EECS 481 — Software Engineering
Winter 2019 — Exam #1

- **Write your UM uniqname and UMID and your name on the exam.**
- There are ten (10) pages in this exam (including this one) and seven (7) questions, each with multiple parts. Some questions span multiple pages. If you get stuck on a question, move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your two page-sides of notes.
- Even vaguely looking at a cellphone or similar device (e.g., tablet computer) during this exam **is cheating**.
- Please write your answers in the space provided on the exam. Clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. We may deduct points if your solution is far more complicated than necessary.
  - *Good Writing Example:* Testing is an expensive activity associated with software maintenance.
  - *Bad Writing Example:* Im in ur class, @cing ur t3stz!1!
- If you leave a non-extra-credit portion of the exam blank, **you will receive one-third of the points for that small portion (rounded down) for not wasting time.**

UM uniqname: ANSWER KEY

UM ID: ANSWER KEY

Name (print): ANSWER KEY

UM uniqname: (yes, again!) ANSWER KEY

UM ID: (yes, again!) ANSWER KEY

Problem	Max points	Points
1 — Software Process Narrative	13	
2 — Test Inputs and Coverage	18	
3 — Short Answer	18	
4 — Mutation Testing	16	
5 — Dataflow Analysis	20	
6 — Quality Assurance Analyses	15	
Extra Credit	0	
TOTAL	100	

How do you think you did? _____

1 Software Process Narrative (13 points)

(1 pt. each) Read the following narrative. Fill in each ____ blank with the single *most specific or appropriate* corresponding concept from the answer bank. (Each ____ blank does have a corresponding answer.) Each option from the answer bank will be used *at most once*.

A. Alpha Testing	B. Beta Testing	C. Call-Graph Profile	D. Comparator
E. Conditional Breakpoint	F. Dataflow Analysis	G. Development Process	H. Effort Estimation
I. Formal Code Inspection	J. Instrumentation	K. Integration Testing	L. Mocking
M. Oracle	N. Passaround Code Review	O. Perverse Incentive	P. Priority
Q. Resolution	R. Severity	S. Software Metric	T. Spiral Development
U. Threat to Validity	V. Triage	W. Watchpoint	X. Waterfall Model

Unrealistic Software is developing a hot new online multiplayer game, *ApexUnknown's Defense League of OverGrounds Legendary Pals 76*.

__G__ Managers decide to plan the organization of various software development activities into distinct phases.

__T__ Managers decide to focus on the construction of an increasingly-complex series of working prototypes.

__S__ Managers decide to use the Maintainability Index to assess the code because it is built in to Visual Studio.

__U__ The Maintainability Index misleads the managers by pointing out code that is large, rather than code that is truly difficult to maintain.

__O__ Developers are instructed to lower the Maintainability Index at any cost, resulting in methods that are very short but almost impossible to comprehend or maintain.

__A__ Developers are worried about the quality of the software prototype. Internally, they use a tool to randomly generate and run inputs.

__M__ Developers decide, for simplicity, that at the very least their software should not crash or throw exceptions on those random inputs.

__V__ A new defect report comes in. QA finds it to be a valid report and not a duplicate. Developers are instructed to address it.

__P__ Management decides that the new defect report must be addressed immediately: the business cost of not fixing it is deemed to be too high.

__H__ Management asks developers to predict how long it will take to localize and fix the defect.

__J__ Developers modify the code to print out which lines are visited during execution.

__W__ The bug report mentions the corruption of a particular memory address, so the developer uses a debugger to halt execution whenever the value at that address changes.

__Q__ However, the code works fine when the developers run it. They cannot reproduce the defect, so they move on to other tasks.

2 Test Inputs and Coverage (18 points)

(4 pts.) Give a smallest test suite for `lamarr()` below that maximizes *statement coverage* for statements S_1 through S_8. (A test for `lamarr()` is a value for `x` and a value for `y`. A test suite is a set of tests.)

```
1  void lamarr(bool x, bool y) {
2    S_1;
3    if (x == y)    S_2;
4    else           S_3;
5    if (x || y)    S_4;
6    else           S_5;
7    if (x && y)    S_6;
8    else           S_7;
9    S_8;
10 }
```

{x=True, y=True} covers S_2, S_4 and S_6.
{x=False, y=False} covers S_2, S_5, and S_7.
{x=False, y=True} covers S_3, s_4, and S_4. Three tests cover all statements.

