

E0:272 Formal Methods in Software Engineering

*3:1, January - April 2022
M-W 2:00 - 3:30*

<http://www.csa.iisc.ac.in/~deepakd/fmse2022>

Deepak D'Souza

K. V. Raghavan

CSA, IISc

05 January 2022

Motivation

- Software is increasingly used in a wide range of domains: business, personal, scientific, embedded control.
- Therefore, software development ought to
 - be efficient and predictable
 - result in high quality (i.e., correct and reliable) software
- The way to achieve this is to use automated analysis **tools**
- Methodology of the course: Hands-on study of a series of advanced tools
- Knowledge of such tools gives
 - Exposure to practical uses of various analysis techniques
 - Generates research ideas for developing better tools
 - Prepares one for career in software-development industry

Software development is hard

Average software-development project [Barry Boehm, ICSE '06 keynote] incurs:

- 90% cost overrun
- 121% time overrun
- delivers only 61% of initially promised functionality

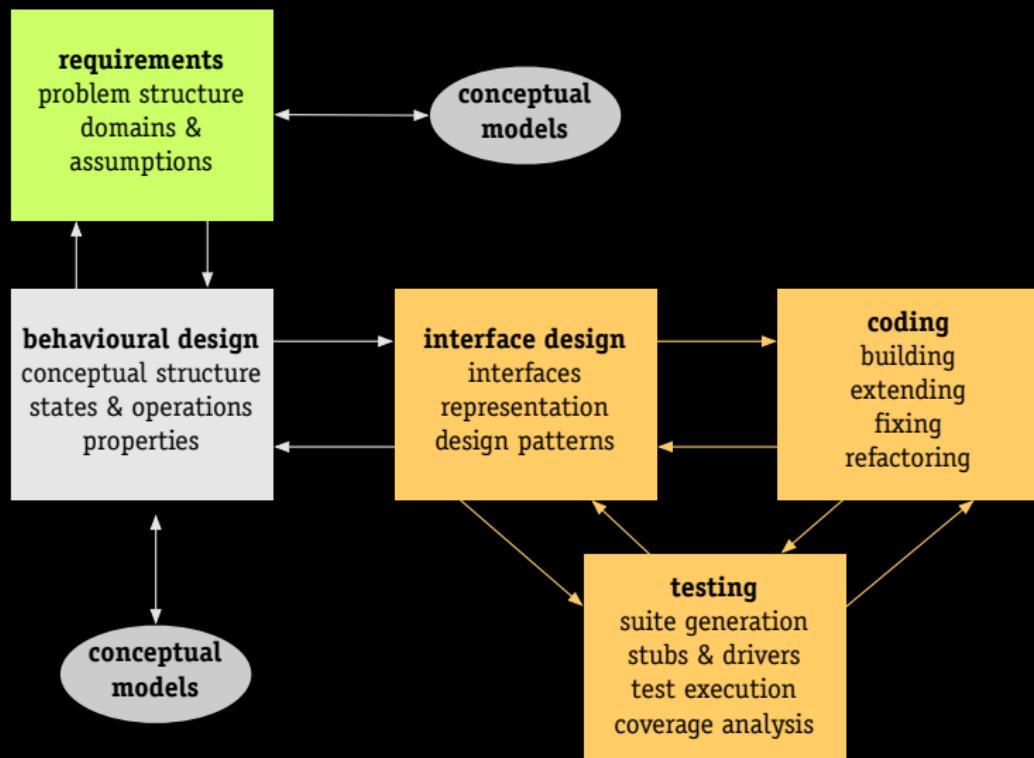
Software defects

- Most large software is buggy
- They cause user dissatisfaction, and sometimes catastrophe (e.g., Ariane 5 rocket explosion)
- Finding and fixing bugs consumes 50% of the total effort in software development!

Causes of software defects

- User's **requirements** not specified properly.
- Software does not meet user's requirements.
- **Implementation** errors
 - Low-level errors, such as null-pointer dereference, array out of bounds
 - Different components of the software (or software and libraries) evolve separately, and become inconsistent.

