Algorithmic Verification

Comp4151
Lecture 1
Ansgar Fehnker

Content

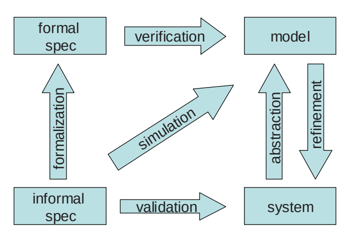
- Welcome
- Who are we?
- Who are you?
- What is the course about?
  - What is algorithmic verification?
  - What went wrong?
  - What is model checking?
  - Where did it come from?
  - What is it good for?
- What about the course?

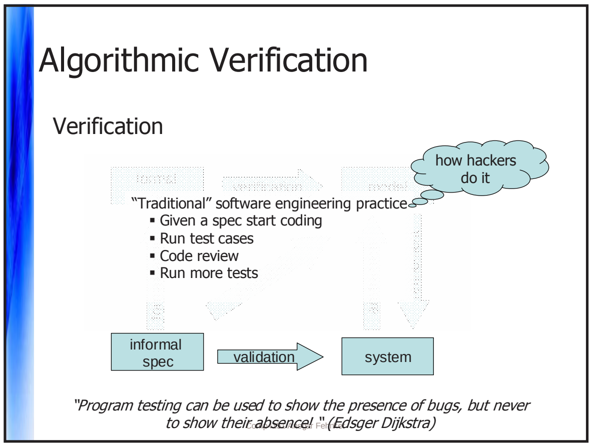
Who are we?

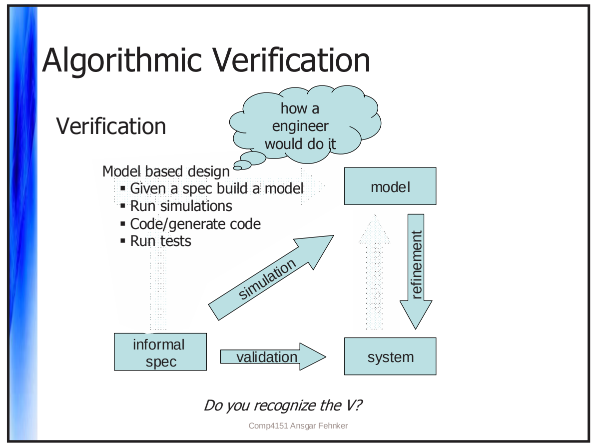
- Ralf Huuck (LiC)
- *Ansgar Fehnker*

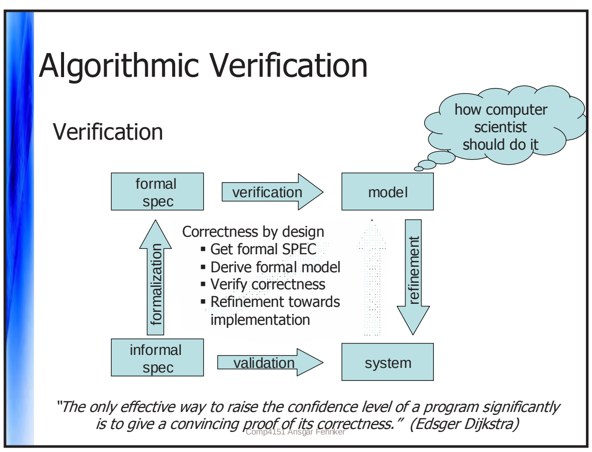
Algorithmic Verification

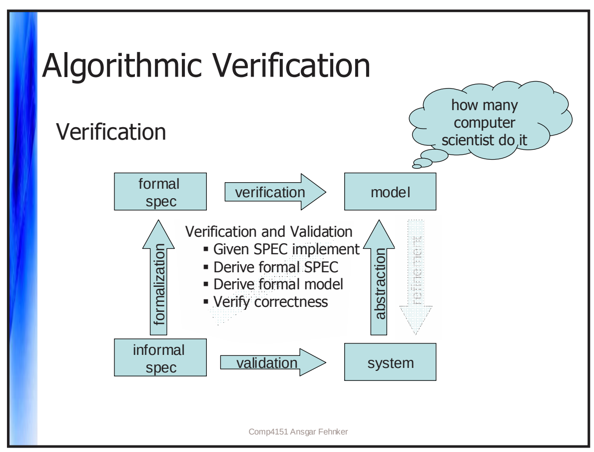
Verification




Algorithmic Verification
Verification
how hackers do it
"Traditional" software engineering practice
Given a spec start coding
Run test cases
Code review
Run more tests
informal spec
validation
system
"Program testing can be used to show the presence of bugs, but never to show their absence! " (Edsger Dijkstra)

