CS1632, Lecture 18: Static Analysis, Part 2

Wonsun Ahn

Kinds of Static Tests

- Code review / walk-through
- Compiling
- Code coverage
- Code metrics
- Linters
- Bug finders
- Formal verification

Formal Verification

- Proving one or the other about a program:
  - Program has no defect
  - Program has defects (and find all of them)
- What!?

Holy Grail of Formal Verification

- Soundness
  - If no defect is reported, then the program does not fail
  - No false negatives
- Preciseness
  - If a defect is reported, then the program does fail
  - No false positives
- Termination
  - The verification terminates

It is impossible to achieve the holy grail in general!

Formal Verification is Undecidable

$x \in$ Variable

$P \in$ Program = assert x | x++ | x-- |

$P_1 ; P_2$ | if x then P_1 else P_2 | **while x P**

- Assertion checking for even this simple language is undecidable!
- "The **Halting Problem** cannot be solved for *all* possible programs (for a Turing-complete language)" - *Alan Turing (1936)*
- Silver-lining: But for *some* programs it can be solved

Methods of Formal Verification

- Theorem Proving
  - Deducing postcondition from precondition through mathematical formal methods

- Model Checking
  - Given a finite state model of a system, exhaustively checking whether this model meets a given specification

Theorem Proving

Deducing postcondition from precondition through mathematical formal methods

Hoare Logic Theorem Proving

- Hoare Logic: Deduces postcondition from precondition through math
- Hoare Triplet: {Precondition} Program {Postcondition}
  - Meaning: Given Precondition and Program, Postcondition is always true
- Examples of Hoare Triplets:
  - { true } x := 5 { x == 5 }
  - { x == y } x := x + 3 { x == y + 3 }
  - { x == a } if (x < 0) then x := -x { x == |a| }
  - { x < 0 } while (x!=0) x := x-1 { ??? } ← No such triple!

Hoare Logic Syntax

English	Formal
false	$\bot$
true	$\top$
not p	$\neg p$
p and q	$p \wedge q$
p or q	$p \vee q$
p implies q	$p \Rightarrow q$
p iff q	$p \Leftrightarrow q$
for all x, p	$\forall x. p$
there exists x such that p	$\exists x. p$

- Idea is to use this syntax to prove with pen and paper your program is correct
- Sounds unappealing? ☹
  - Many programmers would agree!
- There exist “theorem provers” that automate mundane parts of proving
  - But needs human assistance at difficulties
  - Example difficulty: reasoning about recursive data structures (lists, trees, …)

Theorem Proving Advantages

- Can prove large programs with infinite states
  - Remember this Hoare triplet?
    { x < 0 } while (x!=0) x := x-1 { ??? }
  - Model checker will have trouble because it has an infinite number of states
  - But a human or machine theorem prover can tell there is an infinite loop!

- Leads programmer to a deeper understanding of the program
  - After spending weeks proving the program is correct, a natural outcome
  - But really, it does lead to some fundamental insights about your program

Theorem Proving Disadvantages

- Requires (a lot of) human involvement
  - Every time a theorem prover encounters difficulty humans have to step in
  - Requires many hours of highly skilled labor to complete a proof
  - Humans also make mistakes

- Proofs can be obscenely long
  - In one report by Motorola, a proof was 25 MB long (more than 100 pages)
  - Beyond the comprehension limits of a normal human being

Industry Reception

- Advocates want a “formal methods guru” on every project team
  - The education required to produce a “formal methods guru” is very different from the education of a typical software engineer
- Naturally, industry is resistant
- Used in niche markets where correctness is paramount
  - Some embedded systems, cryptography libraries, OS kernels (seL4)

- Industry would like a “push button” solution
  - something that Model Checking provides!

Model Checking

Given a finite state model of a system, exhaustively checking whether this model meets a given specification

The Model Checking Problem

$I \models S$

"implementation"
(system model)

"specification"
(system property)

"satisfies", "implements", "refines"
(satisfaction relation)

Examples of System Properties

- Assertions (invariants)
  - Embedded in source code or part of property-based unit test
- Memory related properties
  - No leaks, double-free, access after free
  - No reading of uninitialized variables
  - No out of bounds array accesses
- Other resource related properties
  - No resource leaks: CreateFile followed by DeleteFile
  - No write of private data to insecure public resources

