E0:272, Formal Methods in Software Engineering

3:1, January - April 2018

CSA 252, M-W 9.30-11

http://www.csa.iisc.ac.in/~deepak/fmse-2018

Deepak D'Souza   K. V. Raghavan

CSA, IISc

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
  - Both research tools, as well as mature, widely-used 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

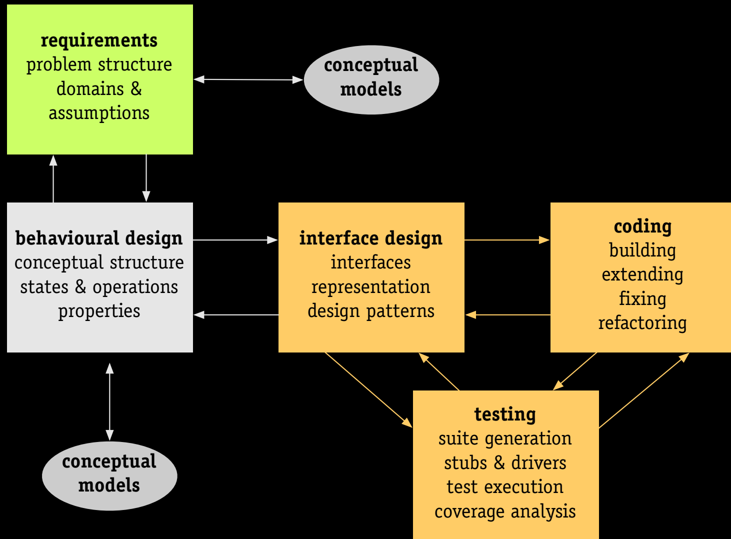
Software defects

- 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
- Design does not meet user requirements.
  - More than 50% of all defects are due to above two reasons
- 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.
  - Logical errors

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!

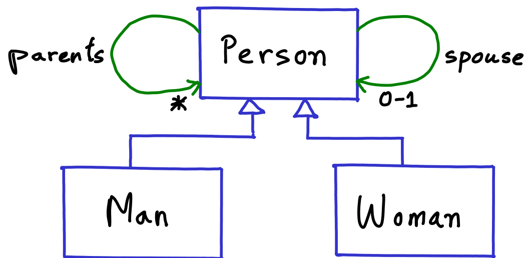
Tools for conceptual modeling and design

- 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

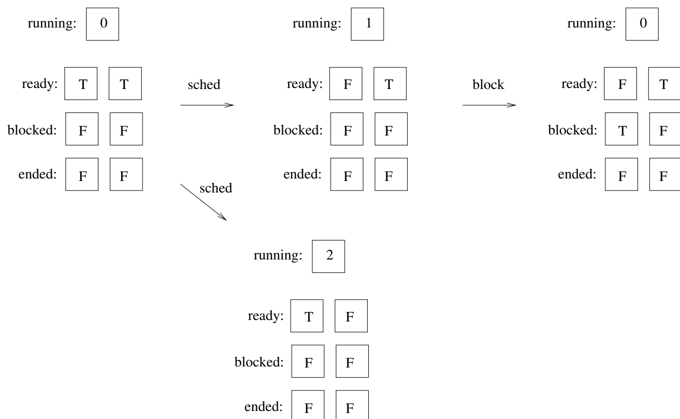
- Our rules are too relaxed. They allow instances wherein a person is their own grandparent.
- We could add an extra rule (i.e., `fact`) that no person be their own grandparent. However, with this, there are no non-trivial instances!

Model checking using Spin/SAL

- 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 satisifes 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.

Example: Simple Operating System Scheduler

- Scheduler has two tasks (task1 and task2) to schedule.
- Events in the system: `sched`, `block`, `unblock`, `fin`.
- Scheduler non-deterministically schedules a task if it is ready.



