INTRODUCTION

Formal Methods

BCS2133

Semester 1 Session 2015/16

Software has become critical in modern life

- Finance (e-trading, e-banking,…)
- Health care (patient monitoring, medical devices control,…)
- Transportation (tickets reservation, air traffic control, ….)
- Manufacturing (robots for car assembling,…)
- Process control (oil, gas, water, ...)
- Defense (weapons control, drones, …)

Software systems are very large

- Millions of Lines of Code (LOCs) in aircraft control software
- Even for cars, e.g. GM Chevrolet Volt contains ~10M LOCs
  - Current cars have hundreds of onboard functions: how we can verify their combination?

    E.g.: does braking when changing the radio station and starting the windscreen wiper, not affect air conditioning?

Failing software costs money

- Thousands of dollars for each minute of factory down-time.
- Huge losses of intellectual investments
  - Rocket boost failure – Arianne 5 (due to reusing soft from Arianne 4 and not taking into account specifics of Arianne 5 rocket)
- Business failures associated with buggy software
  - E.g.: Ashton-tate dBase.

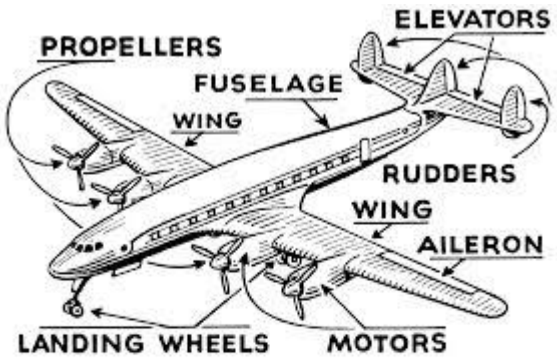
Failing software costs lives

- Potential source of problems – embedded systems:
  - Air-traffic control systems
  - Embedded software in cars
  - Space craft control systems
  - Software used to control nuclear power plants etc.

- A well known and tragic example because of software failure – Therac 25 machine failures (radiation therapy machine, due to error in program patients were given massive overdose of radiation)

Embedded Software

- Most of Embedded systems are safety critical



- Failing software costs not only money, but life!

The peculiarity of software systems

Tiny faults can have catastrophic consequences:

- Arianne 5
- Therac 25
- Denver Airport Luggage Handling System
- Mars Climate Orbiter, Mars Sojourner
- London Ambulance Dispatch System
- Pentium Bug etc.

Motivation

- Building software is what most of you will do after graduation
- You'll develop systems in the context we just mentioned
- Given the increasing importance of software
  - Everybody are liable to errors
  - Your success in job will depend on your ability to produce reliable systems

- How to develop *reliable* software?

Achieving Reliability in Engineering

- Some well-known strategies
  - Precise calculations/estimations of forces, stress, etc.
  - Hardware redundancy (“make it bit stronger than necessary”)
  - Robust design (any airplane flies with dozens of known but minor defects)
  - Clear separation of subsystems (single fault is not catastrophic for whole system)
  - Follows design patterns that are proven to work.

Why This Does Not Work For Software?

- Software systems are discrete. A single bit-flip may change behavior completely.
- Redundancy as replication doesn’t help against bugs. (Redundant SW only viable in extreme cases).
- There is no physical separation of SW subsystems. Local failures often affect whole system.
- Software designs have high logical complexity.
- Design practice for reliable software is not yet mature.
- Most SW engineers untrained in checking correctness.

How to Ensure Software Correctness

- A central strategy : Testing
- Testing against *inherent* SW errors (“bugs”)
  - Development of unit tests
  - Ensure that a system behaves as intended on them
- Testing against *external* faults
  - Inject faults (memory, CPU, communication) e.g. by simulation or radiation.
- Other strategies : strict following processes of SW design, peer reviews, reusing existing templates and libraries etc.

Testing: Static vs Dynamic Analysis

- Static analysis of code → does not require execution of code
  - Lexical analysis of the program syntax, checks the structure and usage of individual statements
  - Often automated and is a first sage of compilation
- Dynamic analysis of code → at run of software system
  - Program runs under controlled conditions
  - Typical scenario is branches testing

Limitation of Testing

- Testing can *show the presence of errors*, *but not their absence*.
- Exhaustive testing is viable only for trivial systems.
- Representativeness of test cases/injected faults is subjective. How to test for the unexpected?
- Testing is labor intensive, hence expensive.

