CpSc 513: Course Overview

Mark Greenstreet

September 10, 2015

Outline:

- What is verification?
- A simple example: binary search
- Course mechanics
  - You will survive.
  - You'll even have fun.

What is verification?

- We use mathematical methods, mostly mathematical logic, to show that a hardware, software, or cyber-physical design has some desired properties.
  - Often, this is seen as “finding bugs.”
  - Bug finding can be very important from a safety and/or cost point of view.
  - Formal methods also allow us to build more highly optimized designs:
- The kinds of techniques that we use:
  - Boolean satisfiability (SAT)
  - SAT augmented with decision procedures for other domains (SMT)
  - Reachability computation: model checking
  - Abstraction, approximation, and refinement.

The Big Picture

	Theory, algorithms, & technology	Applications & examples
propositional ~5 weeks	SAT and SMT intro Symbolic Execution The DPLL algorithm Binary Decision Diagrams	Automatic Exploit Generation Automatic Test Generation Digital circuit equivalence checking
temporal ~5 weeks	Model checking Boolean program abstraction How SMT works Temporal logic	Microprocessor verification (Intel) Cache protocol verification Software model checking (Microsoft) More software analysis tools Verifying fault tolerance (Amazon)
continuous ~3 weeks	Linear Differential Inclusions Interval arithmetic Projectagons	Cyber–physical systems: autonomous vehicles Circuit verification State–of–the–art SW verification

Advanced topics:

Interpolants, bounded model checking (BMC), Abstraction and refinement (e.g. CEGAR), Statistical model checking, concolic execution, interactive theorem proving.

We probably won't cover all of these, but I'll give half–hour intro and overviews along with "for further reading" papers.

Binary Search

- This example is from "Programming Pearls: Writing correct programs", J.L. Bentley, *CACM*, Vol. 26 No. 12 (Dec. 1983), pages 1040-1045. (non-UBC link).
- Bentley had given courses for professional programmers at Bell (now AT&T) and IBM.
- Given an hour to solve the problem, $\sim$ 90% of programmers produced code that failed on a small set of test cases.
- Bently wrote:
  - *"I found this amazing: only about 10 percent of professional programmers were able to get this small program right."*
  - *"This exercise displays many strengths of program verification: the problem is important and requires careful code, the development of the program is guided by verification ideas, and the analysis of correctness employs general tools."*

Binary Search: outline

This section is about 20 slides long. Here's an outline (with links).

- Original code, a test case, and a specification
- Symbolic execution
  - overview and relationship to constraint solving.
  - Symbolic execution with the Z3 SMT solver
    - The first few lines of `search`
    - The `while` loop
    - When loop exits
    - The loop body
      - Unwinding the loop or using an invariant
- Symbolic execution summary

Let's try it

```
# search(key, A): search A for an instance of key.
#     A is a non-decreasing array of integers.
#     return the index of an element of A that matches key
#     or None if no such element exists.
def search(key, A):
    lo = 0
    hi = len(A)
    while(lo < hi):
        mid = (lo + hi)/2
        if(A[mid] == key):  return mid
        elif(A[mid] < key): lo = mid
        else:  hi = mid
    return None
```

Is it right?

- We could try some test cases. For example:

```
def test_it():
    A = [-17, -3, 0, 1, 2, 4, 123456789]
    if(search(2, A) == 4):  print 'Looks good.'
    else:  print 'The test failed.'
```

- Now, execute the test case(s):

```
>>> test_it() Looks good.
```

  The test passed.

- How do we know when we've performed enough tests?

Is it right?

A more formal/systematic approach:

- We want to show:
  - P1: If `search(A, key)` returns `i`, then `A[i] = key`.
  - P2: If `A` does not contain `key`, then `search(A, key)` returns `None`.
  - P3: For any `A` and `key`, `search(A, key)` eventually returns.
- We assume:
  - A1: `key` is an integer.
  - A2: Every element of `A` is an integer.
  - A3: For all $0 \leq i \leq j <$ `len(A)`, `A[i]` $\leq$ `A[j]`.
- Strategy:
  - Write down a formula for what we know at each step in the execution.
  - Check to see if every execution satisfies `P1 ... P3`.

