Bug Catching: Automated Program Verification
15414/15614 Fall 2017
Lecture 1:
Introduction

Matt Fredrikson, André Platzer
{mfredrik, aplatzer}@cs

August 29, 2017

Course Staff

Matt Fredrikson
Instructor

André Platzer
Instructor

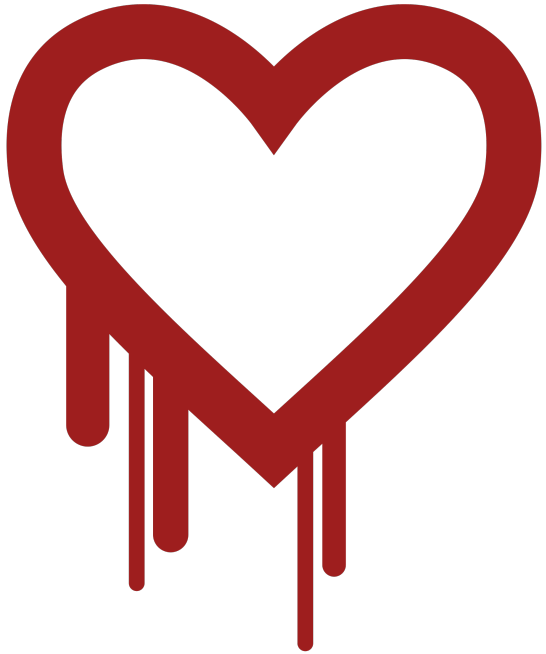
Jonathan Laurent
TA

Tianyu Li
TA

What This Course is About

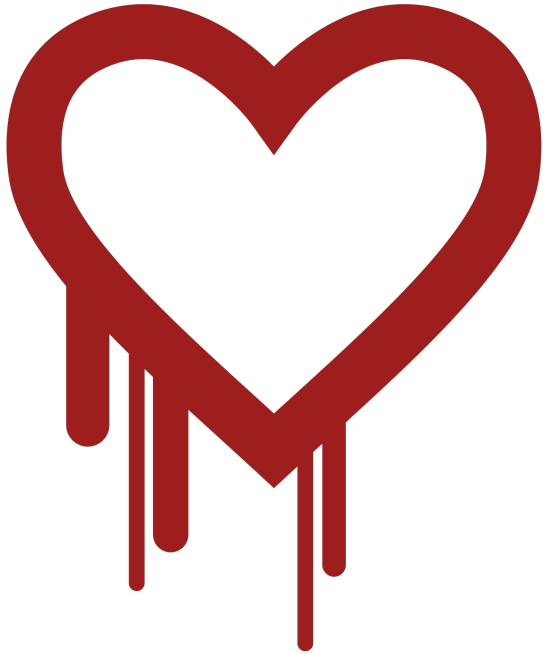
Does the software do what it is supposed to do?

What happens when software misbehaves



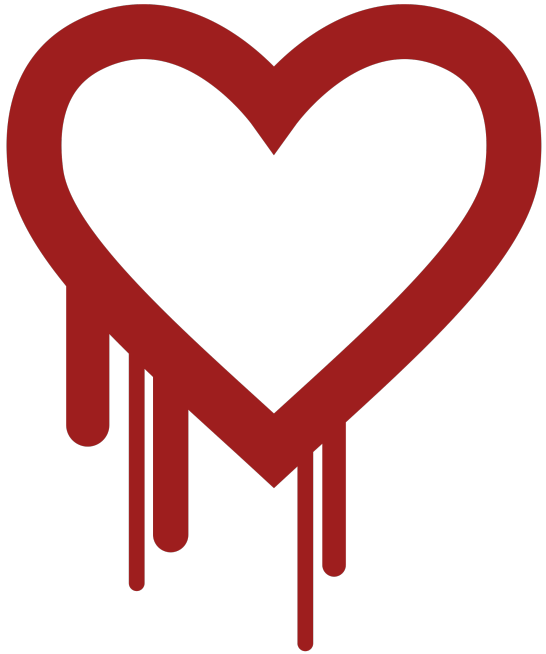
What happens when software misbehaves

- **April, 2014** OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.



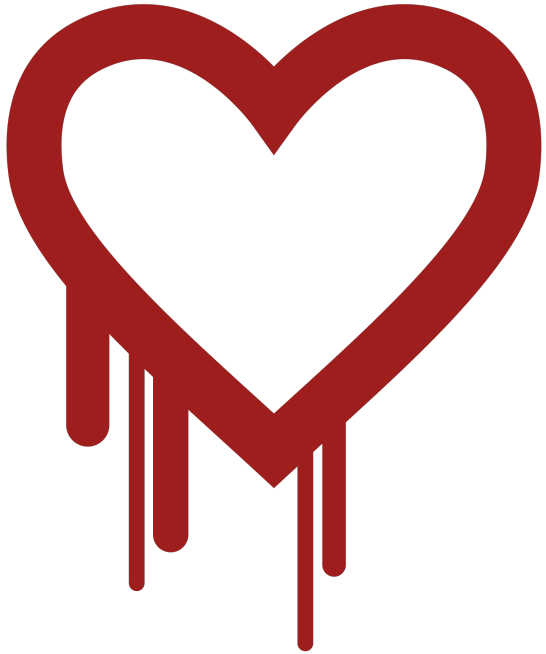
What happens when software misbehaves

- **April, 2014** OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.
- “The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software.”



What happens when software misbehaves

- **April, 2014** OpenSSL announced critical vulnerability in their implementation of the Heartbeat Extension.
- “The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software.”
- “...this allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.”



