Bug Catching: Automated Program Verification
15414/15614 Spring 2024
Lecture 1: Introduction

Ruben Martins

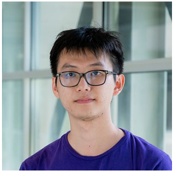
January 16, 2024

Course staff

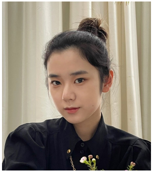
Ruben Martins

Cayden Codel

Pratap Singh



Tony Yu



Ying Sun

Learning objectives

For this lecture

- What is this course about?
- What are the learning objectives for the course?
- How does it fit into the curriculum?
- How does the course work?

Disastrous Software and Hardware Bugs

1962 - Mariner 1 probe

- Destroyed 293 seconds after launch
- Overbar missing in the mathematical specification
- Cost: $18.5 million dollars

1996 - Ariane 5 fight 501

- Destroyed 37 seconds after launch
- Arithmetic overflow from 64-bit floating point to 16-bit signed integer
- Cost: $370 million dollars

1994 - Pentium FDIV bug

- Flaw in the division algorithm:
- 4195835.0/3145727.0 = 1.333 820 449 136 241 000 (Correct)
- 4195835.0/3145727.0 = 1.333 739 068 902 037 589 (FDIV)
- Cost: $475 million dollars

Bad code

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

Bad code

- **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."