Symbolic execution: the big picture

- Use *symbols* for the parameters of `search`.
- For each step of the execution, compute *formulas* in terms of these symbolic parameters.
- Check to see if these formulas satisfy the specification.
- Simple example:

```
def simple(a, b):
    # code                  symbolic execution
    x = a+b                 # x = a+b
    y = a*b                 # y = a*b
    z = x*x - 2*y           # z = (a+b)*(a+b) - 2*(a*b)
    return z
```

  - Does `simple` satisfy the specification `simple(a, b)` ≥ 0?
  - We simplify the formula for `z` and get `z` $=$ `a*a + b*b`.
  - If `a` and `b` are real-valued, then the specification is satisfied.

Symbolic execution by constraint solving

- We can view each statement of a program as adding constraints.
- Each constraint restricts the set of possible executions.
- A constraint solver can give us an example of an execution.
- We can add a constraint: "...and the specification is violated."
  - If the constraint solver can find a solution to the constraints, Then it's shown that the program has a bug (it can violate its specification).
  - If the constraint solver can show that there is no solution to these constraints,
    then all executions satisify the specification.
    We have verified the program.

Another way to think of constraint solving

- It's like a kid's game:
  - ★ "I'm thinking of someone in this room." (the domain)
  - ★ "This person has brown eyes." (a constraint)
  - ★ "This person is shorter than 180cm" (another constraint)
  - ★ ... ...
- The constraints are satisifiable if there is at least one person in the room who satisfies all of them.
  - When using constraints for program executions, it means that execution path is possible.
- If we add a constraint "and the specification is violated"
  - Then a solution to the constraints is an example of a bug.
  - If there is no solution, then all executions (for those constraints) satisfy the specification.

Symbolic execution with the Z3 SMT solver

```
from z3 import *                # get the z3 stuff
s = Solver()                    # s records constraints, finds solutions, etc.
```

You can download z3 from

https://github.com/z3prover/z3/wiki

Go to the `github` link for source code and continue. OSX users can follow the instructions for linux distros.

The solver object is an SMT solver. Here, I constructed one with the default configuration parameters. This gives us a pretty rich set of theories. In the following code, I will declare symbolic constants and then add symbolic constraints to the solver.

Symbolic execution with the Z3 SMT solver

```
from z3 import *                # get the z3 stuff
s = Solver()                    # s records constraints, finds solutions, etc.

# 0: def search(key, A):
key = Int('key')                # declare a symbolic variable for key
```

A bit of explanation may help here. Note that where working with *both* z3 symbolic constants and Python variables. The constructor, `Int('key')` creates a z3 symbolic constant whose name is the string `'key'`. It's a *constant* because for any propositional formula, it has the same value in all occurrences in that formula. When we ask z3 if a formula is satisfiable, it determines if there is a value for this constant (and all other constants appearing in the formula) such that the formula is satisfied. We also have the Python variable, `key`. It is a reference to the object returned by the z3 constructo, `Int('key')`. In this case, I'm using the same name for the z3 constant and the Python variable. Sometimes, that's the clearest approach. I'll give some examples later where we can have Python variables that correspond to z3 expressions.

Symbolic execution with the Z3 SMT solver

```
from z3 import *                # get the z3 stuff
s = Solver()                    # s records constraints, finds solutions, etc.

# 0: def search(key, A):
key = Int('key')                # declare a symbolic variable for key
  A = Array('A', IntSort(), IntSort())  # a symbolic array
lenA = Int('len(A)')            # we'll add a variable for len(A)
s.add(lenA >= 0)                # array lengths can't be negative
```

Here, I had to decide how to represent Python's built-in `len` function. I could probably make it an uninterpreted function from arrays to integers, but I don"t know enough z3 to do that. Instead, I'm using my knowledge that array lengths are non-negative integers, and creating a new variable with that property. In particular, for any array we want to model, there is a value for `lenA` such that `lenA >= 0` and `lenA = len(A)`. The method `s.add(`*`constraint`*`)` adds *constraint* to the set of constraints that the solver is trying to satisfy.