Scheduler model in SAL (1)

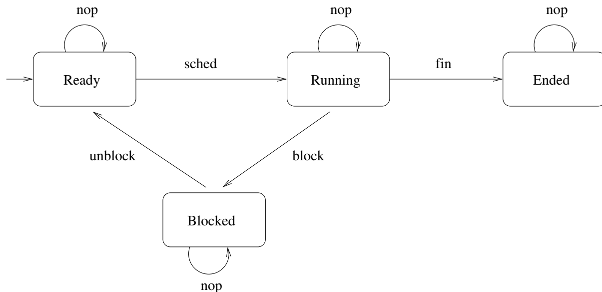
```
scheduler: context =
begin
   Events: type = {nop, sched, fin, block, unblock};

   scheduler: module =
   begin
       input event: Events
       local running: [0..2]
       local ready: array [1..2] of boolean
       local blocked: array [1..2] of boolean
       local ended: array [1..2] of boolean

       initialization
           running = 0;  % no process is running.
           ready[1] = true; % process 1 is ready.
           ready[2] = true; % process 2 is ready.
           blocked[1] = false;
           blocked[2] = false;
           ended[1] = false;
           ended[2] = false;
```

Scheduler model in SAL (2)

```
transition [
    ((event = sched) AND ready[1] AND (running = 0)) -->
                    running' = 1; ready'[1] = false;
  [] ((event = sched) AND ready[2] AND (running = 0)) -->
                    running' = 2; ready'[2] = false;
  [] ((event = block) AND (running /= 0)) -->
                    running' = 0; blocked'[running] = true;
  [] ((event = unblock) AND blocked[1]) -->
                    ready'[1] = true; blocked'[1] = false;
  [] ((event = unblock) AND blocked[2]) -->
                    ready'[2] = true; blocked'[2] = false;
  [] ((event = fin) AND (running /= 0)) -->
                    running' = 0; ended'[running] = true;
  [] else -->  % do nothing
  ]
end;
```

Properties we may want to check of the model

- “nocreate:” Once a task has ended it is never created again.
- “nostarve:” Once a task is ready it eventually runs.
- “stateseq:” Each task follows specified state motion:



Properties in SAL

```
nocreate: theorem scheduler |- G(ended[1] => NOT F(ready[1]));

nostarve: theorem scheduler |- G(ready[1] => F(running = 1));

stateseq: theorem scheduler |- G(((ready[1] AND event = nop) => X(ready[1])) AN
  ((ready[1] AND event = sched AND running = 0 AND NOT ready[2]) =>
                                   X(running = 1)) AND
  ((running = 1 AND event = nop) => X(running = 1)) AND
  ((running = 1 AND event = block) => X(blocked[1])) AND
  ((blocked[1] AND event = nop) => X(blocked[1])) AND
  ((blocked[1] AND event = unblock AND NOT blocked[2]) => X(ready[1])));
```

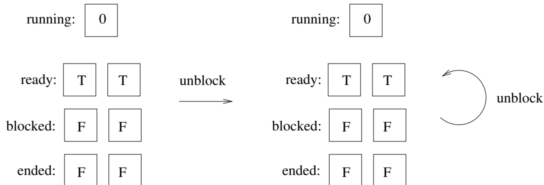
Output of SAL checker

```
$ sal-smc scheduler.sal

Counterexample for 'nostarve' located at [Context: scheduler, line(40), column(0)]:
========================
Path
========================
Step 0:
--- Input Variables (assignments) ---
event = unblock
--- System Variables (assignments) ---
running = 0
ready[1] = true
ready[2] = true
blocked[1] = false
blocked[2] = false
ended[1] = false
ended[2] = false

Step 1:
--- Input Variables (assignments) ---
event = unblock
--- System Variables (assignments) ---
running = 0
ready[1] = true
ready[2] = true
blocked[1] = false
blocked[2] = false
ended[1] = false
ended[2] = false
========================
Begin of Cycle
========================
Step 1:
```

Output of SAL checker (ctd)

```
Summary:
The assertion 'nocreate' located at [Context: scheduler, line(39), column(0)] is valid.
The assertion 'nostarve' located at [Context: scheduler, line(40), column(0)] is invalid.
The assertion 'stateseq' located at [Context: scheduler, line(41), column(0)] is valid.
```

Trace that violates "nostarve" property



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 [Pex, 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

- Demo of tools
- Theory and algorithms behind the 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 (15%), final (25%).
  - Will have practical (lab) component.

Misconduct policy

- Academic misconduct (e.g., copying) will not be tolerated
- Discussion in exams $\Rightarrow$ 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 $\Rightarrow$
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 policy for assignments

- 12 “free” late days for use over all assignments.
- For each late day after free days have been exhausted $\Rightarrow$ 25% penalty on the assignment marks. (Weekends and weekdays treated the same.)
- No late days allowed on final assignment.