(2 pts.) What is the maximum *path coverage* possible for `lamarr()` in practice? (Express your answer as a fraction.)

There are $2^3 = 8$ *paths* through the program from the three serial if statements. Each path can be viewed as a string of three Then and Else branches: TTT, TTE, TET, etc.
{x=True, y=True} covers TTT.
{x=True, y=False} covers ETE.
{x=False, y=True} covers ETE. Note, this is the same path as the last one!
{x=False, y=False} covers TEE.
So even though there are 8 paths and 4 possible inputs, only 3 paths can be covered. So the fractional answer is 3/8.

(4 pts.) Give a smallest test suite for `lamarr()` that maximizes *path coverage*.

Either { TT, TF, FF } or { TT, FT, FF} works (see previous).

(4 pts.) Suppose we define full *boolean coverage* to require that every boolean variable or expression evaluate to true (e.g., on one input) and also to false (e.g., on another input). Give a smallest test suite for `hedy()` below that maximizes *boolean coverage*.

```
1 void hedy(int a, int b) {
2   bool p, q;
3   p = a < b;
4   q = a > b;
5   if (p || q)    S_1;
6   else           S_2;
7 }
```

There are three booleans: $a < b$, $a > c$ and $p||q$. Equivalently, there are three booleans: p, q and $p||q$. We need to make each one both true and false at some point. We could do that with $2^3 = 8$ inputs, but we can also do much better.

{ a=1, b=2 } makes p=True, q=False and p||q=True.

{ a=2, b=1 } makes P=False, q=True and p||q=True.

{ a=1, b=1 } makes p=False, q=False and p||q=False.

So we can cover all of the booleans with three inputs.

(4 pts.) Give a smallest test suite for `hedy()` that maximizes *branch coverage* but *not* boolean coverage.

Since there is only one branch, we only need two inputs to maximize branch coverage.

{ a=2, b=1 } makes the branch visit S_1.

{ a=1, b=1 } makes the branch visit S_2.

3 Short Answer (18 points)

(a) *(4 pts.)* Support or refute the claim that spectrum-based fault localization (such as the Tarantula tool), which is based on differences in coverage between passing and failing runs, would be a useful technique for localizing memory corruption bugs.

Likely "refute". Spectrum-based fault localization is based on measuring the coverage obtained on passing test cases, the coverage obtained on failing test cases, and subtracting those results. It assumes that lines visited on the bad runs and lines visited on the good runs are likely to be different. Suppose you have an array of size 10 and you ask the user which element to access. When the user says 5, everything works fine. When the user says 15, everything fails. But the lines visited are the same in both! Memory corruption bugs are notorious for "showing up" far removed from where the problem actually is: the symptom and the cause are separated, and they tend to be non-deterministic. All of those make spectrum-based fault localization unlikely to work. A technique like CHESS or Eraser would likely be better.

A common mistake was to restate the definition or assumptions of SBFL (e.g., compares coverage on good runs to bad runs, and if the bug causes the program to crash immediately, the coverage difference will pinpoint the bug) *without* linking it to "memory corruption bugs" in a specific way. If your answer text would be the same if you replaced "memory bug" with "logical bug", full credit is unlikely.

(b) *(3 pts.)* For each of *mocking*, *unit testing* and *integration testing*, describe a defect or development scenario for which that technique is likely to work well.

Mocking could be useful in a situation where the full implementation of a method is not yet available. For example, suppose you are testing an embedded system, like car antilock brake software, but you don't have the whole physical brake device built. You could use mocking to test it.

Unit testing can be useful for self-contained data structures, like container classes. You can write unit tests that check internal relationships (e.g., that push push pop pop has the expected LIFO behavior for a stack) that don't depend on external context.

Integration testing is useful when you have different units or libraries developed by separate groups and you want to make sure that they work well together. For example, you might have one group write a game engine and another group write a sound library, and you could use integration testing to make sure they work together.

(c) *(3 pts.)* Explain the relationship between *reopened* bug reports and *regression testing*. What process would you suggest to minimize reopened bug reports?

