Summary of the Course: “The Last Lecture”

Lecture 15: CPEN 400P

Karthik Pattabiraman, UBC

Some Points to Note

This is high-level review of what we learned in the course. This is by no means comprehensive, and it's not meant to replace the actual lecture notes. Also, I've focused only on the main points.

Most topics have equal coverage in this summary, but that doesn't mean they'll have equal coverage in the final exam. We spent more class time on some topics than others.

Please ask me any questions you have at the end. Or else, this'll be quite a short class :)

We'll also do the course evaluations in the middle of the class with a 10 minute break

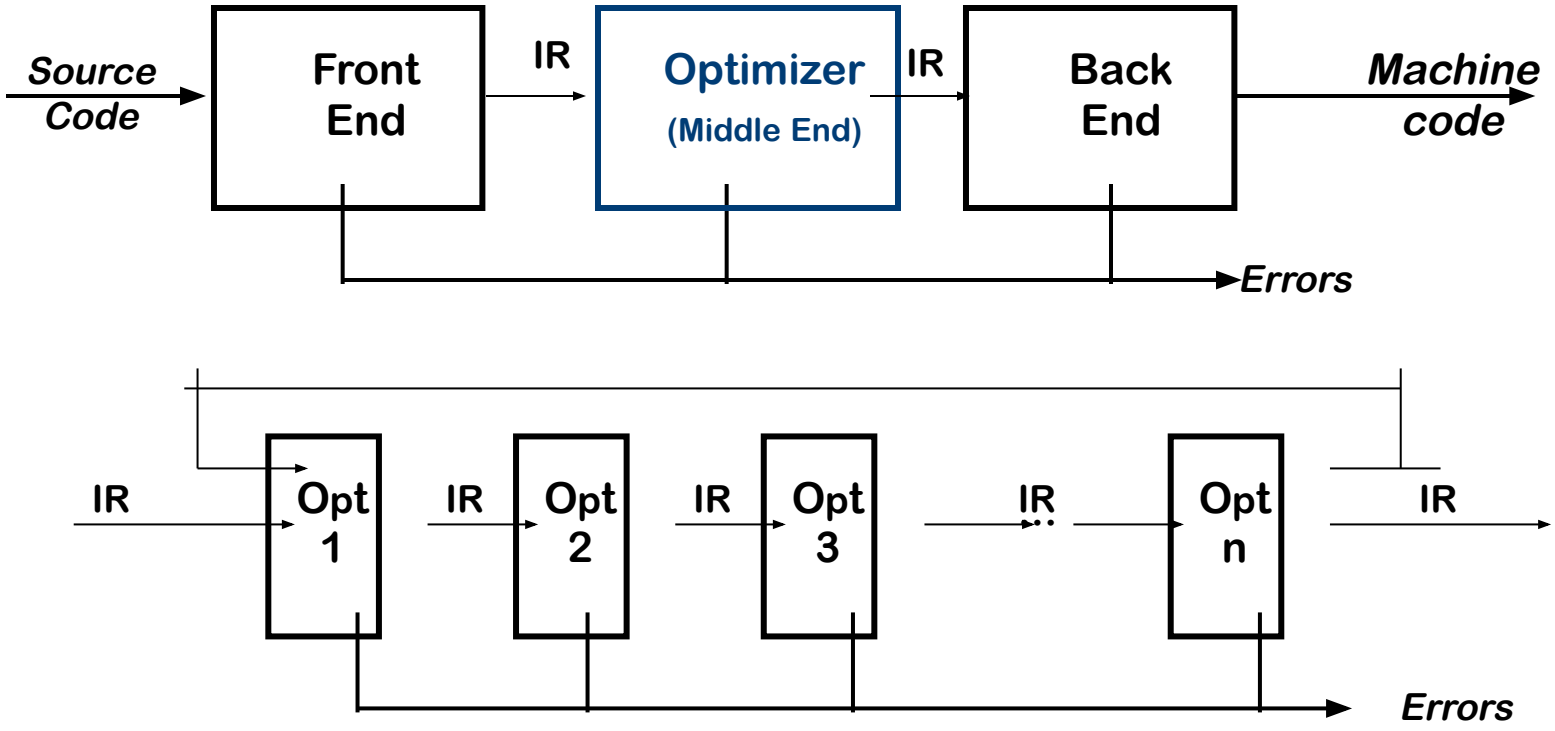
Outline - 1

Static Analysis techniques

- Stages of a compiler, IRs
- Data-flow analysis
- SSA form construction
- SSA-based optimizations
- Inter-procedural analysis
- Pointer/Alias Analysis

