit logical disjunctions in precondition
   - choose representative value for each atomic disjunct
2. Turn these representative program runs into executable test cases
3. Synthesize test oracle from postcondition of contract

Contracts and Test Cases

```
/*@ public normal_behavior
  @ requires Pre;
  @ ensures  Post;
  @*/
  public void m() { ... };
```

All prerequisites for intended behavior contained in requires clause

Unless doing robustness testing, consider behavior violating preconditions irrelevant

Contracts and Test Cases

```
/*@ public normal_behavior
  @ requires Pre;
  @ ensures  Post;
  @*/
  public void m() { ... };
```

All prerequisites for intended behavior contained in requires clause

Unless doing robustness testing, consider behavior violating preconditions irrelevant

Test Generation Principle 1 (Relevance)

State at start of IUT execution must make required precondition true

Multi-Part Contracts and Test Cases

```
/*@ public normal_behavior
  @ requires Pre_1;
  @ ensures  Post_1;
  @ also
  @ ...
  @ also
  @ public normal_behavior
  @ requires Pre_n;
  @ ensures  Post_n;
  @*/
  public void m() { ... };
```

Test Generation Principle 2 (Contract Coverage)

There must be at least one test case for each operation contract

Example

```
public class Traffic {
  private /*@ spec_public @*/ boolean red, green, yellow;
  private /*@ spec_public @*/ boolean drive, brake, halt;

/*@ public normal_behavior
  @ requires red || yellow || green;
  @ ensures  \old(red) ==> halt     &&
  @          \old(yellow) ==> brake;
  @*/
  public boolean setAction() {
  // implementation
  }
}
```

Which test cases should be generated?

Data-Driven Test Case Generation

Generate a test case for each possible value of each input variable

- ✘ Combinatorial explosion (already 2^6 cases for our simple example)
- ✘ Infinitely many test cases for unbounded data structures
- ✘ Some resulting test cases unrelated to specification or IUT

Only feasible in connection with selection and bounding principles

Data-Driven Test Case Generation

Generate a test case for each possible value of each input variable

- ✘ Combinatorial explosion (already 2^6 cases for our simple example)
- ✘ Infinitely many test cases for unbounded data structures
- ✘ Some resulting test cases unrelated to specification or IUT

Only feasible in connection with selection and bounding principles

Restriction to test cases that satisfy precondition (Principle 1)?

Data-Driven Test Case Generation

Generate a test case for each possible value of each input variable

- ✘ Combinatorial explosion (already 2^6 cases for our simple example)
- ✘ Infinitely many test cases for unbounded data structures
- ✘ Some resulting test cases unrelated to specification or IUT

Only feasible in connection with selection and bounding principles

Restriction to test cases that satisfy precondition (Principle 1)?

Insufficient (still too many), but gives the right clue!

Disjunctive Partitioning

```
/*@ public normal_behavior
  @ requires red || yellow || green;
  @ ensures  \old(red) ==> halt     &&
  @          \old(yellow) ==> brake;
  @*/
```

Disjunctive analysis of precondition suggests
minimum of three test cases that relate to precondition

Disjunctive Normal Form

Definition (Disjunctive Normal Form (DNF))

A requires clause of a JML contract is in Disjunctive Normal Form (DNF) when it has the form

$$\texttt{C}_1 \;\texttt{||}\; \texttt{C}_2 \;\texttt{||}\; \cdots \;\texttt{||}\; \texttt{C}_n$$

where each $\texttt{C}_i$ does not contain an explicit or implicit disjunction.

Test Generation Principle 3 (Disjunctive Coverage)

For each disjunct D of precondition in DNF create a test case whose initial state makes D true and as many other disjuncts as possible false

Disjunctive Normal Form

Definition (Disjunctive Normal Form (DNF))

A requires clause of a JML contract is in Disjunctive Normal Form (DNF) when it has the form

$$\texttt{C}_1 \;\texttt{||}\; \texttt{C}_2 \;\texttt{||}\; \cdots \;\texttt{||}\; \texttt{C}_n$$

where each $\texttt{C}_i$ does not contain an explicit or implicit disjunction.

