Introduction
A Truism

Software has become critical to modern life.

- Process Control (oil, gas, water, . . .)
- Transportation (air traffic control, . . .)
- Health Care (patient monitoring, device control . . .)
- Finance (automatic trading, bank security . . .)
- Defense (intelligence, weapons control, . . .)
- Manufacturing (precision milling, assembly, . . .)

Failing software costs money and life!
Failing Software Costs Money

- Thousands of dollars for each minute of factory down-time
- Huge losses of monetary and intellectual investment
  - Rocket boost failure (e.g., Arianne 5)
- Business failures associated with buggy software
  - (e.g., Ashton-Tate dBase)
Failing Software Costs Lives

Potential problems are obvious:

- Software used to control nuclear power plants
- Air-traffic control systems
- Spacecraft launch vehicle control
- Embedded software in cars

A well-known and tragic example:
Therac-25 radiation machine failures
The Peculiarity of Software Systems

Tiny faults can have catastrophic consequences

Software seems particularly prone to faults:

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

Building software is what the majority of you will do after graduation

- You’ll be developing systems in the context we just mentioned
- Given the increasing importance of software,
  - you may be liable for errors
  - your job may depend on your ability to produce reliable systems

What are the challenges in building reliable software?
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 airplane 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 physical or modal **separation** of subsystems
  Local failures often affect whole system
- Software designs have very high logic **complexity**
- Most SW engineers **untrained** in correctness
- Cost efficiency more important than reliability
- Design practice for reliable software in **immature** state
How to Ensure Software Correctness/Compliance?

A Central Strategy: **Testing**
(others: SW processes, reviews, libraries, . . . )

**Testing against inherent SW errors ("bugs")**

- Design test configurations that hopefully are representative and
- ensure that the system behaves intentionally on them

**Testing against external faults**

- Inject faults (memory, communication) by simulation or radiation
Limitations of Testing

- Testing can show the presence of errors, but 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
A Complement to Testing: Formal Verification

A Sorting Program:

```
public static Integer[] sort(Integer[] a) {
    ...
}
```
A Complement to Testing: Formal Verification

A Sorting Program:

```
public static Integer[] sort(Integer[] a) {
    ...
}
```

Testing `sort()`:

- `sort({3, 2, 5}) == {2, 3, 5}`  ✓
- `sort({}) == {}`  ✓
- `sort({17}) == {17}`  ✓
A Complement to Testing: Formal Verification

A Sorting Program:

```java
public static Integer[] sort(Integer[] a) {
    ...
}
```

Testing `sort()`:

- `sort({3, 2, 5}) == {2, 3, 5}  √`
- `sort({}) == {}  √`
- `sort({17}) == {17}  √`

Missed Test Cases!

- `sort({2, 1, 2}) == {1, 2, 2}  ✗`
- `sort(NULL) == {1, 2, 2}  ✗`
Theorem. The program sort() is correct. For any given integer array a, calling the program sort(a) returns an integer array that is sorted \( \text{and is a permutation of } a. \)

However, methodology differs from Mathematics:
1. Formalize the claim in a logical representation
2. Prove the claim with the help of a theorem prover
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 specification
- System implementation

Make formal model of both and use tools to prove **mechanically** that **formal execution model** of the implementation satisfies **formal requirements** of the specification
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 formal system model
- Should ideally be automatic
Formal Methods and Testing

- Run the system at chosen inputs and observe its behaviour
  - Randomly chosen
  - Intelligently chosen (by hand: expensive!)
  - Automatically chosen (need formalized spec)
- What about other inputs? (test coverage)
- What about the observation? (test oracle)

Challenges can be addressed by/require formal methods
Specifications — What the System Should Do

- Simple properties
  - Safety properties
    - Something bad will never happen
  - Liveness properties
    - Something good will happen eventually
  - Non-functional properties
    - Runtime, memory, usability, . . .

- “Complete” behaviour specification
  - Equivalence check
  - Refinement
  - Data consistency
  - . . .
**Formal Specifications**

The expression in some formal language and at some level of abstraction of a collection of properties that some system should satisfy [van Lamsweerde]

- **Formal language**
  - Syntax can be mechanically processed and checked

- **Abstraction**:
  - Above the level of source code
  - Several levels possible
Formal Specifications

The expression in some formal language and at some level of abstraction of a collection of properties that some system should satisfy [van Lamsweerde]

- Properties:
  - Expressed in some formal logic
  - Have a well-defined semantics

- Satisfaction:
  - Ideally (but not usually) decided mechanically
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 source code or, at most, bytecode level
  - Non-formalizable properties

- To replace good design practices

There is no silver bullet!

No correct system w/o clear requirements & good design

This holds as well for Formal Methods
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)
Successful Formal Methods

- are integrated into the development process, in particular at early design stages
- avoid unreasonable new demands or skills from the user
  FM should be learnable as part of Masters in CS
- work at large scale
- save time or money in getting a good quality product out
- increase the feasible complexity of products
Typical Areas

- Saving time
  - Time to market
Typical Areas

- **Saving time**
  - Time to market

- **Saving money**
  - Intel Pentium bug
  - Smart cards in banking
Typical Areas

- Saving time
  - Time to market

- Saving money
  - Intel Pentium bug
  - Smart cards in banking

- More complex products
  - Modern processors, fault tolerant software
Typical Areas

- Saving time
  - Time to market

- Saving money
  - Intel Pentium bug
  - Smart cards in banking

- More complex products
  - Modern processors, fault tolerant software

- Saving human lives
  - Avionics, X-by-wire
Formalisation of system requirements is hard
Difficulties in Creating Formal Models

- Real World
- Abstraction
- Formal Execution Model
- Formal Requirements Specification
Difficulties in Creating Formal Models

Real World

Formal Model

wrong assumption
eg, zero delay
Difficulties in Creating Formal Models

Real World → Formal Model

missing requirement
eg, stack overflow

Difficulties in Creating Formal Models

Real World

Formal Model

misunderstood problem
eg, wrong integer model

Formalization Helps to Find Bugs in Specs

- Wellformedness and consistency of formal specs checkable with tools
- Fixed signature (symbols) helps to spot incomplete specs
- Failed verification of implementation against spec gives feedback on erroneous formalization
Another Fundamental Fact

Proving properties of systems can be hard
Level of System (Implementation) Description

- **Low level**
  - Finitely many states
  - Tedious to program, worse to maintain
  - Automatic proofs are (in principle) possible

- **High level**
  - Complex datatypes and control structures, general programs
  - Easier to program
  - Automatic proofs (in general) impossible!
Expressiveness of Specification

- **Simple**
  - Finitely many cases
  - Approximation, low precision
  - Automatic proofs are (in principle) possible

- **Complex**
  - General properties
  - High precision, tight modeling
  - Automatic proofs (in general) impossible!