Symbolic execution with the Z3 SMT solver

```
from z3 import *                # get the z3 stuff
s = Solver()                    # s records constraints, finds solutions, etc.

# 0: def search(key, A):
key = Int('key')                # declare a symbolic variable for key
  A = Array('A', IntSort(), IntSort())  # a symbolic array
lenA = Int('len(A)')            # we'll add a variable for len(A)
s.add(lenA >= 0)                # array lengths can't be negative
# Now, I'll add the constraint that A is non-decreasing
i,j = Ints(('i', 'j'))          # extra variables for the analysis
s.add(ForAll((i, j),            # A is non-decreasing
              Implies(And(0 <= i, i <= j, j < lenA),
              A[i] <= A[j])))
```

Z3 provides incomplete support for the quantifiers `ForAll` and `Exists`. For many formulas (including this example), z3 can just spot places where the quantified expression is relevant. Z3 then creates the particular instance for that term. This is a rather hand-waving explanation. The technical term is "model-based quantifier instantiation".

Symbolic execution with the Z3 SMT solver

```
from z3 import *                # get the z3 stuff
s = Solver()                    # s records constraints, finds solutions, etc.

# 0: def search(key, A):
key = Int('key')                # declare a symbolic variable for key
  A = Array('A', IntSort(), IntSort())  # a symbolic array
lenA = Int('len(A)')            # we'll add a variable for len(A)
s.add(lenA >= 0)                # array lengths can't be negative
# Now, I'll add the constraint that A is non-decreasing
i,j = Ints(('i', 'j'))          # extra variables for the analysis
s.add(ForAll((i, j),            # A is non-decreasing
              Implies(And(0 <= i, i <= j, j < lenA),
              A[i] <= A[j])))

# 1: lo = 0
lo = Int('lo')
s.add(lo == 0)

# 2: hi = len(A)
hi = Int('hi')
s.add(hi == lenA)
```

Now that I've filled a slide with code

Let's try it:

```
>>> # the stuff from the previous slide
>>> s.check()
sat
>>> s.model()
[lenA = 0,
lo = 0,
hi = 0,
A = [0 -> 0, else -> 0],
k!1 = [0 -> 0, else -> 0]]
>>>
```

So far, so good. The particular solution has:

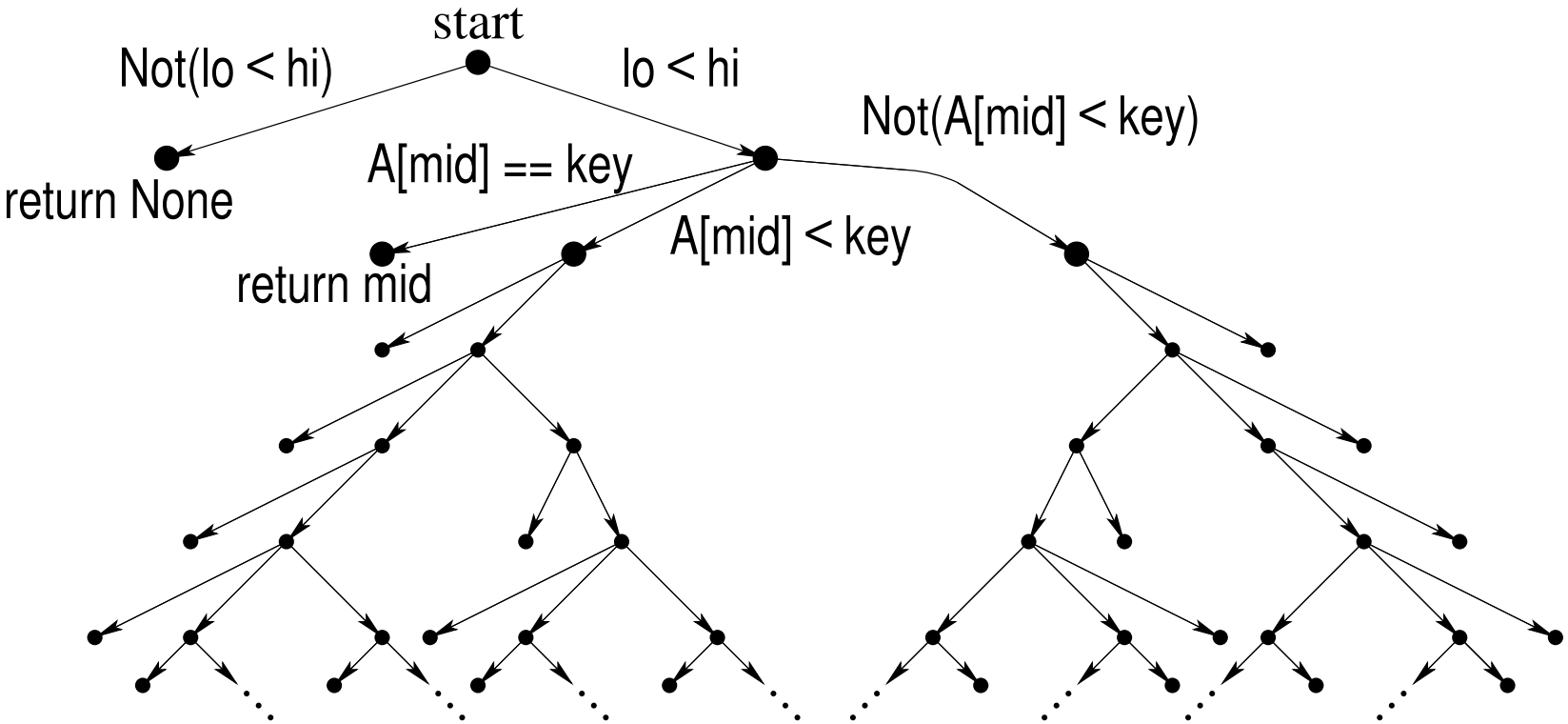
- `A` is an empty array;
- thus, `lo` and `hi` are both 0;
- and the `ForAll` holds vacuously.

I'm guessing that `k!1` is a "Skolem function" for the `ForAll`.

The `while` loop

- Now, we encounter the loop: `while lo < hi`.
- There are two cases to consider:
  - The loop condition is satisfied; the loop body is executed; and the loop condition is tested again.
  - The loop condition is not satisfied; and the exits immediately.
- We can create two subproblems to analyse, according to the loop condition.
  - This leads to the *path explosion* problem (see the next slide).
  - We'll examine various ways to tame path explosions in this class.

The path explosion problem



- Each test of the `while` loop condition splits the path into two subpaths.
- Each comparision of `A[mid]` with `key` creates three subpaths.
- The number of paths grows as 2^k where k is the number of loop iterations.
- Brute-force approaches quickly become intractable.

Constraints for the `while` loop

We can clone our solver for the two cases.

- First case, the while loop exits immediately:

```
#  After the while loop, zero executions of the body
s0x = Solver()                          #  make a new solver
s0x.add(s.assertions())                 #  clone the constraints
s0x.add(Not(lo < hi))                   #  we've exited the loop
s0x.push()              #  we can back out if we make a mistake
s0x.add(Exists(i, A[i] == key))  #  check for an error
print s0x.check()
```

- I tried it and z3 reported `sat`.
- What went wrong?