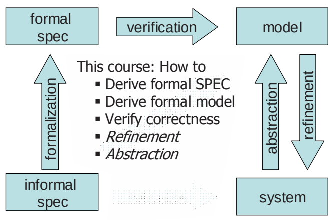

Algorithmic Verification
Verification
how a engineer would do it
Model based design
Given a spec build a model
Run simulations
Code/generate code
Run tests
model
simulation
refinement
informal spec
validation
system
Do you recognize the V?
Comp4151 Ansgar Fehnker


Algorithmic Verification
Verification
how computer scientist should do it
formal spec
verification
model
Correctness by design
Get formal SPEC
Derive formal model
Verify correctness
Refinement towards implementation
formalization
refinement
informal spec
validation
system
"The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness." (Edsger Dijkstra)
Comp4151 Ansgar Fehnker


Algorithmic Verification
Verification
how many computer scientist do it
formal spec
verification
model
Verification and Validation
Given SPEC implement
Derive formal SPEC
Derive formal model
Verify correctness
formalization
abstraction
informal spec
validation
system
Comp4151 Ansgar Fehnker

Algorithmic Verification

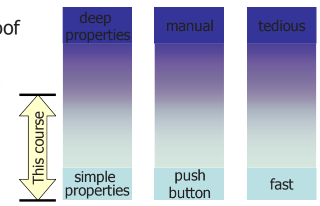
Verification



Algorithmic Verification

Verification techniques

- Mathematical proof
- Theorem proving
- Model checking
- Static analysis



Algorithmic Verification

- Testing and simulation have proven to work
- Why should we care about formal correctness?

Algorithmic Verification

- Testing and simulation have proven to work
- Why should we care about formal correctness?

Comp4151 Ansgar Fehnker

Windows

```
An exception  06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) +
00001660.  This was called from 0028:C11B40C8 in VxD voltrack(04) +
00000000.  It may be possible to continue normally.

*  Press any key to attempt to continue.
*  Press CTRL+ALT+RESET to restart your computer.  You will
   lose any unsaved information in all applications.

                    Press any key to continue
```

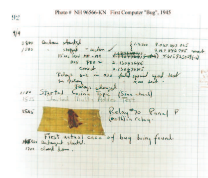
"We must not put mistakes into programs because of sloppiness, we have to do it systematically and with care." *(Edsger Dijkstra)*

Correctness

Famous bugs

First computer bug (1945)

- "First actual case of bug being found" by Grace Hopper
- Operators noticed an error in the Mark II
- It was caused by a moth trapped in a relay
- Bug on display in the Smithsonian

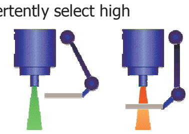


Correctness

Famous bugs

Therac-25 Accident :

- X-ray machine with two modes
  - X-rays, generated high energy electron-beam directed on metal shield (between beam and patient)
  - Low energy electron-beam without metal target
- A software error let operator inadvertently select high energy beam without metal shield.
- Results: At least five patients die.



Correctness

Famous bugs

Pentium bug (1994)

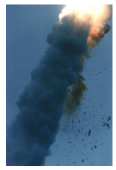
- First release of Intel Pentium chip
- Mistakes when dividing floating-point numbers that occur within a specific range
- Estimated 3 million to 5 million defective chips
- PR nightmare for Intel
- Cost : $475 million

Correctness

Famous bugs

Ariane 5 (1996)

- Ariane 5 used software used prior in Ariane 4
- 64-bit floating-point to 16-bit integer generated conversion an overflow
- Error was caught, sub-system shut down
- Back-up systems failed for the same reason.
- Rocket veered off course.
- Control system decided to abort mission.
- Result: Rocket self-destructed
- Cost : $400 million payload



Correctness

Famous bugs

USS Yorktown (1998)

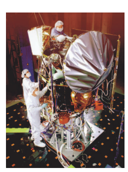
- A program did not check for valid input.
- A crew member entered by mistake zero.
- Resulted in division by zero.
- Lead eventually to shut down of the ship's propulsion system
- Result: The ship was dead in the water for several hours

Correctness

Famous bugs

Mars Climate Orbiter (1999)

- One development team used pound/second in their code while the other used Newton/second
- Vlaues passed from one module to another witout conversion
- Result: Loss of the craft
- Cost: $ 125 million

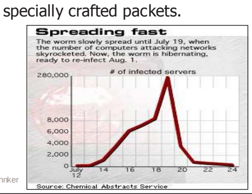


Correctness

Famous bugs

Code Red:

- Potential buffer over-flow in Microsoft Internet Information Server
- Worm uses exploit. It sends specially crafted packets.
- Triggering a buffer overflow
- Giving worm administrative privileges to the worm
- Cost: > $2 billion.



