Automated Program Verification and Testing 15414/15614 Fall 2016 Lecture 1: Introduction

Matt Fredrikson
mfredrik@cs.cmu.edu

August 30, 2016

Course Staff

Matt Fredrikson
Instructor

Ryan Wagner
TA

What This Course is About

Does the software do what it is supposed to do?

Does this do what it is supposed to?

```java
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

Does this do what it is supposed to?

```
public static int binarySearch(int[] a, int key) {
    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

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

Then `mid = (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—with proof! The code, another story...

Bugs make software insecure

Bugs make software insecure

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

Bugs make software insecure

- **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.”

Bugs make software insecure

- **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

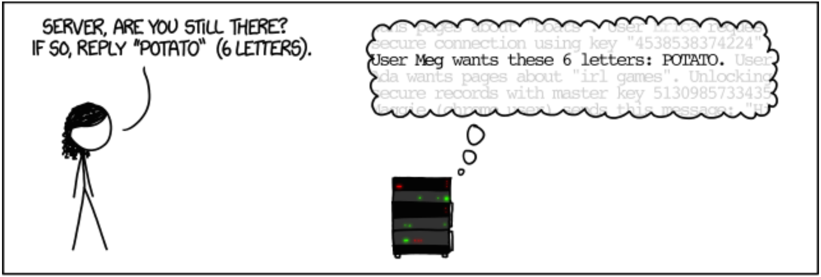


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

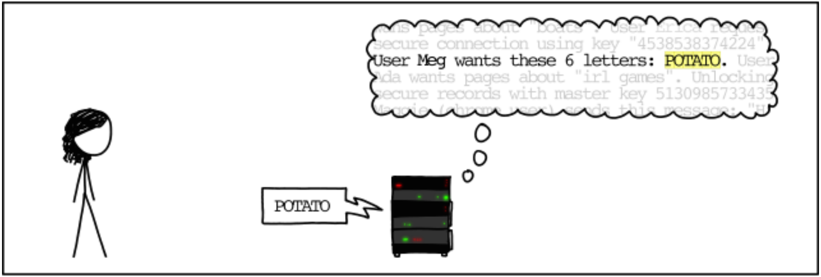


Image source: Randall Munroe, xkcd.com

Image source: Randall Munroe, xkcd.com

Heartbleed, explained

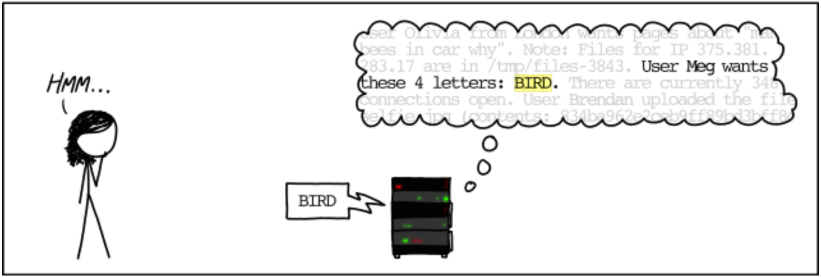


Image source: Randall Munroe, xkcd.com

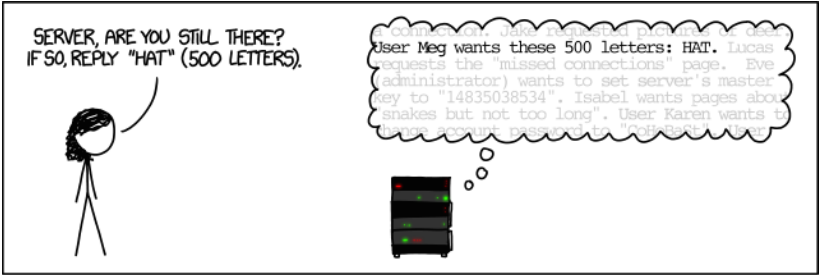


Image source: Randall Munroe, xkcd.com

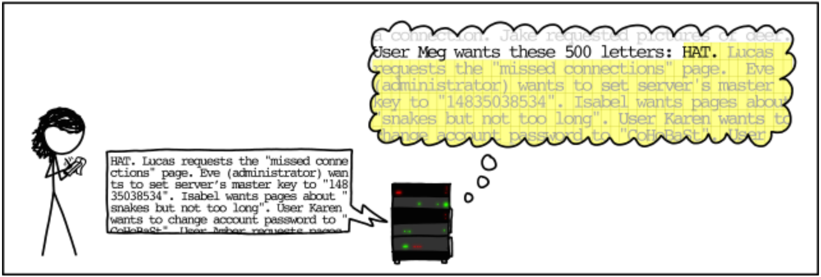


Image source: Randall Munroe, xkcd.com

Many, many bugs

Many, many bugs

1996, Ariane 5
Numerical overflow

Many, many bugs

1996, Ariane 5
Numerical overflow

2016, Nissan
1m recalls for buggy airbag code

Many, many bugs

1996, Ariane 5
Numerical overflow

2016, Nissan
1m recalls for buggy airbag code

2000-2010, Toyota
“Unintended acceleration” bug

Many, many bugs

1996, Ariane 5
Numerical overflow

2000-2010, Toyota
“Unintended acceleration” bug

2016, Nissan
1m recalls for buggy airbag code

2012, Knight Capital
Lost $440m in 30 minutes

Formal Verification

All about proof

Formal Verification

All about proof

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

Formal Verification

All about proof

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

- Specifications must be *unambiguous*
- *Meaning* of implementation must be well-defined

Formal Verification

All about proof

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

- Specifications must be *unambiguous*
- *Meaning* of implementation must be well-defined

When done well, gives strong indication of correctness