Test Generation Principle 3 (Disjunctive Coverage)

For each disjunct D of precondition in DNF create a test case whose initial state makes D true and as many other disjuncts as possible false

Disjunctive Normal Form

Definition (Disjunctive Normal Form (DNF))

A requires clause of a JML contract is in Disjunctive Normal Form (DNF) when it has the form

$$\texttt{C}_1 \;||\; \texttt{C}_2 \;||\; \cdots \;||\; \texttt{C}_n$$

where each $\texttt{C}_i$ does not contain an explicit or implicit disjunction.

Test Generation Principle 3 (Disjunctive Coverage)

For each disjunct D of precondition in DNF create a test case whose initial state makes D true and as many other disjuncts as possible false

Example

```
@ requires red || yellow || green;
```

Gives rise to three test cases `red=true; yellow=green=false;`, etc.

Disjunctive Normal Form

Definition (Disjunctive Normal Form (DNF))

A requires clause of a JML contract is in Disjunctive Normal Form (DNF) when it has the form

$$\texttt{C}_1 \ \texttt{||} \ \texttt{C}_2 \ \texttt{||} \ \cdots \ \texttt{||} \ \texttt{C}_n$$

where each $\texttt{C}_i$ does not contain an explicit or implicit disjunction.

Test Generation Principle 3 (Disjunctive Coverage)

For each disjunct D of precondition in DNF create a test case whose initial state makes D true and as many other disjuncts as possible false

Importance of Establishing DNF Syntactically

Implicit logical disjunctions must be made explicit by computing DNF:

Replace `A ==> B` with `!A || B`, etc.

Test Coverage Criteria

Example

```
requires red || yellow || green;
```

is true even for `red=yellow=green=true;`

Possible to generate a test case for each state making precondition true

Test Coverage Criteria

Example

```
requires red || yellow || green;
```

is true even for `red=yellow=green=true;`

Possible to generate a test case for each state making precondition true

How many different test cases to generate?

Logic Expression Coverage Criteria

Create test cases that make parts of precondition true

- At least one test per spec case (Decision Coverage)
- One for each disjunct (Clause Coverage) ⇒ what we go for
- All disjunct combinations (Multiple Condition Coverage)
- Criteria based on making predicates true/false, etc.

Consistent Test Cases

Example (Class invariant specified in JML)

```
public class Traffic {
  /*@ public invariant (red ==> !green && !yellow) &&
    @                  (yellow ==> !green && !red) &&
    @                  (green ==> !yellow && !red);
    @*/
  private /*@ spec_public @*/ boolean red, green, yellow;

  /*@ public normal_behavior
    @ requires red || yellow || green;
    @ ...
```

The program state `red=yellow=green=true;` violates the class invariant

Consistent Test Cases

Example (Class invariant specified in JML)

```
public class Traffic {
  /*@ public invariant (red ==> !green && !yellow) &&
    @                  (yellow ==> !green && !red) &&
    @                  (green ==> !yellow && !red);
    @*/
  private /*@ spec_public @*/ boolean red, green, yellow;

  /*@ public normal_behavior
    @ requires red || yellow || green;
    @ ...
```

The program state `red=yellow=green=true;` violates the class invariant

As class invariants are supposed to hold whenever a method is called, there is no point to generate test cases for program states violating it

Consistent Test Cases

Example (Class invariant specified in JML)

```
public class Traffic {
  /*@ public invariant (red ==> !green && !yellow) &&
    @                  (yellow ==> !green && !red) &&
    @                  (green ==> !yellow && !red);
    @*/
  private /*@ spec_public @*/ boolean red, green, yellow;

  /*@ public normal_behavior
    @ requires red || yellow || green;
    @ ...
```

The program state `red=yellow=green=true;` violates the class invariant

Test Generation Principle 4 (Consistency with Invariant)

Generate test cases from states that do not violate the class invariant

Dealing with Existential Quantification

Example (Square root)

```
/*@ public normal_behavior
  @  requires n>=0 && (\exists int r; r >= 0 && r*r == n);
  @  ensures ...
  @*/
  public static final int sqrt(int n) { ... }
```

Dealing with Existential Quantification

Example (Square root)

```
/*@ public normal_behavior
  @  requires n>=0 && (\exists int r; r >= 0 && r*r == n);
  @  ensures ...
  @*/
  public static final int sqrt(int n) { ... }
```

Where is the disjunction in the precondition?