Comparison with Property-Based Testing

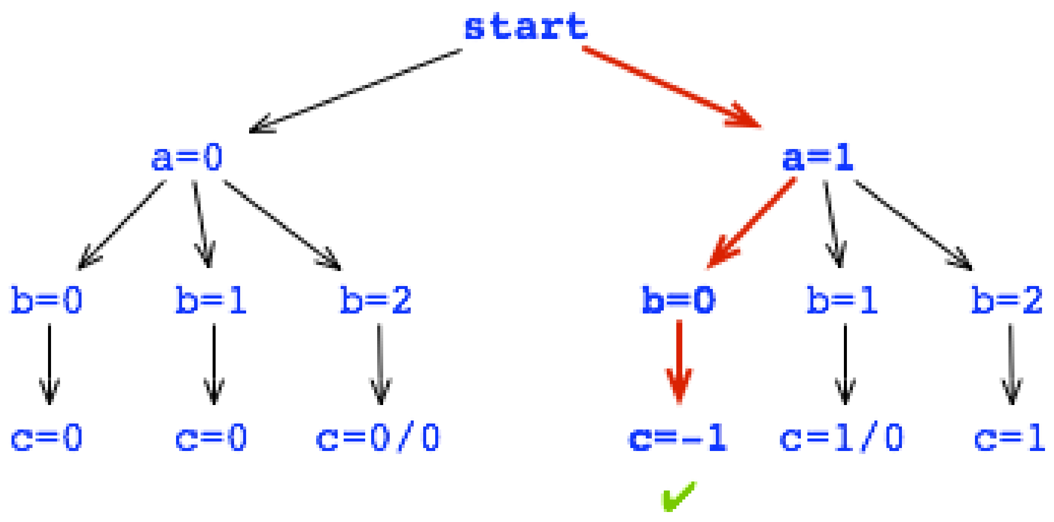
- Similarity
  - Model checking also tests a property, not an output value
- Difference
  - With stochastic testing, we tested (a few) randomized input values
  - With model checking, all states are check *exhaustively*
  - With model checking, states are visited in a *systematic* way

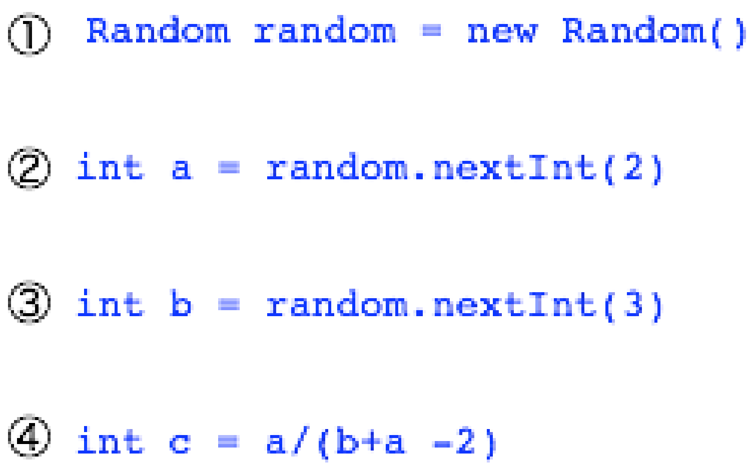
Stochastic Testing (a Single Trial)

Given this code:

```
int a = random.nextInt(2);
int b = random.nextInt(3);
int c = a/(b+a -2);
```

If unlucky, bug is never found!