Complementary Testing : Formal Verification

A Sorting Program:

```
int* sort (int* a) {
 ...........
  }
```

Example of Testing

A Sorting Program:

```
int* sort (int* a) {
 ...........
 }
```

Testing sort() ;

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

Example of Testing

A Sorting Program:

```
int* sort (int* a) {
 …………
 }
```

Testing sort() ;

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

Missing test cases!

- sort ({2,1,2})=={1,2,2}
- sort (NULL) == exception

Formal Verification as Theorem Proving

Theorem : The program $\mathrm{sort}\,()$ is correct;

For any given non-null integer array $\boldsymbol{a}$, calling the program $\mathrm{sort}\,(\boldsymbol{a})$ returns an integer array that is sorted in ascending order and is a permutation of $\boldsymbol{a}$.

Methodology:

- Formalize this claim in a logical representation
- Prove this claim with the help of an automated reasoner.

Definition – Formal Methods

- In computer science, formal methods refers to mathematically based techniques for the specification, development and verification of computational (software and hardware) systems.

What Are Formal Methods?

- Specification language + formal reasoning technique
- The technique is supported by
  - Mathematical notation (Z, TLA, UPPAAL)
  - Reasoning tools and checkers (Z-Eves, TLC, UPPAAL tool)

Formal Methods

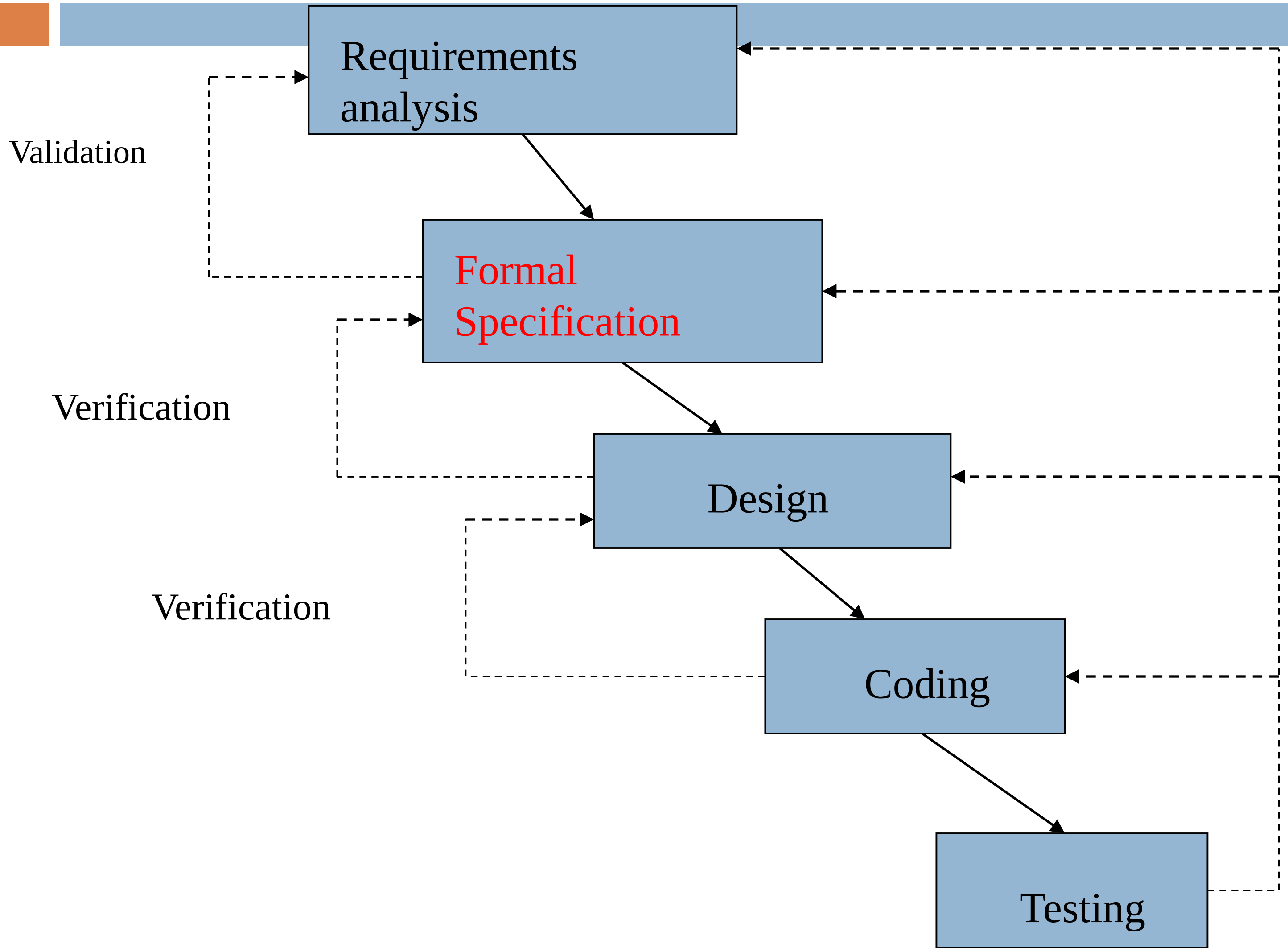
- Based on discrete math (mostly, logic and set theory)
- Allow to increase confidence in the correctness/robustness/security of the system
- Consider two main artifacts:
  - System requirements (1)
  - System implementation (2)
- Are based on
  - A formal specification of (1)
  - A formal execution model of (2)
- Use tools to verify that (2) satisfy (1)

