Combining Verification Approaches

http://d3s.mff.cuni.cz

Pavel Parízek
Verification approaches

- Model checking programs
  - Explicit state (Java Pathfinder)
  - Abstraction-based (CEGAR, ...)
- Symbolic execution (concolic testing)
- Deductive methods (Spec#/Boogie)
- Static analysis (data-flow, pointers)
- Abstract interpretation
- Dynamic analysis (runtime)

- Classical testing (e.g., JUnit)
Evaluation

- **Advantages**
  - **Model checking**
    - path-sensitive, very precise, does not scale well (state explosion)
  - **Static analysis**
    - explores all program behaviors, limited precision, highly scalable

- **Limitations**
  - Abstraction-based model checking and deductive methods
    - Problem with concurrency (limited support for threads)
    - Very good at checking properties related to data values
  - Explicit state model checking
    - Supports threads well (detecting concurrency errors)
    - Does not handle data non-determinism very well
Categories

- **Search for errors**
  - testing, symbolic execution, dynamic analysis

- **Search for proofs**
  - program model checking, deductive methods
Search for errors

- Program executed concretely on many inputs
  - Finds only real errors
  - Achieves small coverage

- Abstract execution tracking only some facts
  - Covers all the program paths
  - Reports many false positives

- Intermediate solutions
  - Example: directed concolic testing
Goal: find the safe over-approximation

Model checking: reachable state space

Deductive methods: inductive invariant

Limitations
- Verification procedure might not terminate
- State explosion (many thread interleavings)

Recent solutions: CEGAR
Combining tests and program verification

Detecting some bugs in web applications

Static taint (data flow) analysis for Android

Program termination and checking liveness

Program synthesis: overview, current state
Combining tests and verification

- Search for errors and proofs at the same time
- Using results of one search also in the other

Example: SYNERGY

Example program

\[ x = 0; \]

\[
\text{while } (x < 1000) \{ \\
\quad x = x + 1; \\
\}
\]

\[ \text{assert } (x > 1000); \]
Combining tests and verification

- Goal: compute **inductive invariant** (safety proof) or find a real **counterexample**

- Verification: over-approximation (may)
  - Refine abstraction of the transition relation (abstract state space)

- Tests: under-approximation (must)
  - Generalize inductive invariant from a finite set of finite paths (execution traces)

- Key property of algorithms: **convergence**
Combining tests and verification

- Selected literature
Property-driven reachability (PDR)

- Specific algorithm: IC3


- (... and lot more)
Checking dynamic web applications

- Dynamic programming languages
  - Features: dynamically typed programs, `eval()`
- Implicit input parameters (GET, POST)
- Persistent state (database, cookies)
- Complex patterns of user interactions
- On-the-fly generating of source code
- Control flows through the HTML pages
  - forms, buttons, input events (keyboard, mouse)
Example: Apollo

Example program

```php
<?php
    if (!isset($_GET['step'])) $step = 1;
    else $step = $_GET['step'];
    if ($_GET['login'] == 1) validateAuth();
    switch ($step) {
        case 1: require('login.php'); break;
        case 2: require('news.php'); break;
        case 3: require('inbox.php'); break;
        default: die("wrong input!");
    }
?>
```
Example: **FlowDroid**

- [https://blogs.uni-paderborn.de/sse/tools/flowdroid/](https://blogs.uni-paderborn.de/sse/tools/flowdroid/)
Convergence

• Classic model checking
  □ Program model: abstract reachability tree
  □ Path-sensitive: never joins different paths

• Static program analysis
  □ Program model: control flow graph (inter-proc)
  □ Path-insensitive: losing precision at join points
Generalization

- Abstract domain
- Transfer functions
- Merge operator
- Termination check

- Based on this research paper