Regression testing checks to make sure that new commits do not re-trigger bugs that were previously identified and fixed. The first time a bug is fixed, a regression test could be created to test it. Later, when new commits are made, they can be checked against that test. If you fail to do this, you might end up with the bug reappearing without you

noticing it, at which point users will report it later, at which point the old bug report will be reopened. To avoid this, developers are encouraged to take the time to make regression tests when they fix bugs (and also to link those regression tests to the bug tracking system, so that if the test fails later devs know which bug report to reopen, etc.).

A common mistake involved implicitly or explicitly asserting that failed regression tests *cause* reopened bug reports. Regression testing is typically done by developers internally (cf. continuous integration testing, pull requests, etc.) *before* bug reports get involved. In modern SE, a change that would fail a regression test typically "does not get checked in" rather than "gets checked in and then has a bug report re-opened against it".

(d) *(4 pts.)* Support or refute the claim that modern passaround code review should be performed *after* the new code is subjected to unit and regression testing if goals of code review are taken to be "code improvement" and "reduce costs".

Likely "refute". Running unit and regression tests is expensive, and many companies do them only after the change has passed some cursory human review. See the flowchart near Slide 24 of the Code Inspection and Review Slideset, for example. By contrast, linting and some cheap static analyses may be done before human code review. Later in that same slideset we see that "code improvement" involves aspects like "better coding practices" and "improving readability" — areas that regression and unit testing really don't help you with. So doing that testing early does not help code improvement and activley hurts reducing costs. However, students can marshal very good arguments (often citing the slides and papers on expectations and outcomes of code review) for either side.

(e) *(4 pts.)* Choose a modern passaround code review requirement from a large company (such as Facebook's "all changes have at least two reviewers" or Google's "you must have a readability badge in the programming language") and support or refute the claim that that requirement aids the goals of "defect finding" and "knowledge transfer".

Many options. Here are some examples:

Google's "you must be experienced with this language" is likely to help with "knowledge transfer" because "knowledge transfer" involves things like pointers to internal and external documentation (e.g., library APIs) (see near Slide 38 of the Code Review slideset). Other developers familiar enough with the language to get the badge are more likely to be able to point you to the right documentation.

Facebook's "two reviewers" approach is likely to help you with "defect finding". Linus's Law suggests that "given enough eyes, all bugs are shallow', and the evidence from formal code inspection suggests that techniques with bigger teams can help find bugs (see the last 10 slides of that slideset).

Full credit required addressing both "defect finding" *and* "knowledge transfer", as per the question.

4 Mutation Testing (16 points)

Consider the following implementation of ascending bubble sort and some associated test inputs, oracles and suites:

```
1  def BubbleSort(arr):
2    n = len(arr)                    # reminder: len([5,6]) = 2
3    for i in range(n):              # reminder: range(3) = [0,1,2]
4      for j in range(i):
5        if (arr[j] >= arr[j+1]):
6          arr[j], arr[j+1] = arr[j+1], arr[j] # cool python swap
7
8  testInput1 = [11, 22]             # oracle1 = [11, 22]
9  testInput2 = [88, 66, 77]         # oracle2 = [66, 77, 88]
10 testInput3 = [55, 33, 0, 99]      # oracle3 = [0, 33, 55, 99]
11
12 testSuiteA = [ testInput1 ]
13 testSuiteB = [ testinput2, testInput3 ]
```

(6 pts.) Suppose the only mutation operator available to you is *single variable renaming*: on line A change the Bth instance of variable C to variable D. For example, one mutation might be "on line 6 change the 3rd instance of `j` to `i`". You may only mutate the method body (lines 2–6), not the tests. Give an example of a single mutation such that both `testSuiteA` and `testSuiteB` kill that mutant (i.e., that mutant fails at least one test in `testSuiteA` and also fails at least one test in `testSuiteB`) *without* any Python runtime errors (i.e., no mutants that reference undefined variables and no index out of bounds accesses, etc.).

"On line 5 change the 1st instance of j to i."
That produces:

```
1  def BubbleSort(arr):
2    n = len(arr)
3    for i in range(n):
4      for j in range(i):
5        if (arr[i] >= arr[j+1]): # LOOK HERE
6          arr[j], arr[j+1] = arr[j+1], arr[j]
```

Informally, this mutation causes the first swap to "happen too often" (even if it "shouldn't" for a correct ascending order sort).

