Program Analysis

CSCE 747 - Lecture 16 - 04/04/2017

Axiom of Testing

“Program testing can be used to show the presence of bugs, but never their absence.”

- Dijkstra

Holy Grail of Verification

- Ability to prove whether any property holds over the software.
  - Finite State Verification can do this as long as we can abstract the software to a simple enough model.
  - Symbolic execution can form the basis of techniques that analyze the real source code.
    - Can exhaustively check for *particular* properties.
    - Can extract and summarize information for inspection and automated test generation.

Program Analysis

- Testing is weak at detecting faults that rarely cause failures.
  - Program fails under difficult to control conditions.
    - Race conditions - thread synchronization.
    - Memory access and allocation faults.
- Program analysis can detect these by abstracting the program down to a relevant finite model.

Concurrency Faults

Two types of subtle faults:

- Deadlock - threads are blocked, waiting for another thread to release the lock.
- Data Races - threads access a shared resource while other threads are modifying that resource.
- Concurrent threads can execute non-deterministically.
  - Same execution sequence may not result in the same failure.

Concurrency Faults

Can be prevented through safe programming:

- In critical regions, do not allow more than one thread to write to shared memory.
- In Java, synchronized blocks protect shared variables.
  - Threads entering the block are locked out until the thread in the block exits.

```
public synchronized void
     add(int value){
       this.count += value;
}
```

- Synchronized Method
  - One thread at a time.

```
public void add(int value){

     synchronized(this){
        this.count += value;
     }
  }
```

- Synchronized Block
  - One thread can execute that block at a time.

Synchronized Example

```
public class Counter{
      long count = 0;
      public synchronized void
            add(long value){
        this.count += value;
      }
}


public class CounterThread extends Thread{
      protected Counter counter = null;
      public CounterThread(Counter counter){
         this.counter = counter;
      }
      public void run() {
            for(int i=0; i<10; i++){
                  counter.add(i);
            }
      }
}
```

```
public class Example {
      public static void main(String[] args){
            Counter counter = new Counter();
            Thread  threadA = new
                        CounterThread(counter);
            Thread  threadB = new
                        CounterThread(counter);
            threadA.start();
            threadB.start();
      }
}
```

Memory Faults

- Dynamic memory access and allocation can cause particular types of faults.
  - Null pointer dereferencing
  - Illegal access
  - Memory leaks
- Can lead to memory corruption, exhaustion, incorrect results, or illegal access to data.
- Hard to detect when testing. May not cause a failure immediately.

Memory Fault - Example

```
if(c==’+’){
     *dptr = ‘ ‘;
}else if(c==’%’){
     /* Case 2: ‘%xx’ is hex for
                    character xx */
     int digit_high = Hex_Values[*(++eptr)];
     int digit_low = Hex_Values[*(++eptr)];
     ...
}
```

- Increments pointer twice without checking for buffer termination.
- If ‘%x’ is fed as input, program scans beyond end of input string.
- Can corrupt memory.
  - Failure may not occur until that memory is *used*.

Memory Faults

- If deallocation is required (or allowed):
  - Deallocating memory still accessible through pointers may result in *dangling pointers*.
  - Failing to deallocate memory that has become inaccessible can cause *memory leaks*.
- Many modern languages limit memory faults by preventing explicit allocation and deallocation, automatically checking for array index and null pointer access.

Program Analysis

- Static Analysis
  - Exhaustively analyzes the source code and verify properties over all possible executions.
  - Prone to false alarms.
  - Can include infeasible paths.
- Dynamic Analysis
  - Use execution traces to verify properties.
  - Do not include infeasible paths.
  - Cannot examine the execution space exhaustively.

Efficiency Vs. Accuracy

- Two directions of trade-off:
  - Examine a summary of all possible behaviors.
    - Pessimistic inaccuracy
    - Leads to false alarms
    - Common in static analysis
  - Examine a sampling of possible behaviors.
    - Optimistic inaccuracy
    - Leads to incomplete results
    - Common in dynamic analysis

Static Analysis

Symbolic Execution

- Bridge between complex program behavior and analyzable logical structures.
  - Enables complex analyses of programs through abstraction to a model of execution.

Program Execution

- Execute the program with actual values.
- Statements compute new values for variables.
- Program state can be characterized by the values of variables.