Dynamic Analysis techniques

Stages of a compiler



Intermediate Representations (IRs)

Many kinds of IRs used in compilers

- Two-address code
- Three-address code
- Stack-based
- Directed Acyclic Graph (DAG)
- Abstract Syntax Tree (AST)
- Control-flow Graphs (CFG)
- SSA form
- Memory-to-Memory and Register-to-Register Models

Data-flow Analysis - 1

Dominators

A node *n* dominates *m* iff *n* is on every path from n_0 to *m*

- **Every node dominates itself**
- ***All of m*'s predecessors *need to be dominated by n***

$$\text{DOM}(n_0) = \{ n_0 \}$$

$$\text{DOM}(n) = \{ n \} \cup (\cap_{p \in preds(n)} \text{DOM}(p))$$

LiveOut

- LiveIn is the set of variables that are live on *entry* to a basic block
- LiveOut is the set of variables that are live on *exit* from a basic block

$$LIVEOUT(n) = \emptyset, \forall n$$

$$LIVEOUT(b) = \cup_{s \in succ(b)} LIVEIN(s)$$

$$LIVEIN(b) = UEVAR(b) \cup (LIVEOUT(b) \cap VARKILL(b))$$

Dataflow Analysis - 2

Many kinds of data-flow problems - each comes with unique characteristics

But many characteristics are common (MOP operator, backward Vs forward etc.)

Can be combined into a common framework for code reuse etc.

Classification (Direction/MOP)	**May**	**Must**
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Anticipable Expressions

SSA Form - 1

Dominance Frontier

Where does an assignment in block *n* induce ϕ–functions in SSA form?

- ***n DOM m* ⇒ no need for a ϕ–function in *m***
  - **Definition in *n* blocks any previous definition from reaching *m***
- **If *m* has multiple predecessors, and *n* (strictly) dominates some of them, but not all of them, *m* needs a ϕ–function for each definition in *n***

This is also known as the dominance frontier of m - these are the locations at which phi nodes need to be inserted (with some minor optimizations)

SSA Form - 2

Insert phi nodes in all blocks that are in the DF(b), where b is the block in which the variable ‘x’ is defined

- DF(b) represents the first join-point in the CFG in which ‘x’ does NOT dominate any downstream uses of ‘x’.
- Phi-node ensures that the invariant of SSA is preserved, every def dominates all its uses
- Can be pruned if the variable ‘x’ is not Live across multiple basic blocks (any variable that is live across multiple blocks is added to *Globals)*
- Variables that are live across basic blocks can be computed as UEVar(b)

SSA-based Optimization - 1 (SSCP)

Lattice Operator (meet): Represented as $\wedge$, and has the following rules:

- **$c_1 \wedge c_2 = c_1$ if $c_1 = c_2$, else $\perp$**
- **$c_1 \wedge \top = c_1$**
- **$c_1 \wedge \perp = \perp$**
- **$\top \wedge \perp = \perp$**

Intuition: When you do a meet of two elements *x* and *y*, you descend the lattice to find a "greatest lower bound" of them

- **Represents facts about what is known and unknown in the program**

SSA-based Optimization - 2 (SSCP)

Only propagate constant information on edges of vars that have been modified

- Use SSA form to easily find all the uses of a variable
- Keep adding edges to a worklist until you run out of edges

Variables assigned to constants are initially marked to a constant value

When you come to a Phi-Node, perform a meet operation over the operands

For all other nodes, it depends on how complex are the semantics we implement

- Need to encode simple arithmetic rules (e.g., const + const = const)
- Can be quite complex to capture all possible permutations