Heartbleed, explained

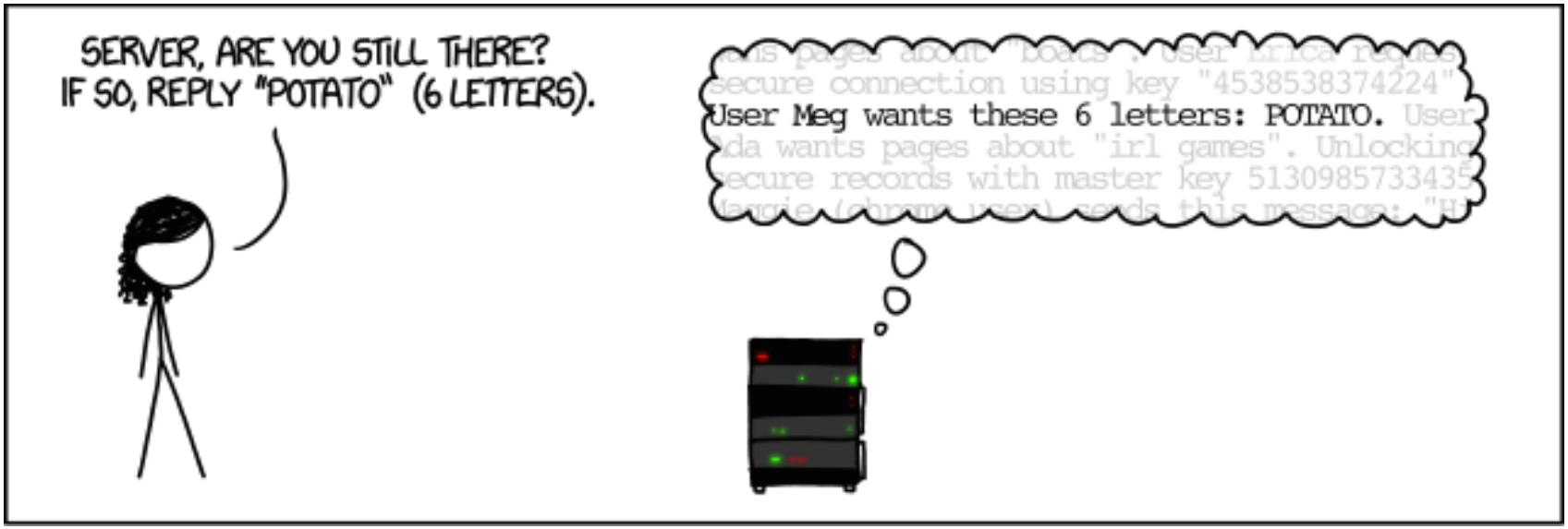


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

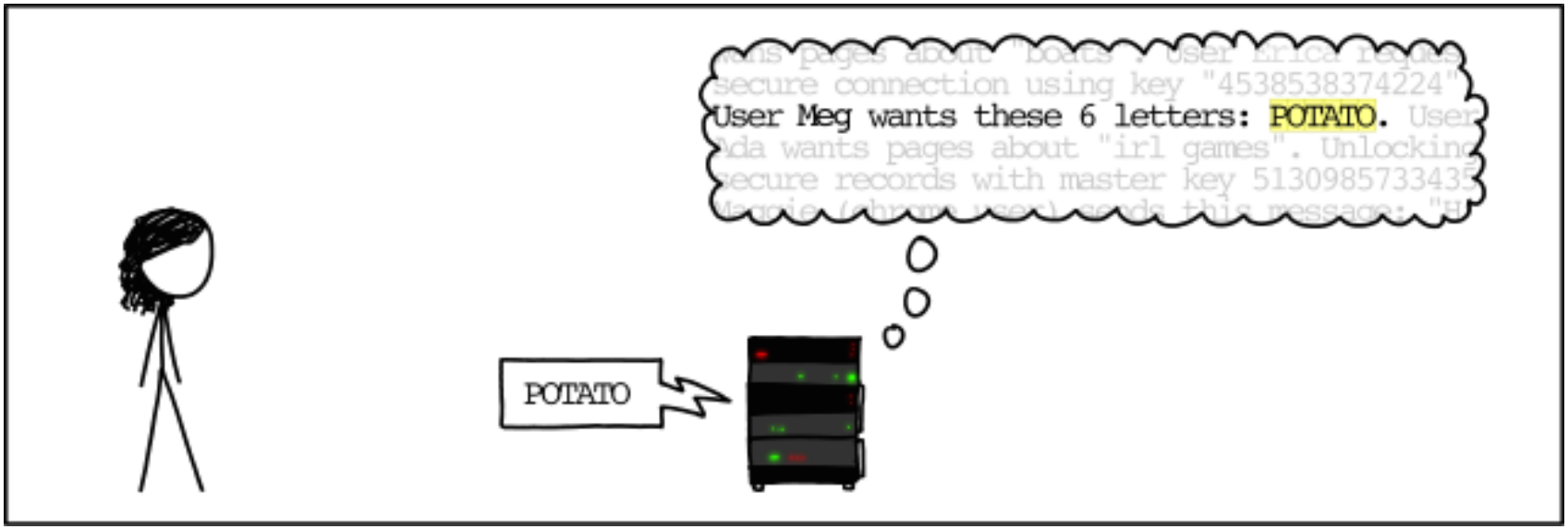


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

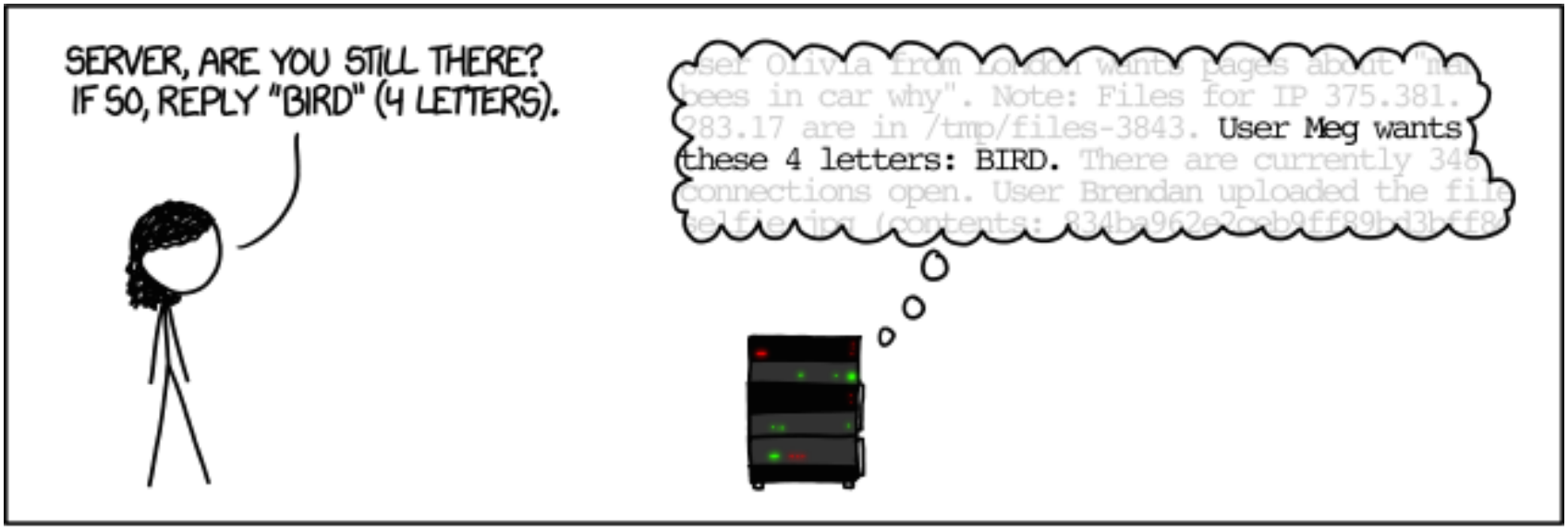


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

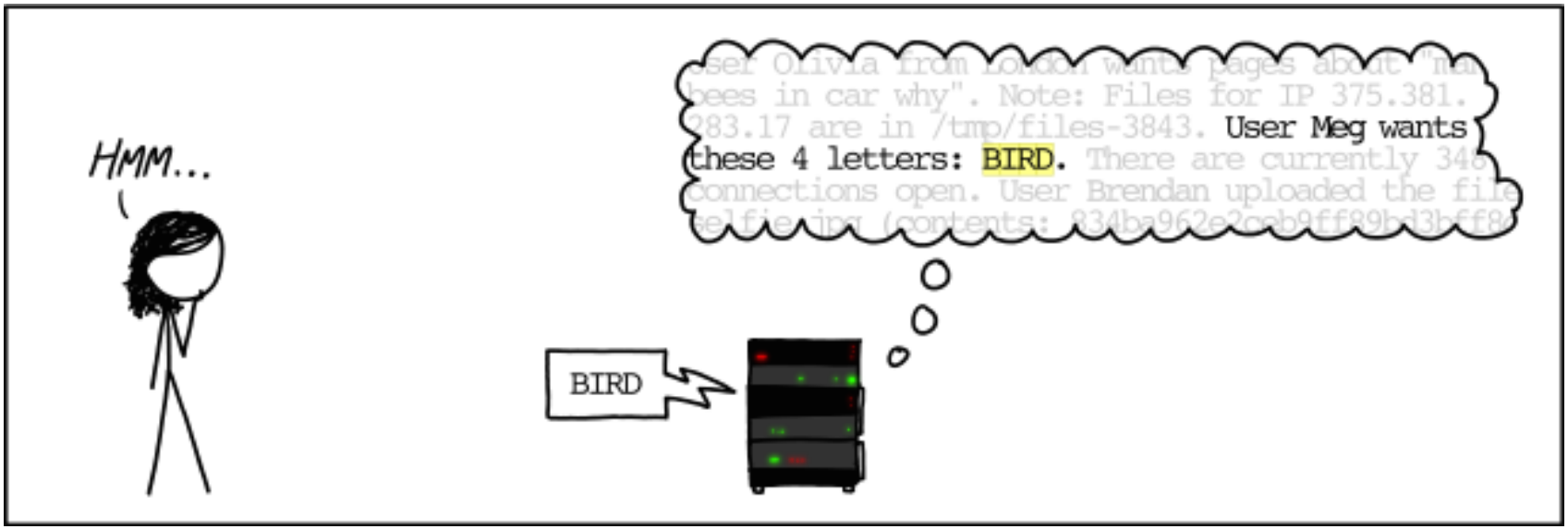


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

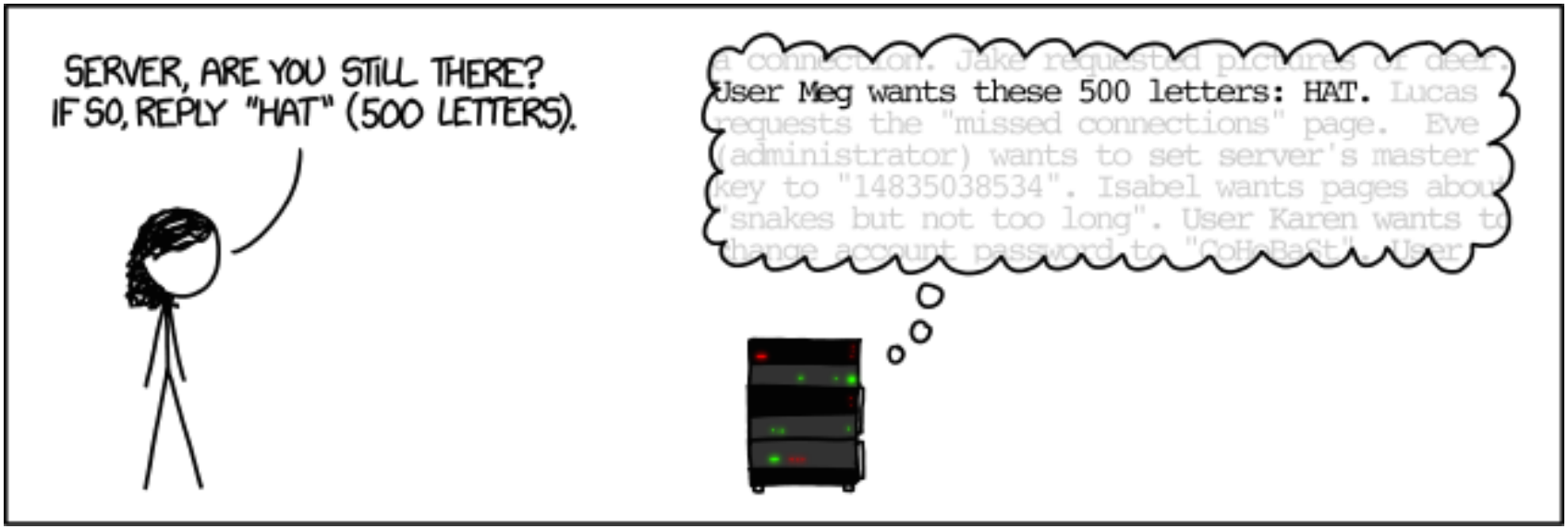


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

