Summer Term 2026

Program Verification

Part 1 – Introduction

René Thiemann

Department of Computer Science

Organization

Lecture (VO 3)

- LV-Number: 703083
- lecturer: René Thiemann
  consultation hours: Tuesday 10:15–11:15
  ICT-building, 2nd floor, 3M09
- time: Monday, 8:30–11:00, with break(s) in between
- place: SR 12
- course website: `http://cl-informatik.uibk.ac.at/teaching/ss26/pv/`
- slides are available online and contain links
- online registration required before June 30
- lecture will be in German

Schedule

lecture 1	March	2	lecture 8	May	4
lecture 2	March	9	lecture 9	May	11
lecture 3	March	16	lecture 10	May	18
lecture 4	March	23	lecture 11	June	1
lecture 5	April	13	lecture 12	June	8
lecture 6	April	20	Q & A	June	15
lecture 7	April	27			
1st exam	June	22			

lecture 7 might be video only

Proseminar (PS 2)

- LV-Number: 703084
- time and place: Tuesday, 13:45–15:15 in SR 12
- online registration was required before February 21
- late registration directly after this lecture by contacting me
- exercises available online on Tuesday evening at the latest
- solved exercises must be marked in OLAT
  (deadline: Monday 5pm)
- solutions will be discussed in proseminar groups
- first exercise sheet: today
- proseminar starts on March 10
- attendance is mandatory (2 absences tolerated without giving reasons)
- exercise sheets will be in English, solutions can be in either English or German

Weekly Schedule

- Monday 8:30–11:00: lecture n on topic n
- Monday 17:00: deadline for marking solved exercises of sheet $n - 1$ in OLAT
- Tuesday 13:45–15:15: proseminar on exercise sheet $n - 1$
- Tuesday evening: exercise sheet n is available
- next Monday 8:30–11:00: lecture $n + 1$ on topic $n + 1$
- ...

Grading

- separate grades for lecture and proseminar
- lecture
  - written exam (closed book)
  - 1st exam on June 22, 2026
  - online registration required
    - opening 5 weeks before exam
    - closing 2 weeks before exam
    - deregister until Thursday evening before exam
  - 2nd and 3rd exam in September and February (on demand)
- proseminar
  - 80 %: scores from weekly exercises
  - 20 %: presentation of solutions

Literature

- slides
  - no other topics will appear in exam ...
  - ... but topics need to be understood thoroughly
    - read and write specifications and proofs
    - apply presented techniques on new examples
    - not only knowledge reproduction

- Nipkow and Klein: Concrete Semantics with Isabelle/HOL. Springer.
- Huth and Ryan: Logic in Computer Science, Modelling and Reasoning about Systems. Second Edition. Cambridge.
- Robinson and Voronkov: Handbook of Automated Reasoning, Volume I. MIT Press.

Motivation

What is Program Verification?

- program verification
  - method to prove that a program meets its specification
  - does not execute a program
  - incomplete proof: might reveal bug, or just wrong proof structure
  - verification often uses simplified model of the actual program
  - requires human interaction
- testing
  - executes program to detect bugs, i.e., violation of specification
  - cannot prove that a program meets its specification
  - similar to checking 1 000 000 possible assignments of propositional formula with 100 variables, to be convinced that formula is valid (for all 2^{100} assignments)
- program analysis
  - automatic method to detect simple propositions about programs
  - does not execute a program
  - examples: type correctness, detection of dead-code, uninitialized variables
  - often used for warnings in IDEs and for optimizing compilers
- program verification, testing and program analysis are complementary

Verification vs Validation

- verification: prove that a program meets its specification
  - requires a formal model of the program
  - requires a formal model of the specification
- validation: check whether the (formal) specification is what we want
  - turning an informal (textual) specification into a formal one is complex
  - already writing the formal specification can reveal mistakes, e.g., inconsistencies in an informal textual specification