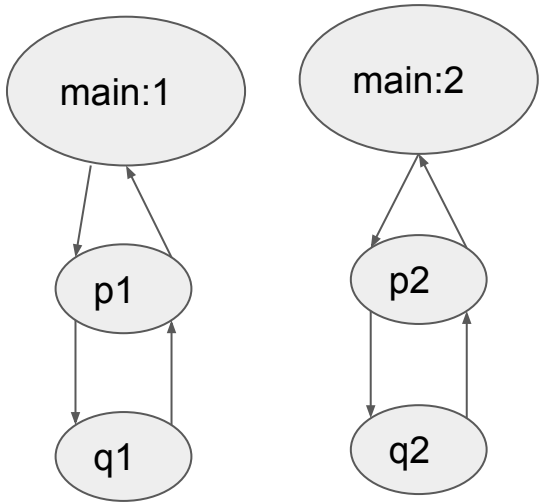
Interprocedural Analysis - 1

Analysis can trade-off precision for cost depending on the types of sensitivity

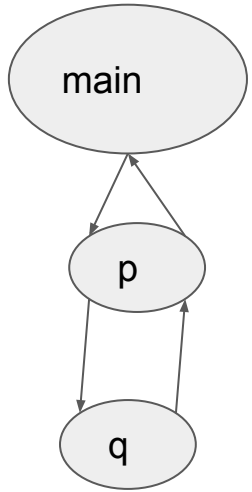
1. Flow-sensitive
2. Path sensitive
3. Context sensitive

```
main() {
        1: p(7);
        2: p(42);
}

p(int n) {
        3: q(n);
}

q(int k) {
        return k;
}
```

Context-sensitive



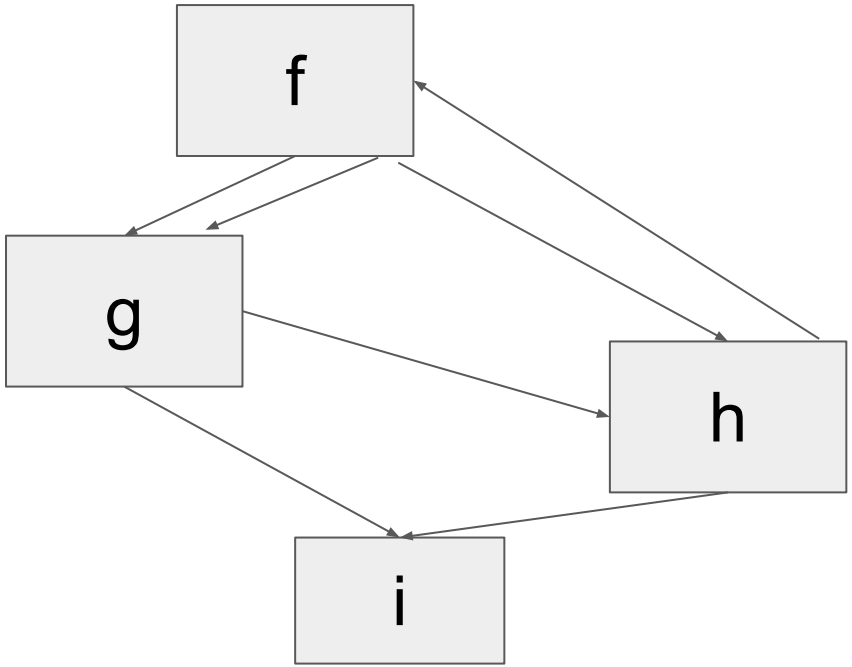
Context-insensitive



Interprocedural Analysis - 2

Call Graph Construction: Represents all possible function call paths in program

```
f() {
    1: g();
    2: g();
    3: h();
}

g() {
   4: h();
   5: i();
}

h() {
    6: f();
    7: i();
}

 i() { … }
```



Pointer Analysis - 1

1. Pointers to pointers, which can occur in many ways:
   - Take address of pointer
   - Pointer to structure containing pointer
   - Pass a pointer to a procedure by reference
2. Aggregate objects: structures and arrays containing pointers
3. Recursive data structures (lists, trees, graphs, etc.)