Formal Methods : The Objectives

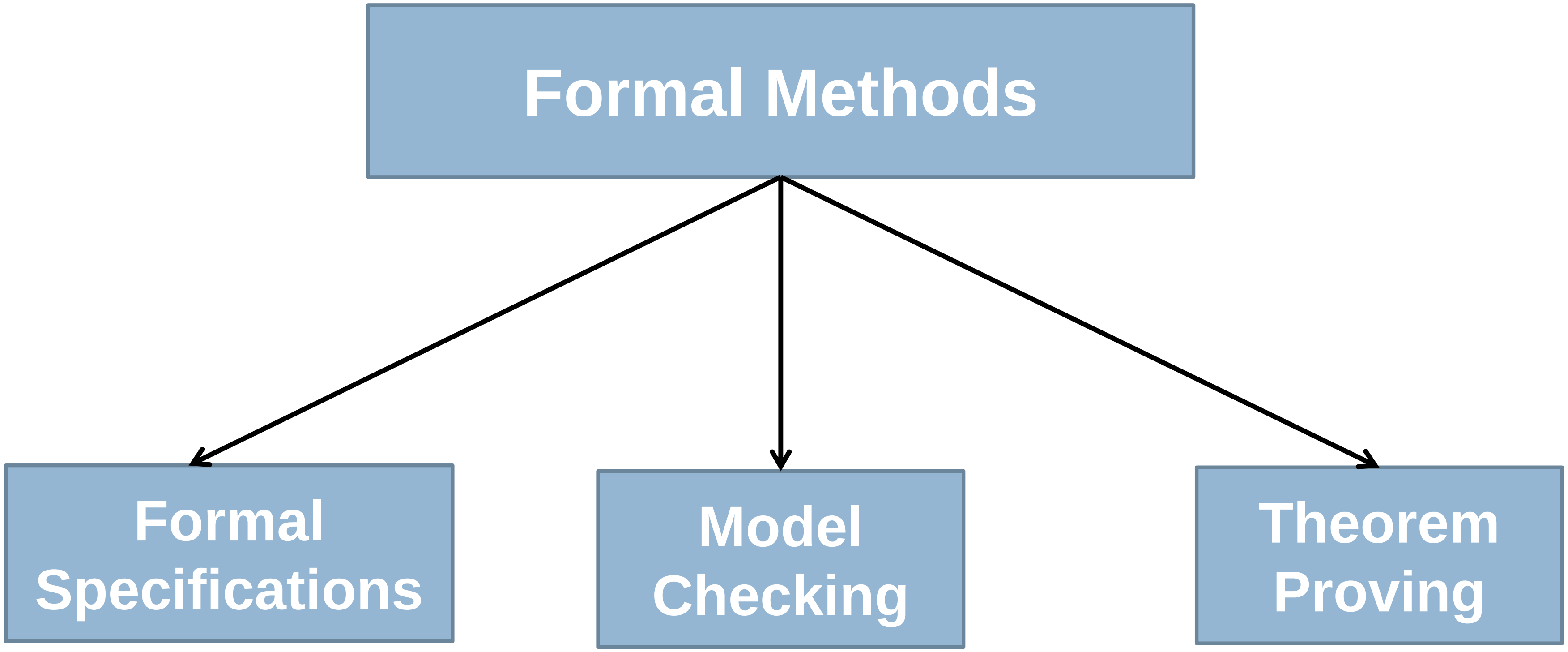
- Requirements specification
  - clarify customer’s requirements
  - reveal ambiguity, inconsistency, incompleteness
- Software design
  - decomposition
    - specification of components structural relationships
    - specification of components behavior
  - refinement

