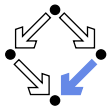
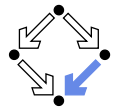
Formal Methods in Software Development

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at



Core Claim

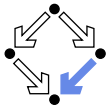


Computer programs/systems are subject to exact reasoning.

> *Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductic reasoning.*
>
> C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", 1969.

A strong claim; not everyone might agree to it (we will rephrase it later).

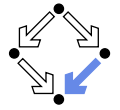
Example



```
static int sum(int[] a)
{
  int n = a.length;
  int s = 0;
  {n = length(a) ∧ s = 0}
  for (int i=0; i<n; i++)
  {
    {n = length(a) ∧ s = ∑_{j=0}^{i-1} a[j] ∧ 0 ≤ i < n}
    s = s+a[i];
  }
  {n = length(a) ∧ s = ∑_{j=0}^{n-1} a[j]}
  return s;
}
```

There are rules to reason why in every possible program run the denoted properties hold at the corresponding program points.

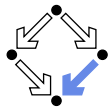
Demonstration



```
class Swap
{
  // swap a[i] and a[j]
  static void swap(int[] a, int i, int j)
  {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
  }

  // swap the first two elements of a
  static void swapFirst(int[] a)
  {
    swap(a, 0, 1);
  }
}
```

Tools may help us to investigate what can go wrong.

Demonstration (Contd)

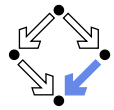


```
class Main
{
 public static void main(String[] args)
  {
    int n = parseInt(args[0]);
    int[] a = new int[n];
    init(a, n);
    print(a, n);
    Swap.swapFirst(a);
    print(a, n);
  }

  static int parseInt(String s)
  {
    return Integer.parseInt(s);
  }
```

```
  static void init(int[] a, int n)
  {
    for (int i=0; i<n; i++)
      a[i] = i;
  }

  static void print(int[] a, int n)
  {
    for (int i=0; i<n; i++)
    {
      System.out.print(a[i]);
      System.out.print(' ');
    }
    System.out.println("");
  }
}
```

Methods need to be specified (and correspondingly corrected).

Aspects

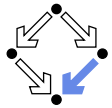


This reasoning has various aspects:

- Specification. State the properties that a program shall have.
- Verification. Argue why a particular program satisfies a given specification (in every execution).
- Falsification. Detect that a program violates a specification (in some execution).
- Transformation. Transform a program such that it preserves its behavior (but e.g. improves its execution time).
- Derivation. Construct a program in such a way that it is guaranteed to satisfy a given specification.

This course deals with specification and verification/falsification.

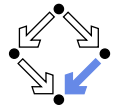
Properties of Reasoning



- Principal advantage: clarity and generality.
  - The subjects of discussion are well defined.
  - There are clear rules for the correctness of the arguments.
  - The arguments may apply to infinitely many situations.
  - Example: $\{x \geq 0\}$`x = x+1`$\{x > 0\}$
    (for all x, to show that x is greater than zero after the execution of the assignment statement, it suffices to show that x is greater than or equal to zero before).
- Principal disadvantage: abstraction.
  - We reason about models of the real world.
  - Questionable whether the model adequately describes the real world.
  - Example: the machine code generated by the compiler for `x = x+1` may be wrong, or the processor may have an error in the implementation of the $+$ operation, or ...

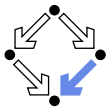
All reasoning is relative to the assumptions of the model.

Alternatives to Reasoning



- Inspection. Let multiple people look at program and discuss it.
  - Actually, this *is* reasoning (but without a formal basis).
  - Advantages: reduces programmer's "system blindness".
  - Disadvantage: without a formal basis, it is unclear what rules guide the discussion.
- Testing. Run the program with sample inputs and observe the external effects (see what happens).
  - Advantage: the program is shown to work in certain situations.
  - Disadvantage: you never know what happens with other inputs.
  - In concurrent programs, you even do not now whether the same behavior is always exhibited with the same input.
- Simulation/Visualization. Similar to testing, but also observe the internal behavior of the program.
  - Advantage: consideration not limited to external effects.
  - Disadvantages: same as for testing.