That mutation causes Tests 1 and 3 to fail, which causes both test suites to fail:

```
>>> testInput1 = [11, 22]
>>> BubbleSort(testInput1)
>>> print testInput1
[22, 11]
>>> testInput2 = [88, 66, 77]
>>> BubbleSort(testInput2)
```

```
>>> print testInput2
[66, 77, 88]
>>> testInput3 = [55, 33, 0, 99]
>>> BubbleSort(testInput3)
>>> print testInput3
[0, 55, 99, 33]
```

(6 pts.) You are still limited to single variable renaming mutations. Give an example of a single mutation such that *one of* `testSuiteA` or `testSuiteB` kills the mutant *but not both* (with no Python runtime errors) — and say which one kills the mutant.

"On line 6, change the 1st instance of j to i."
That produces:

```
1 def BubbleSort(arr):
2   n = len(arr)
3   for i in range(n):
4     for j in range(i):
5       if (arr[j] >= arr[j+1]):
6         arr[i], arr[j+1] = arr[j+1], arr[j] # LOOK HERE
```

Informally, this mutation causes every swap to just copy the 0th element around. Since Test 1 does not ever invoke swaps (i.e., it never gets to line 6) it is not killed by this mutant. By contrast, Tests 2 and 3 are:

```
>>> testInput1 = [11, 22]
>>> BubbleSort(testInput1)
>>> print testInput1
[11, 22]
>>> testInput2 = [88, 66, 77]
>>> BubbleSort(testInput2)
>>> print testInput2
[88, 88, 88]
>>> testInput3 = [55, 33, 0, 99]
>>> BubbleSort(testInput3)
>>> print testInput3
[55, 55, 55, 55]
```

(4 pts.) You may now consider any and all mutation operators. Give two advantages of higher-order mutation (i.e., multiple mutations per mutant) and two disadvantages of higher-order mutation. Use at most four sentences total.

+ Higher-order mutants can simulate more realistic human bugs (coupling effect hypothesis).

+ Higher-order mutants can tease apart test suites of similar quality.

+ Higher-order mutants can reduce the number of mutants needed, reducing the cost of mutation testing (if you're careful).

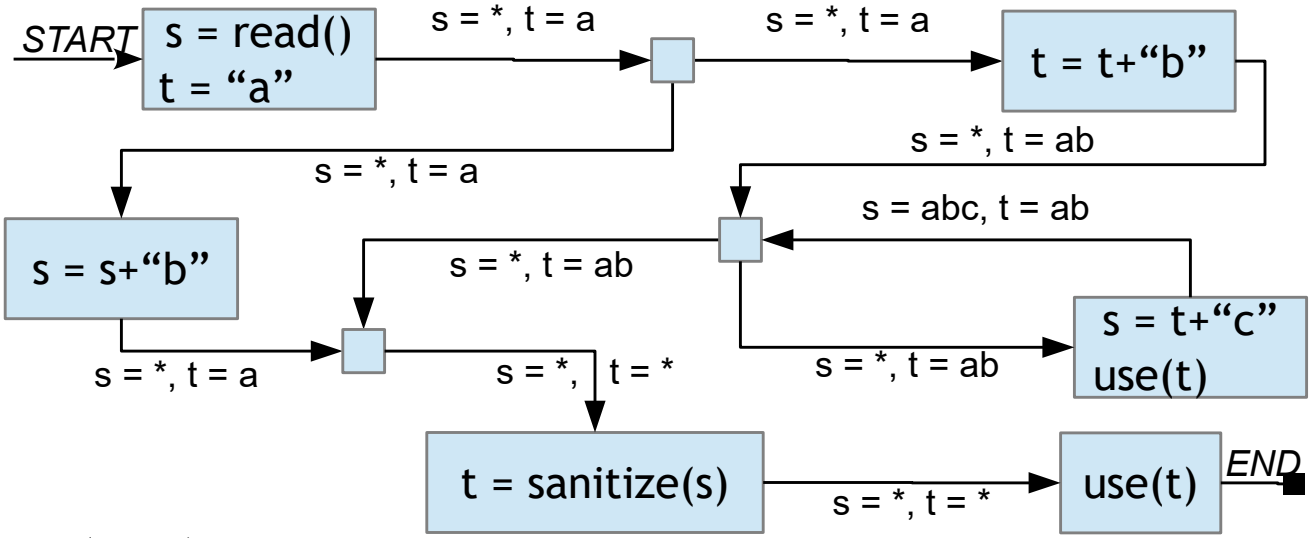
- Higher-order mutants can be more expensive to create (more coin tosses and passes per mutant created, potentially).

