Program Verification

Automated Test Case Generation, Part I

Reiner Hähnle

27 November 2007

Introduction

Now we can formally specify program behavior. How to make use of it?

Introduction

Now we can formally specify program behavior. How to make use of it?

Automated Test Case Generation (ATCG)

- ✔ tool support for creating test cases
- ✔ ensuring test case coverage methodically

Introduction

Now we can formally specify program behavior. How to make use of it?

Automated Test Case Generation (ATCG)

- ✔ tool support for creating test cases
- ✔ ensuring test case coverage methodically

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

ATCG Principle

- Specialize contract/code to representative selection of concrete runs
- 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, deduction, symbolic execution, model finding

Automated Test Generation Framework: Unit Tests

Test a single method or function, the implementation under test (IUT)

Create test case for popular JAVA unit test framework: JUNIT

Test Cases in Unit Testing

- Initialisation of test data (test fixture/preamble): create program state from which IUT is started
- Invoke IUT
- Inspection of result: test oracle: tell whether test succeeded: PASS or FAIL

Black box vs White Box Testing

Black box testing

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

Black box vs White Box Testing

Black box testing

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

White box testing

The IUT is analyzed to generate test data for it

Black box vs White Box Testing

Black box testing

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

White box testing

The IUT is analyzed to generate test data for it

Specific Pros and Cons

- ✔ White box testing can use additional information from code
- ✘ White box testing does require source code

Black box vs White Box Testing

Black box testing

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

White box testing

The IUT is analyzed to generate test data for it

Specific Pros and Cons

- ✔ White box testing can use additional information from code
- ✘ White box testing does require source code
- ✔ Black box testing does not require source code
- ✘ Black box testing can be irrelevant/insufficient for IUT

Program States and JML Expressions

Reminder

A given program state S makes a boolean JML expression true or false

Example

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

Then "`arr.length==2 && search(arr, 1)==0`" is true in S

Program States and Test Cases

A desired program state can be reached by suitable test case preamble

Program States and Test Cases

A desired program state can be reached by suitable test case preamble

Example

Assume that `int[] arr` has value $\{1,2\}$ in S
This state can be reached by the following preamble:

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

Program States and Test Cases

A desired program state can be reached by suitable test case preamble

Example

Assume that `int[] arr` has value $\{1,2\}$ in S
This state can be reached by the following preamble:

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

Assume we can compute such initialization code automatically

Specification-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, BZ-TT, JML-TT, UniTesK, JTest,
  TestEra, Cow_Suite, UTJML, ...
- Various specification languages: B, Z, Statecharts, JML, ...
- Detailed formal specification/system model required

Test Generation Principle

View JML contract as formal description of all anticipated runs

Specification-Based Test Generation Principle

- Specialize JML contract to representative selection of concrete runs
- Turn these program runs into executable test cases

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 unintended behavior 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 unintended behavior irrelevant

Test Generation Principle 1

Test data must make required precondition true

Multi-Part Contracts and Test Cases

```
/*@ public normal_behavior
  @ requires Pre1;
  @ ensures  Post1;
  @ also
  @ ...
  @ also
  @ public normal_behavior
  @ requires Pren;
  @ ensures  Postn;
  @*/
  public void m() { ... };
```

Test Generation Principle 2

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 2^5 cases for our simple example)
- ✘ Infinitely many test cases for unbounded data structures
- ✘ Test cases unrelated to specification or IUT

Data-Driven Test Case Generation

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

- ✘ Combinatorial explosion (already 2^5 cases for our simple example)
- ✘ Infinitely many test cases for unbounded data structures
- ✘ Test cases unrelated to specification or IUT

Restriction to test cases that satisfy precondition?

Data-Driven Test Case Generation

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

- ✘ Combinatorial explosion (already 2^5 cases for our simple example)
- ✘ Infinitely many test cases for unbounded data structures
- ✘ Test cases unrelated to specification or IUT

Restriction to test cases that satisfy precondition?