The Power and Limits of Reasoning



Assume a correct proof that program P satisfies specification S.

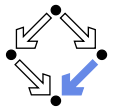
- We have shown
  - that every model execution of P satisfies S,
  - which is also true for all real executions of P if the model is adequate.
- We have not shown
  - that any real execution of P satisfies S,

    Beware of bugs in the above code; I have only proved it correct, not tried it. Donald E. Knuth, 1977.
  - that S captures your expectation of a real world behavior.
- On the other hand, with a successful test run
  - we demonstrate that some real execution of P satisfies an expectation,
  - we cannot show that this is true for all real executions of P.

    Program testing can be used to show the presence of bugs, but never to show their absence! E.W.Dijkstra, 1972.

Reasoning and testing have both their place.

Core Claim Rephrased

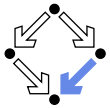


True interpretation of Hoare's statement:

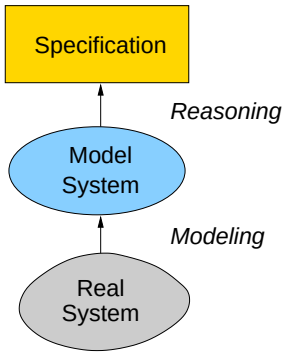
> Computer programming is an exact science in that all the properties of a model program and all the consequences of executing it in any given model environment can, in principle, be found out from the text of the program itself by means of purely deductic reasoning.

Still a strong claim, but now everyone should be able to agree to it.

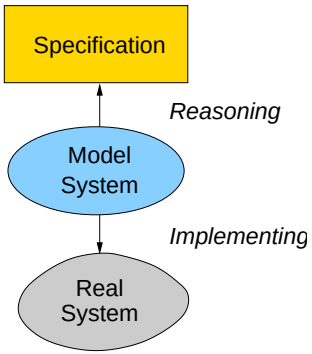
The Role of Reasoning



Often: Post Factum

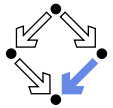


Better: Pre Factum



Write a specification, design a model system, reason about its correctness, then implement the design, then test the implementation.

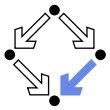
The Role of Reasoning



- So the main role of reasoning is on program/system designs:
  - Design a principal solution to a given problem.
  - Argue about the correctness of the design.
  - Thus find design errors; repeat process until you are satisfied.
- The main role of testing is on program/system implementations:
  - Map the design to real computers using real programming languages.
  - Run the implementation on sample inputs and check its behavior.
  - Thus find implementation errors; repeat process until you are satisfied.

This is a big difference from "proving that a program is correct".

The Role of Reasoning

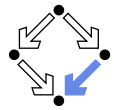


Apparently, there are various approaches/tools that *seem* to apply reasoning to real systems ...

- Typical goal: prevent runtime errors.
  - Division by zero, array index out of bounds, null pointer dereferences, memory corruptions (buffer overflows).
  - Tool that takes program as input and detects program errors.
- Actually: operate on automatically constructed models.
  - Focus is on falsification (finding errors).
  - Investigate whether "bad" states might occur in model.
  - Detected/assumed error in model may imply error in real system.
- Application to mission-critical/important pieces of software.
  - Airplane control software.
  - MS Windows device driver.

We always reason about models.

The Origins: Sequential Programs

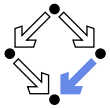


A personal selection.

- Robert W. Floyd, 1967: Assigning Meaning to Programs.
- C.A.R. Hoare, 1969: An Axiomatic Basis for Computer Programming.
- C.A.R. Hoare, 1972: Proof of Correctness of Data Representations.
- Edsger W. Dijkstra, 1975: Guarded Commands, Nondeterminacy and Formal Derivation of Programs.
- Edsger W. Dijkstra, 1976: A Discipline of Programming.
- Luckham et al, 1979: Stanford PASCAL Verifier - User Manual.
- David Gries, 1981: The Science of Programming.