Bad code

- **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

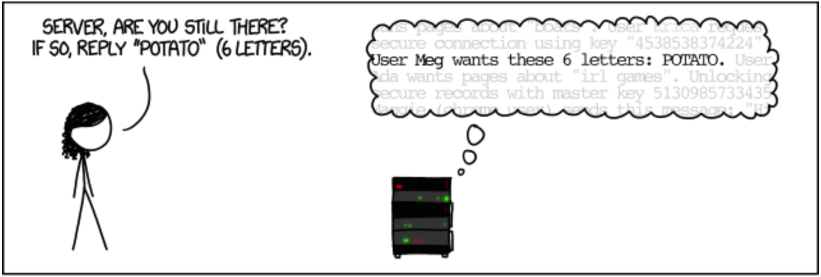


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

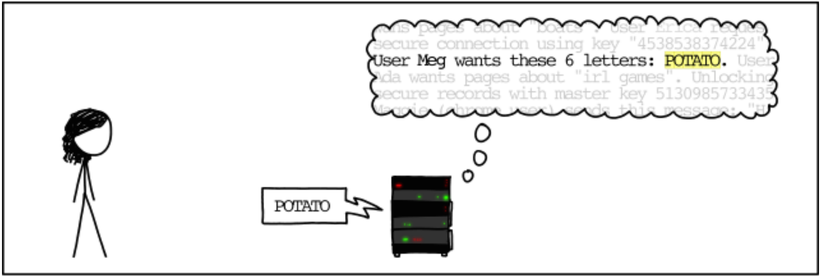


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

Image source: Randall Munroe, xkcd.com

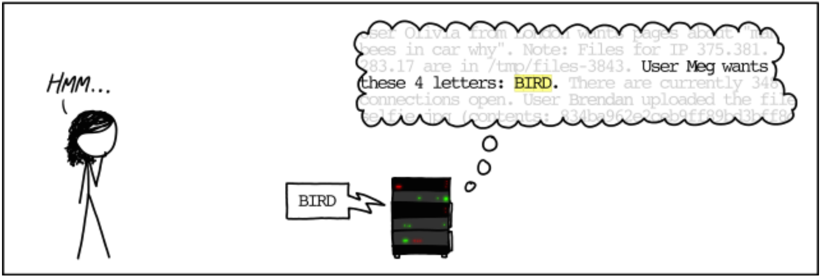


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

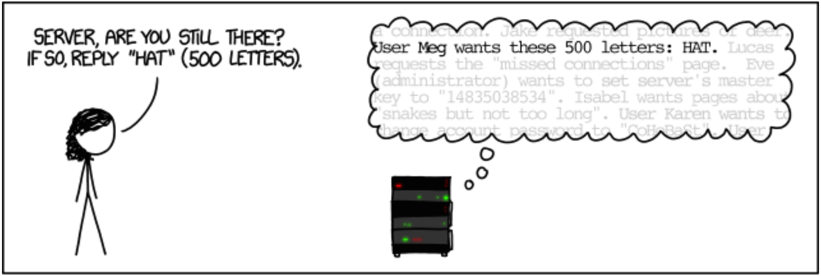


Image source: Randall Munroe, xkcd.com

Heartbleed, explained

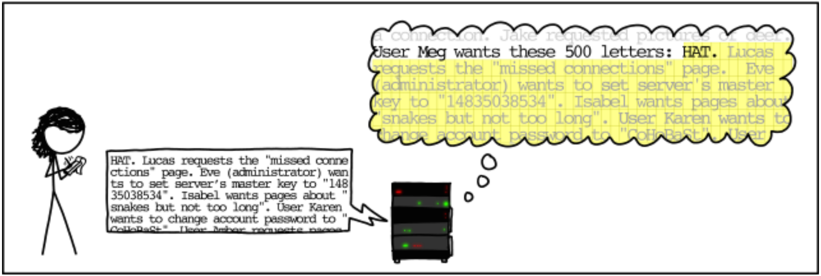


Image source: Randall Munroe, xkcd.com

The cost

- Hard to say, but estimates are ~$500 million

The cost

- Hard to say, but estimates are ~$500 million
  - Stolen data
  - Certificate revocation
  - Bandwidth
  - Engineering effort
  - ...

The cost

- Hard to say, but estimates are ~$500 million
  - Stolen data
  - Certificate revocation
  - Bandwidth
  - Engineering effort
  - ...

Tech giants spend millions to stop another Heartbleed

25 April 2014

The cost

- Hard to say, but estimates are ~$500 million
  - Stolen data
  - Certificate revocation
  - Bandwidth
  - Engineering effort
  - ...

Tech giants spend millions to stop another Heartbleed

25 April 2014

EXPLOITS AND VULNERABILITIES | NEWS

Five years later, Heartbleed vulnerability still unpatched

Posted: September 12, 2019 by Gilad Maayan

Algorithms vs. code

```
int binarySearch(int key, int[] a, int n) {
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = (low + high) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -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 (that is, arbitrary) 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 (that is, arbitrary) behavior

Algorithm may be correct—but we run code, not algorithms.

How do we fix it?

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

How do we fix it?

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

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

The fix

```
int binarySearch(int key, int[] a, int n) {
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

The fix

```
int binarySearch(int key, int[] a, int n)
//@requires 0 <= n && n <= \length(A);
{
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

The fix

```
int binarySearch(int key, int[] a, int n)
//@requires 0 <= n && n <= \length(a);
/*@ensures (\result == -1 && !is_in(key, A, 0, n))
  @      || (0 <= \result && \result < n
  @            && A[\result] == key); @*/
{
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

The fix

```
int binarySearch(int key, int[] a, int n)
//@requires 0 <= n && n <= \length(a);
//@requires is_sorted(a, 0, n);
/*@ensures (\result == -1 && !is_in(key, A, 0, n))
  @      || (0 <= \result && \result < n
  @           && A[\result] == key); @*/
{
    int low = 0;
    int high = n;

    while (low < high) {
        int mid = low + (high - low) / 2;

        if(a[mid] == key) return mid; // key found
        else if(a[mid] < key) {
            low = mid + 1;
        } else {
            high = mid;
        }
    }
    return -1;  // key not found.
}
```

How do we know if it's correct?

How do we know if it's correct?

One solution: testing

How do we know if it's correct?

One solution: testing

- Probably incomplete
- Never really sure what's left...

How do we know if it's correct?

One solution: testing

- Probably incomplete
- Never really sure what's left...

Another: code review

How do we know if it's correct?

One solution: testing

- Probably incomplete
- Never really sure what's left...