Look at the counter-example

```
>>> print s0x.model()
[lenA = 0,
lo = 0,
hi = 0,
A = [else -> 2],
i!15 = 1,
key = 2,
k!16 = [else -> 2]]
```

In English:

`[lenA = 0`: `A` is an empty array.

`A = [else -> 2]`: `A[i]` is 2 for any choice of i.

`key = 2`: `key` is 2.

`i!15 = 1`: `i` is 1.

Ah ha! I forgot to constrain `i` and the counter-example is an array bounds violation.

Try again

```
s0x.pop()    # restore the solver
s0x.add(Exists(i, And(0 <= i, i < lenA,
                      A[i] == key)))
print s0x.check()
```

- This time, z3 reports `unsat` – it proved the desired property.
- We've now shown that if the while-loop executes 0 times (i.e. `A` is empty), then `search` returns the right answer.
- Now, let's look at when the loop body is executed.

The loop body

We start with

```
# 3: while(lo < hi):
s0b = Solver()      # the solver for the loop body
s0b.add(s.assertions())
s0b.add(lo < hi)    # the loop test passed

# 4: mid = (lo + hi)/2
mid = Int('mid')
s0b.add(mid == (lo + hi)/2)

# 5: if(A[mid] == key):  return mid
s0r = Solver()      # solver for the 'return' case
s0r.add(s.assertions())
s0r.add(A[mid] == key)
```

Now, we just need to add a constraint that we are returning the wrong answer and check for unsatisfiability:

```
>>> s0r.add(Not(A[mid] == key))
>>> print s0r.check()
unsat
```

Yay! We passed another test.

The `A[mid] < key` case

- How do we model `lo = mid`?
  - We need a fresh, z3 symbolic constant.
  - Let's call it `lo$1`.
  - We'll need a `hi$1` as well.
- We now get:

```
s0b.add(Not(A[mid] == key))  # condition for 1st if is false
lo_next, hi_next = Ints(('lo$1', 'hi$1'))
s0b.add(lo1 == If(A[mid] < mid, mid, lo))
s0b.add(hi1 == If(A[mid] < mid, hi,  mid))
```

- We've made it to the end of the first loop body execution. How do we continue?
  - The "right" answer is "invariants".
  - We'll get there, but . . .
    It's much easier to write an invariant if you already know what it should be!

Unwinding the loop

- Strategy: write Python code to execute the loop a small number of times (e.g. 5).
  - If `len(A)` is small enough (e.g. `len(A)` < 8) the code should exit. We can show this by showing that the final version of the solver for continuing the loop body precludes `lenA` < 8.
  - Code in

    `http://www.cs.ubc.ca/~mrg/cs513/2015-1/notes/09.10/src/bf0.py`.
- The termination test fails:

  `key`: has the value 6927.
  `A`: is an array of 6 elements, all of which have the value 6926.
  `lo`: starts at 0, goes 3, 4, 5, 5, ...

Pause to think (but not too much)

- We're using `lo` and `hi` to bracket the part of the array that might hold an element that matches `key`.
- In more detail, we consider `A[i]` for $\texttt{lo} \leq \texttt{i} < \texttt{hi}$.
  - But when we set `lo = mid` we set `lo` to an index that is definitely lower than any one that could have `key`.
  - We should set `lo = mid+1` instead.
- What about `hi`?
  - Setting `hi` to `mid` seems OK.
  - `A[mid]` is greater than `key`, but
  - `A[mid-1]` could match.

Revised code

```
# search(key, A): search A for an instance of key.
#     A is a non-decreasing array of integers.
#     return the index of an element of A that matches key
#     or None if no such element exists.
def search(key, A):
    lo = 0
    hi = len(A)
    while(lo < hi):
        mid = (lo + hi)/2
        if(A[mid] == key):  return mid
        elif(A[mid] < key): lo = mid+1  # ← changed this line
        else:  hi = mid
    return None
```

- I created bf1.py by changing:

  ```
  s.add(lo_next == If(A[mid] < key, mid, lo)) # see slide 20
  ```

  to

  ```
  s.add(lo_next == If(A[mid] < key, mid+1, lo))
  ```

- I re-ran with the revised code, and all the tests passed.