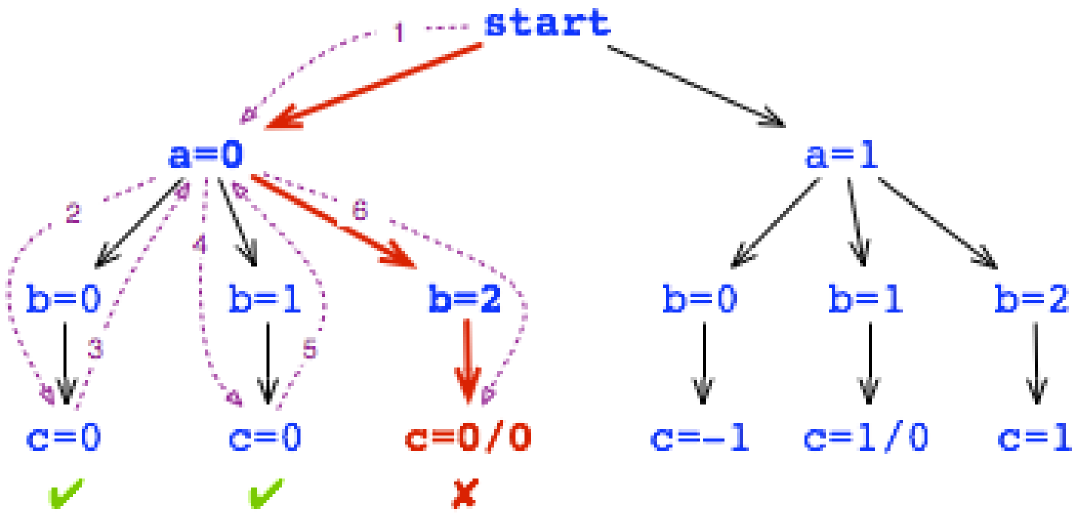


Model Checking

Given this code:

```
int a = random.nextInt(2);
int b = random.nextInt(3);
int c = a/(b+a -2);
```

Bug is always found!
(through exhaustive searching)
If none found, guaranteed correct!



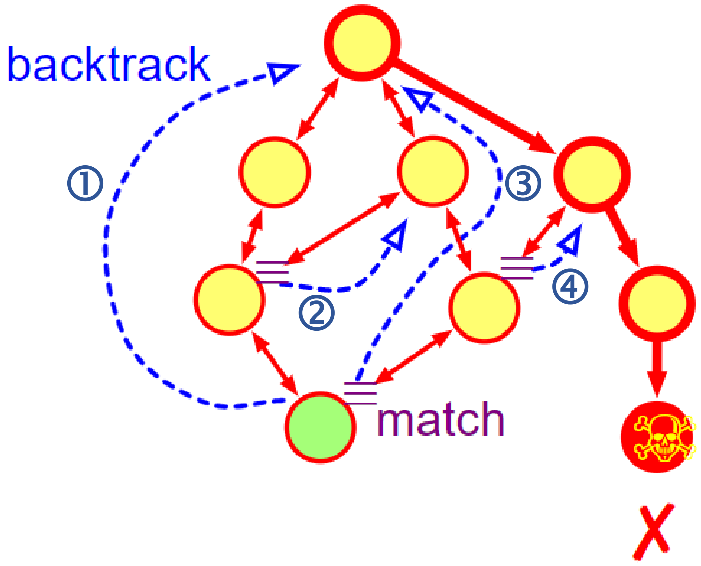
① `Random random = new Random()`

② `int a = random.nextInt(2)`

③ `int b = random.nextInt(3)`

④ `int c = a/(b+a -2)`

Systematic Exploration



Circles: Program states
Arrows: State transitions

- Example of depth-first-search of state space
- State transitions happen as a result of executing a program statement
- Backtrack
  - On termination or when there is a match
  - Go to a previous state with unexplored transitions
- Match
  - When next state matches a previously visited state
  - Backtrack to not waste work

Systematic Exploration Algorithm

```
Hashtable states_seen;
Queue pending;

pending.add(initial_state);
while(!pending.empty()){
      current = pending.remove();
      if(current in states_seen)
            continue;   // match! Backtrack.
      check current for correctness;
      states_seen.insert(current);
      for transition T in current {
            successor = execute transition T on current;
            pending.add(successor);
            restore_state(current);
      }
}
```

State Explosion Problem

- States num = $O(D^N)$, where N = variables, D = data type domain size
  - E.g. if there are N Boolean variables, number of states = 2^N
- You may even end up with an *infinite* number of states
  - If your program has the potential to create an infinite number of objects
- Single reason preventing wide adoption of model checking
  - May run into memory limitations (can’t contain entire state graph)
  - State exploration may take too long to be usable
  - Especially a big problem for sizable programs (> 10,000 lines of code)

Concrete and Abstract Model Checking

- Concrete model checking
  - States in model are actual concrete program states
  - As in, your program stack and your program heap

- Abstract model checking
  - States in model are some abstraction of actual program states
  - Abstraction is done in hopes of reducing the state explosion problem
  - Typically tradeoffs accuracy for efficiency

Concrete and Abstract Model Checking

- Concrete model checking
  - States in model are actual concrete program states
  - As in, your program stack and your program heap

- Abstract model checking
  - States in model are some abstraction of actual program states
  - Abstraction is done in hopes of reducing the state explosion problem
  - Typically tradeoffs accuracy for efficiency

Abstract Model Checking

- Requires an intermediate description of abstract model
  - Describes the system at a high level
  - Throws away implementation details