Symbolic Execution

- Execute the program with symbolic values
- Statements compute new symbolic expressions
- Program state can be characterized by predicates made of symbolic expressions

Symbolic Execution in Analysis

- Can be used to form proofs of correctness.
  - Identify pre/post-conditions, invariants, and path conditions.
  - Solve constraints over the gathered state predicates.
  - Very expensive and to difficult to apply widely.
- Very effective at finding limited classes of faults - i.e., memory/concurrency issues.
  - Do not require complete specifications.
  - Fold the state space

Symbolic Testing

- Execution with symbolic values can be applied like in testing.
  - Values of variables summarized to a symbolic set based on context of analysis.
  - Values of a pointer: {null, not null, invalid, unknown}
  - Other variables may be represented by a constant.
    - Or left out entirely.
- Explore paths, searching for violations of the property of interest.

Symbolic Testing

- Reduce number of possible paths by exploring all paths to a pre-set depth or pruning paths.
  - Heuristics based on likelihood that a path is executable and leads to a potential failure.
- If not enough information is retained to determine the outcome of a branch, either choose one or take both.

Analysis Sensitivity

- Path-sensitive analysis
  - May obtain different symbolic state by reaching a concrete state through different paths.
- May be context-sensitive.
  - Explores execution through different procedure call and return sequences.
- Combination is a strength:
  - Can produce a detailed warning.
  - Cost can be reduced by memoizing entry and exit conditions.

False Alarms

- Abstraction can lead to situations where a “fault” is not possible.
  - Problem with 0 loop executions, but loop always executes once.
- False alarms degrade trust in tool.
- To reduce issues:
  - Suppress warnings that have previously been marked as false.
  - Prune execution paths whose conditions are too complex.
  - Prioritize warnings by likelihood + severity.

Summarizing Execution Paths

- Pruning paths can lead to incompleteness.
- Alternative - fold the state space down to a manageable size.
    - Build a FSM with states abstracting data values.
    - Operations cause transitions between states.
- Summarizes executions of the system.

Pointer Analysis Example

- Values:
  - invalid, may-be-null, not-null.
- Allocation transitions may-be-null, invalid to not-null.
- Deallocation transitions not-null to invalid.
  - Deallocation in may-be-null is a potential misuse.
  - Dereference in may-be-null or invalid is a potential misuse.
- Testing a pointer for not-null triggers transition from may-be-null to not-null.

Summarizing Paths

- Important choice - whether to merge states obtained along different execution paths.
  - Data flow techniques merge all states encountered at a program location.
  - Finite state verification techniques are path sensitive and do not merge states.
  - Merging shrinks state space, but loses context.
- Keeping context information reduces false alarms, but increases cost of analysis.

Dynamic Analysis

Dynamic Analysis

- Analysis of actual program executions.
  - Execute test cases.
  - Monitor execution to analyze behavior with respect to certain properties of interest.
    - Such as potential memory corruption.
  - *Instruments* the program with additional code to collect information about the execution.
- Amplifies the usefulness of test execution
  - Can detect issues even if testing does not result in failure.

Spot the Fault

```
int main (int argc, char *argv[]){
     char pre[] = “2B2B2B2B2B”;
     char subject[] =
               “AndPlus+%26%2B+%0D%”;
     char post[] = “26262626”;
     char *outbuf = (char *) malloc(10);
     int return_code =
          cgi_decode(subject,outbuf);
     return_code =
          cgi_decode(argv[1], outbuf);

     ...
}
```

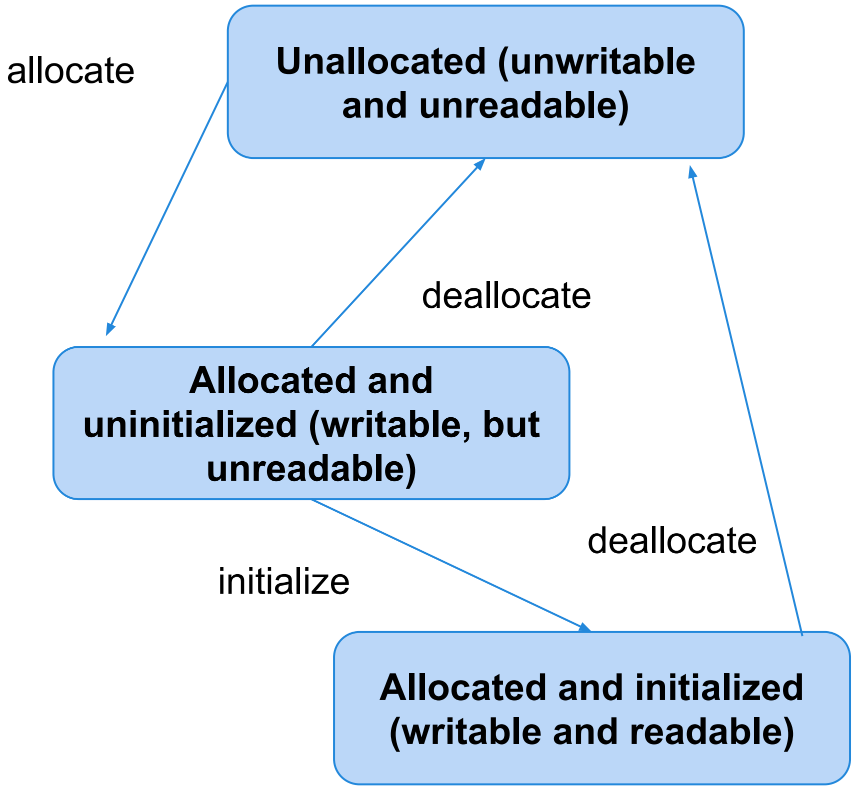
```
...

[E] ABWL: Late detect array bounds write
{1 occurrence}
Memory corruption detected, 14 bytes at
0x00e74b02
Address 0x00e74b02 is 1 byte past the end
of a 10 byte block at 0x00e74af8
Address 0x00e74b02 points to a malloc'd
block in heap 0x00e70000
63 memory operations and 3 seconds since
last-known good heap state

Detection location - error occurred before
the following function call
printf [MSVCRT.dll]
```