Invariants

- How can we reason about:

```
0:    preamble
1:    while(condition):
2:        loop-body
4:    epilogue
```

- We find an *invariant*, `Inv`, such that:
  - `Inv` holds after executing `preamable` (need to show).
  - From *any* state where `Inv` and `condition` hold before executing `loop-body`, `Inv` will hold after executing `loop-body` (need to show).
  - We use `And(Inv, condition)` as the condition that holds after exiting the loop (i.e. just before executing `epilog`).

An invariant for binary search

- From the previous examples, we see that a key property is that whenever we test the condition of the while loop, then if there is an element of `A` that matches `key`, then there must be such an element with an index in the range [`lo`, `hi`).
- Our proposed invariant:

```
Implies(Exists(i, And(0  <= i, i < lenA, A[i] == key))
        Exists(j, And(lo <= j, j < hi,   A[j] == key)))
```

  - If we try this, it fails:
  - We need to add a clause that `A` remains non-decreasing.
  - We need to add a clause that $0 \leq$ `lo` $\leq$ `hi` $\leq$ `lenA`.
- Our revised invariant:

```
And(non_decreasing(A), 0 <= lo, lo <= hi, hi <= lenA,
    Implies(Exists(i, And(0  <= i, i < lenA, A[i] == key))
            Exists(j, And(lo <= j, j < hi,   A[j] == key))))
```

  - This invariant passes.
  - See http://www.cs.ubc.ca/~mrg/cs513/2015-1/notes/09.10/search_invariant.py.

In the z3 code

- To do list:
  - Check termination: add check that `hi - lo` is always non-negative and decreases with each iteration.
  - Check array bounds.
  - Check with bit-vector arithmetic.
- Remarks about automation:
  - I manually wrote the constraints for this program.
  - The constraints describing what each program statement does can be generated automatically.
  - Check for array bounds violations and many other common errors can be generated automatically.
  - Humans are still needed to add annotations that require understanding the algorithm.

Binary Search: summary

- Seemingly simple code can be prone to errors because humans aren't good at handling lots of corner-cases.
- Symbolic execution handles all of the cases systematically.
- Constraint solvers such as Z3's SMT solver make symbolic execution practical.
- Issues like path explosion remain a concern.
- This simple example illustrates, in a small code fragment, many of the topics that we'll cover this semester.

Course mechanics

- We'll cover roughly one paper per lecture.
  - The reading list is at
    http://www.cs.ubc.ca/~mrg/cs513/2015-1/reading/ReadingList.html
  - Starting Sept. 17, you will be expected to write a short summaries for the papers. See slide 30.
- Each person will present one paper: see slide 31.
- There will be 4 ± 1 homework assignments: see slide 33.
- There will be a project: see slide 34.
  - Should take $\sim$ 40 hours to your time.
  - The final output is a M.Sc. thesis proposal (or equivalent). They are fine choices for papers to present.

Grading

- 10% paper summaries. You need to do 15 summaries to get full credit.
- 15% class presentation.
- 30% homework
- 45% course project.

Paper summaries

- Each summary should address each of the questions below:
  1. What problem does the paper address?
  2. What is the key insight/idea in the paper's solution to this problem?
  3. What did the authors do to demonstrate their claims? (e.g. implement a tool, present a proof, run simulations, etc.)
  4. Is the support for the claims convincing? Why or why not?
  5. What are your questions or other comments about the paper?
- You can address each point with 1–3 sentences. These summaries should be short: at most one page.
- Paper summaries are due at 10am of the day that the paper is covered in class.
  - Submit your summaries by e-mail.
  - Plain text is preferred. PDF is acceptable.
- If you found a paper too hard to read:
  - Write a description of where you got stuck.
  - Write some questions that would help you understand the paper.