- Higher-order mutants can end up being killed by everything if you are not careful (which means they do not tell apart test suites).

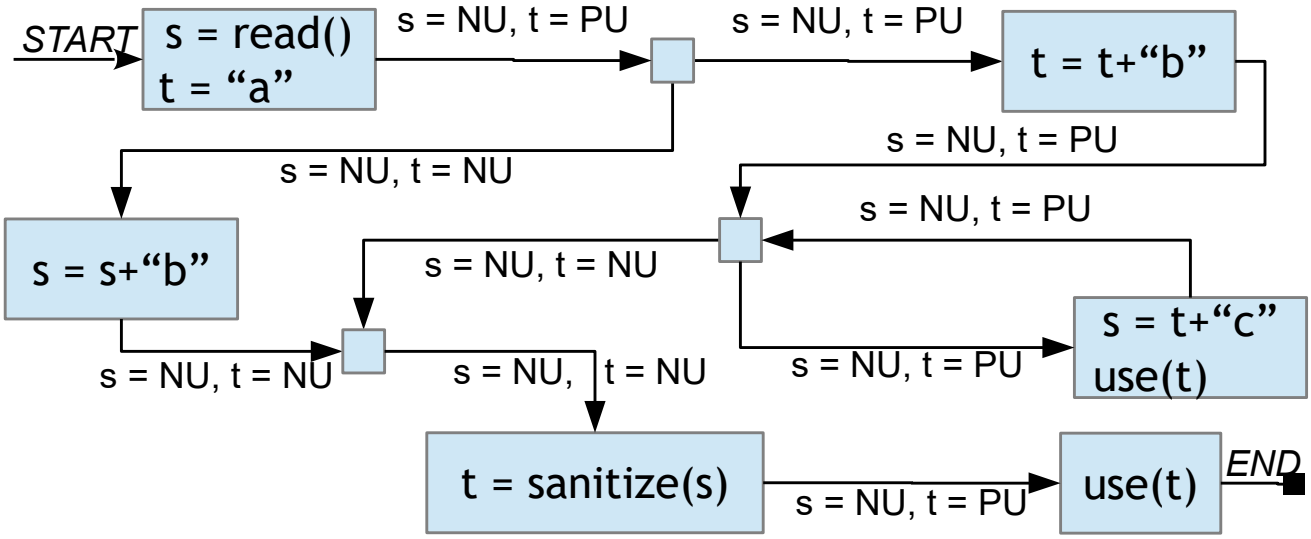
5 Dataflow Analysis (20 points)

Consider a *constant propagation* dataflow analysis for string values. We associate with each variable an analysis fact: either * ("the variable holds a value, but our analysis cannot be certain which value"), # ("this point has not yet been reached by our analysis") or a particular constant string ("at this point in the program, we are certain this variable holds exactly `hello`"). We also include simple string concatenation. For example, if `x = 'ada '` before the statement `y = x+'lovelace'`, we conclude that `y = 'ada lovelace'` after it.

(10 pts.) Complete this *forward* string constant propagation dataflow analysis below by filling in each blank for both string variables `s` and `t`.



(10 pts.) We also want to known if uncontrolled string values may be used un-sanitized (i.e., in an unsafe manner). It is a bug if *even part* of the string passed to the `use()` function comes from an un-sanitized, unsafe read. We associate with each variable either PU for "possibly used *unsafely* later" or NU for "not used *unsafely* later". For the same program, complete this *backward* dataflow analysis by filling in each blank for both string variables.



6 Quality Assurance Analyses (15 points)

(5 pts.) CHESS and Eraser are dynamic analysis approaches that find bugs in concurrent or multi-threaded programs. How are they affected by test quality? List one other advantage of each approach and one other disadvantage of each approach.

CHESS and Eraser both involve running the program on already-existing inputs. For example, if you don't have any inputs that visit the use of a shared variable, Eraser will never be able to compute any lockset information about that variable, so it will never be able to meaningfully warn you about race conditions on it. The story is simialr for CHESS: both of these tools rely on you already having high-coverage test inputs.