Memory Analysis

- Instrument program to detect memory access.
- Track status of memory locations.
- Flag incompatible accesses.
    - Write/read when memory is unallocated.
    - Read when uninitialized.
- Can check array bounds
    - Add memory blocks before/after array with state “unallocated”.



Detecting Memory Leaks

- Garbage Detectors detect memory leaks.
  - Identify and free unused memory locations.
- Recursively follow pointers from the data and stack segments into the memory heap.
- Mark all referenced blocks.
  - If a block is allocated, but no longer references, are potential memory leaks.

Lockset Analysis

- Often too difficult to detect for testing and static analysis, but can be handled with dynamic analysis.
- Data races can be prevented using a *locking discipline*.
  - Every shared variable must be protected by a mutual exclusion lock.
- Lockset analysis reveals potential data races by detecting violation of the locking discipline

Lockset Analysis

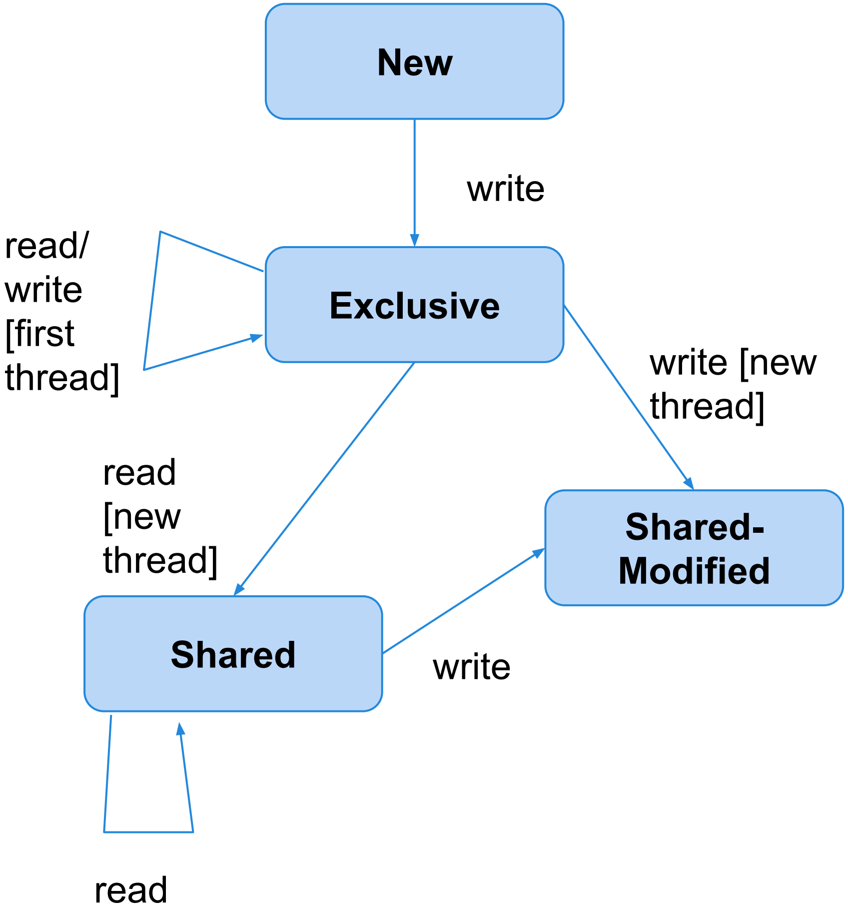
- Identifies the set of mutual exclusion locks.
  - At start: lockset for each variable associated with all known locks.
  - At access: update lockset to be only those currently also held by the accessing thread.
  - At end: lockset indicates the set of locks that were always held by threads when accessing the variable.
    - Empty = locking violation

