Program Analysis for Software Security

Lecture 1

Grading

Seminar activity (2-3 practical assignments + 1 oral presentation) :**--50%**

Final exam: **-- 50%**

- Final Written Exam (open books): **--50%**

- To pass the exam you have to obtain minim 5 at the final exam and the final grade is minimum 5

Rules

- seminar activity will be done at the group level
- groups consist of max 2 students
- final exam is individual and is an open book exam (you can have access at the lecture notes and the seminar notes)

Course Content

Please join the Course Teams- code:

st0nimi

In this course we will discuss about

Automated Program Verification

using the **VIPER tool**.

Important NOTE

Some of the slides are taken from

Peter Muler, ETH Zurich

and

Christoph Matheja, DTU, Denmark

-

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger W. Dijkstra (1970)

Outline

1. Why Program Verification?
2. Course Overview
3. Starting with Viper

How much confidence do we have in computer systems?

more confidence

no confidence

extensive testing

Testing is insufficient

- 1994 Intel® Pentium® Floating-point Division bug
- Estimate: 1 in 9 billion floating-point divisions inaccurate
- Issue: missing entries in the lookup table
- Recall losses: $475 million (> 5 billion DKK in 2019)
- Bug was detected during experiments on number theory

How much confidence do we have in computer systems?

more confidence

extensive testing

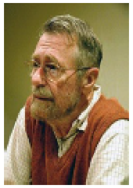
no confidence

OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case*

Stijn de Gouw[1,2], Jurriaan Rot[3,1], Frank S. de Boer[1,3], Richard Bubel[4], and Reiner Hähnle[4]

- TimSort: default sorting algorithm in OpenJDK and Android SDK
- Certain large arrays (>= 67M) lead to index-out-of-bounds errors
- Multiple attempts to fix related errors were ineffective

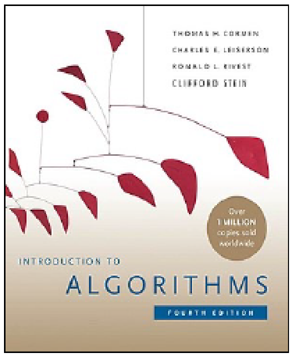
> Program testing can be very effective to show the presence of bugs, but it is hopelessly inadequate for showing their absence.



Edsger W. Dijkstra

How much confidence do we have in computer systems?

more confidence



correctness arguments

extensive testing

no confidence

> The only effective way to raise the confidence level of a program is to give a convincing proof of its correctness.

Edsger W. Dijkstra

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

At the beginning of each loop iteration:

1. If $p \leq k \leq i$, then $A[k] \leq x$.
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

credits: Cormen et al., Introduction to Algorithms, 2009

Textbook-style correctness arguments are insufficient

- 2006 implementation of binary search in `java.util.Arrays`
  - Problem: `mid` might overflow!
  - This bug was part of the Java standard library for approximately nine years
- Faithful implementation of algorithm from *Programming Pearls, Bentley, 1986*
  - Correctness argument: "sets `mid` to the average of `low` and `high`, truncated down to the nearest integer"
  - It was inconceivable at the time to have arrays of length 2^{30} or greater
- Same issue arises for other divide-and-conquer algorithms

```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
}
```

How much confidence do we have in computer systems?

more confidence

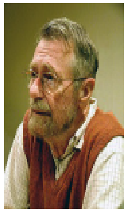
rigorous proofs

correctness arguments

extensive testing

no confidence

> The only effective way to raise the confidence level of a program is to give a convincing proof of its correctness.



Edsger W. Dijkstra,
ACM Turing Lecture 1972

```
{ true }
if (x < 0) {
  { -x + 1 = |-x| + 1 }
  x := -x
  { x + 1 = |x| + 1 }
}
{ x + 1 = |x| + 1 }
y := x + 1
{ y = |x| + 1 }
```

Handwritten proofs are insufficient

Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Ion Stoica*, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan†
MIT Laboratory for Computer Science
chord@lcs.mit.edu
http://pdos.lcs.mit.edu/chord/

- Chord is a distributed hash table developed at MIT

> Three features that distinguish Chord from many other peer-to-peer lookup protocols are its simplicity, provable correctness, and provable performance.

- None of the seven properties claimed invariant of the original version is actually an invariant

> *"Beware of bugs in the above code;*
> *I have only proved it correct, not tried it."*
> Donald Knuth, 1977

How much confidence do we have in computer systems?