Dealing with Existential Quantification

Example (Square root)

```
/*@ public normal_behavior
  @  requires n>=0 && (\exists int r; r >= 0 && r*r == n);
  @  ensures ...
  @*/
  public static final int sqrt(int n) { ... }
```

Where is the disjunction in the precondition?

Existential quantifier as disjunction

- Existentially quantified expression `(\exists int r; P(r))`
- Rewrite as: `P(MIN_VALUE)|| ... || P(0)|| ... || P(MAX_VALUE)`

Dealing with Existential Quantification

Example (Square root)

```
/*@ public normal_behavior
  @  requires n>=0 && (\exists int r; r >= 0 && r*r == n);
  @  ensures ...
  @*/
  public static final int sqrt(int n) { ... }
```

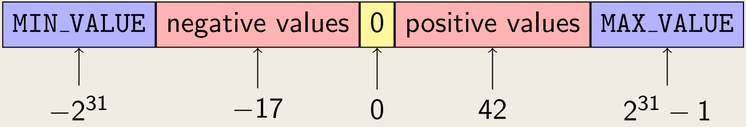
Where is the disjunction in the precondition?

Existential quantifier as disjunction

- Existentially quantified expression `(\exists int r; P(r))`
- Rewrite as: `P(MIN_VALUE)|| ... || P(0)|| ... || P(MAX_VALUE)`
- Get rid of those `P(i)` that are false: `P(0)|| ... || P(46340)`
- Stil too many cases...

Equivalence Classes on Large Input Domains

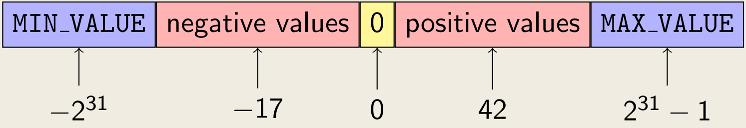
Partition large/infinite domains in finitely many equivalence classes



... and create test case for only one representative of each

Equivalence Classes on Large Input Domains

Partition large/infinite domains in finitely many equivalence classes



... and create test case for only one representative of each

Partitioning is a heuristic method

- Partitioning tries to achieve that the same computation path is taken for all input values within a potential equivalence class.
  Then, one value from each class is sufficient to check for defects.
- As we don't know the IUT, correct partitioning is in general unattainable
- Judicious selection and good heuristics can make it work in practice

Go for Boundary Values

Example (Square)

```
/*@ public normal_behavior
  @  requires n>=0 && n*n >= 0;
  @  ensures \result >=0 && \result == n*n;
  @*/
  public static final int square(int n) { ... }
```

Test Generation Principle 5 (Boundary Values)

Include boundary values of ordered domains as class representatives

Go for Boundary Values

Example (Square)

```
/*@ public normal_behavior
  @  requires n>=0 && n*n >= 0;
  @  ensures \result >=0 && \result == n*n;
  @*/
  public static final int square(int n) { ... }
```

Test Generation Principle 5 (Boundary Values)

Include boundary values of ordered domains as class representatives

Which are suitable boundary values for `n` in this example?

Go for Boundary Values

Example (Square)

```
/*@ public normal_behavior
  @  requires n>=0 && n*n >= 0;
  @  ensures \result >=0 && \result == n*n;
  @*/
  public static final int square(int n) { ... }
```

Test Generation Principle 5 (Boundary Values)

Include boundary values of ordered domains as class representatives

Which are suitable boundary values for `n` in this example?

But. . .