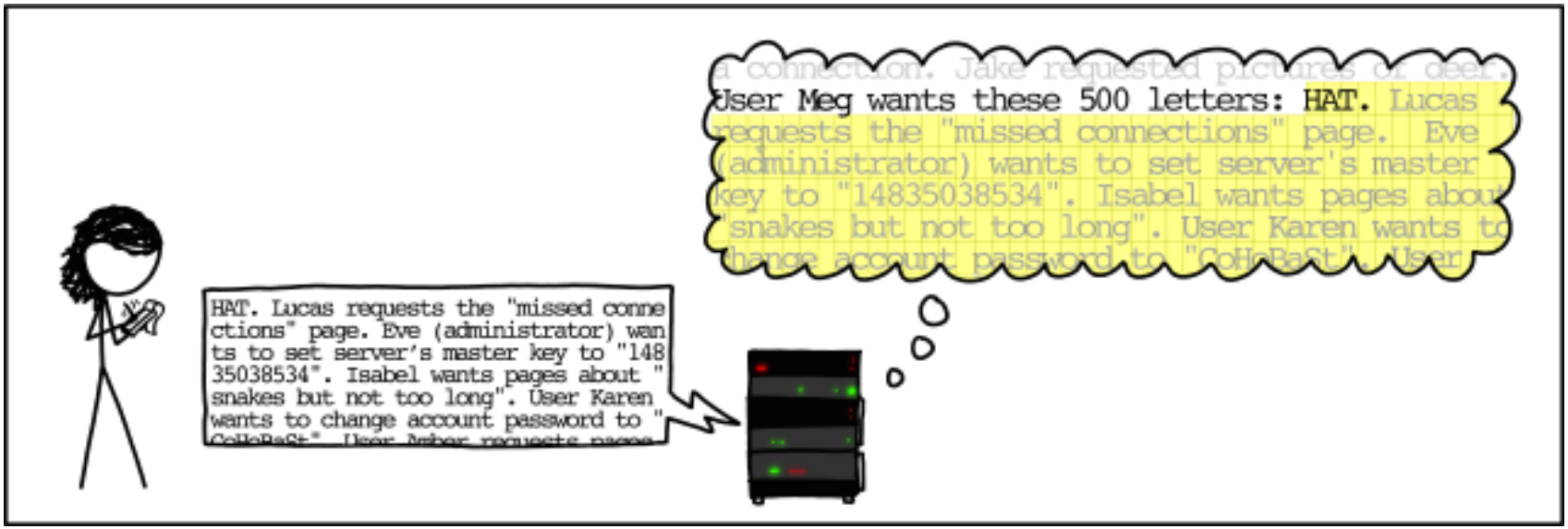


Image source: Randall Munroe, xkcd.com

Does this do what it is supposed to?

```
1  int binarySearch(int key, int[] a, int n) {
2      int low = 0;
3      int high = n;
4
5      while (low < high) {
6          int mid = (low + high) / 2;
7
8          if(a[mid] == key) return mid; // key found
9          else if(a[mid] < key) {
10             low = mid + 1;
11         } else {
12             high = mid;
13         }
14     }
15     return -1;  // key not found.
16 }
```

Does this do what it is supposed to?

```
1  int binarySearch(int key, int[] a, int n) {
2      int low = 0;
3      int high = n;
4
5      while (low < high) {
6          int mid = (low + high) / 2;
7
8          if(a[mid] == key) return mid; // key found
9          else if(a[mid] < key) {
10             low = mid + 1;
11         } else {
12             high = mid;
13         }
14     }
15     return -1;  // key not found.
16 }
```

Code Matters

This is a correct binary search algorithm.