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 suggests at least three test cases related to precondition

Disjunctive Normal Form

Disjunctive Normal Form (DNF)

Assume the requires clause has the form

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

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

Disjunctive Normal Form

Disjunctive Normal Form (DNF)

Assume the requires clause 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

For each disjunct of precondition in DNF create test case making it true

Disjunctive Normal Form

Disjunctive Normal Form (DNF)

Assume the requires clause 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

For each disjunct of precondition in DNF create test case making it true

Example

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

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

Disjunctive Normal Form

Disjunctive Normal Form (DNF)

Assume the requires clause has the form

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

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

Test Generation Principle 3

For each disjunct of precondition in DNF create test case making it true

Importance of Establishing DNF

Implicit 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;`

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

(Specification-based) Test Coverage Criterion

How many different test cases to create that make precondition true?

- At least one (Decision Coverage)
- ...
- All (Multiple Condition Coverage)

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

If the class invariant always holds when a method is called, there is no point to generate test cases from 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

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

Dealing with Large Datatypes (First-Order Logic)

Example (Square root)

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

Dealing with Large Datatypes (First-Order Logic)

Example (Square root)

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

Where is the disjunction?

Dealing with Large Datatypes (First-Order Logic)

Example (Square root)

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

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(MAX_VALUE)`

Equivalence Classes on Input Domains

Example (Square root)

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

Equivalence Classes on Input Domains

Example (Square root)

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

Too many test cases from existential quantifier!

`n = 0*0;`, `n = 1*1;`, ..., `n = MAX_VALUE*MAX_VALUE;`

Equivalence Classes on Input Domains

Example (Square root)

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

Partition large/infinite domains in finitely many equivalence classes

MIN_VALUE	negative values	0	positive values	MAX_VALUE
-2^{31}	-17	0	42	$2^{31}-1$

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

Boundary Values

Example (Square root)

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

Choice of `r=MAX_VALUE` exhibits defective spec for overflow

Boundary Values

Example (Square root)

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

Choice of `r=MAX_VALUE` exhibits defective spec for overflow

Test Generation Principle 5

Include boundary values of ordered domains as class representatives

Boundary Values

Example (Square root)

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

Choosing exact boundary value for `n` amounts to computing result

Computing exact boundary values can be difficult or impossible!

Test Generation Principle 5

Include boundary values of ordered domains as class representatives

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` in precondition!

Data Generation Principles

Test Generation Principle 6

Values of variables without explicit quantification can be freely chosen

Data Generation Principles

Test Generation Principle 6

Values of variables without explicit quantification can be freely chosen

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 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

- Use data generation principle for **int** arrays
- Choose equivalence classes and representatives of **int**, **int**[]:

  int[] empty array, singleton, two elements

  int 0, 1
- Generate all 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 only array element
  `array = { 0 }; target = 0;`
  `array = { 1 }; target = 1;`
- two-element sorted array, target occurs in array, four tests
  `array = { 0,0 }; target = 0;`
  `array = { 0,1 }; target = 0;`
  etc.

Implicit Disjunctions, Part II

Example (Copy)

```
/*@ public normal_behavior
  @ requires src != null && dst != null;
  @  ensures ...
  @*/
static void java.util.Collections.copy (List src,List dst)
```

Implicit Disjunctions, Part II

Example (Copy)

```
/*@ public normal_behavior
  @ requires src != null && dst != null;
  @  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`

- Admission of aliasing often unintended in contract

Forgotten protection against runtime exceptions

`src.length <= dst.length`

Implicit Disjunctions, Part II

Example (Copy)

```
/*@ public normal_behavior
  @ requires src != null && dst != null;
  @  ensures ...
  @*/
static void java.util.Collections.copy (List src,List dst)
```

Test Generation Principle 7

Generate separate test cases that enforce aliasing and raising exceptions

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 tells exactly this
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 tells exactly this
provided that "requires" clause is true for given test case

Test Generation Principle 1

Test data must make required precondition true

Test Generation Principle 8

Use "ensures" clauses (postconditions) of JML contracts as test oracles

Executable JML Expressions

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

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-based Test Generation
- Systematic test case generation from JML contracts guided by 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, decision 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 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 have no relation to implementation

What Next?

Remaining Problems of ATCG

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

1. Tools `jml-junit` and `jtest` discussed in Exercises
2. Code-based test generation that uses symbolic execution of IUT