- ...but nothing is absolute
- Specifications and models must be validated
- Excellent complement to testing, other engineering practices

Algorithmic Approaches

Algorithmic Approaches

Formal proofs are tedious, error-prone

Algorithmic Approaches

Formal proofs are tedious, error-prone

We want algorithms to:

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

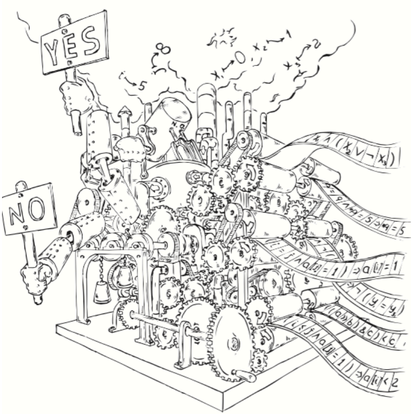


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures: An Algorithmic Point of View*

Algorithmic Approaches

Formal proofs are tedious, error-prone

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*

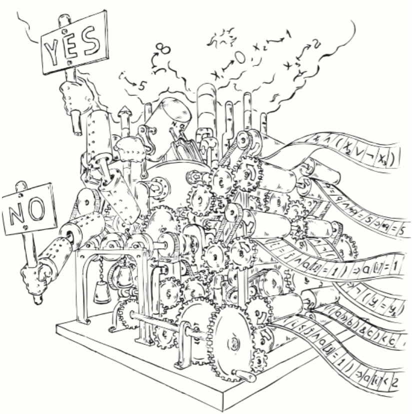


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures: An Algorithmic Point of View*

This course

Understand the principles and algorithms behind verification tools

This course

Understand the principles and algorithms behind verification tools

Gain experience using tools to write machine-checked code

This course

Understand the principles and algorithms behind verification tools

Gain experience using tools to write machine-checked code

Three high-level topics:

- Decision procedures for automated reasoning
- Techniques for proving program correctness
- Algorithms and tools for automatic verification

This course, in more detail

In this course, we'll cover:

- Propositional and first-order logic
- First-order theories commonly used in software verification
- Satisfiability decision procedures for propositional and first-order logic with theories
- Well-founded and structural induction
- Specifications of program correctness
- Hoare Logic, verification conditions, and predicate transformers
- Techniques for proving termination
- Automated inductive verification
- Static analysis techniques for inferring useful invariants
- Software model checking and temporal logic
- Symbolic execution for testing

Decision Procedures

Decision Procedure

An algorithm that, when given a decision problem, terminates with a yes/no answer.

Decision Procedures

Decision Procedure

An algorithm that, when given a decision problem, terminates with a yes/no answer.

Decision problems:

Decision Procedures

Decision Procedure

An algorithm that, when given a decision problem, terminates with a yes/no answer.

Decision problems:

- Is x a prime?

Decision Procedures

Decision Procedure

An algorithm that, when given a decision problem, terminates with a yes/no answer.

Decision problems:

- Is x a prime?
- Is w a word in L?