Code Matters

This is a correct binary search algorithm.

But what if `low + high` $> 2^{31} - 1$?

Code Matters

This is a correct binary search algorithm.

But what if $\texttt{low + high} > 2^{31} - 1$?

Then $\texttt{mid} = (\texttt{low + high}) / 2$ becomes negative

Code Matters

This is a correct binary search algorithm.

But what if `low + high` $> 2^{31} - 1$?

Then `mid = (low + high) / 2` becomes negative

- Best case: `ArrayIndexOutOfBoundsException`

Code Matters

This is a correct binary search algorithm.

But what if `low + high` $> 2^{31} - 1$?

Then `mid = (low + high) / 2` becomes negative

- Best case: `ArrayIndexOutOfBoundsException`
- Worst case: undefined behavior

Code Matters

This is a correct binary search algorithm.

But what if `low + high` $> 2^{31} - 1$?

Then `mid = (low + high) / 2` becomes negative

- Best case: `ArrayIndexOutOfBoundsException`
- Worst case: undefined behavior

Algorithm may be correct. The code, another story...

How do we fix it?

The culprit: `mid = (low + high) / 2`

How do we fix it?

The culprit: `mid = (low + high) / 2`

Need to make sure we don't overflow at any point

How do we fix it?

The culprit: `mid = (low + high) / 2`

Need to make sure we don't overflow at any point

Solution: `mid = low + (high - low)/2`

The fix

```
1  int binarySearch(int key, int[] a, int n) {
2      int low = 0;
3      int high = n;
4
5      while (low < high) {
6          int mid = low + (high - low) / 2;
7
8          if(a[mid] == key) return mid; // key found
9          else if(a[mid] < key) {
10             low = mid + 1;
11         } else {
12             high = mid;
13         }
14     }
15     return -1;  // key not found.
16 }
```

The fix

```
1  int binarySearch(int key, int[] a, int n)
2  //@requires 0 <= n && n <= \length(A);
3  {
4      int low = 0;
5      int high = n;
6
7      while (low < high) {
8          int mid = low + (high - low) / 2;
9
10         if(a[mid] == key) return mid; // key found
11         else if(a[mid] < key) {
12             low = mid + 1;
13         } else {
14              high = mid;
15         }
16     }
17     return -1;  // key not found.
18 }
```

The fix

```
1  int binarySearch(int key, int[] a, int n)
2  //@requires 0 <= n && n <= \length(a);
3  //@requires is_sorted(a, 0, n);
4  /*@ensures (\result == -1 && !is_in(key, A, 0, n))
5    @      || (0 <= \result, \result < n
6    @           && A[\result] == key); @*/
7  {
8      int low = 0;
9      int high = n;
10
11     while (low < high) {
12         int mid = low + (high - low) / 2;
13
14         if(a[mid] == key) return mid; // key found
15         else if(a[mid] < key) {
16             low = mid + 1;
17         } else {
18              high = mid;
19         }
20     }
21     return -1;  // key not found.
22 }
```

How do we know if it's correct?

How do we know if it's correct?

One solution: test the code

How do we know if it's correct?

One solution: test the code

- Possibly incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing usually not feasible

How do we know if it's correct?

One solution: test the code

- Possibly incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing usually not feasible

Better: **prove** that that it's correct

$$\textit{Specification} \iff \textit{Implementation}$$

How do we know if it's correct?

One solution: test the code

- Possibly incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing usually not feasible

Better: **prove** that that it's correct

$$\textit{Specification} \iff \textit{Implementation}$$

- Specifications must be precise, unambiguous
- Meaning of code must be well-defined

How do we know if it's correct?

One solution: test the code

- Possibly incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing usually not feasible

Better: **prove** that that it's correct

$$\textit{Specification} \iff \textit{Implementation}$$

- Specifications must be precise, unambiguous
- Meaning of code must be well-defined

When done well, gives strong indication of correctness

How do we know if it's correct?

One solution: test the code

- Possibly incomplete $\longrightarrow$ uncertain answer
- Exhaustive testing usually not feasible

Better: **prove** that that it's correct

$$\textit{Specification} \Longleftrightarrow \textit{Implementation}$$

- Specifications must be precise, unambiguous
- Meaning of code must be well-defined

When done well, gives strong indication of correctness

- Specifications must be validated
- Proofs must be correct
- Reasoning must be sound

Algorithmic Approaches

Formal proofs are tedious,
labor-intensive

Algorithmic Approaches

Formal proofs are tedious,
labor-intensive

We want algorithms to:

- Check our work
- Fill in low-level details
- Give diagnostic info
- Verify the system (if possible)

Algorithmic Approaches

Formal proofs are tedious,
labor-intensive

We want algorithms to:

- Check our work
- Fill in low-level details
- Give diagnostic info
- Verify the system (if possible)

This is called
algorithmic verification

Algorithmic Approaches

Formal proofs are tedious, labor-intensive

We want algorithms to:

- Check our work
- Fill in low-level details
- Give diagnostic info
- Verify the system (if possible)