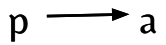
   closely related problem: anonymous heap locations
4. Control-flow: analyzing different data paths
5. Inter-procedural analysis is crucial

Pointer Analysis - 2

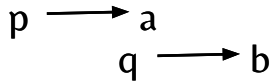
Anderson's (No unification)

Steensgard (unification)

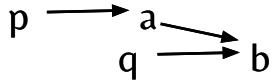
p = &a



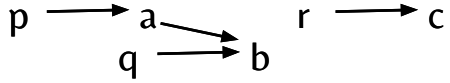
q = &b



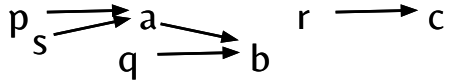
*p = q;



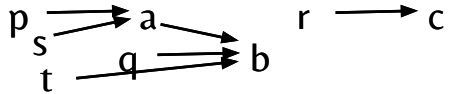
r = &c;



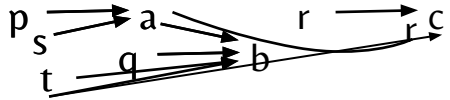
s = p;

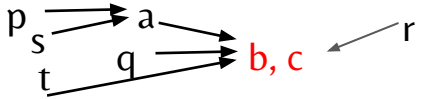


t = *p;



*s = r;





Dynamic Analysis - 1

Can analyze the actual execution of the program and obtain runtime values

Much more **precise** compared to static analysis

- No need to over-approximate across all paths
- No need to worry about scalability of the analysis
- No need to be conservative when the analysis cannot determine a value

Many common code constructs are difficult to statically analyze (e.g., threads)

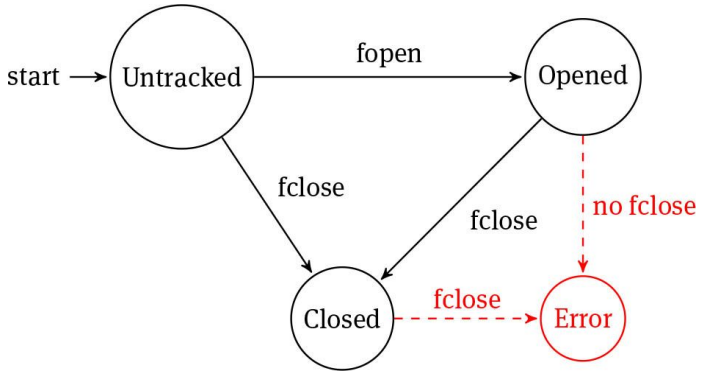
Can take external environment into account (e.g., interrupts, I/O operations etc.)

Does not need the source code of libraries or 3rd party component

Dynamic Analysis - 2

Steps in dynamic analysis:

1. Instrument the program to capture the “property of interest”
2. Execute the program under one or more inputs
3. Monitor program for the property violation at runtime
4. (Optional) Use static analysis results to optimize the dynamic execution monitoring



Outline - 2

Testing and Test generation

Fuzzing

Symbolic Execution and Concolic Execution

Model-checking

Fault Injection

Safety Analysis

Final Thoughts

Path Forward

Testing - 1

Black Box Testing: Generate tests based on specs alone without the code

- Advantage: Tests are not biased towards the implementation

Kinds of Test Cases

- Test each clause of the spec. Individually
- Test boundary conditions
- Test invalid cases (out of spec.)

Testing - 2

White-box Testing: Use the code to generate tests

- Advantage: Test can cover the code better
- Attempt to maximize different types of coverage

Three types of code coverage

- Statement coverage
- Edge coverage (of a CFG)
- Path coverage

Tests can be automatically generated using pre-conditions & post-conditions

Fuzzing - 1

- Run program on many random, abnormal inputs and look for bad behavior in the responses
  - Bad behaviors such as crashes or hangs
- Approach
  - Generate random inputs
  - Run lots of programs using random inputs
  - Identify crashes of these programs
  - Correlate random inputs with crashes
- Problem: It can take a long time to generate “interesting” inputs

Fuzzing - 2

Types of Fuzz Testing

- Black-box fuzzing: Fuzz without knowing details of the implementation
- White-box fuzzing: Design fuzzing based on internals of the system

