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?
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?

```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.
}
```
This is a correct binary search algorithm.
This is a correct binary search algorithm.

But what if \( \text{low} + \text{high} > 2^{31} - 1 \)?
This is a correct binary search algorithm.

But what if \(\text{low} + \text{high} > 2^{31} - 1\)?

Then \(\text{mid} = (\text{low} + \text{high}) / 2\) becomes negative.
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
This is a correct binary search algorithm.

But what if \( \text{low} + \text{high} > 2^{31} - 1 \)?

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

- **Best case**: `ArrayIndexOutOfBoundsException`
- **Worst case**: undefined behavior
This is a correct binary search algorithm.

But what if \( \text{low} + \text{high} > 2^{31} - 1 \)?

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

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

Algorithm may be correct—with proof! The code, another story...
Bugs make software insecure

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.

Matt Fredrikson
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
Heartbleed, explained

SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).

User Olivia from London wants pages about "her bees in car why". Note: Files for IP 375.381.283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 348 connections open. User Brendan uploaded the file self.jpg (contents: b34ba962e2ebbf89f43f5ff)

Image source: Randall Munroe, xkcd.com
Heartbleed, explained

User Olivia from London wants pages about "hass bees in car why". Note: Files for IP 375.381. 283.17 are in /tmp/files-3843. User Meg wants these 4 letters: BIRD. There are currently 34 connections open. User Brendan uploaded the file 'file.jpg' (contents: 331ba962e25ebc8f89b13b7f8).

Image source: Randall Munroe, xkcd.com
Heartbleed, explained

Image source: Randall Munroe, xkcd.com
Heartbleed, explained

User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about snakes but not too long. User Karen wants to change account password to "ColHoRaSt". User Jacob requests pages.

Image source: Randall Munroe, xkcd.com
Many, many bugs

Numerical overflow

"Unintended acceleration" bug

Lost $440m in 30 minutes
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

2016, Nissan
1m recalls for buggy airbag code

2000-2010, Toyota
“Unintended acceleration” bug

2012, Knight Capital
Lost $440m in 30 minutes
All about proof
All about proof

Specification $\iff$ Implementation
All about proof

Specification $\iff$ Implementation

- Specifications must be *unambiguous*
- *Meaning* of implementation must be well-defined
All about proof

*Specification* ↔ *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

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.
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*
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
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

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 $\phi$ satisfiable?

We will focus on satisfiability procedures.

We'll look at examples that are:

- Expressive enough to model real problems.
- Still decidable.
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 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 $\phi$ 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 $\phi$ 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 $\phi$ satisfiable?

We will focus on *satisfiability procedures*.

We’ll look at examples that are:
- Expressive enough to model real problems.
- Still decidable.
## Propositional Logic

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>¬</td>
<td>Not</td>
</tr>
<tr>
<td>∧</td>
<td>And</td>
</tr>
<tr>
<td>∨</td>
<td>Or</td>
</tr>
<tr>
<td>→</td>
<td>Implies</td>
</tr>
<tr>
<td>⇔</td>
<td>Equivalent</td>
</tr>
</tbody>
</table>
## Propositional SAT

<table>
<thead>
<tr>
<th>Propositional Logic</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>¬</td>
<td>Not</td>
</tr>
<tr>
<td>∧</td>
<td>And</td>
</tr>
<tr>
<td>∨</td>
<td>Or</td>
</tr>
<tr>
<td>→</td>
<td>Implies</td>
</tr>
<tr>
<td>↔</td>
<td>Equivalent</td>
</tr>
</tbody>
</table>

### SAT Problem

Given a propositional formula $F$ over variables $p_1, p_2, \ldots$, find an assignment $I = [p_1 \mapsto \_, p_2 \mapsto \_, \ldots]$ that satisfies $F$. Lots of important applications…