Formal Methods : The Objectives

- Validation
  - Are we building the right system?
  - Check correspondence of specifications to requirements; Use specification to determine test cases for testing and debugging
- Verification
  - Are we building the system right?
  - Proving that a specific realization satisfies a system specifications
- Documentation
  - Communication among stakeholders for better understanding


Requirements analysis
Validation
Formal Specification
Verification
Design
Verification
Coding
Testing

Formal Methods: The Concepts



Program Specification

- A **program specification** is the definition of what a computer program is expected to do

Types of Specifications

- Informal
  - Free form, natural language
  - Ambiguity and lack of organization lead to incompleteness, inconsistency and misunderstandings.
- Formatted
  - Standard syntax
  - Basic consistency and completeness checks
  - Imprecise semantics implies errors

Types of Specifications

- Formal
  - Translate non-mathematical description (usually, English text, diagrams, tables) into formal language.
  - Syntax and semantics rigorously defined. Precise mathematical form, allowing eliminate ambiguity.
  - Precise description of behavior and properties of a system.
  - Strict semantics of a language support formal deduction.
  - Provide basis for verifying equivalence between specification and implementation.
  - Hard to read and understand without special training.

An example of *informal* specification:

“A software system for an Automated Teller Machine (ATM) needs to provide services on various accounts. The services include operations on current account, operations on savings account, transferring money between accounts, managing foreign currency account, and change password. The operations on a current or savings account include deposit, withdraw, show balance, and print out transaction records.”

A better (formatted) way to write the same specification

“A software system for an automated teller machine (ATM) needs to provide services on various accounts.

The services include

① operations on current account
② operations on savings account
③ transferring money between accounts
④ managing foreign currency account,
⑤ change password.

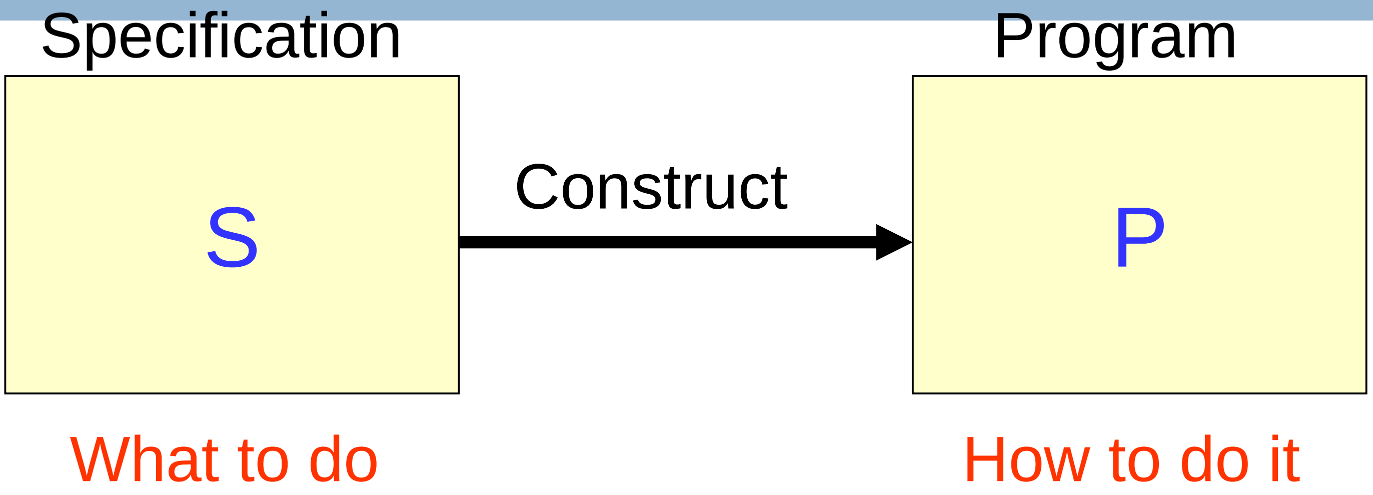
The operations on a current or savings account include