This is called
algorithmic verification

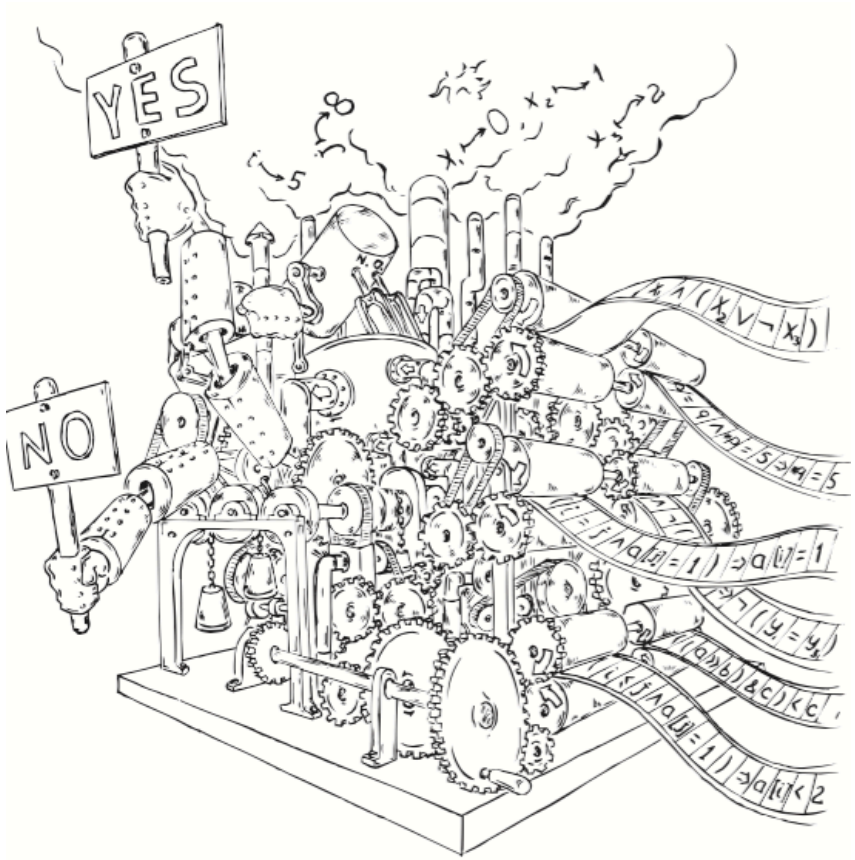


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Course objectives

Course objectives

- Identify and formalize program correctness

Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs

Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
- Apply mathematical reasoning to program correctness

Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end

Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end
- Use automated tools that assist verifying your code

Course objectives

- Identify and formalize program correctness
- Understand the formal semantics of programs
- Apply mathematical reasoning to program correctness
- Learn how to write correct software, from beginning to end
- Use automated tools that assist verifying your code
- Understand how verification tools work

Reasoning about correctness

```
1  int[] array_copy(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures \length(\result) == n;
4  {
5    int[] B = alloc_array(int, n);
6
7    for (int i = 0; i < n; i++)
8    //@loop_invariant 0 <= i;
9    {
10     B[i] = A[i];
11   }
12
13   return B;
14 }
```

Reasoning about correctness

Functional Correctness

- Specification
- Proof

```
1  int[] array_copy(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures \length(\result) == n;
4  {
5    int[] B = alloc_array(int, n);
6
7    for (int i = 0; i < n; i++)
8    //@loop_invariant 0 <= i;
9    {
10     B[i] = A[i];
11   }
12
13   return B;
14 }
```

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise
- Amenable to proof

```
1  int[] array_copy(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures \length(\result) == n;
4  {
5    int[] B = alloc_array(int, n);
6
7    for (int i = 0; i < n; i++)
8    //@loop_invariant 0 <= i;
9    {
10     B[i] = A[i];
11   }
12
13   return B;
14 }
```

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise
- Amenable to proof

Systematic proof techniques

- Based on language semantics
- Exhaustive proof rules
- Ideally, automatable

```
1  int[] array_copy(int[] A, int n)
2  //@requires 0 <= n && n <= \length(A);
3  //@ensures \length(\result) == n;
4  {
5    int[] B = alloc_array(int, n);
6
7    for (int i = 0; i < n; i++)
8    //@loop_invariant 0 <= i;
9    {
10     B[i] = A[i];
11   }
12
13   return B;
14 }
```

Why3

Deductive verification platform

- Programming language
- Automated verification tools

Why3

Deductive verification platform

- Programming language
- Automated verification tools

Rich specification language

- Pre and postconditions, assertions
- Pure mathematical functions
- Termination metrics

Why3

Deductive verification platform

- Programming language
- Automated verification tools

Rich specification language

- Pre and postconditions, assertions
- Pure mathematical functions
- Termination metrics

Programmer writes specification, proof annotations

Compiler checks correctness automatically*!

Binary search in Why3

```
let binary_search (a : array int) (v : int)
  requires { sorted(a) }
  ensures  { 0 <= result < length a && a[result] = v }
  raises   { Not_found -> forall i:int. 0 <= i < length a -> a[i] <> v }
= try
    let l = ref 0 in
    let u = ref (length a - 1) in
    while !l <= !u do
      invariant { 0 <= !l /\ !u < length a }
      invariant { forall i : int. 0 <= i < length a -> a[i] = v -> !l <= i <= !u }
      variant { !u - !l }
      let m = !l + div (!u - !l) 2 in
      assert { !l <= m <= !u };
      if a[m] < v then
        l := m + 1
      else if a[m] > v then
        u := m - 1
      else
        raise (Break m)
    done;
    raise Not_found
  with Break i ->
    i
  end
```

Automated Verification

Algorithms for proving that programs match their specifications

Automated Verification

Algorithms for proving that programs match their specifications

Basic idea:

1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a decision procedure

Automated Verification

Algorithms for proving that programs match their specifications

Problem is undecidable!

