INTRODUCTION

Formal Methods

BCS2133

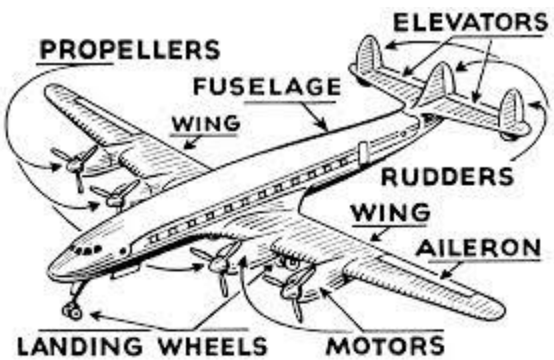
Semester 1 Session 2014/2015

Software has become critical in 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,…)

Embedded Software

- Most of Embedded systems are safety critical







- Failing software costs money and life!

Software systems are very large

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

    E.g.: does braking when changing the radio station and starting the windscreen wiper, will affect air conditioning?

Failing software costs money

- Thousands of dollars for each minute of factory down-time.
- Huge losses of monetary and intellectual investment
  - 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:
  - Air-traffic control systems
  - Embedded software in cars
  - Space craft vehicle control
  - Software used to control nuclear power plants

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

The peculiarity of software systems

Tiny faults can have catastrophic consequences:

- Arianne 5
- Therac 25
- Mars Climate Orbiter, Mars Sojourner
- London Ambulance Dispatch System
- Denver Airport Luggage Handling 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 from engineering:
  - Precise calculations/estimations of forces, stress, etc.
  - Hardware redundancy(“make it bit stronger than necessary”)
  - Robust design (single fault is not catastrophic)
  - Clear separation of subsystems (any airplane flies with dozens of known but minor defects)
  - Follows design patterns that are proven to work.

Why This Does Not Work For Software?

- Software systems are discrete. 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

How to Ensure Software Correctness

- A central strategy : Testing
- Others : flowing Process of SW design, peer reviews, using existing templates and libraries
- 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) by simulation or radiation.

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 first sage of compilation
- Dynamic analysis of code → at run of software system
  - Program run formally under controlled conditions
  - Branch 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:

$$\textbf{int*}\ \mathrm{sort}\ (\textbf{int*}\ \mathrm{a})\ \{$$

…………

$$\}$$

Complementary Testing : Formal Verification

A Sorting Program:

$\textbf{int*}$ sort ($\textbf{int*}$ a) {

....

}

Testing sort() ;

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

Complementary Testing : Formal Verification

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 a, calling the program $\mathrm{sort}\ (\mathrm{a})$ returns an integer array that is sorted wrt ≤ and is a permutation of a.

Methodology:

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

What Are Formal Methods?

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

Formal Methods

- Rigorous design and development methods for computational (hardware/software) systems
- 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)

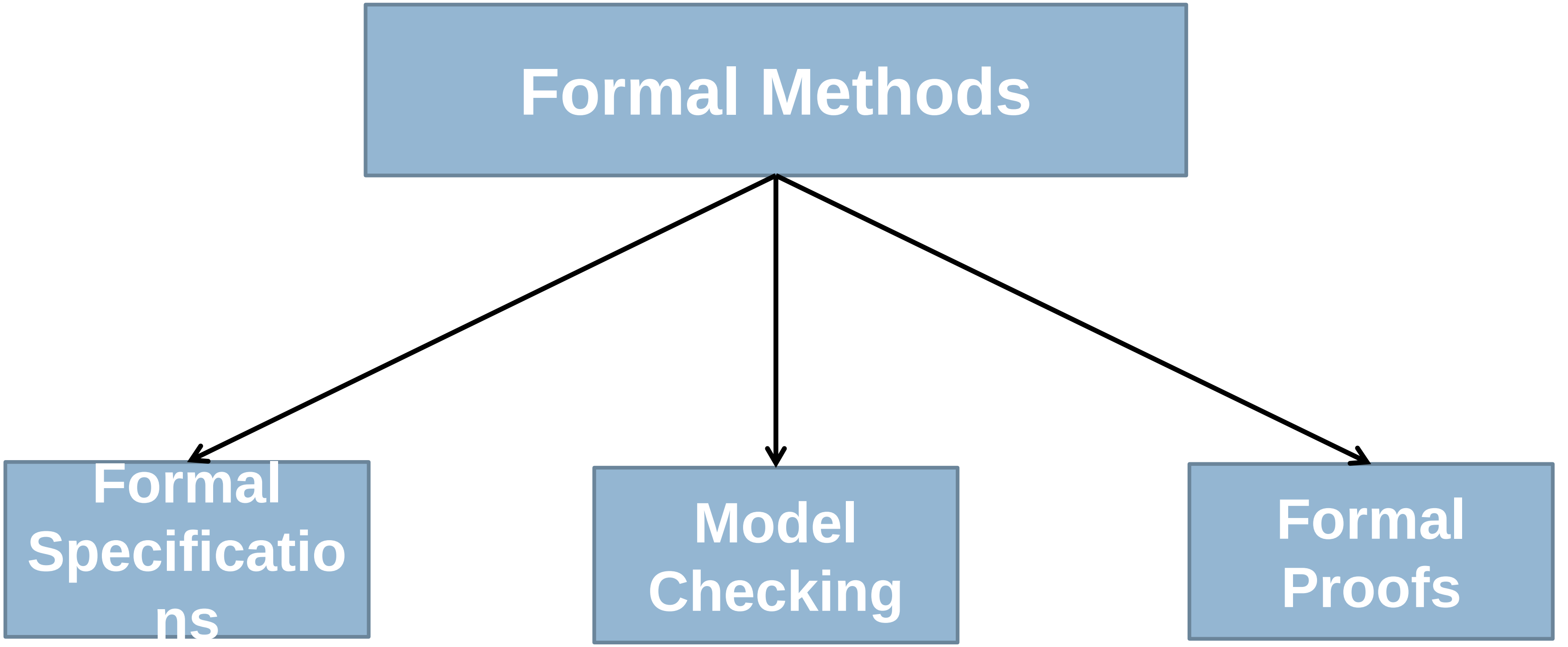
Formal Methods : The Objectives

- 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

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

Formal Methods: The Concepts



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
  - Syntax and semantics rigorously defined. Precise mathematical form, allowing eliminate imprecision and ambiguity
  - Translate non-mathematical description (usually, English text, diagrams, tables) into formal specification language.
  - 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

Formal Proofs

- Complete and convincing proving validity of some property of the system
- Constructed as a series of steps, each of which is justified by a small set of rules
- Eliminates ambiguity and subjectivity inherent when drawing informal conclusions
- May be manual but usually constructed with automated assistance

Model Checking

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

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

Benefits of Formal Specification in Software Development

- Formal specifications ground the software development process in the well-defined basis of computer science
- Orientation goes from customer to developer
- Formal specifications are expressed in languages with formally defined syntax and semantics
  - hierarchical decomposition
  - mathematical foundation
  - graphical presentation
  - It 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
- Formal methods are not a panacea, 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

- Are not used on large-scale software
- Require highly trained mathematicians
- Application is needed if benefits are to exceed costs.
- Formal Methods and problem domain expertise must be integrated to achieve positive results.

Conclusion

- FM are no panacea
- 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

Please ask your questions!