Paper Presentations

- Each person will present one paper.
- There are many “For further reading” papers proposed on the reading list. They are fine choices for papers to present.
- Or you can present a paper related to your project.
- Or you can present another paper.
- Claim a paper by sending e-mail to me:
  - Tell me what paper and what date.
  - The *first* person to claim a paper gets it.
  - I’ll send you a confirmation and update the reading list.

Presentations Guidelines

- A presentation should take about 20 minutes and be like a conference talk.
- An effective conference talk is an **advertisement** for the paper:
  - State why the problem matters.
  - State the main contributions.
  - Sketch how the contributions are validated — focus on one or two *interesting* points about what they did.
  - Add your own comments, questions, and criticisms. Connect the paper with other papers that we've studied in class.
- You can prepare slides and/or use the whiteboard.
  - I *strongly* recommend giving a practice run of your talk to another student.
- Note: I'll make a few presentations like this to touch on some of the "advanced topics" (see slide 3) that we won't have time to cover in detail.

Homework

- There will be 4 ± 1 homework assignments.
- Homework problems include:
  - Trying something with real, formal verification tools.
  - Some “pencil-and-paper” problems to complement the programming.
- The first assignment should go out on Sept. 17.

Projects

The goal of the project is to produce a Master's thesis proposal including:

- A statement of what problem you are addressing.
- A review of relevant research (at least five papers)
- A simple experiment (e.g. write some code) to show that your idea will probably be feasible.
- A timeline for the thesis research and write-up.
- Identify the any resources you would need:
  - Special equipment.
  - Access to proprietry data.
  - Anything else
- Identify where the risks are in the research and how you plan to handle them.
  - If you knew the results ahead of time, it wouldn't be research.

Project ideas

- I'm very happy to see projects for any aspect of formal verification.
- I also like projects that connect formal verification with other areas.
- See

  http://www.cs.ubc.ca/~mrg/cs513/2015-1/project.html

  for some project ideas.
- I prefer individual projects, but I'll consider a proposal for a group project if:
  - A clear reason is given for why the project should be done as a group and not as separate projects.
  - Clear criteria are given for evaluating each member's contribution to the project.

Project Deadlines

- October 29: proposals due.
  - State the problem you plan to address.
  - State what approach you expect to use to address the problem.
  - List at least three papers that you plan to include in your literature survey.
- December 3: intermediate report due.
  - List the papers that you have read.
  - Describe what the progress you have made on evaluating the feasibility of your idea. For example, if you're writing a program, describe what the status of developing that code.
  - Describe any issues that have come up that could impact the project.
  - This report should be 1 or 2 pages long plus short summaries of the papers that you've read.
- December 14: Final project report due.

Project Grading

$<$ 80: You didn't do what you proposed, and you didn't try to revise the project goals if you encountered some unforeseen difficulty. Note that you can always check with me to revise the project proposal if you need to.

80 – 84: You did what you proposed, but you didn't demonstrate that you explored the topic in a way so that you learned something significant in the process.

85 – 89: You clearly learned something significant by doing the project. Make sure that your report clearly states what you learned by doing the project.

90 – 94: I learned something significant by reading your report.

95 – 100: This work is worthy of writing a paper that I expect to be a landmark in the field.

Will I survive this class?

Yes.

- Formal verification spans many areas of computer science and other fields including:
  - Mathematical logic, programming languages, digital logic design, computational complexity, computer architecture, temporal logic, robotics, differential equations, optimization.
- I expect that most of these will be new to most students in the class
  - Lectures will include many tutorials.
  - I will emphasize showing the connections between what you already know and other branches of computer science.
- The goals of the course are:
  - Give you an introduction to formal methods so you can do research in the area if you want to.
  - Give you a background so you can see how these methods are useful in other areas where you may end up doing your thesis research or working after you graduate.
  - Have fun exploring how formal methods solve challenging, real-world problems.