0.1 Test Automation & Tool Support

0.1.1 Automated Test Case Generation (ATCG)

As writing test cases with good coverage is very time consuming, automated test case generation (ATCG) is often used instead.

Fundamental Approaches

White Box: (**Code Based**) Code of IUT is analyzed to achieve coverage.
- **Syntactic Approach:** Scan for conditions, evaluation. Achieves logic-based criteria.
- **Symbolic Execution:** Unwinding CFG with symbolic values. Achieves structural coverage criteria.
  → **Under-Approximation:** Unreached code.

Black Box: Analysis of input data or model of IUT.

0.1.2 Further Test Automation

Test Coverage Recording

1. Instrument IUT
2. Run test suite, collect information during test runs
3. Analyze and display achieved coverage statistics

Does not help with writing tests, but helps with knowing when to stop.

Test Oracle Synthesis

- **Human Oracle:** Time consuming and error-prone
- **Verdict as Code (assert):** Need expertise, hard to maintain

→ Write test oracle in **formal specification language**, sythesize code from specification

0.1.3 Automatic Static Verification Techniques

Static Checking

Typically based on CFG of IUT and constraint solving
- Runtime exceptions, liveness, information flow
- Fully automated
- Over-approximation, possibly many false positives
- Scales reasonably

Bug Finding

Based on pattern matching and heuristics
- Fast, scales well, handles full Java
- Over- and Under approximation (false positives and incomplete)

SpotBugs is a static analysis tool utilizing bug patterns to find bugs in Java programs. It works at a byte code level.
It sources its bug patterns from **complex language features**, **misunderstood API methods or invariants** and **typos and wrong usage of operators**

0.1.4 Formal Verification

Formal Approaches

- Mathematical Foundation (logic and set theory)
- Sound relative to formal model (strong guarantee)
- Not necessarily complete (not all true properties can be proven)

Often checked for by external programs that use either **Model Checking** or **Deductive Verification** by feeding it the source code and the formal specifications.

0.1.5 Design-by-Contract

Design-by-Contract Example in JML

```
/*@ private normal_behavior
    // What needs to be true for this method to work correctly
    requires 0 <= low <= up <= a.length;
    requires (\forall int x,y;
            0 <= x < y < a.length; a[x] <= a[y]);
    // What this method guarantees to be true after execution
    ensures \result == -1 || low <= \result < up;
    ensures (\exists int idx;
            low <= idx < up; a[idx] == v) ?
        \result >= low && a[\result] == v
        : \result == -1;
    // What the method may modify
    assignable \nothing;
    // Specifies the termination metric
    measured_by up - low;
@*/
private int binSearch(int v, int low, int up) {...}
```

Design-by-Contract

Formal specification:

- Pre- and Postconditions, side effects for each method
- Class and loop invariants

Verification tool proves that each method

- fulfills its contract in all possible runs
- preserves loop and object invariants

0.1.6 Java Modelling Language (JML)

JML is a **contract-based specification language** tailored to java

General JML Philosophy

Integrate

- JML specification
- Java implementation

within a single language

JML is not external to Java, but integrated

0.1.7 Deductive Verification

Working Principle: Path Exploration

Symbolic execution explores all paths in CFG of straight-line programs (no jumps, no loops, no method calls)

- Finite number of paths
- Uses symbolic values to represent all inputs
- Loops approximated by all invariants

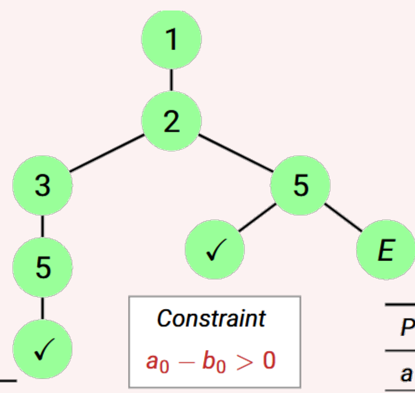
Scalability of Deductive Verification

Approximate effect of a method call with a contract

- During symbolic execution, replace called method with contract
- Substitution and first-order deduction instead of path exploration

Symbolic Execution Example

```
a = a - b;
if (a < 0) {
    a = -a;
}
r = b / a;
```



Constraint: $-a_0 + b_0 < 0$

PV	Value
a	$-a_0 + b_0$
b	b_0
r	$b_0/(-a_0 + b_0)$

Constraint: $a_0 - b_0 > 0$

PV	Value
a	$a_0 - b_0$
b	b_0
r	$b_0/(a_0 - b_0)$

PV	Value
a	$a_0 - b_0$
b	b_0
r	r_0

Constraint: $a_0 - b_0 = 0$