Types of Black-box fuzzing

1. Mutation-based: Mutate a well-formed input into another one
2. Generation-based: Generate an input based on a user-provided specification

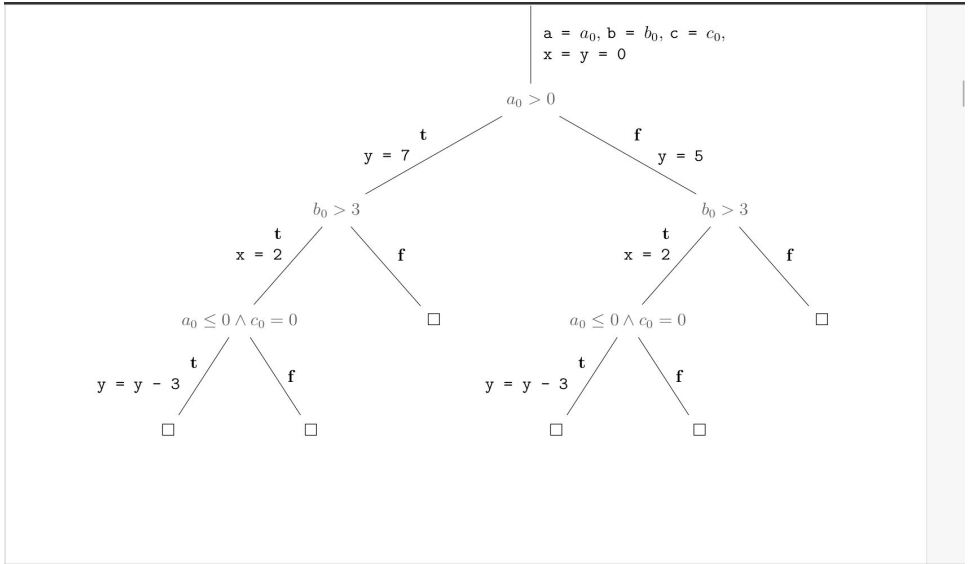
AFL: Coverage-based white-box (grey-box) fuzzer

- Instruments edges and attempts to increase edge coverage

Symbolic Execution - 1

Execution Tree: Capture all possible execution paths in program

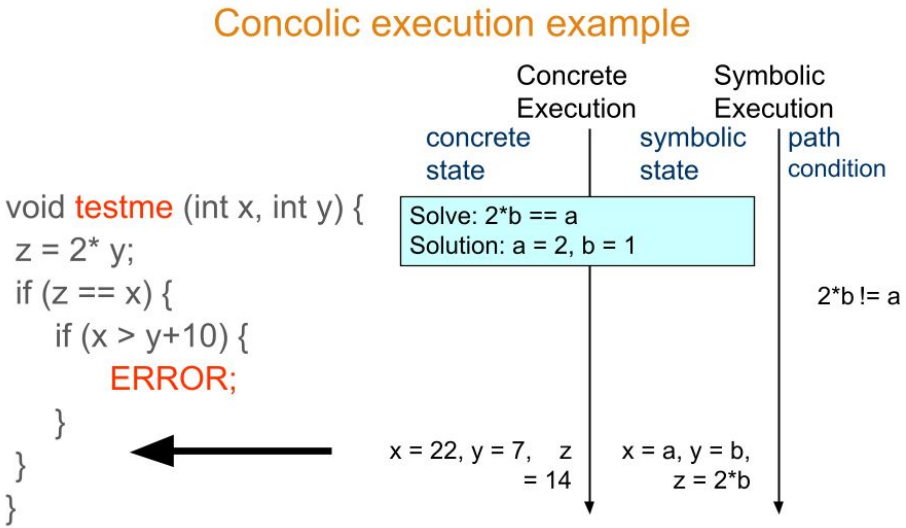
- Model every branch as Taken or Not Taken
- Determine (in)feasibility of each path by constraint solving