Example: Sorting Algorithm

- objective: formulate that a function is a sorting algorithm on arrays
- specification via predicate logic:

$$\begin{aligned} &sorting\_alg(f) \longleftrightarrow \forall xs\ ys : [int].\\ &\quad f(xs) = ys \longrightarrow\\ &\quad \forall i.\, 0 < i \longrightarrow i < length(ys) \longrightarrow ys[i-1] \leq ys[i] \end{aligned}$$

- specification is not precise enough, think of the following algorithms
  - algorithm which always returns the empty array
    consequence: add $length(xs) = length(ys)$ to specification
  - the algorithm which overwrites each array element with value 0
    consequence: need to specify that xs and ys contain same elements

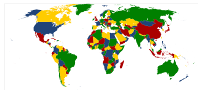
Necessity of Verification – Software

- buggy programs can be costly:
  crash of Ariane 5 rocket ($\sim$ 370 000 000 $)
  - parts of 32-bit control system was reused from successful Ariane 4
  - Ariane 5 is more powerful, so has higher acceleration and velocity
  - overflow in 32-bit integer arithmetic
  - control system out of control when handling negative velocity
- buggy programs can be fatal:
  - faulty software in radiation therapy device led to 100x overdosis and at least 3 deaths
  - system error caused Chinook helicopter crash and killed all 29 passengers
- further problems caused by software bugs

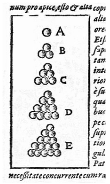
  https://raygun.com/blog/costly-software-errors-history/

Necessity of Verification – Mathematics

- programs are used to prove mathematical theorems:
  - 4-color-theorem: every planar graph is 4-colorable

    - proof is based on set of 1834 configurations
    - the set of configurations is unavoidable
      (every minimal counterexample belongs to one configuration in the set)
    - the set of configurations is reducible (none of the configurations is minimal)
    - original proof contained the set on 400 pages of microfilm
    - reducibility of the set was checked by program in over 1000 hours
    - no chance for inspection solely by humans, instead verify program
  - Kepler conjecture

    - statement: optimal density of stacking spheres is $\pi/\sqrt{18}$
    - proof by Hales works as follows
    - identify 5000 configurations
    - if these 5000 configurations cannot be packed with a higher density than $\pi/\sqrt{18}$, then Kepler conjecture holds
    - prove that this is the case by solving $\sim 100\,000$ linear programming problems
    - submitted proof: 250 pages + 3 GB of computer programs and data
    - referees: 99 % certain of correctness

Successes in Program Verification

- mathematics:
  - 4-color-theorem
  - Kepler conjecture

  both the constructed set of configurations as well as the properties of these sets have been guaranteed by executing verified programs
- software:
  - CompCert: verified optimizing C-compiler (ACM Software System Award 2021)
  - seL4: verified microkernel (ACM Software System Award 2022)
    free of implementation bugs such as
    - deadlocks
    - buffer overflows
    - arithmetic exceptions
    - use of uninitialized variables
  - Nitro Isolation Engine: Amazon Web Services use a formally verified hypervisor (2025)
    - workloads are isolated from each other and AWS operators

Program Verification Tools

- doing large proofs (correctness of large programs) requires tool support
- proof assistants help to perform these proofs
- proof assistants are designed so that only small part has to be trusted
- examples
  - academic: Isabelle/HOL, ACL2, Rocq (formerly: Coq), HOL Light, Why3, Key,...
  - industrial: Lean (Microsoft), Dafny (Microsoft), PVS (SRI International, NASA), ...
  - generic tools: Isabelle/HOL (seL4, Kepler, Nitro IE), Roqc (CompCert, 4-Color-Thm), ...
  - specific tools: Key (verification of Java programs), Dafny, ...
- master courses on Interactive theorem proving:
  include more challenging examples and tool usage
- this course: focus on program verification on paper
  - learn underlying concepts
  - freedom of mathematical reasoning ...
  - ... without challenge of doing proofs exactly in format of particular tool
  - also included: a glimpse of Isabelle/HOL

Example Proof

- program (defined over lists via constructors Nil and Cons)

$$\mathsf{append}(\mathsf{Nil}, ys) = ys \quad (1)$$
$$\mathsf{append}(\mathsf{Cons}(x, xs), ys) = \mathsf{Cons}(x, \mathsf{append}(xs, ys)) \quad (2)$$