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallible, bugs are subtle
- What's the specification?

How do we know if it's correct?

One solution: testing

- Probably incomplete
- Never really sure what's left...

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallible, bugs are subtle
- What's the specification?

Another: proof

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

How do we know if it's correct?

One solution: testing

- Probably incomplete
- Never really sure what's left...

Another: code review

- Correctness definitely important, but not the only thing
- Humans are fallible, bugs are subtle
- What's the specification?

Another: proof

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

- Specification must be **precise**
- Meaning of code must be **comprehensive**
- Reasoning must be **sound**

Course objectives

Course objectives

- Identify and formalize program correctness
- Understand language semantics
- 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

Course objectives

- Identify and formalize program correctness
- Understand language semantics
- 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
- Make you better programmers

Course outline

Course outline

Part I: Reasoning about programs: from 122 and 150 to 414

- Gain intuitive understanding of language and methodology

Course outline

Part I: Reasoning about programs: from 122 and 150 to 414

- Gain intuitive understanding of language and methodology

Part II: From inform to formal reasoning

- Specifying meaning of programs
- Specifying meaning of propositions
- Formal reasoning and its justification

Course outline

Part I: Reasoning about programs: from 122 and 150 to 414

- Gain intuitive understanding of language and methodology

Part II: From inform to formal reasoning

- Specifying meaning of programs
- Specifying meaning of propositions
- Formal reasoning and its justification

Part III: Mechanized reasoning

- Techniques for automated proving

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

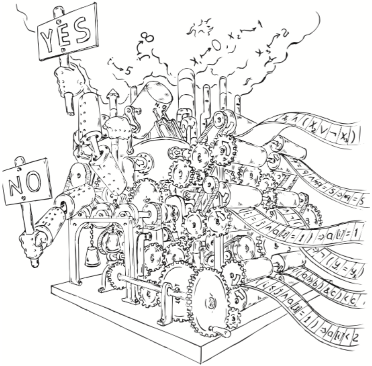


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info

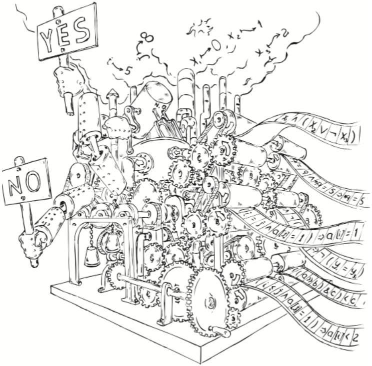


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info

They usually cannot:

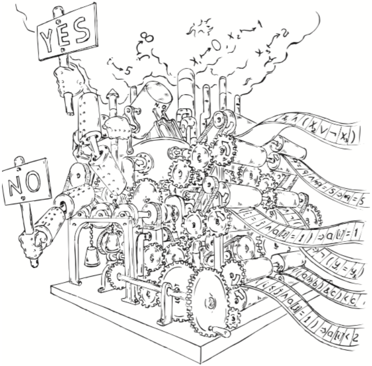


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info

They usually cannot:

- Verify "everything" for us
- Generate specification, invariants
- Tell us how to fix bugs

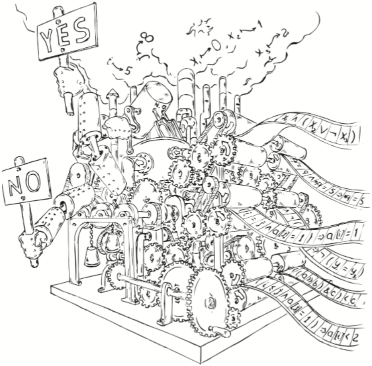


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Algorithmic approaches

Formal proofs are tedious

Automatic methods can:

- Check our work
- Fill in low-level details
- Give diagnostic info

They usually cannot:

- Verify "everything" for us
- Generate specification, invariants
- Tell us how to fix bugs

This is what you will learn!