- Good for checking designs, rather than implementations
  - Success stories: hardware circuits, cache-coherence protocols

- Problem: Specifying an abstract model is HARD for large systems
  - What you check is not what you run!
  - As the system evolves model has to be updated
  - Manual extraction of abstract model can miss or introduce errors

Automatically Extracting the Model

- Statically analyze the code to generate a model
  - Models usually mimic the implementation

Murphi abstract model

```
Rule "PI Local Get (Put)"
 1:Cache.State = Invalid
    & ! Cache.Wait
 2: & ! DH.Pending
 3: & ! DH.Dirty ==>
  Begin
 4: Assert !DH.Local;
 5: DH.Local := true;
 6: CC_Put(Home, Memory);
  EndRule;
```

Flash Memory Driver Implementation

```
void PILocalGet(void) {
     // ... Boilerplate setup
2   if (!hl.Pending) {
3     if (!hl.Dirty) {
4!      // ASSERT(hl.Local);
         ...
5       hl.Local = 1;
6       PI_SEND(F_DATA, F_FREE, F_SWAP,
                 F_NOWAIT, F_DEC, 1);
```

Automatic Model Extraction

- Examples
  - FeaVer : C program -> Promela (SPIN) model
  - Bandera: Java -> Bandera model
- Features
  - Sophisticated property-driven slicing techniques
  - Can throw away state unrelated to property that is being proved
- Problems
  - Not all primitives are available in the modeling language
    - Pointers, dynamic object creation, dynamic threads, exceptions
  - A precise-enough slice could be as large as the program itself

Concrete and Abstract Model Checking

- Concrete model checking
  - States in model are actual concrete program states
  - As in, your program stack and your program heap

- Abstract model checking
  - States in model are some abstraction of actual program states
  - Abstraction is done in hopes of reducing the state explosion problem
  - Typically tradeoffs accuracy for efficiency

Concrete Model Checking

- Code as the model – directly execute the code!
- Concrete model checkers
  - Verisoft (C/C++) – Bell Laboratories
  - CBMC (C/C++) – Oxford University
  - Java Path Finder (Java) – NASA
- State space can be infinite (or very large)
  - Try exploring as much behaviors as possible (likely you can’t explore all)
  - Focus on precision (finding defects accurately)

State Space and Programming Language

- Definition of program state depends on programming language
  - What is the state that your program can access and modify?

- C/C++ programs: essentially your entire memory space!
- Java programs: abstract state maintained by the Java Virtual Machine
  - Java bytecode works on an abstract stack maintained by JVM + heap
  - That state is *much* smaller than your entire memory space
  - Gets some benefits of abstract model checking while still being concrete!

State Space and Programming Language

- Remember I said choice of language is important?
  - This is one example: it is much easier to model check Java than C/C++!

- Fortunately, you can convert most languages to Java bytecode
  - JavaScript, Python, Ruby, Lua, …
  - Even (for a limited set of) C / C++
  - And then, you can model check the bytecode using a JVM
  - But not equivalent since at deployment it will not execute as Java bytecode

- But regardless, state space grows *very* quickly

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

State collapsing

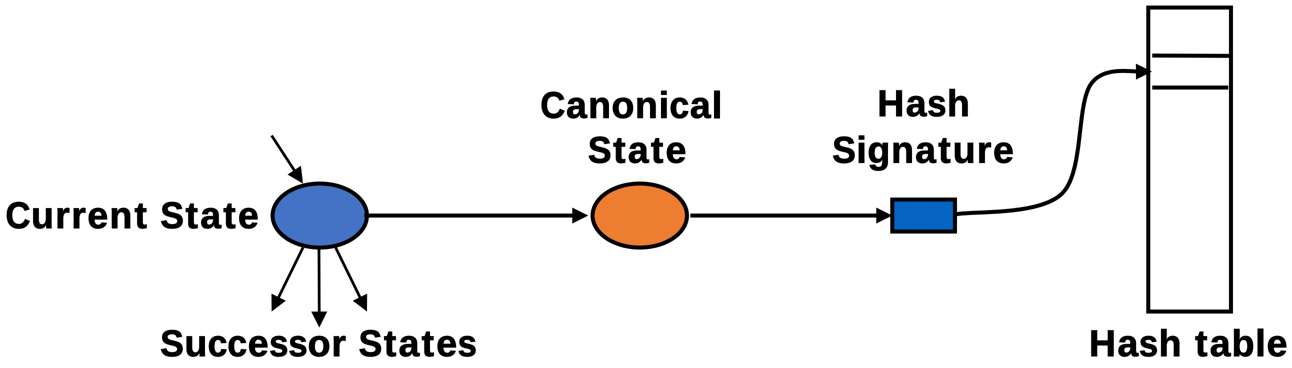
- Typically a state transition involves changing very small state
  - Updating a local variable in the stack
  - Creating a new object on the heap
- Instead of storing the entire state each time in hash table …
- Store the delta (change) from the previous state in hash table