- property: associativity of append:

$$\mathsf{append}(\mathsf{append}(xs, ys), zs) = \mathsf{append}(xs, \mathsf{append}(ys, zs))$$

- proof via equational reasoning by structural induction on xs
  - base case: $xs = \mathsf{Nil}$

$$\begin{aligned} & \mathsf{append}(\mathsf{append}(\mathsf{Nil}, ys), zs) && (1)\\ = {} & \mathsf{append}(ys, zs) && (1)\\ = {} & \mathsf{append}(\mathsf{Nil}, \mathsf{append}(ys, zs)) \end{aligned}$$

Example Proof Continued

- program

$$\mathsf{append}(\mathsf{Nil}, ys) = ys \quad (1)$$
$$\mathsf{append}(\mathsf{Cons}(x, xs), ys) = \mathsf{Cons}(x, \mathsf{append}(xs, ys)) \quad (2)$$

- property: $\mathsf{append}(\mathsf{append}(xs, ys), zs) = \mathsf{append}(xs, \mathsf{append}(ys, zs))$
- proof by structural induction on xs
  - step case: $xs = \mathsf{Cons}(u, us)$
    induction hypothesis: $\mathsf{append}(\mathsf{append}(us, ys), zs) = \mathsf{append}(us, \mathsf{append}(ys, zs))$ (IH)

$$\begin{aligned} & \mathsf{append}(\mathsf{append}(\mathsf{Cons}(u, us), ys), zs) && (2)\\ = {} & \mathsf{append}(\mathsf{Cons}(u, \mathsf{append}(us, ys)), zs) && (2)\\ = {} & \mathsf{Cons}(u, \mathsf{append}(\mathsf{append}(us, ys), zs)) && (\mathrm{IH})\\ = {} & \mathsf{Cons}(u, \mathsf{append}(us, \mathsf{append}(ys, zs))) && (2)\\ = {} & \mathsf{append}(\mathsf{Cons}(u, us), \mathsf{append}(ys, zs)) \end{aligned}$$

Questions

- what is equational reasoning?
- what is structural induction?
- why was that a valid proof?
- how to find such a proof?
- these questions will be answered in this course, but they are not trivial

Equational Reasoning

- idea: extract equations from functional program and use them to derive new equalities
- problems can arise:
  - program

$$\mathsf{f}(x) = 1 + \mathsf{f}(x) \quad (1)$$

  - property: $0 = 1$
  - proof:

$$\begin{aligned} & 0 && (arith)\\ = {} & \mathsf{f}(x) - \mathsf{f}(x) && (1)\\ = {} & (1 + \mathsf{f}(x)) - \mathsf{f}(x) && (arith)\\ = {} & 1 \end{aligned}$$

- observation: blindly converting functional programs into equations is unsound!
- solution requires precise semantics of functional programs

Another Example Proof

- property: algorithm computes the factorial function
- proof using Hoare logic and loop-invariants

$\langle n \geq 0 \rangle$

```
f := 1;
x := 0;
```

$\langle f = x! \wedge x \leq n \rangle$

```
while (x < n) {
  x := x + 1;
  f := f * x;
}
```

$\langle f = n! \rangle$

- questions
  - what statement is actually proven?
  - do you trust this proof? what must be checked?
  - tool support?

Hoare Style Proofs

- problematic proof:

$\langle \textit{True} \rangle$

```
while (0 < 1) {
  x := x + 1;
}
```

$\langle \textit{False} \rangle$

- questions
  - did we prove that True implies False?
  - no, since execution never leaves the while-loop

Soundness = Partial Correctness + Termination

- in both problematic examples the problem was caused by non-terminating programs
- there are several proof-methods that only show partial correctness:
  if the program terminates, then the specified property is satisfied
- for full correctness (soundness), we additionally require a termination proof

Content of Course

- logic for program specifications
- semantics of functional programs
- termination proofs for functional programs
- partial correctness of functional programs
- semantics of imperative programs
- termination proofs for imperative programs
- partial correctness of imperative programs