Axiomatic program semantics, program verification.

The Origins: Concurrent Programs

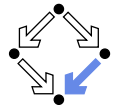


A personal selection.

- Dijkstra, 1968: Cooperating Sequential Processes.
- Ashcorft and Manna, 1971: Formalization of Properties of Parallel Programs.
- Owicki, Gries, 1976: An Axiomatic Proof Technique for Parallel Programs.
- C.A.R. Hoare, 1978: Communicating Sequential Processes.
- Armin Pnueli, 1979: The Temporal Logic of Programs.
- Leslie Lamport, 1980: The "Hoare Logic" of Concurrent Programs.
- Robin Milner, 1982: A Calculus of Communicating Systems.

Axiomatic semantics, process calculus/algebra, temporal logic.

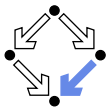
Later History



- In the 1980s, program verification was considered "dead" by many.
  - Little automation, much manual effort by programmer.
  - Handling of toy programs only.
- In the 1990s, model checkers became successful.
  - Fully automatically check (finite state) models.
  - Verification of hardware and of communication protocols.
  - Systems with state spaces of size 10^{100} and beyond.
- Since the late 1990s, interest in software verification revived.
  - Model checking/proving of critical pieces of software.
  - Focus often more on falsification than on verification.
- New applications such as proof-carrying-code (PCC).
  - Ship machine code together with proof that code satisfies certain safety/security properties.
- New issues such as security considered.

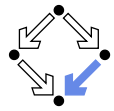
Formal methods are a *hot* topic today (also in industry).

Languages and Systems



- Specification Languages
  - General: VDM (Vienna Development Method), Z, ASM (Abstract State Machines), OCL (Object Constraint Language for UML), ...
  - Bound to programming language: SPARK (Ada), Larch/C++, JML (Java), Spec# (C#), ACSL (C), ...
  - Algebraic/axiomatic: OBJ, ACT One, Larch, CASL, ...
  - Concurrent: Unity, Estelle, Lotos, TLA, ...
  - Mobile: pi-Calculus, ...
- Model Checkers
  - Spin, SMV, BLAST, Bandera, SLAM, VeriSoft, ...
- Proving Assistants
  - PVS, HOL, Isabelle, Coq, Theorema, ...
- Verification Environments
  - Edinburgh Concurrency Workbench, STeP (Stanford Temporal Prover), KIV (Karlsruhe Interactive Verifier), JIVE (Java Interactive Verification Environment), LOOP, Krakatoa/Why, KeY, Mobius, ...

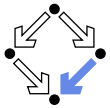
Course Outline



- Specifying and verifying sequential programs.
  - Predicate logic, computer-assisted reasoning (*RISC ProofNavigator*).
  - Hoare calculus, predicate transformers, state relations.
  - Computer-supported program verification (*RISC ProgramExplorer*).
- Specifying and verifying Java programs.
  - Java modeling language, runtime assertions (*JML toolset*, *JML4c*).
  - Extended static checking of Java programs (*ESC/Java2*).
  - Verifying Java programs (*KeY*).
- Specifying and verifying concurrent systems.
  - State graphs for synchronous and asynchronous systems.
  - Specifying properties of concurrent systems in temporal logic.
  - Verifying and model-checking concurrent systems (*Spin*).
- Three lectures mandatory only for KV4 (recommended for KV3).
  - Mykola Nikitchenko (Taras Shevchenko National University of Kyiv): *Compositional Approach for Program Specification and Verification.*

12 lectures of which 3 are mandatory only for KV4.

Grading System



Combined lecture (KV3 for software engineering, KV4 for computer mathematics): exercises and final exam.

- 10 exercises.
  - 8 exercises from Schreiner, 2 from Nikitchenko.
  - KV3: best 7 are used for grading (350 points needed).
  - KV4: best 8 are used for grading (400 points needed).
    - At least one exercise from Nikitchenko.
- Final exam.
  - KV3: 4 questions.
  - KV4: 5 questions.

Each part must be positive; each accounts for 50% of the overall grade.