Decision Procedures

Decision Procedure

An algorithm that, when given a decision problem, terminates with a yes/no answer.

Decision problems:

- Is x a prime?
- Is w a word in L?
- Does M halt on every input?

Decision Procedures

Decision Procedure

An algorithm that, when given a decision problem, terminates with a yes/no answer.

Decision problems:

- Is x a prime?
- Is w a word in L?
- Does M halt on every input?
- Is ϕ satisfiable?

Decision Procedures

Decision Procedure

An algorithm that, when given a decision problem, terminates with a yes/no answer.

Decision problems:

- Is x a prime?
- Is w a word in L?
- Does M halt on every input?
- Is ϕ satisfiable?

We will focus on *satisfiability procedures*.

Decision Procedures

Decision Procedure

An algorithm that, when given a decision problem, terminates with a yes/no answer.

Decision problems:

- Is x a prime?
- Is w a word in L?
- Does M halt on every input?
- Is ϕ satisfiable?

We will focus on *satisfiability procedures*.

We'll look at examples that are:

- Expressive enough to model real problems.
- Still decidable.

Propositional SAT

Propositional Logic	
0	False
1	True
$\neg$	Not
$\wedge$	And
$\vee$	Or
$\rightarrow$	Implies
$\leftrightarrow$	Equivalent

Propositional SAT

Propositional Logic	
0	False
1	True
$\neg$	Not
$\wedge$	And
$\vee$	Or
$\rightarrow$	Implies
$\leftrightarrow$	Equivalent

SAT Problem

Given a propositional formula F over variables $p_1, p_2, \ldots$, find an assignment $I = [p_1 \mapsto \cdot, p_2 \mapsto \cdot, \ldots]$ that satisfies F.

Propositional SAT

Propositional Logic	
0	False
1	True
$\neg$	Not
$\wedge$	And
$\vee$	Or
$\rightarrow$	Implies
$\leftrightarrow$	Equivalent

SAT Problem

Given a propositional formula F over variables $p_1, p_2, \ldots$, find an assignment $I = [p_1 \mapsto \cdot, p_2 \mapsto \cdot, \ldots]$ that satisfies F.

Lots of important applications...

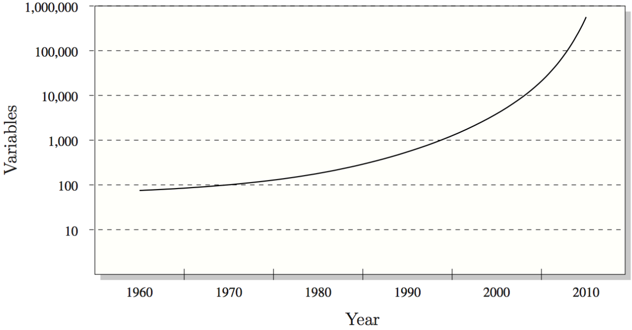
- Verification
- Program synthesis
- Test generation
- Equivalence checking
- Combinatorial design
- Cryptanalysis

Isn’t SAT too hard?

Isn’t SAT too hard?

3-SAT is the canonical NP-Complete problem

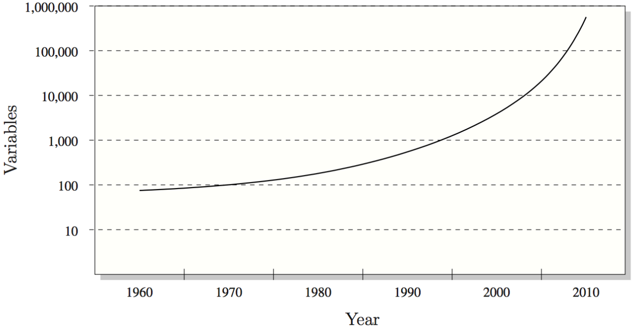
Isn’t SAT too hard?



3-SAT is the canonical NP-Complete problem

...but procedures routinely solve very large instances

Isn't SAT too hard?