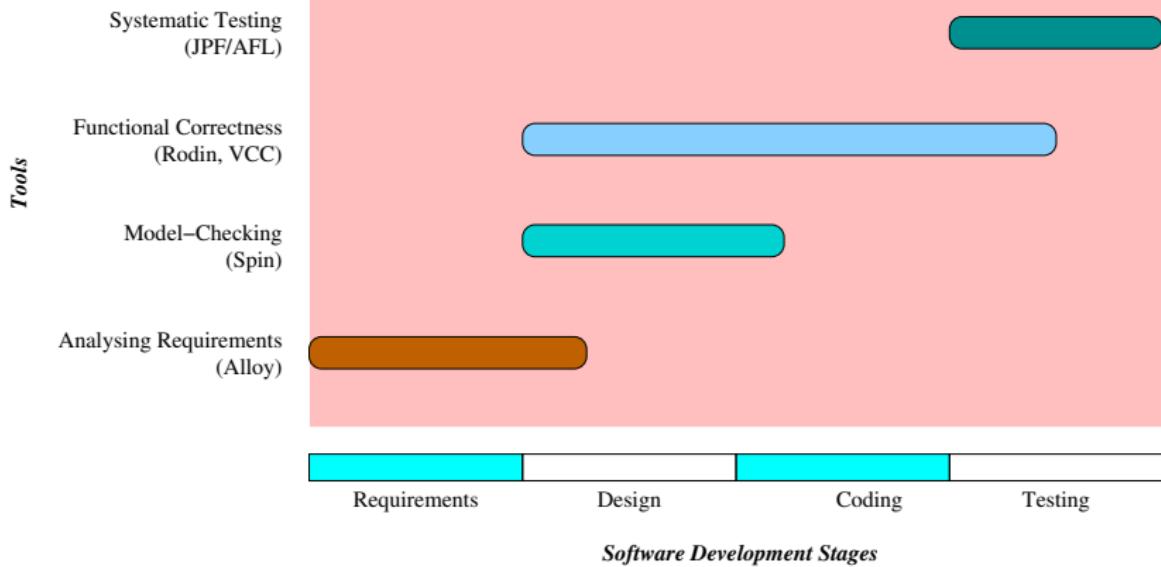
two kinds of design



The solution to the problem: tools

- Tools are available for all stages of software development
- Benefits of tools
 - Tackle complexity by providing abstract views of software
 - Identify errors by exhaustive analysis
 - Provide reliable way to make changes
 - Make software development more like engineering, and less of an art
- Our focus is mostly on
 - Formal tools, that provide definitive guarantees, and
 - Involve non-trivial capability for analysis or transformation.
 - We will cover only a small selection of the tools available!

Methods and tools covered in this course



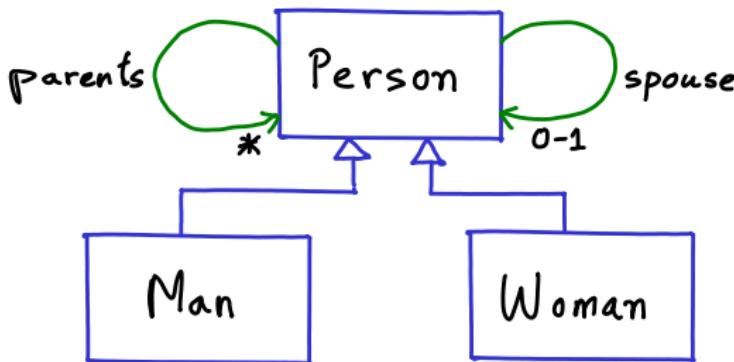
Tools for conceptual modeling and design

- **Alloy** (Conceptual modeling)
 - Formalize key **entities** in the domain, their **relationships** between them, and the **operations** that can be performed on them.
 - Helps **identify inconsistencies and incompleteness** in requirements.
- **Rodin** (Modeling/design)
 - Step-wise refinement of a conceptual model, with guarantee of preservation of all properties.
 - Ultimately: refine to code level.
- **Spin** (Modeling/design)
 - Specify **states** in the system, and the **transitions** between them.
 - Identify the properties of the sequences of states that can arise.

Overview of Alloy

- Formal modeling of entities and relationships, using sets and relations
- Modeling of invariants/constraints on the entities
- Analyzing consistency of the model, and identifying errors

Example – keeping track of family relationships



Examples of desired constraints

- Every person has two parents, one man and one woman
- Parents of any child are married
- Cannot marry a sibling or a parent
- Every person is married to at most one person
- a married to $b \Rightarrow b$ is married to a
- A man can only marry a woman, and vice versa

Key elements of Alloy model

- abstract sig Person
 - Person is a *abstract* entity (i.e., with no concrete instances).
- sig Man, Woman extends Person {}
 - Man, Woman are subtypes of Person.
 - No other subtypes. Therefore, every instance of Person is an instance of either Man or Woman.
- spouse is a relation mapping each Person to zero or one Person
- parents is a relation mapping each Person to zero or more Persons
- Constraints

```
fact {
    all p: Person | one mother: Woman | one father: Man |
        p.parents = mother + father // every person has a mother and father
        spouse = ~spouse // spouse is symmetric
        Man.spouse in Woman && Woman.spouse in Man
        // a man's spouse is a woman and vice versa
}
```

Results of using Alloy on above example

- Our rules are too relaxed. They allow instances wherein a person is their own grandparent.
- We could add an extra rule (i.e., fact) that no person be their own grandparent. However, with this, there are no non-trivial instances!