1. Require annotations
2. Relieve manual burden by inferring some annotations

Basic idea:

1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a decision procedure

Automated Verification

Algorithms for proving that programs match their specifications

Problem is undecidable!

1. Require annotations
2. Relieve manual burden by inferring some annotations

Verifiers are non-trivial tools

Basic idea:

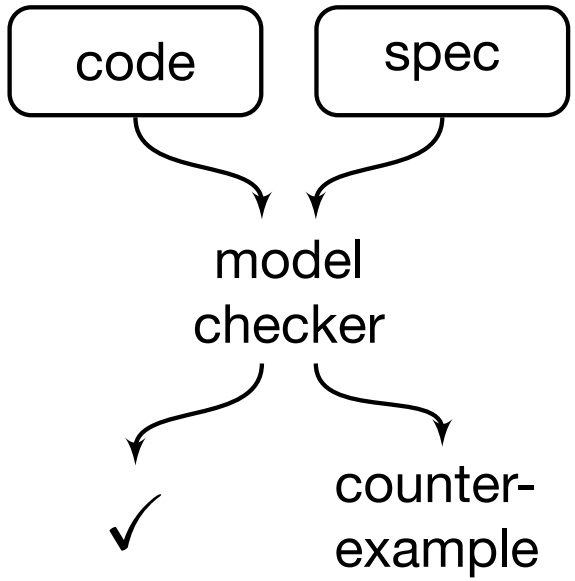
1. Translate programs into *proof obligations*
2. Encode proof obligations as satisfiability
3. Solve using a decision procedure

Model Checking

Automatic **techniques for finding bugs (or proving their absence)**

Model Checking

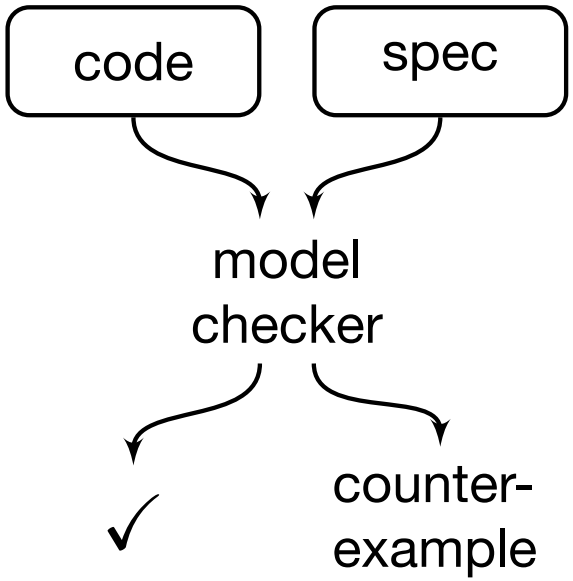
***Automatic* techniques for finding bugs (or proving their absence)**



Model Checking

***Automatic* techniques for finding bugs (or proving their absence)**

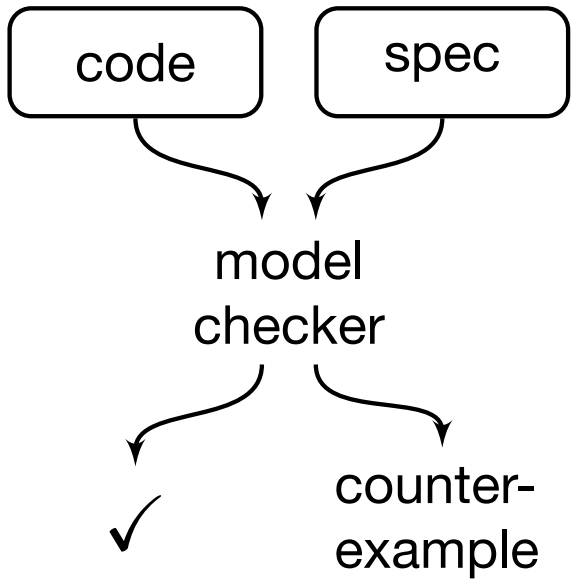
- Specifications written in *propositional temporal logic*



Model Checking

***Automatic* techniques for finding bugs (or proving their absence)**

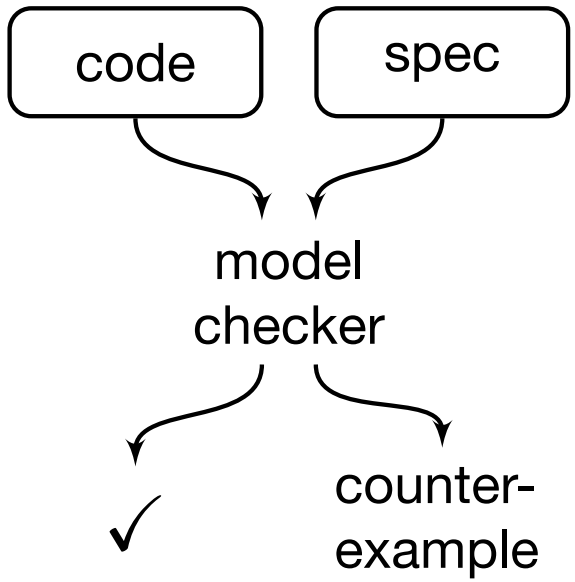
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search