3-SAT is the canonical NP-Complete problem

...but procedures routinely solve very large instances

Key: combine search and deduction for common-case efficiency

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

Beyond SAT: Modulo Theories

SAT is a good foundation for automated reasoning

Beyond SAT: Modulo Theories

SAT is a good foundation for automated reasoning

Finite problems:

1. "Bit blast" the problem to propositional logic
2. Use latest-and-greatest SAT solver to find a solution
3. Translate back to original domain

Beyond SAT: Modulo Theories

SAT is a good foundation for automated reasoning

Finite problems:

1. "Bit blast" the problem to propositional logic
2. Use latest-and-greatest SAT solver to find a solution
3. Translate back to original domain

SMT: Sat Modulo Theories

Beyond SAT: Modulo Theories

SAT is a good foundation for automated reasoning

Finite problems:

1. "Bit blast" the problem to propositional logic
2. Use latest-and-greatest SAT solver to find a solution
3. Translate back to original domain

SMT: Sat Modulo Theories

Richer way to model problems:

Beyond SAT: Modulo Theories

SAT is a good foundation for automated reasoning

Finite problems:

1. "Bit blast" the problem to propositional logic
2. Use latest-and-greatest SAT solver to find a solution
3. Translate back to original domain

SMT: Sat Modulo Theories

Richer way to model problems:

- Allow predicates from selected *background theories*

$$(x_1 \geq 0) \wedge (x_1 \leq 10) \wedge \mathsf{rd}(\mathsf{wr}(P, x_2, x_3), x_1 + x_2) = x_3 + 1$$

Beyond SAT: Modulo Theories

SAT is a good foundation for automated reasoning

Finite problems:

1. "Bit blast" the problem to propositional logic
2. Use latest-and-greatest SAT solver to find a solution
3. Translate back to original domain

SMT: Sat Modulo Theories

Richer way to model problems:

- Allow predicates from selected *background theories*
- Combine theory-specific reasoning with approaches from SAT

$$(x_1 \geq 0) \wedge (x_1 \leq 10) \wedge \mathsf{rd}(\mathsf{wr}(P, x_2, x_3), x_1 + x_2) = x_3 + 1$$

Beyond SAT: Modulo Theories

SAT is a good foundation for automated reasoning

Finite problems:

1. "Bit blast" the problem to propositional logic
2. Use latest-and-greatest SAT solver to find a solution
3. Translate back to original domain

SMT: Sat Modulo Theories

Richer way to model problems:

- Allow predicates from selected *background theories*
- Combine theory-specific reasoning with approaches from SAT
- Supports infinite domains

$$(x_1 \geq 0) \wedge (x_1 \leq 10) \wedge \mathsf{rd}(\mathsf{wr}(P, x_2, x_3), x_1 + x_2) = x_3 + 1$$

Reasoning About Programs

```
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
  int[] B = alloc_array(int, n);

  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    B[i] = A[i];
  }

  return B;
}
```

Reasoning About Programs

Functional Correctness

- Specification
- Proof

```
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
  int[] B = alloc_array(int, n);

  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    B[i] = A[i];
  }

  return B;
}
```

Reasoning About Programs

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise
- Amenable to proof

```
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
  int[] B = alloc_array(int, n);

  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    B[i] = A[i];
  }

  return B;
}
```

Reasoning About Programs

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise
- Amenable to proof

Systematic proof techniques

- Based on language semantics
- Well-defined proof rules
- Ideally, automatable

```
int[] array_copy(int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@ensures \length(\result) == n;
{
  int[] B = alloc_array(int, n);

  for (int i = 0; i < n; i++)
  //@loop_invariant 0 <= i;
  {
    B[i] = A[i];
  }

  return B;
}
```

Dafny

A language and verifier for functional correctness

Dafny

A language and verifier for functional correctness

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

Dafny

A language and verifier for functional correctness

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

```
predicate sorted(a: array<int>)
   requires a != null
   reads a
{
   forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
method BinarySearch(a: array<int>, val: int) returns (idx: int)
   requires a != null && 0 <= a.Length && sorted(a)
   ensures 0 <= idx ==> idx < a.Length && a[idx] == val
   ensures idx < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != val
{
   var low, high := 0, a.Length;
   while low < high
```

Dafny

A language and verifier for functional correctness

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

Compiler checks everything statically!

- SMT solver under the hood

```
predicate sorted(a: array<int>)
   requires a != null
   reads a
{
   forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
method BinarySearch(a: array<int>, val: int) returns (idx: int)
   requires a != null && 0 <= a.Length && sorted(a)
   ensures 0 <= idx ==> idx < a.Length && a[idx] == val
   ensures idx < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != val
{
   var low, high := 0, a.Length;
   while low < high
```

Dafny

A language and verifier for functional correctness

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

Compiler checks everything statically!

