CSED601
Dependable Computing
Lecture 6

Jong Kim

Dept. of CSE

POSTECH

References

– George Candea, Stanford University
  http://www.stanford.edu/~candea/teaching/cs444a-fall-2003/slides/formal.pdf

Overview

- Justification for formal methods
- Program analysis
  - Domain-specific programming rules
  - Dataflow analysis
- Formal specifications
- Model checking
- Theorem proving
- Myths about formal methods

Why Formal Methods?

- Sometimes you cannot provide software “as is”
- Give the customer a sense of confidence
- They offer legitimate benefits
  - Catch inconsistencies in requirements
  - Catch design bugs
  - May even catch code bugs
- Reality check: they cannot guarantee anything in the absolute
- A good complement to defensive programming, recovery-oriented coding, etc.

Program Analysis

- Your compiler does it
- Lexical analysis → syntactic analysis → semantic analysis
- gcc –Wall
- Main goal of detailed program analysis
  - optimization

Domain-Specific Programming Rules

- Easy to map directly to code:
  - Always check permissions on special ops
  - Check length of incoming strings
  - Don’t use freed memory
  - … ??
- Metacompilation extends gcc to check for this kind of rules
  - Programmer writes specific checkers
  - Later work: infer programming rules
- Free memory example: track state of variables

Using Freed Memory

State of variable `connection`

```
  ...

  connection->buffer = malloc( ... );        allocated
  if (NULL == connection->buffer) {
       free( connection );                   freed
       printf( ``Ran out of memory.'' );
       goto done;
  }

  ...

  /* connection establishment */

  ...

done:
  return connection;                         bug!
```

Dataflow Analysis

```
print STDERR ``Enter file name:'';
$x=<STDIN>;                       # $x is tainted (user input)

... more code ...

$z="/tmp/safe_file.txt";          # $z is clean
$y="$sysdir/$x";                  # $y is tainted
system("cat $y");                 # disallowed!
system("cat $z");                 # OK
```

Why might tainted data be bad?

Pros and Cons

- Pros
  - Done at compilation stage
  - Don’t need to execute code (like in Q/A testing)
  - Can analyze all execution paths
  - Don’t need to understand the intention of the code (mostly)

- Cons
  - Checked properties are shallow (close to the code)
  - Predictive flavor
    - Consequence: more aggressive → more false positives

Formal Methods

1. Write specification of the system (often you need one for the environment as well)
2. Formalize the desired properties of the system
3. Prove that the specification satisfies the properties

- Sometimes spec = system spec/model + properties (and we prove that it is consistent)
- Specs:
  - Written in formal, computer-understandable language
  - Concise
  - Unambiguous
  - Complete
  - Say “what” without saying “how” (FSSRs =counter example)

Writing Specifications I

- Define system states (input / output)
- Define all legal state-modifying operations

```
result: TYPE = [output: BRKTYPE, state: BRKTYPE]
ABS_apply_brakes( command: CMDTYPE,
                  brk_state: BRKTYPE ) : result =
   (IF (command == APPLY)
       THEN return [brk_state,
                    [~brk_state[left], ~brk_state[right]]
       ELSE return [brk_state, brk_state]
    )
```

Writing Specifications II

- Axiomatic approach: implicit statements about operations
- Pre- and pos-conditions

  binary_search (Array, Key) → Index

  PRE: ordered (Array) intersect (Key in Array)

  POST: Array[Index] == Key
- Invariants: pre- and post-condition for all operations

  INVAR: ordered(Array) intersect size(Array) < 100

Success Stories

- IBM CICS
  - OLTP system, precursor of TP monitor and app servers
  - 1980s: IBM Research + Oxford University used Z to specify parts of CICS and weed out bugs
  - Motivator for subsequent projects
- TCAS
  - Electronic eyes for pilot (up to 40 miles away)
  - Advisor for pilots, to not get confused and collide
  - Uses RSML for all official documentation
  - Spec checked for completeness and consistency
  - No mid-air collisions since 1990 in the US

Pros and Cons

- Pros:
  - Writing a spec furthers your understanding of the system
  - Stupid computers force explicit domain-specific knowledge
    - Space shuttle HAC case: major_mode=602 IMPLIES iphase>4
  - Writing uncovers design flaws, inconsistencies, ambiguity, and/or incompleteness
  - Spec provides a precise form of communication
  - Useful artifact: can use spec in verification
- Cons:
  - Very hard to specify full systems
  - Very hard to keep system and spec synchronized
  - Developers hate writing specs

Model Checking Inputs

- An example of verification (take specs a step further)
- Describe system with a state-machine transition func:

  State Machine: Inputs x State → Outputs x State

```
result: TYPE = [output: BRKTYPE, state: BRKTYPE]
ABS_apply_brakes( command: CMDTYPE,
                  brk_state: BRKTYPE ) : result =
   (IF (command == APPLY)
       THEN return [brk_state,
                     [~brk_state[left], ~brk_state[right]]
       ELSE return [brk_state, brk_state])
```

