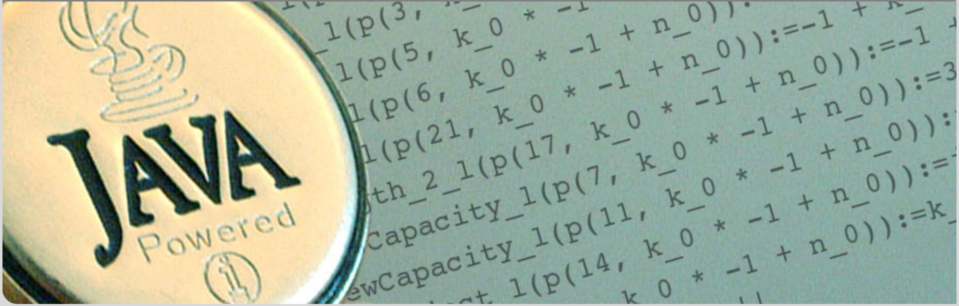
Specification & Formal Analysis of Java Programs

Introduction

Prof. Dr. Bernhard Beckert | ADAPT 2010

KIT – INSTITUT FÜR THEORETISCHE INFORMATIK


JAVA
Powered
1(p(5, k_0 * -1 + n_0)):=-1
1(p(6, k_0 * -1 + n_0)):=-1
1(p(21, k_0 * -1 + n_0)):=3
th_2_1(p(17, k_0 * -1 + n_0)):
Capacity_1(p(7, k_0 * -1 + n_0)):=
ewCapacity_1(p(11, k_0 * -1 + n_0)):=k_
1(p(14, k_0 * -1

Karlsruhe Institute of Technology

Merger of

- Karlsruhe University (state funded)
- Research Center Karlsruhe (funded by federal government)

Figures

Karlsruhe Institute of Technology

Employees

8,500

19,700

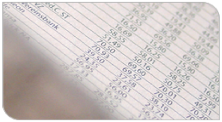
Students

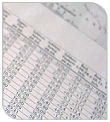
364

Professors

650

Annual Budget in Million Euros





Motivation:
Software Defects Cause BIG Failures

Tiny faults in technical systems can have catastrophic consequences

In particular, this goes for software systems

- Ariane 5
- Mars Climate Orbiter, Mars Sojourner
- London Ambulance Dispatch System
- Denver Airport Luggage Handling System
- Pentium Bug
- EC Card Bug

Motivation: Software Defects cause OMNIPRESENT Failures

Ubiquitous Computing results in Ubiquitous Failures

Software these days is inside just about anything:

- Mobiles
- Smart devices
- Smart cards
- Cars
- Aviation

⇒ *software—and specification—quality is a growing legal issue*

Achieving Reliability in Engineering

Some well-known strategies from civil engineering

- Precise calculations/estimations of forces, stress, etc.
- Hardware redundancy (“make it a bit stronger than necessary”)
- Robust design (single fault not catastrophic)
- Clear separation of subsystems
  Any air plane flies with dozens of known and minor defects
- Design follows patterns that are proven to work

Why This Does Not Work For Software

- Software systems compute non-continuous functions
  Single bit-flip may change behaviour completely
- Redundancy as replication doesn’t help against bugs
  Redundant SW development only viable in extreme cases
- No clear separation of subsystems
  Local failures often affect whole system
- Software designs have very high logic complexity
- Design practice for reliable software in immature state
  for complex, particularly, distributed systems
- Extremely short innovation cycles

Limitations of Testing

- Testing adaptive systems is difficult
- Testing shows the presence of errors, in general not their absence
  (exhaustive testing viable only for trivial systems)
- Representativeness of test cases/injected faults subjective
  How to test for the unexpected? Rare cases?
- Testing is labor intensive, hence expensive

Formal Methods: The Scenario

- Rigorous methods used in system design and development
- Mathematics and symbolic logic $\Rightarrow$ formal
- Increase confidence in a system
- Two aspects:
  - System implementation
  - System requirements
- Make formal model of both and use tools to prove mechanically
  that formal execution model satisfies formal requirements

Formal Methods: The Vision

- Complement other analysis and design methods
- Are good at finding bugs
  (in code and specification)
- Reduce development (and test) time
- Can *ensure* certain *properties* of the system model
- Should ideally be as automatic as possible

Various Properties
(Require Different Verification Techniques)

- Simple properties
  - Safety properties
    Something bad will never happen (eg, mutual exclusion)
  - Liveness properties
    Something good will happen eventually
- General properties of concurrent/distributed systems
  - deadlock-free, no starvation, fairness
- Non-functional properties
  - Runtime, memory, usability, . . .
- Full behavioural specification
  - Code satisfies a contract that describes its functionality
  - Data consistency, system invariants
    (in particular for efficient, i.e. redundant, data representations)
  - Modularity, encapsulation
  - Refinement relation

The Main Point of Formal Methods is **Not**

- To show “correctness” of entire systems
  What *IS* correctness? Always go for specific properties!
- To replace testing entirely
  - Formal methods work on models, on source code, or, at most, on bytecode level
  - Many non-formalizable properties
- To replace good design practices

There is no silver bullet!

- No correct system w/o clear requirements & good design
- One can’t formally verify messy code with unclear specs

But . . .

- Formal proof can replace (infinitely) many test cases
- Formal methods can be used in automatic test case generation
- Formal methods improve the quality of specs
  (even without formal verification)
- Formal methods guarantee specific properties of a specific system model

Formal Methods Aim at:

- Saving money
  Intel Pentium bug
  Smart cards in banking
- Saving time
  otherwise spent on heavy testing and maintenance
- More complex products
  Modern μ-processors
  Fault tolerant software
- Saving human lives
  Avionics, X-by-wire
  Washing machine

Tool Support is Essential

Some Reasons for Using Tools

- Automate repetitive tasks
- Avoid clerical errors, etc.
- Cope with large/complex programs
- Make verification certifiable