Software Engineering
Prof. Dr. Bertrand Meyer
March 2007 – June 2007

Slides: Based on KSE06 - With kind permission of Peter Müller

Static program checking and verification
Correctness

class ArraySet implements Set {
    private int[] array;
    private int next;
    ...

    public void insert(int i) {
        for (int j = 0; j < next; j-- )
            if array[ j ] == i then return true;
        return false;
    }
}

Behavioral Specification

Semantic Rules

Context Conditions

Syntax Rules

Software Engineering, lecture 20: Static program checking and verification
Aspects of correctness

Semantics
- Behavioral Specification
- Semantic Rules

Context Conditions

Syntax
- Syntax Rules

Test, Verification
- Semantic Analysis, Type Checking
- Scanning, Parsing
Test and verification

**Test**

**Objective**
- Detect bugs

**Examples**
- White box test
- Black box test

**Problems**
- Successful test does not guarantee correctness

**Verification**

**Objective**
- Prove correctness

**Examples**
- Formal verification based on a logic
- Symbolic execution

**Problems**
- Expensive
- Formal specification of behavior is required
Levels of coverage

<table>
<thead>
<tr>
<th>Coverage</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type checking</td>
<td></td>
</tr>
<tr>
<td>Extended static checking</td>
<td></td>
</tr>
<tr>
<td>Program verification</td>
<td></td>
</tr>
</tbody>
</table>

Decidability ceiling
Extended static checking

**ESC/Java** developed at DEC, Compaq, and HP Research

**Fully automated** tool

**Tries to verify**

- Absence of runtime exceptions and common mistakes  
  e.g. null dereference, array bounds, type cast errors, deadlocks
- Simple user-specified contracts  
  invariants, pre/postconditions, loop invariants, assertions

**Program with specifications**  
**Program Checker/Verifier**  
**Error messages**

Bag.java:18: Array index possibly too large
Program checker design tradeoffs

Objectives

- Fully automated reasoning
- As little annotation overhead as possible
- Performance

Not sound
- Errors may be missed

Not complete
- Warnings do not always report errors (false alarms)

Goal

- Cost-effective tool
- Find source of possible bugs quickly

Main reason why it’s called checker and not verifier
Tool architecture

Annotated Java program

Translator

Verification condition

Automatic Theorem Prover

Counterexample context

Post Processor

Warning messages

Valid

Resource exhausted
Theorem prover: “Simplify”

Automatic: **No user interaction**

**Refutation based:** To prove $\varphi$ it will attempt to satisfy $\neg \varphi$

- If this is possible, a counterexample is found, and we know a reason why $\varphi$ is invalid
- If it fails to satisfy $\neg \varphi$ then $\varphi$ is considered to be valid
Time limits

Logic used in Simplify is **semi-decidable**

- Each procedure that proves all valid formulas loops forever on some invalid ones

Simplify works with a **time limit**

- When time limit is reached, counterexample is returned
- Longer computation might show that returned counterexample is inconsistent

Time limits are a source of **incompleteness**

- Spurious counterexamples lead to spurious warnings
ESC/Java2

Successor of ESC/Java

Eclipse integration

Made specification language compatible with JML

Made open source

Give it a try!
http://secure.ucd.ie/products/opensource/escjava2
Spec#

Program verification tool developed at MS Research

Superset of C#
- non-null types
- pre- and postconditions
- object invariants

Tool support
- more type checking
- compiler-emitted run-time checks
- static program verification
- fully integrated into Visual Studio .NET 2005
Spec# vs. ESC/Java(2)

Similarities
- Architecture
- Full automation (even theorem prover is the same)
- Essential contract language

Differences
- Spec# is sound
- Spec# does modular reasoning
  - price to pay: need to understand methodology
Non-null types

T x;
The value of x is
- null or
- reference to object whose type is a subtype of T.

T! y;
The value of y is
- reference to object whose type is a subtype of T,
  and not null.
Types versus assertions

Without non-null types:

\[
\text{Person(}\text{string name)} \\
\text{requires name} \neq \text{null};
\]

With non-null types:

\[
\text{Person(}\text{string! name)}
\]
Comparing against null

```java
public void M(T x) {
    if (x == null) {
        ...
    } else {
        T! y = x;
        ...
    }
}
```

Spec# performs a data-flow analysis to allow this
Spec# DEMO
References

ESC/Java
- Flanagan et al.: Extended Static Checking for Java

ESC/Java2
- http://secure.ucd.ie/products/opensource/ESCJava2

Spec#
- Barnett et al.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs
- http://research.microsoft.com/specsharp

Rustan Leino's lectures
- http://research.microsoft.com/~leino/talks.html