- Make use of these methods
- How (and when) they work

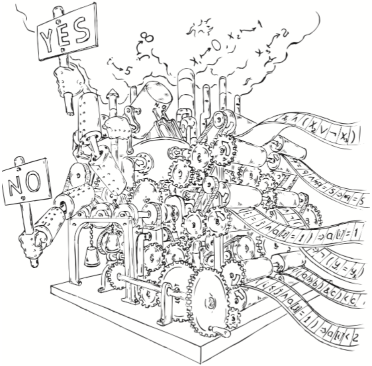


Image source: Daniel Kroening & Ofer Strichman, *Decision Procedures*

Different traditions and techniques

Functional programming: dependent types

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Why3 (this course) supports both!

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Why3 (this course) supports both!

- Functional and imperative code in WhyML

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Why3 (this course) supports both!

- Functional and imperative code in WhyML
- Automated provers for VC (Z3, CVC, alt-ergo, ...)

Different traditions and techniques

Functional programming: dependent types

- Proofs are expressed in programs (Agda)
- Proof tactics are expressed as programs (Coq)

Imperative programming: logical contracts

- Properties are expressed in contracts
- Reduce correctness to logical propositions (verification condition)
- Use automated theorem provers to prove VC

Why3 (this course) supports both!

- Functional and imperative code in WhyML
- Automated provers for VC (Z3, CVC, alt-ergo, ...)
- Interactive provers for VC (Coq)

We focus on automated proving

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Systematic proof techniques

- Derived from semantics
- Exhaustive proof rules
- Automatable*

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Systematic proof techniques

- Derived from semantics
- Exhaustive proof rules
- 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;
}
```

Reasoning about correctness

Functional Correctness

- Specification
- Proof

Specify behavior with logic

- Declarative
- Precise

Systematic proof techniques

- Derived from semantics
- Exhaustive proof rules
- 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;
}
```

But . . .

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Rich specification language

- Pre- and post-conditions, loop invariants, assertions
- Pure mathematical functions
- Termination metrics

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Rich specification language

- Pre- and post-conditions, loop invariants, assertions
- Pure mathematical functions
- Termination metrics

Programmer writes specification, partial annotations

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Rich specification language

- Pre- and post-conditions, loop invariants, assertions
- Pure mathematical functions
- Termination metrics

Programmer writes specification, partial annotations

Compiler proves correctness

Why3

Deductive verification platform

- Programming language (WhyML, derived from OCaml)
- Verification toolchain

Rich specification language

- Pre- and post-conditions, loop invariants, assertions
- Pure mathematical functions
- Termination metrics

Programmer writes specification, partial annotations

Compiler proves correctness

When it works! (It's not quite like a type-checker ...)

Automated verifiers

Systems that prove 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 verifiers

Systems that prove that programs match their specifications

Problem is undecidable!

1. Prover needs "hints" from programmer
2. Finding the right set of hints can be challenging

Basic idea:

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

Automated verifiers

Systems that prove that programs match their specifications

Problem is undecidable!

1. Prover needs "hints" from programmer
2. Finding the right set of hints can be challenging

Verifiers are complex systems

- We'll deep-dive into selected components
- Understand "big picture" for the rest

Basic idea:

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

Recent developments

Recent developments

```
Implement a `sat` function in Why3 that matches this signature:

    sat (cnf : cnf) : option valuation

If `cnf` is satisfiable, then `sat` should return `Some rho`,
where `rho` is a satisfying assignment for `cnf`.
Otherwise, it should return `None`.
```