Model Checking

***Automatic* techniques for finding bugs (or proving their absence)**

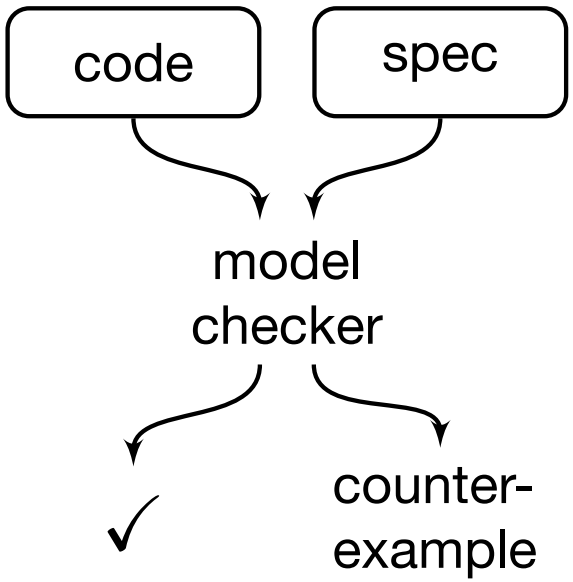
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples



Model Checking

***Automatic* techniques for finding bugs (or proving their absence)**

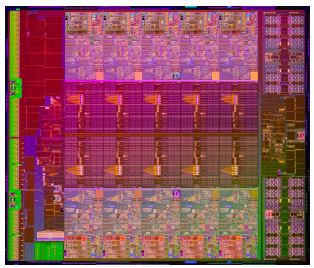
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No proofs!

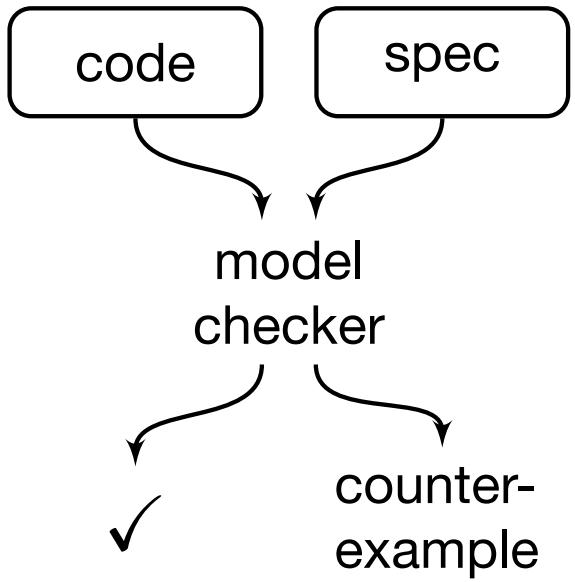


Model Checking

***Automatic* techniques for finding bugs (or proving their absence)**

- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No proofs!
- **Downside**: "State explosion"





Model Checking Gets Results

Clever ways of dealing with state explosion:

Model Checking Gets Results

Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic exploration
- Abstraction & refinement

Model Checking Gets Results

Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic exploration
- Abstraction & refinement

Now widely used for verification & bug-finding:

- Hardware, software, protocols, …
- Microsoft, Intel, Cadence, IBM, NASA, …

Model Checking Gets Results

Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic exploration
- Abstraction & refinement

Now widely used for verification & bug-finding:

- Hardware, software, protocols, …
- Microsoft, Intel, Cadence, IBM, NASA, …

Ed Clarke
Turing Award,
2007

Grading

Breakdown:

- 40% labs
- 25% written homework
- 30% exams (15% each, midterm and final)
- 5% participation

5 labs

Weekly written homework

In-class exams, closed-book

Participation:

- Come to lecture
- Ask questions, give answers
- Contribute to discussion

Grading

For the labs, you will:

- Implement some functionality (usually)
- Specify correctness for that functionality
- Prove it correct by annotating your implementation

Grading

For the labs, you will:

- Implement some functionality (usually)
- Specify correctness for that functionality
- Prove it correct by annotating your implementation

Most important criterion is **correctness.**

Grading

For the labs, you will:

- Implement some functionality (usually)
- Specify correctness for that functionality
- Prove it correct by annotating your implementation

Most important criterion is **correctness.**

Full points when you provide the following

- Correct implementation
- Correct specification
- Correct annotations
- Sufficient annotations for verification

Grading

For the labs, you will:

- Implement some functionality (usually)
- Specify correctness for that functionality
- Prove it correct by annotating your implementation

Most important criterion is **correctness.**

Full points when you provide the following

- Correct implementation
- Correct specification
- Correct annotations
- Sufficient annotations for verification

Partial credit depending on how many of these you achieve

Clarity & conciseness is necessary for partial credit!

Grading

Written homeworks focus on theory and fundamental skills

Grading

Written homeworks focus on theory and fundamental skills

Grades are based on:

- Correctness of your answer
- How you present your reasoning

Grading

Written homeworks focus on theory and fundamental skills

Grades are based on:

- Correctness of your answer
- How you present your reasoning

Strive for **clarity & conciseness**

- Show each step of your reasoning
- State your assumptions
- Answers without well-explained reasoning don't count!

Late Policy

No late days on written homework

- Not intended to be time-intensive
- 25% deduction for each day past deadline

Late Policy

No late days on written homework

- Not intended to be time-intensive
- 25% deduction for each day past deadline

Can earn back missed points for proofs on labs

- Must submit original lab by the deadline
- Resubmit **once** within three days of deadline
- If proof is complete & correct, earn back points *only on the proof*

Logistics

Website: `http://www.cs.cmu.edu/~15414`

Course staff contact: Piazza or
`15414-staff@lists.andrew.cmu.edu`
Lecture: Tuesdays & Thursdays, 10:30-11:50 GHC 4211

Matt Fredrikson, André Platzer

- Location: CIC 2126, GHC 9103
- Office Hours: TBD
- Email: mfredrik@cs, aplatzer@cs

Jonathan Laurent, Tianyu Li

- Office Hours: TBD
- Email: jonathan.laurent@cs, tli2@cs