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

Heuristic state approximation

- Explore one out of a (large) set of equivalent states
- Canonicalize (unify) states before hashing



- Example: suppose an int value has two equivalence classes
  - When hashing to check for match with an already visited state, Unify all values in equivalence class to one chosen value

  → Leads to a drastic reduction in visited states

  → Can also lead to missed defects – **unsound**

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

Hash compaction

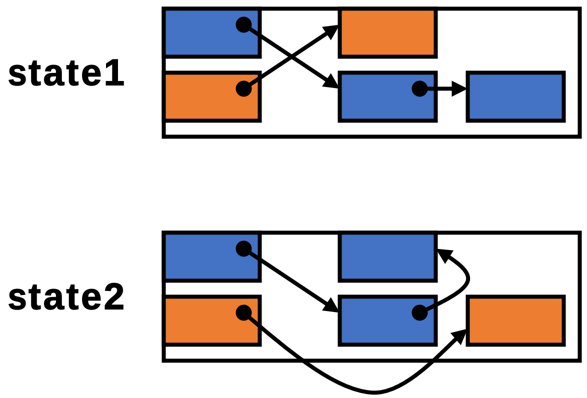
- Only store hash of state in the hash table
  - Do not store the actual state
  - A state match is determined solely by an equality check on the hash
- Might miss defects due to hash collisions – **unsound**
  - Two states that are different may be stored as the same hash
  - Means some states will not be visited as a result
- But orders of magnitude memory savings
  - Can compact 100 kilobyte state to 4-8 bytes!

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

Heap canonicalization

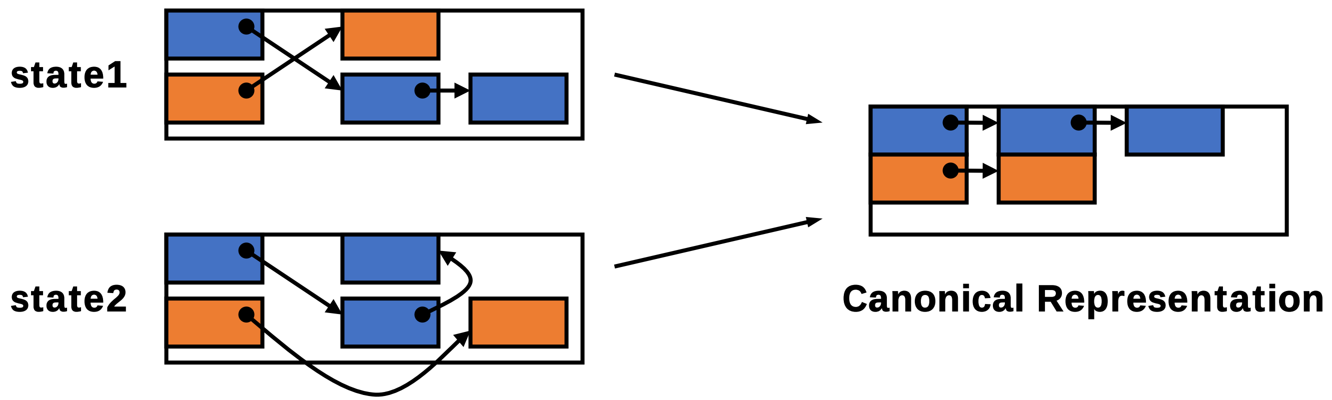
- Problem: two logically equivalent program states appear different because of differences in heap layout
- Example:



- Are the two states logically different?
  - No! But appears different due to different reference pointer values.

Heap canonicalization

- Solution: Canonicalize heap to unify layout
- Example:



- Canonical layout can be found by doing a fixed traversal of heap
  - DFS: Depth first search, or BFS: Breadth first search
- Note: can do it incrementally on each heap modification w/o full traversal

State Space Reduction Techniques

- State collapsing
- Heuristic state approximation
- Hash compaction
- Heap canonicalization
- Symbolic execution

To be continued …