Example Lockset Analysis

Thread	Program Trace	Locks Held	Lockset(x)
		{}	{lck1, lck2}
thread A	lock(lck1)		
		{lck1}	
	x = x+1;		
			{lck1}
	unlock(lck1)		
		{}	
thread B	lock(lck2)		
		{lck2}	
	x = x+1;		
			{}
	unlock(lck2)		
		{}	

Extending Lockset Analysis

- Delay analysis until initialization is complete.
  - State not modified until a second thread attempts to read or write.
- Violations reported if they occur in the shared-modified state.
  - Multiple readers are allowed.
  - Distinguish locks held in all accesses from locks held in write accesses.



Extracting Behavior Models

- Executing a test reveals information about the program.
    - This information can be used to synthesize a model that describes those - and other - executions.
- Models can be used:
    - As oracles (build a model from “correct” executions, apply to future tests)
    - To evaluate thoroughness of testing (coverage)
    - For program analysis.
    - During debugging (fault localization)

Extracting Predicates

- Can extract predicates on the values of variables at selected execution points.
- Example - AVL Tree insertion operation:

```
father > left
father < right
diffHeight is one of {-1,0,1}
```

- Allows us to examine and understand program behaviors.
  - Checks thoroughness of the test suite.

Building Predicates

Start with initial predicates generated from templates.

Over any variable x:	
constant	x = a
uninitialized	x = uninit
small value set	x = {a,b,c} for a small set of values
Over two numeric variables, x and y:	
linear relationship	y = ax+b
ordering relationship	x <= y, x < y, x = y, x !=y
functions	x = fn(y)
Over the sum of two numeric variables, x and y:	
in a range	x + y >= a, x+y <= b, a <= x+y <= b
nonzero	x + y != 0

Building Predicates

- Instantiating every template for every variable can get very expensive.
- Can instead indicate points in the program where we want to extract predicates, and the variables we want to examine at that point.

```
…
node.height = max(
    height(node.left),
    height(node.right))
    + 1;
recordData(node,
    node.left,node.right);
return node;
}
```

Building Predicates

- Eliminate generated predicates violated during test execution.

```
static void testCaseSingleVals(){
    AvlTree t = new AvlTree();
    t.insert(5);
    t.insert(2);
    t.insert(7);
}

static void testCaseRandom(int n){
    AvlTree t = new AvlTree();
    for(int i =1; i < n; i++){
        t.insert((int) Math.round(
                Math.random()*100));
    }
}
```

Model: testCaseSingleVals

- father, one of {2,5,7}
- left == 2
- right == 7
- leftHeight == rightHeight == diffHeght
- leftHeight, rightHeight == 0
- fatherHeight, one of {0,1}

Model: testCaseRandom

- father, left >= 0
- right > father > left
- left < right
- fatherHeight >= 0
- leftHeight, rightHeight >= 0
- fatherHeight > leftHeight, rightHeight, diffHeight
- rightHeight >= diffHeight
- diffHeight, one of {-1, 0, 1}
- leftHeight - rightHeight + diffHeight == 0

Building Predicates

- Representation of what has been observed.
  - More executions will refine the predicates.
- Some predicates are coincidental.
  - Associate probability of coincidence with predicates.
    - Estimated by number of executions where a predicate is tested.
      - Probability of 0.5 if verified by one execution.
      - Probability of 0.5^n if verified by *n* executions.
  - Omit predicates that do not meet a threshold.

Daikon Example

- Daikon is a tool that detects predicates from Java, C, C++, C#, Perl, and Eiffel programs.
  - http://plse.cs.washington.edu/daikon/
- Follows the process outlined:
  - Form an initial set of predicates from templates.
  - Execute the code and take observations.
  - Learn the “likely” predicates from these executions.

We Have Learned

- Testing is not enough to find faults that are only triggered under specific or non-deterministic circumstances.
    - Memory leaks
    - Data races
    - Deadlock
- Program analysis can be used to ensure that the SUT is free from certain types of faults.

We Have Learned

- Static Analyses
  - Based on symbolic execution.
  - Summarize execution paths.
  - Exhaustively examine a portion of the state space for violations of properties.
- Dynamic Analyses
  - Observe executions of the system.
  - Compare collected information to a model of “ideal” behavior for property of interest.
  - Augment testing with targeted analyses.

Next Time

- Automated Test Case Generation

- Homework:
  - Reading Assignment 3 - due tonight
  - Assignment 4 - due April 11.