① deposit
② withdraw
③ show balance
④ print out transaction records.”

The major problems with informal specifications:

- Informal specifications are likely to be ambiguous, which causes misinterpretations.
- Informal specifications are difficult to be used for inspection and testing of programs because of the big gap between the functional descriptions in the specifications and the program structures.
- Informal specifications are difficult to be analyzed for their consistency and validity.
- Information specifications are difficult to be supported by software tools in their analysis, transformation, and management (e.g., search, change, reuse).

Problems in software development



- How to ensure that S is not ambiguous so that it can be correctly understood by all the people involved?
- How can S be effectively used for inspecting and testing P?
- How can software tools effectively support the analysis of S, transformation from S to P, and verification of P against S?

A solution to these problems are:

Formal Methods

Formal methods allow to achieve desirable properties of formal specifications

- Unambiguous
  - exactly one (set of) properties satisfies it
- Consistency
  - No contradictions between requirements
- Completeness
  - all aspects of a system are specified
- Inference
  - Can be used to prove properties of a system

Model Checking

- Use Finite State Machine (FSM) model of a system in a one of notations (e.g. TLA, UPPAAL)
- Model checker determines if a model satisfies specifications expressed as formulas in a given logic
- Basic method is to explore all reachable paths in a tree of states of the model

Theorem Proving

- **Automated theorem proving** is proving of mathematical theorems by a computer program.

Benefits of Formal Specification in Software Development

- Formal specifications put the software development process on the well-defined basis of computer science
- Formal specifications are expressed in languages with formally defined syntax and semantics
  - hierarchical decomposition
  - mathematical foundation
  - graphical presentation
  - (it also can be accompanied by informal description)

Benefits of Formal Specifications

- Higher level of abstractions enables a better understanding of the problem
- Defects are covered that would likely go unnoticed with traditional specification methods
- Identify defects earlier in life cycle

Benefits of Formal Specifications

- Formal specification enable formal proofs which can establish fundamental system properties and invariants
- Repeatable analysis means reasoning and conclusions can be checked by colleagues
- Encourages an abstract view of systems– focusing on *what* a proposed system should accomplish as opposed on *how* to accomplish it
- Abstract formal view helps separate specification from design

Limitation to Formal Methods

- Used as an *addition* to, *not a replacement* for standard quality assurance methods (e.g. testing);
- Formal methods are not a universal solution, but can increase confidence in a product’s reliability if applied with care and skill;
- Very useful for consistency checks, but can not assure completeness of a specification

Why Use Formal Methods

- Potential to improve both quality and productivity in software development.
- Becoming best and required practice for developing safety-critical and mission-critical software systems.
- To ensure that systems meet regulations and standards.
- To avoid legal liability repercussions.

Myths and limitations of Formal Methods

- Able to guarantee that software is prefect

- it only allows to validate a model of software

- Increase the cost of development

- So mostly used for safety-critical systems

- Application is only needed if benefits are to exceed costs.

- Are not used on large-scale software
- Require highly trained mathematicians

Conclusion

- FM can detect defects earlier in life cycle
- FM can be applied at various levels of software systems developments
- FM can be integrated with existing process models of software systems development
- FM can improve quality assurance when applied judiciously to appropriate projects
- FM are no universal solution

The question of our interest is:

How to write a formal specification?

There are many formal notations have been developed for writing formal specifications, e.g. Z, TLA, UPPAAL.

Next Lecture – Z notation

- A formal specification technique developed at Oxford
- Uses mathematical notation to provide exact definitions of a system
- System is described in a number of small Z modules, called schemas

Please ask your questions!