- SMT solver under the hood

Used to build real systems

```
predicate sorted(a: array<int>)
   requires a != null
   reads a
{
   forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
}
method BinarySearch(a: array<int>, val: int) returns (idx: int)
   requires a != null && 0 <= a.Length && sorted(a)
   ensures 0 <= idx ==> idx < a.Length && a[idx] == val
   ensures idx < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != val
{
   var low, high := 0, a.Length;
   while low < high
```

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 systems

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 systems

See how to build them for:

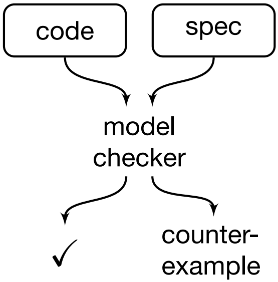
- Efficiency
- Extensibility

Basic idea:

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

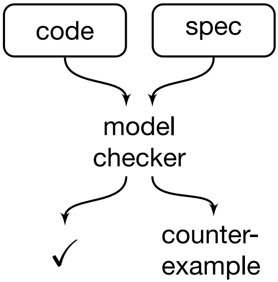
Automatic **techniques for finding bugs (or proving their absence)**

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



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

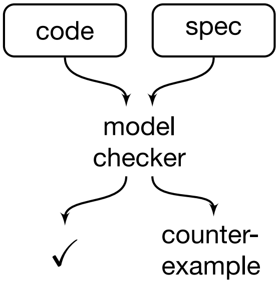
- Specifications written in *propositional temporal logic*



Model Checking

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

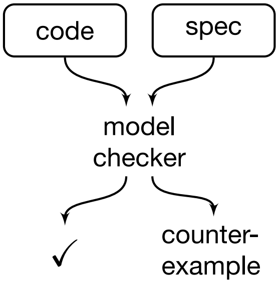
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search



Model Checking

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

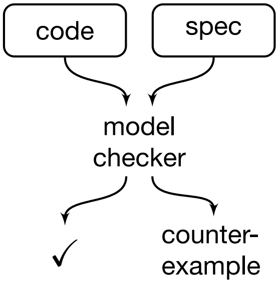
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples



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 manual proofs!



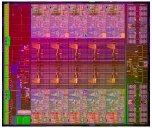
Model Checking

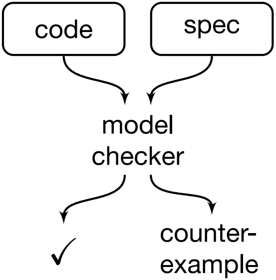
***Automatic* techniques for finding bugs (or proving their absence)**

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

10^{70} atoms

10^{500000} states





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, …

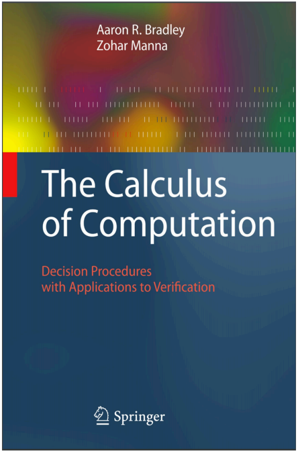
Invented here at CMU

Ed Clarke
Turing Award,
2007

Textbook

Free PDF available on campus network

Buy hardcover from Amazon, Springer



`http://vufind.library.cmu.edu/vufind/Record/1607219`

Grading

Breakdown:

- 50% assignments
- 25% final exam
- 20% midterm
- 5% participation

Between 6-8 assignments

Some pen-and-paper, some programming

Written portions: hand in PDF from LaTeX

In-class exams

Participation:

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

Late Policy

Two days of “grace period” throughout semester

- We count in days, not hours or minutes
- One assignment, two days late
- Two assignments, one day late
- You decide...

Notify **both** instructor and TA when handing in late

Assignments receive no credit if turned in late:

- without notification, or
- past grace period

Logistics

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

Lecture: Tuesdays & Thursdays, 10:30-11:50 GHC 4211

Matt Fredrikson

- Location: CIC 2126
- Office Hours: Mondays & Wednesdays 1-2pm, or by appointment
- Email: mfredrik@cs

Ryan Wagner

- Location: Wean 4109
- Office Hours: Tuesdays & Thursdays 1-2pm
- Email: rrwagner@cs

Next Lecture

Propositional Logic

Reading: Chapter 1 of Bradley & Manna, through 1.5