A solution

Microsoft Powerpoint EULA Point 11

- **11**. EXCLUSION OF INCIDENTAL, CONSEQUENTIAL AND CERTAIN OTHER DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, **IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER** (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS OR CONFIDENTIAL OR OTHER INFORMATION, FOR BUSINESS INTERRUPTION, FOR PERSONAL INJURY, FOR LOSS OF PRIVACY, FOR FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE, AND FOR ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) **ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT,** THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, OR OTHERWISE UNDER OR IN CONNECTION WITH ANY PROVISION OF THIS EULA, EVEN IN THE EVENT OF THE FAULT, TORT (INCLUDING NEGLIGENCE), STRICT LIABILITY, BREACH OF CONTRACT OR BREACH OF WARRANTY OF MICROSOFT OR ANY SUPPLIER, AND EVEN IF MICROSOFT OR ANY SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

A solution

The GPL

- **11.** BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. **THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU**. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
- **12. IN NO EVENT** UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING **WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM** AS PERMITTED ABOVE, **BE LIABLE TO YOU FOR DAMAGES**, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES **ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM** (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The problem

The software crisis

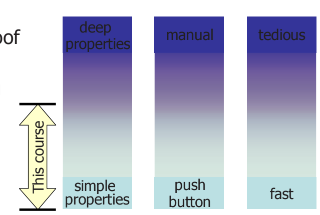
- Computer become more powerful (Moore's law)
- The quality of programs cannot keep up
  - Up to 80% of all software development time is spent on locating and correcting defects
  - About 70% of all cost in hardware design go to verification and validation
  - Rework due to defects identified accounts for between 40% and 50% of total project cost

"When there were no computers programming was no problem. When we had a few weak computers, it became a mild problem. Now that we have gigantic computers, programming is a gigantic problem." (Edsger Dijkstra)

Algorithmic Verification

Verification techniques

- Mathematical proof
- Theorem proving
- Model checking
- Static analysis



What is model checking?

Model checking

The basic idea

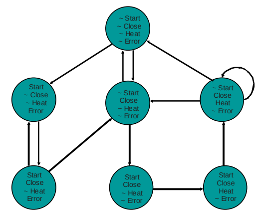
- Given a model of the system
  - Kripke structre, FSM, automaton, Petri net, …
- Given a formal specification
  - LTL, CTL, mu-calculus, …
  - another simpler model
- Calculate whether model satisfies specification

No proofs. (But you need math to build a model checker)
Fast (compared to other rigorous approaches)
Gives counter-examples (help with debugging, too)

Model Checking

Microwave Oven Example

The model
- state-transition graph
- describes system evolving over time.

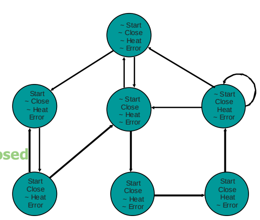


Model Checking

Microwave Oven Example

The property
- The oven doesn't heat up until the door is closed.
- Not **heat_up** until **door_closed**
- In temporal logic

(~ **heat_up**) **U** **door_closed**



Property Specifications

Temporal Logic

- Express properties of event orderings in time
- Basic operators
  - Let "**p**" atomic proposition, e.g. "**Device Enabled**".

Fp - p holds sometime in the ***future.***
Gp - p holds ***globally*** in the future.
Xp - p holds ***next*** time.
p**U**q - p holds ***until*** q holds.

Property Specifications

Temporal Logic

- Express properties of event orderings in time

- Linear Time
  - Every moment has a unique successor
  - Infinite sequences (words)
  - Linear Time Temporal Logic (LTL)
- Branching Time
  - Every moment has several successors
  - Infinite tree
  - Computation Tree Logic (CTL)

Property Specifications

Safety and Liveness

- Safety properties
  - Invariants, deadlocks, reachability, etc.
  - Can be checked on finite traces
  - "something bad never happens"
- Liveness Properties
  - Fairness, response, etc.
  - Infinite traces
  - "something good will eventually happen"

Temporal Logic Model Checking

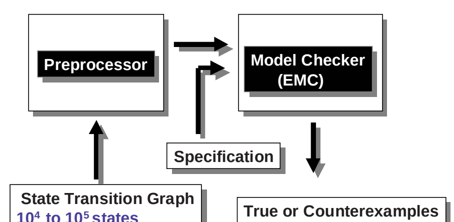
History

- Model checking introduced as *automatic verification technique* for *finite state concurrent systems*.
- Developed independently by *Clarke, Emerson, and Sistla* and by *Queille and Sifakis* in early 1980's.
- Specifications are written in *propositional temporal logic*.
- Verification procedure is an *exhaustive search of the state space* of the design.