+ CHESS can help find arbitrary heisenbugs, not just locking-based race conditions. + CHESS can be very efficient (e.g., uses constraint solving to prune the search space). + CHESS can scale to large programs (e.g., integrated into Microsoft development).

- CHESS requires control of the scheduler (so that it can try out different thread interleavings). You may not have that control.

- CHESS can still suffer from state-space explosion problems.

+ Eraser's algorithm and results are easy to understand (compared to other tools).

+ Eraser can find locking-related race conditions, and locks are the most common kind of concurrency control.

- Eraser needs to understand what locks are, which means you need to integrate with system libraries.

- Eraser still has non-trivial overhead.

- Eraser can have false positives where you do not always use the same lock to guard the same shared variable yet you don't actually have a race condition.

(5 pts.) List two scientific or algorithmic challenges (e.g., a theoretical or "whiteboard" concern) associated with deploying a static bug-finding tool like FindBugs at a large company like Google. List three "software engineering" (e.g., process, people, management, etc.) challenges associated with deploying such a tool at such a company.

A. Some static analysis algorithms may not scale to real-world code. FindBugs chose not to use some of the more impressive bug-finding algorithms at Google for efficiency reasons.

A. The halting problem is relevant here: any static analysis will inevitably have false positives or false negatives or both on real-world software (which will enough to include loops).

B. Humans hate false positives. The tools must often be tweaked to avoid them or developers will stop using them.

B. The tools must often be integrated with existing software development practices, such as code review or continuous integration testing.

B. For some more exotic examples, the optional Coverity paper lists many of these: you can't count on them using your favorite build process; the person responsible for running the tool is not the one punished if things break; you can't count on consistent header files; companies may not care about bugs; and so on. If you haven't read it, go back and take a look.

(5 pts.) Support or refute the claim that an effective approach to finding security defects (e.g., leaked passwords, buffer overruns, etc.) would be to identify frequently-executed regions of code via sampling-based profiling and apply formal code inspection to them.

Likely "refute". While formal code inspection is a great way to actually find bugs (see last 10 slides of that lecture), it has to be targeted to the right area of code. Sampling-based profiling finds the methods that you execute most frequently — it is likely to identify your innermost loop. But that area is often not where the security concern is. For example, imagine you are a webserver that sends big music files but also does authentication. You spend most of your time sending the big music files (so profiling will point to that) but the password-leak security error is almost certainly near the authentication code, which only runs once and is fairly quick. Event-based profiling (e.g., that highlights when you make security-critical calls) might work, but sampling-based profiling highlights a mismatch between "the code you spend the most time on" and "the code that is the most dangerous": they probably aren't the same!

7 Extra Credit **(1 pt each; we are tough on reading questions)**

(Feedback) What is one thing you would change about this class for next year? What is one thing you like about this class?

snowflakes that stay on my nose and eyelashes // silver-white winters that melt into spring

(Free/Participation) Make up one "believable" (but possibly outlandish) claim about the class, professor or course staff. If we get enough interesting responses, we'll post them.

Example: "The professor has written a romance novel under a false name."

(Your Choice Reading 1) Identify one of the optional readings. Write a sentence about it that convinces us that you read it critically. (Our subjective judgment applies here — sorry!).

(Your Choice Reading 2) Identify a different optional reading. Write a sentence about it that convinces us that you read it critically. (Our subjective judgment applies here — sorry!).

(My Choice Reading 1) In "Automated Whitebox Fuzz Testing", for what sort of "well-formed input" did they need to use the whitebox information to generate inputs?

Many programs take inputs that are governed by context-free grammars. For example, a calculator program might take in valid arithmetic expressions like $1+(2*3)+5$. If you just make up random inputs, you're likely to get the parentheses wrong. So they look inside the grammar (the rules for +, -, *, parentheses, etc.) and use that to construct some well-formed inputs.

(My Choice Reading 2) In "A Decade of Software Model Checking with SLAM", give one key technique used by that static analysis to find API violations in software such as device drivers.

They convert C programs to Boolean programs in a sound manner. They use Model Checking to look at those Boolean programs. They use automated theorem proving and counter-example guided abstraction refinement to make better and better models. They use a special specification language, SLIC, to write down good and bad behavior. They use environment models to be more precise in the real world. And so on.