Recent developments

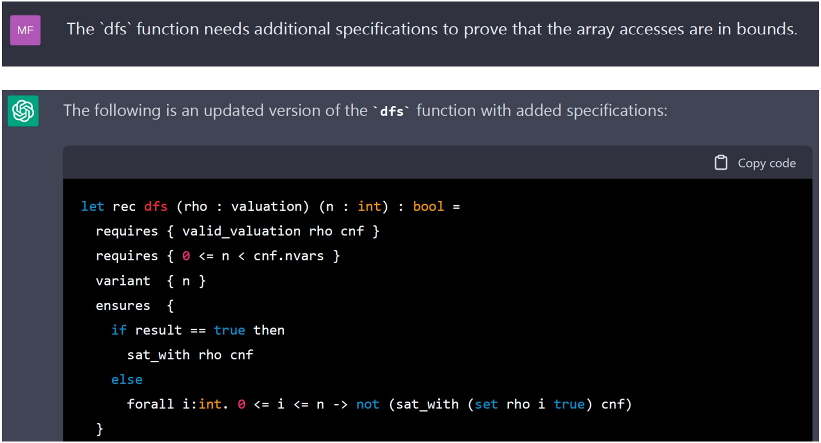
```
let rec sat (cnf : cnf) : option valuation =
  ensures  {
    match result with
    | None -> unsat cnf
    | Some rho -> sat_with rho cnf
  }
  let rho = Array.make cnf.nvars False in
  let rec dfs (n : int) : bool =
    requires { valid_valuation rho cnf }
    requires { 0 <= n <= cnf.nvars }
    variant  { cnf.nvars - n }
    ensures  {
      if result == true then
        sat_with rho cnf
      else
        forall rho':valuation. array_eq_sub rho' rho 0 n -> not (sat_with rho' cnf)
    }
```

Recent developments

MF

The `dfs` function needs additional specifications to prove that the array accesses are in bounds.

Recent developments



Recent developments

MF

The `ensures` specification on `dfs` is not correct. Specifying `sat_with (set rho i true) cnf` for all `i` in the range from `0` to `n` is not relevant; what is relevant is that if `result <> true`, then `unsat cnf`.

Recent developments

MF The `ensures` specification on `dfs` is not correct. Specifying `sat_with (set rho i true) cnf` for all `i` in the range from `0` to `n` is not relevant; what is relevant is that if `result <> true`, then `unsat cnf`.

The following is an updated version of the **`dfs`** function with a corrected **`ensures`** specification:

Copy code

```
let rec dfs (rho : valuation) (n : int) : bool =
  requires { valid_valuation rho cnf }
  requires { 0 <= n < cnf.nvars }
  variant  { n }
  ensures  {
    if result == true then
      sat_with rho cnf
    else
      unsat cnf
  }
```

(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Convincing it to do so was not trivial!

(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Convincing it to do so was not trivial!

If you use a code model/assistant in this course:

(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Convincing it to do so was not trivial!

If you use a code model/assistant in this course:

- Document your approach thoroughly

(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Convincing it to do so was not trivial!

If you use a code model/assistant in this course:

- Document your approach thoroughly
- Describe the relevant parts in your solution

(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Convincing it to do so was not trivial!

If you use a code model/assistant in this course:

- Document your approach thoroughly
- Describe the relevant parts in your solution
- Come talk to me about it

(Limitations of) recent developments

Eventually, ChatGPT produced a verified implementation

Convincing it to do so was not trivial!

If you use a code model/assistant in this course:

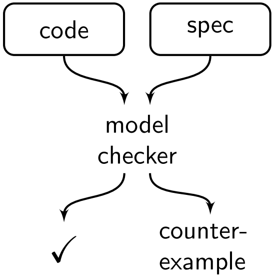
- Document your approach thoroughly
- Describe the relevant parts in your solution
- Come talk to me about it

If you are interested in research related to this, let me know!

Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

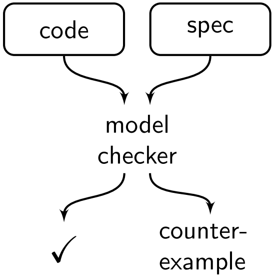
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples



Model Checking

Fully-automatic techniques for finding bugs (or proving their absence)

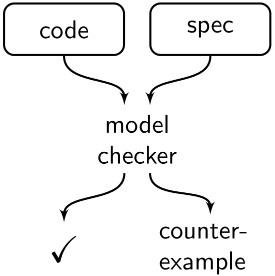
- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No proofs!