Model checking using Spin

- Given an abstract model like a state machine, and a specification of desired behaviour of all traces (typically in temporal logic), the tool tries to verify that the model satisfies the property.
- If not, it produces a **counterexample**: a behaviour of the model that violates the property.
- Similar to Alloy, but checks traces of a state machine rather than entities and relationships.

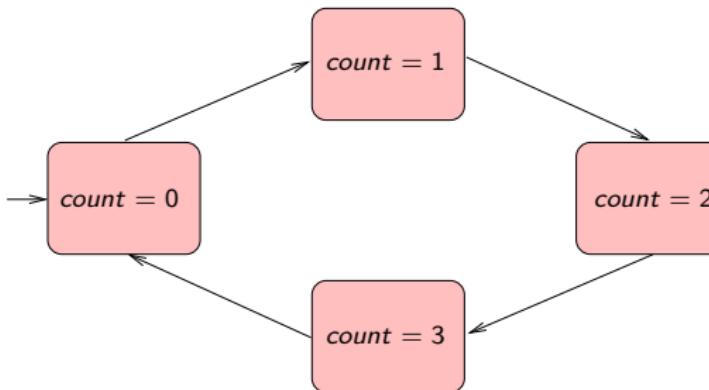
Model-Checking and Spin

Spin is a **model-checking tool**, in which we can

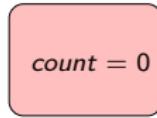
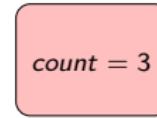
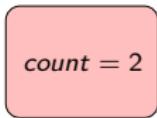
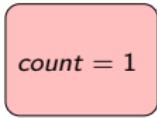
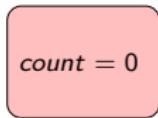
- **Describe** transition system models.
 - Suited for concurrent protocols, supports different synchronization constructs.
- **Simulate** them, explore paths in them.
- **Describe** desirable properties of the system in temporal logic.
- **Check** that the system satisfies these properties.
 - Proves that property is satisfied
 - Produces counter-examples (execution that violates property).

Example transition system: a mod-4 counter

Diagrammatic representation



Example run:



Example properties for counter model

```
byte count = 0;

proctype counter() {
    do
        :: true -> count = (count + 1) % 4;
                    assert (count <= 3);
    od
}

init {
    run counter();
}

ltl prop1 { [](count <= 3) };
ltl inc { []((count == 1) -> X(count == 2)) }
ltl prop3 { ((count == 0) || (count == 1)) U (count == 2));
ltl prop4 { [](count == 0) };
```

Tools for implementation and testing

- **Verification** tools [VCC]
 - Guarantee that a program returns correct output in all runs.
 - Programmer needs to specify formally correctness of program output
 - Programmer also needs to specify intermediate properties that need to hold at various program locations in order for final output to be correct. Otherwise, tool may fail to work or may report false warnings.
- **Automated testing** tools [JPF, AFL]
 - Based on **actually executing** the program on test-cases.
 - Both tools **automatically** generate test inputs one after the other, to try to **reach** more and more parts of the program.
 - Developer would need to specify a way to check correctness of output from each run. However, developer need not annotate intermediate program locations with intermediate properties.
 - All bugs found are true bugs.
 - However, can miss bugs.

Prerequisites

- Discrete structures such as sets, relations, partially ordered sets, functions
- Mathematical logic (propositional, first-order)
- General mathematical maturity: comfort with notation, understanding and writing proofs
- Familiarity with languages C/C++ and Java
- (Moderate) programming experience

Lecture format

- Theory and algorithms behind the tools
- Demo of tools

Assignments and exams (tentative)

- Assignments (60%). Each assignment will involve
 - applying one or more tools practically, and
 - a few written problems
- Exams: mid-sem (20%), final (20%).
 - Will have practical (lab) component.

Misconduct policy

- Academic misconduct (e.g., copying) will not be tolerated
- Discussion in exams ⇒ automatic fail grade for both students
- Assignments
 - Work individually.
 - If necessary, you can seek clarifications on basic concepts from other students.
 - However, you must develop the solutions to the given problems **on your own** (without discussions), and write the programs or answers on your own.
 - **No looking at others solutions, no showing your solution to others!**
 - If you refer to materials other than class lecture notes and text books, mention them.
- Penalties
 - For *each* instance of a violation of above policy ⇒ Zero for the entire assignment, *plus* one grade-point reduction in final grade (for the one who copied).
 - Grade-point reductions over multiple violations will accumulate.
 - Grading: Your marks for each assignment will be based on your written answers *and* on a subsequent viva.

Late submission policy for assignments

- For each late day 20% penalty on the assignment marks.
(Weekends and weekdays treated the same.)

Attendance

- Students are expected to have at least 75% attendance. As per Institute rules students with less than 75% attendance are liable to get an **F** grade.