- domains may be non-continuous or non-ordered
- boundary values often expensive to compute

Implicit Disjunctions, Part I

Example (Binary search, target not found)

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 < i && i < array.length
  @                            ==> array[i-1] <= array[i]);
  @          (\forall int i; 0 <= i && i < array.length
  @                            ==> array[i] != target);
  @  ensures  \result == -1;
  @*/
  int search( int array[], int target ) { ... }
```

Implicit Disjunctions, Part I

Example (Binary search, target not found)

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 < i && i < array.length
  @                            ==> array[i-1] <= array[i]);
  @          (\forall int i; 0 <= i && i < array.length
  @                            ==> array[i] != target);
  @  ensures  \result == -1;
  @*/
  int search( int array[], int target ) { ... }
```

No disjunction in precondition!?

Implicit Disjunctions, Part I

Example (Binary search, target not found)

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 < i && i < array.length
  @                           ==> array[i-1] <= array[i]);
  @          (\forall int i; 0 <= i && i < array.length
  @                           ==> array[i] != target);
  @  ensures  \result == -1;
  @*/
  int search( int array[], int target ) { ... }
```

We can freely choose `array`, `length`, and `target` in precondition!

Data Generation Principles

Test Generation Principle 6 (Unbound Locations)

Values of locations without explicit quantification can be freely chosen (Amounts to implicit existential quantification over possible values)

How choose representatives from types of unbound locations?

- There are infinitely many different arrays . . .
- Before defining equivalence classes, need to enumerate all values

Data Generation Principles

Test Generation Principle 6 (Unbound Locations)

Values of locations without explicit quantification can be freely chosen
(Amounts to implicit existential quantification over possible values)

Systematic enumeration of values by data generation principle

Assume declaration: `int[] ar;`, then the array `ar` is

1. either the **null** array: `int[] ar = null;`
2. or the empty array of type **int**: `int[] ar = new int[0];`
3. or an **int** array with one element
   - 3.a `int[] ar = { MIN_VALUE };`
   - 3.b ...
   - 3.ω `int[] ar = { MAX_VALUE };`
4. or an **int** array with two elements ...

n. or an **int** array with $n-2$ elements ...

Combining the Test Generation Principles

Example (Binary search, target found)

```
requires (\exists int i; 0 <= i && i < array.length
                         &&  array[i] == target)      &&
         (\forall int i; 0 < i && i < array.length
                         ==> array[i-1] <= array[i]);
```

Apply test generation principles

1. Use data generation for unbound int array
2. Choose equivalence classes and representatives for:

   array type: `int[]` empty array, singleton, two elements
   usually, need to stop here ...
   `target: int` (include boundaries)
3. Generate test cases that make precondition true

Combining the Test Generation Principles

Example (Binary search, target found)

```
requires (\exists int i; 0 <= i && i < array.length
                         && array[i] == target)    &&
         (\forall int i; 0 < i && i < array.length
                         ==> array[i-1] <= array[i]);
```

- empty array: precondition cannot be made true, no test case
- singleton array, target must be the only array element
  `array = { 0 }; target = 0;`
  `array = { 1 }; target = 1;`
- two-element sorted array, target occurs in array
  `array = { 0,0 }; target = 0;`
  `array = { 0,1 }; target = 0;`
  `array = { 1,1 }; target = 1;`

Implicit Disjunctions, Part II

Example (List Copy)

```
/*@ public normal_behavior
  @ requires true; // src, dst non-Nullable by default
  @  ensures ...
  @*/
static void java.util.Collections.copy(List src, List dst)
```

Implicit Disjunctions, Part II

Example (List Copy)

```
/*@ public normal_behavior
  @ requires true; // src, dst non-Nullable by default
  @  ensures ...
  @*/
static void java.util.Collections.copy(List src, List dst)
```

Aliasing and Exceptions

In JAVA object references `src`, `dst` can be aliased, ie, `src==dst`

- Aliasing usually unintended—exclusion often forgotten in contract

Preconditions can be (unintentionally) too weak

- Exception thrown when `src.length > dst.length`

Implicit Disjunctions, Part II

Example (List Copy)

