An Introduction to
Analysis and Verification of Software

Anders Møller

University of Aarhus

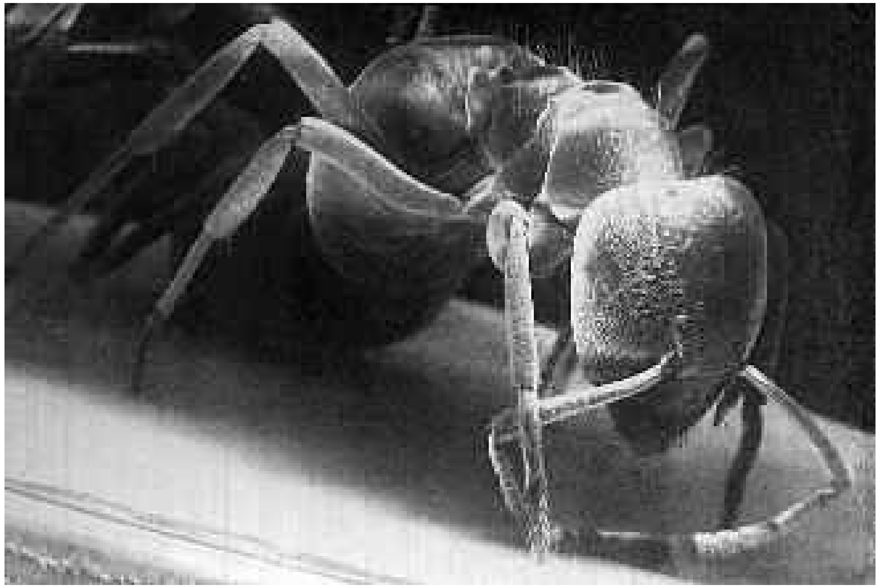
BRICS

Goals of this talk

- To give a brief overview of the area
- Establish some terminology, including some "common knowledge"
- Propose study group projects
- Present some aspect by which the various projects may be evaluated and compared

Bugs

bug: a mismatch between **implementation** and **specification**



- If there is no spec, it's not a bug but a feature :-)

- Catching bugs after deployment can be expensive

- Detecting bugs vs. guaranteeing correctness?

Properties of Interest

Where does (semi-)**automated** analysis and verification have a chance?

- resource management (memory, files, allocation, locking)
- temporal properties (event ordering, concurrency, deadlocks, safety/liveness)
- datatype invariants (shapes, memory errors)
- security (integrity, confidentiality)
- numerical computations
- ...

Typical bugs? See e.g. bugzilla.mozilla.org or bugzilla.kernel.org

Rice's Theorem

"*Everything interesting about the behavior of programs is undecidable.*"

[paraphrase of H.G. Rice, 1953]

- Approximations (perferably conservative)
- Annotations (invariants)

Verification vs. Analysis

- Verification: *checking* invariants
- Analysis: *detecting* invariants

  - this is just one possible definition of the words
  - in practical tools, there is a large overlap
  - analysis is not necessarily “harder” than verification

Relation to Debugging

- **Debugging**: you know there is a bug,
  but not exactly where it is
- **Verification**: you hope there are no bugs,
  but you want to be sure

- Analysis tools can be used to enhance program understanding (e.g. “program slicing”)
- Verification tools can often provide counterexamples

Relation to Testing

“*Program testing can be used to show the presence of bugs, but never to show their absence.*” [Dijkstra, 1972]

Relation to Program Optimization

- Many of the same program analysis techniques apply

- **Moore's law:** “the performance of microprocessors doubles every 18 months”
- **Proebsting's law:** “compiler technology doubles the performance of typical programs every 18 years”

- Conclusion: analysis/verification is more fun than optimization ☺
- or: invent new high-level languages that need new optimization techniques!

Relation to Programming Language Design

- Programming at a higher level of abstraction can reduce the possibility of errors

- Examples:
  - type systems (note: types are invariants!)
  - abstract data-types
  - object oriented encapsulation and inheritance
  - domain-specific languages

A Spectrum of Techniques

- Light-weight
  - **unsound** and **incomplete**
  - bug searching via simulation
  - simple properties, efficient and easy to use
- Medium-weight
  - **sound** but **incomplete**
  - analysis via fixed-point computation, type checking/inference
- Heavy-weight
  - **sound** and **complete**
  - verification via theorem proving and user annotations
  - complex properties, resource demanding and difficult to use

Aspects

- domain of applicability (model heap, object state, events, numerical, parallelism, hardware/software, sequential/concurrent, reactive/transformational...)
- expressive power (fixed properties vs. using logic/automata in specs)
- degree of automation, annotations, theorem prover guidance
- modularity (interprocedural, whole-program, iteration, ...)
- scalability (program size)
- efficiency (time & space), theoretical complexity
- learnability (training requirements)
- soundness/completeness, spurious errors, heuristics, transparency (e.g. type systems should be transparent)
- error tracking (precision of error messages), counterexamples
- division between program abstraction and solution computation (verification conditions)

Approaches

- Axiomatic Semantics
  - Hoare logic, weakest precondition, separation logics
  - often used for sequential transformational programs
- Abstract Interpretation
  - data-flow analysis, constraint-based analysis
- Model Checking
  - state-space exploration with modal logics
  - often used for concurrent reactive systems
- Type Systems
  - type checking and inference

Hot Research Projects and Topics

- SLAM - model checking for boolean programs and C, abstraction refinement
- Java PathFinder - model checking for Java
- Bandera - model checking for Java
- Blast - model checking for C, abstraction refinement
- SPIN - LTL model checking
- ESP - control-flow + data-flow analysis on C/C++
- 3-Valued Logic Analysis Engine - TVLA - shape analysis
- BANE/Banshee - constraint-based program analysis
- Abstract Interpretation - foundation for program analysis
- Behave! - checking behavioral properties using type systems and Pi-calculus
- Vault - type system for a safe version of C
- Cyclone - type system for a safe version of C
- CQual - type qualifiers for C
- PVS - verification system based on theorem proving with higher-order logic
- ESC/Java - theorem proving for light-weight properties
- Proof-Carrying Code - safe execution of untrusted code
- Meta-Level Compilation / Metal - finding bugs in system code, e.g. Linux kernel