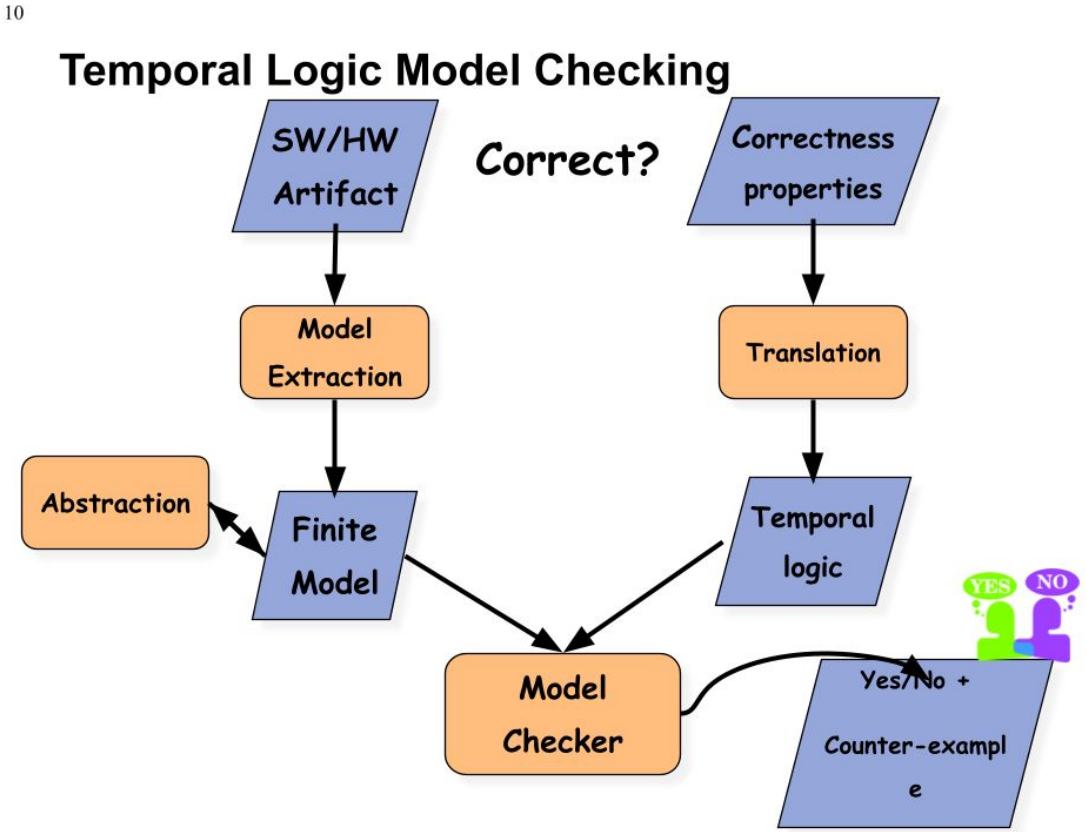
Symbolic Execution - 2

Difficult to scale to large programs, and programs with complex interactions

- Solution: Use concrete input together with symbolic input (**Concolic**)
- Solve symbolic constraints to negate path condition with concrete values



Model Checking - 1



Model Checking - 2

CTL: Computational Tree Logic (CTL)

- Propositional temporal logic with explicit quantification over possible futures

  True and *False* are CTL formulas;
  propositional variables are CTL formulas;

  If ϕ and ψ are CTL formulae, then so are: $\neg \phi$, $\phi \wedge \psi$, $\phi \vee \psi$