```
/*@ public normal_behavior
  @ requires true; // src, dst non-Nullable by default
  @  ensures ...
  @*/
static void java.util.Collections.copy(List src, List dst)
```

Test Generation Principle 7 (Aliasing, Exceptions)

Generate test cases that enforce/prevent aliasing and throwing exceptions (when not excluded by contract)

The Postcondition as Test Oracle

Oracle Problem in Automated Testing

How to determine automatically whether a test run succeeded?

The Postcondition as Test Oracle

Oracle Problem in Automated Testing

How to determine automatically whether a test run succeeded?

The "ensures" clause of a JML contract provides verdict on success provided that "requires" clause is true for given test case

The Postcondition as Test Oracle

Oracle Problem in Automated Testing

How to determine automatically whether a test run succeeded?

The "ensures" clause of a JML contract provides verdict on success provided that "requires" clause is true for given test case

Test Generation Principle 1 (Relevance)

State at start of IUT execution must make required precondition true

$\Downarrow$

Test Generation Principle 8 (Oracle Synthesis)

Use "ensures" clauses of contracts (and class invariant) as test oracles

Executable JML Expressions

How to determine whether a JML expression is true in a program state?

It is expensive to check whether a JML expression is true in a state

- Corresponds to first-order model checking, because JML $\sim$ FOL
- PSPACE-complete problem, efficient solutions exist only for special cases
- Identify a syntactic fragment of JML that can be mapped into Java

Executable JML Expressions

How to determine whether a JML expression is true in a program state?

Example

```
\exists int i; 0 <= i && i < ar.length && ar[i] == target
```

is of the form

```
\exists int i; guard(i) && test(i)
```

- `guard()` is JAVA guard expression with fixed upper/lower bound
- `test()` is executable JAVA expression

Executable JML Expressions

How to determine whether a JML expression is true in a program state?

Example

`\exists int i; 0 <= i && i < ar.length && ar[i] == target`

is of the form

`\exists int i; guard(i) && test(i)`

- `guard()` is Java guard expression with fixed upper/lower bound
- `test()` is executable Java expression

Guarded existential JML quantifiers as Java (Example)

```
for (int i = 0; 0 <= i && i < ar.length; i++) {
  if (ar[i]==target) { return true; }
} return false;
```

Executable JML Expressions

How to determine whether a JML expression is true in a program state?

Example

`\exists int i; 0 <= i && i < ar.length && ar[i] == target`

is of the form

`\exists int i; guard(i) && test(i)`

- `guard()` is JAVA guard expression with fixed upper/lower bound
- `test()` is executable JAVA expression

Guarded existential JML quantifiers as Java (General)

```
for (int i = lowerBound; guard(i); i++) {
  if (test(i)) { return true; }
} return false;
```

Executable JML Expressions

How to determine whether a JML expression is true in a program state?

Example

`\exists int i; 0 <= i && i < ar.length && ar[i] == target`

is of the form

`\exists int i; guard(i) && test(i)`

- `guard()` is JAVA guard expression with fixed upper/lower bound
- `test()` is executable JAVA expression

Guarded JML quantifiers as Java

- Universal quantifiers treated similarly (exercise)
- Alternative JML syntax for quantifiers ok as well:

  `\exists int i; guard(i) ; test(i)`

Summary

- Black box vs White box testing
- Black box testing $\sim$ Specification/Model-based Test Generation
- Systematic test case generation from JML contracts
  guided by a few Test Generation Principles
  - Only generate test cases that make precondition true
  - Each operation contract and each disjunction in precondition
    gives rise to a separate test case
  - Coverage criteria, clause coverage
  - Large/infinite datatypes represented by class representatives
  - Values of free variables supplied by Data Generation Principle
  - Create separate test cases for potential aliases and exceptions
- Postconditions of contract and class invariants provide test oracle
- Turn pre- and postconditions into executable JAVA code

What Next?

Remaining Problems of ATCG

1. How to automate specification-based test generation?
2. Generated test cases bear no relation to implementation