- Provide desired properties (model):

```
INVAR: ( brake_state[left] == brake_state[right] )
```

Model Checking Algorithm

- Exhaustively search states of system, and verify desired properties

```
BRKSTATUS: TYPE = ONEOF("applied", "released")
BRKTYPE: TYPE = [ BRKSTATUS, BRKSTATUS ]
result: TYPE = [output: BRKTYPE, state: BRKTYPE]
ABS_apply_brakes( command: CMDTYPE,
                  brk_state: BRKTYPE ) : result =
   (IF (command == APPLY)
       THEN return[ brk_state,
                    [~brk_state[left], ~brk_state[right] ]
       ELSE return[ brk_state, brk_state ])

INVAR: ( brake_state[left] == brake_state[right] )
```

- Does invariant hold ?
- Finite model → guaranteed termination

Success Stories

- Cache coherence protocols
  - IEEE Futurebus+ → found bugs, although it was mature
  - IEEE Scalable Coherent Interface
    - Model of system taken directly from C implementation
    - Just a small fraction of system modeled, still found bugs
- Active Mass Damper (AMD) system
  - Protects high-rises during earthquakes
  - Sensor pick up vibration, hydraulic actuators counter-act; all driven by computer
  - Model checking found one major bug

Pros and Cons

- Pros:
  - Completely automatic (unlike theorem proving)
  - Provides counter examples = executions that take system into state that doesn't satisfy properties
  - Can check partial specifications (good for large systems)
  - Can check more interesting properties than static analysis (higher level view of system)
- Cons:
  - State space explosion: exponential
  - Writing and maintain abstract model is hard
  - Coarse models far from actual implementation

Theorem Proving

1. Specify system using logic with a “context” (axioms + inference rules)
2. Specify desired properties using logic (theorems)
3. Machine generates proof of theorem based on system and context

- Theorem prover = fancy pattern matcher
- Needs user guidance in the form of lemmas and definitions

Simple (Fake) Example

- Modified ABS system

```
RULE: x div by 2 → (x+2) div by 2

ABS_apply_brakes( command: CMDTYPE,
                  brk_state: BRKTYPE ) : result =
   ...
   brk_state[left] += 10
   brk_state[right] += 10
   ...

INVAR: ( for all i, brake_state[i] div by 2 )
```

- Helper Lemma….

Helping the Theorem Prover

```
LEMMA: x div by 2 → (x+10) div by 2
   PROOF: x div by 2 → x+2 div by 2
          x+2 div by 2 → (x+2)+2 div by 2
          ... (inductive proof) ...
          x div by 2 → (x+10) div by 2
```

- Using this lemma, can prove desired property

```
ABS_apply_brakes( command: CMDTYPE,
                  brk_state: BRKTYPE ) : result =
   ...
   brk_state[left] += 10
   brk_state[right] += 10
   ...

INVAR: ( for all i, brake_state[i] div by 2 )
```

- All OK ?

Insufficient Axioms in Context

- Must say addition is associative (obvious to us…)

```
AXIOM: (x+y)+z == x+(y+z)
```

- Must provide transitivity rule

```
AXIOM: (P1 → P2 AND P2 → P3)
        →
        (P1 → P3)
```

- Now we can indeed prove

```
LEMMA: x div by 2 → (x+10) div by 2
   PROOF: x div by 2 → x+2 div by 2
          x+2 div by 2 → (x+4) div by 2  (associativity)
          ... (inductive proof) ...
          x div by 2 → (x+10) div by 2   (transitivity)
```

Success Stories

- Reaction Control System Jet Selection (JS)
- Motorola Complex Arithmetic Processor (CAP)
  - DSP processor
  - 3-stage pipeline, 6 independent memories, >250 programmer-visible registers, rich instruction set
  - Theorem prover allowed simpler CAP model for certain class of programs (could prove equivalence)

Pros and Cons

- Pros:
  - Can deal with **unbounded # of states**
  - Solid proofs of properties if assumptions are correct
- Cons:
  - Specs are very abstract, hard to translate real system into correct spec
  - Theorem prover **requires guidance**
    - Slow process
    - Error prone
    - Requires user to have an idea about the proof

Examples of Theorem Provers

- Java bytecode verification
  - Bytecode = language for modeling the program
  - Properties to be proven:
    - Types are used correctly and preserved
    - Object access restrictions are not violated
    - Pointers are not forged
- Proof-Carrying Code
  - Mobile code carries proof of its safety
  - Code receiver verifies proof quickly
  - Burden of proof is on code generator

Proving Correctness of an Internet App

- What would you need to prove in order to gain confidence in the application you're running?
  - Prove correctness of
    - Application code
    - Libraries
    - Operating system
    - Drivers
    - Hardware devices
    - Environment: power grid, administrators, nature
    - Laws of physics (God-preserved invariants)