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

Matt Fredrikson

Model Checking
Propositional SAT

Propositional Logic

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>¬</td>
<td>Not</td>
</tr>
<tr>
<td>∧</td>
<td>And</td>
</tr>
<tr>
<td>∨</td>
<td>Or</td>
</tr>
<tr>
<td>→</td>
<td>Implies</td>
</tr>
<tr>
<td>↔</td>
<td>Equivalent</td>
</tr>
</tbody>
</table>

SAT Problem

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

Lots of important applications…

- Verification
- Program synthesis
- Test generation
- Equivalence checking
- Combinatorial design
- Cryptanalysis
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.
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*
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
SAT is a good foundation for automated reasoning

SMT: Sat Modulo Theories

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
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

\[(x_1 \geq 0) \land (x_1 \leq 10) \land \text{rd}(\text{wr}(P, x_2, x_3), x_1 + x_2) = x_3 + 1\]
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

\[(x_1 \geq 0) \land (x_1 \leq 10) \land \text{rd}(\text{wr}(P, x_2, x_3), x_1 + x_2) = x_3 + 1\]
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

\[(x_1 \geq 0) \land (x_1 \leq 10) \land \text{rd}(\text{wr}(P, x_2, x_3), x_1 + x_2) = x_3 + 1\]
Reasoning About Programs

Specify behavior with logic
▶ Declarative
▶ Precise
▶ Amenable to proof

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

Matt Fredrikson

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

```java
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

```c
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 \forall j < i \implies B[j] = A[j];
    {
        B[i] = A[i];
    }
    return B;
}
```
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

```java
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;
}
```
A language and verifier for functional correctness
A language and verifier for functional correctness

- Pre- and postconditions, assertions
- Pure mathematical functions
- Termination metrics
A language and verifier for functional correctness

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

```csharp
1 predicate sorted(a: array<int>)
2   requires a != null
3   reads a
4 {
5    forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
6 }
7 method BinarySearch(a: array<int>, val: int) returns (idx: int)
8   requires a != null && 0 <= a.Length && sorted(a)
9   ensures 0 <= idx ==> idx < a.Length && a[idx] == val
10  ensures idx < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != val
11 {
12    var low, high := 0, a.Length;
13    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

```d
1 predicate sorted(a: array<int>)
2     requires a != null
3     reads a
4 {
5     forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
6 }
7 method BinarySearch(a: array<int>, val: int) returns (idx: int)
8     requires a != null && 0 <= a.Length && sorted(a)
9     ensures 0 <= idx ==> idx < a.Length && a[idx] == val
10    ensures idx < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != val
11 {
12    var low, high := 0, a.Length;
13    while low < high
```
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

```plaintext
1 predicate sorted(a: array<int>)
2   requires a != null
3   reads a
4 {
5     forall j, k :: 0 <= j < k < a.Length ==> a[j] <= a[k]
6 }
7 method BinarySearch(a: array<int>, val: int) returns (idx: int)
8   requires a != null && 0 <= a.Length && sorted(a)
9   ensures 0 <= idx ==> idx < a.Length && a[idx] == val
10  ensures idx < 0 ==> forall k :: 0 <= k < a.Length ==> a[k] != val
11 {
12    var low, high := 0, a.Length;
13    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
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 \textit{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)

- Specifications written in propositional temporal logic
- Verification by exhaustive state space search
- Diagnostic counterexamples
- No manual proofs!
- Downside: "State explosion"
Automatic techniques for finding bugs (or proving their absence)

- Specifications written in propositional temporal logic

Diagram:
- Code
- Specification
- Model checker
- Counterexample

Matt Fredrikson
Automatic techniques for finding bugs (or proving their absence)

- Specifications written in \textit{propositional temporal logic}
- Verification by exhaustive state space search
**Automatic** techniques for finding bugs (or proving their absence)

- Specifications written in *propositional temporal logic*
- Verification by exhaustive state space search
- Diagnostic counterexamples
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!
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
Clever ways of dealing with state explosion:
Clever ways of dealing with state explosion:

- Partial order reduction
- Bounded model checking
- Symbolic exploration
- Abstraction & refinement
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, …
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
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
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
Propositional Logic

Reading: Chapter 1 of Bradley & Manna, through 1.5