Model Checking

Fully-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"



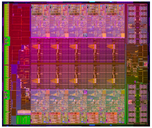
Model Checking

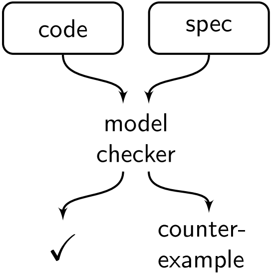
Fully-automatic techniques for finding bugs (or proving their absence)

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

10^{70} atoms

10^{500000} states





Model Checking

Clever ways of dealing with state explosion:

Model Checking

Clever ways of dealing with state explosion:

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

Model Checking

Clever ways of dealing with state explosion:

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

Now widely used for bug-finding:

- Hardware, software, protocols, ...
- Microsoft, Intel, Amazon, Google, NASA, ...

Model Checking

Clever ways of dealing with state explosion:

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

Now widely used for bug-finding:

- Hardware, software, protocols, . . .
- Microsoft, Intel, Amazon, Google, NASA, . . .

Ed Clarke, 1945–2020
Turing Award, 2007

Model Checking

Clever ways of dealing with state explosion:

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

Now widely used for bug-finding:

- Hardware, software, protocols, ...
- Microsoft, Intel, Amazon, Google, NASA, ...

Ed Clarke, 1945–2020
Turing Award, 2007
First developed this course!

Grading

Breakdown:

- 50% assignments
  (written + programming)
- 15% mini-project 1
- 15% mini-project 2
- 20% final exam

8 assignments
done individually

2 mini-projects
pick from small menu
can work with a partner

Participation:

- Come to lecture
- Answer questions
  (in class and on Piazza!)
- Contribute to discussion

Written parts of assignments

Written homeworks focus on theory and fundamental skills

Written parts of assignments

Written homeworks focus on theory and fundamental skills

Grades are based on:

- Correctness of your answer
- How you present your reasoning

Written parts of assignments

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 these $\longrightarrow$ no points

Programming parts of assignments

For the programming, you will:

- Implement some functionality (data structure or algorithm)
- Specify correctness for that functionality
- Use Why3 to prove it correct

Programming parts of assignments

For the programming, you will:

- Implement some functionality (data structure or algorithm)
- Specify correctness for that functionality
- Use Why3 to prove it correct

Most important criterion is **correctness**.

Programming parts of assignments

For the programming, you will:

- Implement some functionality (data structure or algorithm)
- Specify correctness for that functionality
- Use Why3 to prove it correct

Most important criterion is **correctness**.

Full points when you provide the following

- Correct implementation
- Correct specification
- Correct contracts
- Sufficient contracts for verification

Programming parts of assignments

For the programming, you will:

- Implement some functionality (data structure or algorithm)
- Specify correctness for that functionality
- Use Why3 to prove it correct

Most important criterion is **correctness**.

Full points when you provide the following

- Correct implementation
- Correct specification
- Correct contracts
- Sufficient contracts for verification

Partial credit depending on how many of these you achieve

Clarity & conciseness is necessary for partial credit!

Mini-Projects

Mini-projects are intended to build proficiency in:

- Writing good specifications
- Applying course principles to practice
- Making effective use of automated tools
- **Writing useful & correct code**

Mini-Projects

Mini-projects are intended to build proficiency in:

- Writing good specifications
- Applying course principles to practice
- Making effective use of automated tools
- **Writing useful & correct code**

Gradual progression to sophistication:

1. Familiarize yourself with Why3
2. Implement and prove something
3. Work with more complex data structures
4. Implement and prove something really interesting
5. Optimize your implementation, still verified

Late Policy

Late days

- 5 late days to use throughout the semester
- No more than 2 late days on any assignment
- Late days do not apply to mini-projects!

Logistics

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

Course staff contact: Piazza
`https://piazza.com/cmu/spring2024/15414`

Lecture: Tuesdays & Thursdays, 12:30pm-1:50pm, HH B131

Office Hours: TBD, schedule on website and course calendar soon

Assignments: Gradescope
`https://www.gradescope.com/courses/709329`
(entry code: RK3262)