  EX ϕ : ϕ holds in some next state

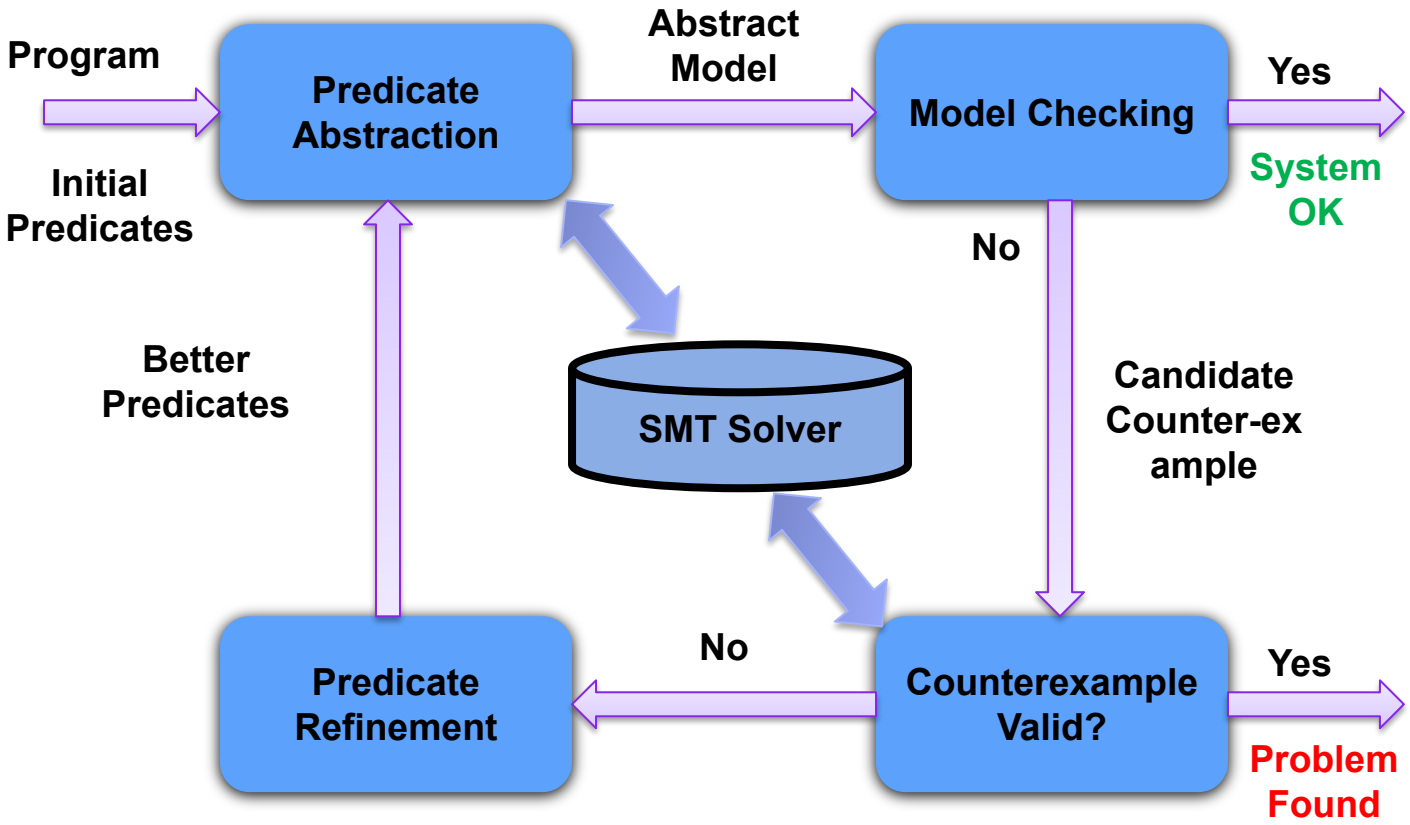
  EF ϕ : along some path, ϕ holds in a future state

  E[ϕ U ψ] : along some path, ϕ holds until ψ holds

  EG ϕ : along some path, ϕ holds in every state

  • Universal quantification: AX ϕ , AF ϕ , A[ϕ U ψ], AG ϕ

Model Checking - 3 (Counter-Example Guided Abstraction Refinement)



Localization Reduction, Kurshan, Bell Labs

Counterexample-guided Abstraction Refinement for Symbolic Model Checking, Clarke et al., CMU

Software Model Checking, SLAM Project, Microsoft, Ball & Rajamani

Fault Injection - 1

Fault Injection: Act of deliberately introducing faults into the system in a controlled and scientific manner, in order to study the system's response to the fault

- Can be used to estimate resilience (e.g., detection, recovery)
- To obtain reliability estimates of the system prior to deployment

Three types of Software-based Fault Injection (SWiFI)

- Compile-time
- Runtime
- Hybrid (LLFI)

Fault Injection - 2

Testing Cloud Applications: Also known as Chaos Engineering (ChaosMonkey)