Model Checking

EMC

The first model checker by Clarke and Emerson



Model Checking

State Explosion Problem

- The size of the model grows exponentially
- Example
  - A 50 x 50 wireless network.
  - Each node has 3 states: wait, send, sleep
  - Composed system has $3^{250} \approx 10^{125}$ states
  - Compare to 10^{78} atoms in universe

- 25 years of research to combat state explosion problem

Model Checking

SMV (Ken McMillan, CMU, 1987)

- First breakthrough by symbolic model checking
- Using *Binary Decision Diagrams* to represent state transition systems more efficiently.
- Could handle large state spaces
  - Heuristics to handle search spaces well
  - Specification: CTL (and later LTL)
  - by far the most useful technique in the hardware domain

Model Checking

SPIN (Holzmann, Bell Labs, ’90s)

- Explicit-state model checker
- Uses PROMELA modeling language
- Heuristics to control state-space explosion
  - Partial order reduction
  - Hashing and approximate search
  - Specification: LTL / Buechi automata
- Succesful in protocol verification

Model Checking

- Advent of SAT tools (2000)
  - Check if a boolean formaula is satisfiable
  - zChaff (Princeton) first tool
  - Handles formulas with 100000 variable, and millions of clauses!

Model Checking

SAT-based techniques

Bounded model checking
- Is there a path of length *k* that reaches an unsafe state?
- Transform problem to a satisfiablity problem.

Counterexample guided abstraction refinement
- Use small abstraction to compute potential counterexamples
- Use efficient SAT-solver to check potential counterexamples

SAT-solvers are used by most modern model checkers

Model Checking

SAT-based tools

SLAM (Ball and Rajamani, 2000)
- Developed by Microsoft Research
- Verifies device drivers against formal specifications

C-BMC (Kroening, 2002)
- Bounded model checker for ANSI-C

Model Checking

Static analysis

- Static analysis to find patterns of bad programming practice in systems code.
- Very successful in terms of errors found
  - 100s of bugs (incl security) found in Linux/BSD
  - Errors in various protocols, drivers.
  - Explicit-state analysis on CFG.

Model checking

Hardware vs software model checking

Hardware model checking
- BDD-based model checking was the enabling technology
- Hardware is typically synchronous and regular
- Known semantics
- The Intel Pentium bug, got model checking on the map

Software
- Focus until the late 90's on design, rather than programs
- Fuzzy program semantics
- Contrary to tradition: Code first, test later.
- Catching bugs early is more cost-effective
- SAT and abstraction based techniques state-of-the-art

Model Checker Performance

State-of-the-art

- Model checkers today can routinely handle systems with between *100* and *1000 state variables*.
- Systems with 10^{120} *reachable states* have been checked.
- By using appropriate abstraction techniques, systems with an essentially *infinite number* of states can be checked.
- There are many *successful examples* of the use of model checking in *hardware and protocol* verification.

Algorithmic Verification

Hardware verification
  - Verifying microprocessor designs, cache coherence protocols
  - Tools: SMV, nuSMV, VIS, Mocha, FormalCheck
- Protocol verification
  - Network/Communications protocol implementations
  - Tools: Spin
- Software verification
  - Apply directly to source code (e.g., device drivers)
  - Tools: SLAM, Blast, Magic
- Embedded and real time systems
  - Tools: Uppaal, HyTech, Kronos, Charon, Phaver
- Static Analysis
  - Tools: Coverity, Polyspace, Flexelint, UNO, Klocwork, *Goanna*

The course

Content

- *Introduction*
- Modelling Systems
- Temporal Logic
- *CTL Model Checking*
- NuSMV
- *LTL Model Checking*
- Spin
- Partial order and symmetry reduction
- *SAT-based model checking*
- Static Analysis
- *Model checking Timed Automata*
- *Beyond time*

The course

Homework 1
- 3rd to 4th week of March

Verification Project
- 2nd week of April to 1st week of May

Homework 2
- 3rd to 4th week of May

Exam in June

Assessment Criteria
- Homework: 25%
- Verification Project: 25%
- Final Exam: 50% (2h, written)

The course

- When and Where
  - Tues 14:00 - 16:00 (K-B11B-8)
  - Thu 14:00 - 15:00 (APPSC G02)
- Office hour:
  - Thu 15:00-16:00
- Dr. Ralf Huuck (LiC)
  - Email: rhuuck
  - Phone: 8306 0493
  - Office: Room E523, L5 Building
- Dr. Ansgar Fehnker
  - Email: ansgar
  - Phone: 8306 0490
  - Office: Room E520, L5 Building

Questions?

http://www.cse.unsw.edu.au/~cs4151/