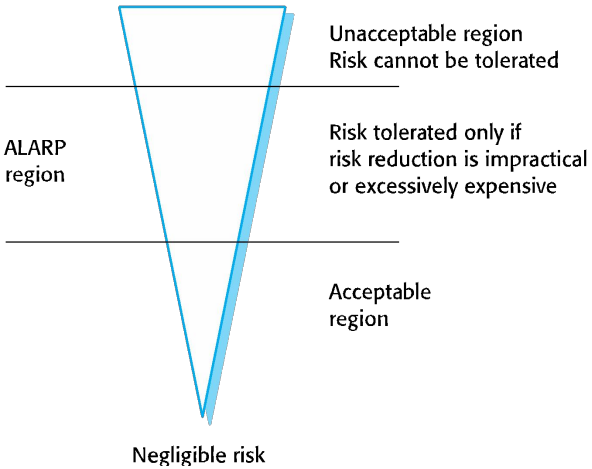
Can inject different types of faults

- Hardware failures
- Functional bugs
- State transmission errors (e.g., inconsistency of states between sender and receiver nodes)
- Network latency and partition
- Large fluctuations in input (up or down) and retry storms
- Resource exhaustion
- Unusual or unpredictable combinations of interservice communication
- Byzantine failures (e.g., a node believing it has the most current data when it actually does not)
- Race conditions
- Downstream dependencies malfunction

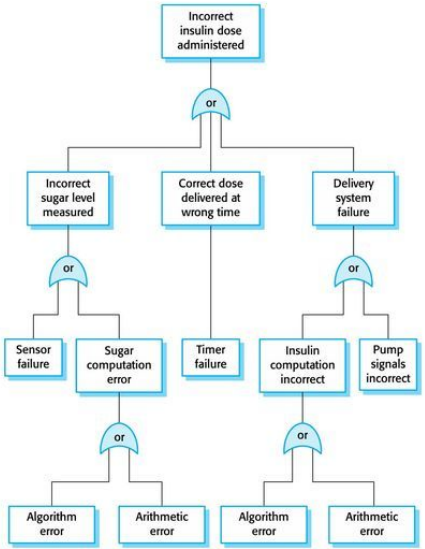
Need to limit blast radius of the fault

Safety Analysis - 1

ALARP - As Low as Reasonably Practicable



An example of a software fault tree



Safety Analysis - 2

Construction of Safety Arguments

- Establish the safe exit conditions for a component or a program.
- Starting from the END of the code, work backwards until you have identified all paths that lead to the exit of the code.
- Assume that the exit condition is false.
- Show that, for each path leading to the exit that the assignments made in that path contradict the assumption of an unsafe exit from the component.

Final Thoughts

Prepare well for the exam; make sure you really understand the concepts

Problem solving type questions will be very similar to the class activities

M/C questions will test background knowledge about different topics

Weightage will be more or less evenly split between the pre- and post- midterm

I'll be online on Piazza and I'll try to answer questions promptly until April 19th

If you need it, I can also setup Zoom sessions on demand (private note on Piazza)

Class participation marks will be assigned only after the final exam

Path Forward

Hope you enjoyed this course - this is the first time it was taught (at UBC or elsewhere)

- Feel free to send me suggestions or comments (anonymously or not, no difference)
- I'd appreciate it if you can complete the online course evaluations as well

Hope you get to apply these techniques in industry (or at least be influenced by them)

- I tried to choose topics/techniques that have a proven history of application….
- Not all topics are relevant for all industry, and I had to leave some topics out
- If you find any of these useful or relevant in your jobs, I'd appreciate hearing from you (or better still, come and give a guest lecture at a future offering of this course)
- If any of you is interested in grad school, I'd be happy to chat (send me an email)

Outline - 1

Static Analysis techniques

- Stages of a compiler, IRs
- Data-flow analysis
- SSA form construction
- SSA-based optimizations
- Inter-procedural analysis
- Pointer/Alias Analysis

Dynamic Analysis techniques

Outline - 2

Testing and Test generation

Fuzzing

Symbolic Execution and Concolic Execution

Model-checking

Fault Injection

Safety Analysis

Final Thoughts

Path Forward