more confidence

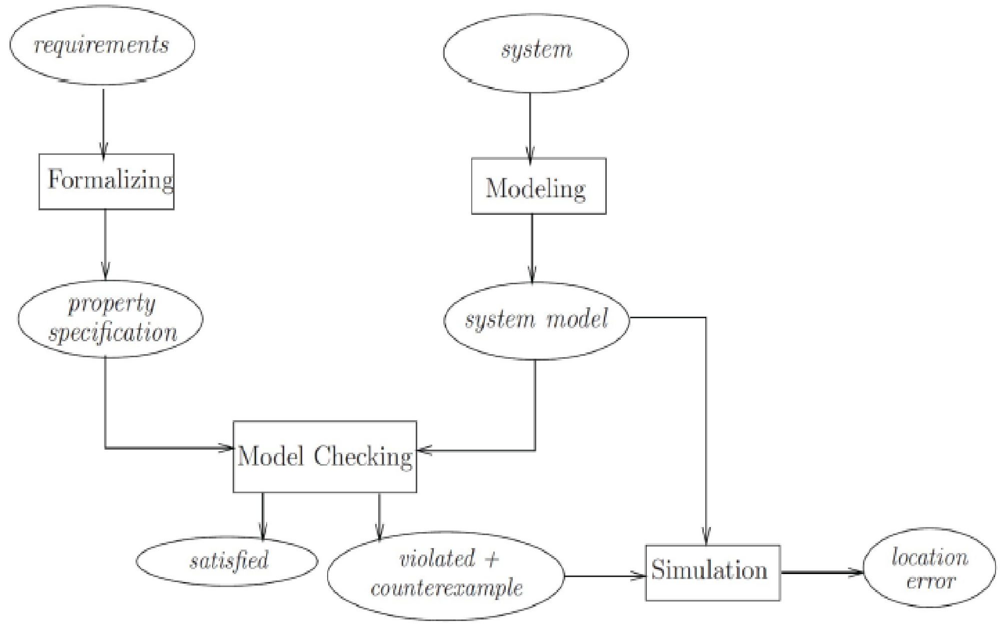
verified models

rigorous proofs

correctness arguments

extensive testing

no confidence



if model = system, we attempt to generate
a rigorous proof by exhaustive testing

credits: Baier & Katoen, Principles of Model Checking, 2008

Verification of system models is insufficient

> *"Any verification using model-based techniques is only as good as the model of the system." – Baier & Katoen 2008*

- No guarantees for the actual system
- Model checking suffers from the state-space explosion problem
- Not generally applicable to infinite-state systems
  - Ill-suited if data ranges over infinite domains (e.g., dynamic memory allocation)
  - Does not work well for systems with an arbitrary number of components

How much confidence do we have in computer systems?

more confidence

machine-checked proofs

our focus: deductive verification tools

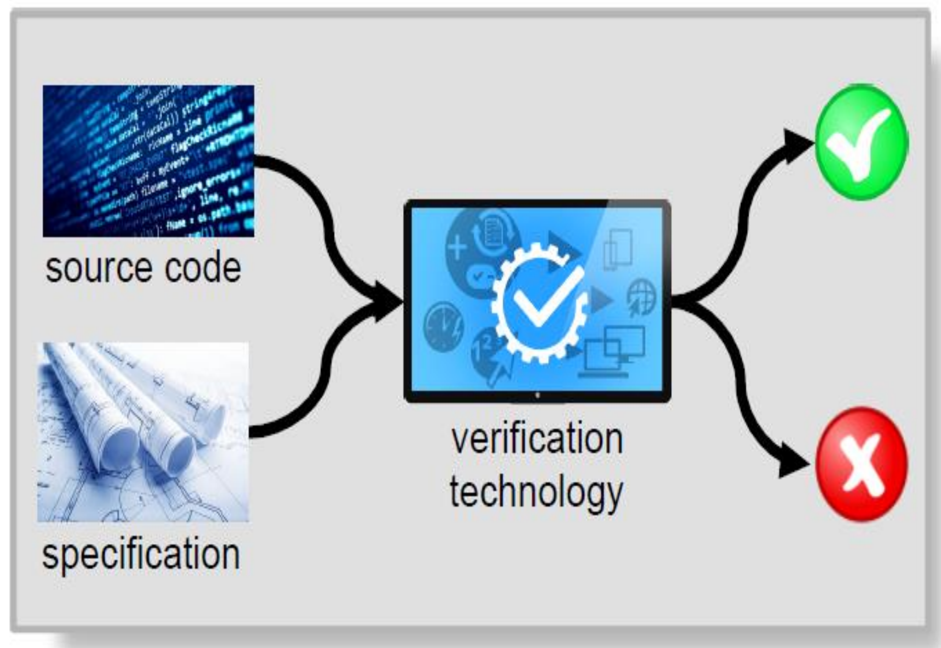
verified models

(handwritten) rigorous proofs

correctness arguments

extensive testing

no confidence



Interactive verification

- Successful large-scale applications of interactive verification tools:
  - CompCert: a formally verified C compiler (2008)
  - seL4: a formally verified high-performance operating system microkernel (2009)
  - EveryCrypt: a formally verified crypto library (2020)

- Strengths
  - Expressive foundation (higher-order logic)
  - Can handle complex systems and properties

- Weaknesses
  - Requires expert knowledge
  - Very labor-intensive (CompCert required more than 6 person years)

Automated (or auto-active) Verification

- Idea: "use verification like compilation"
  - Specifications take the form of source code annotations
  - Analogies: TypeScript, Rust ownership & traits, Python type hints

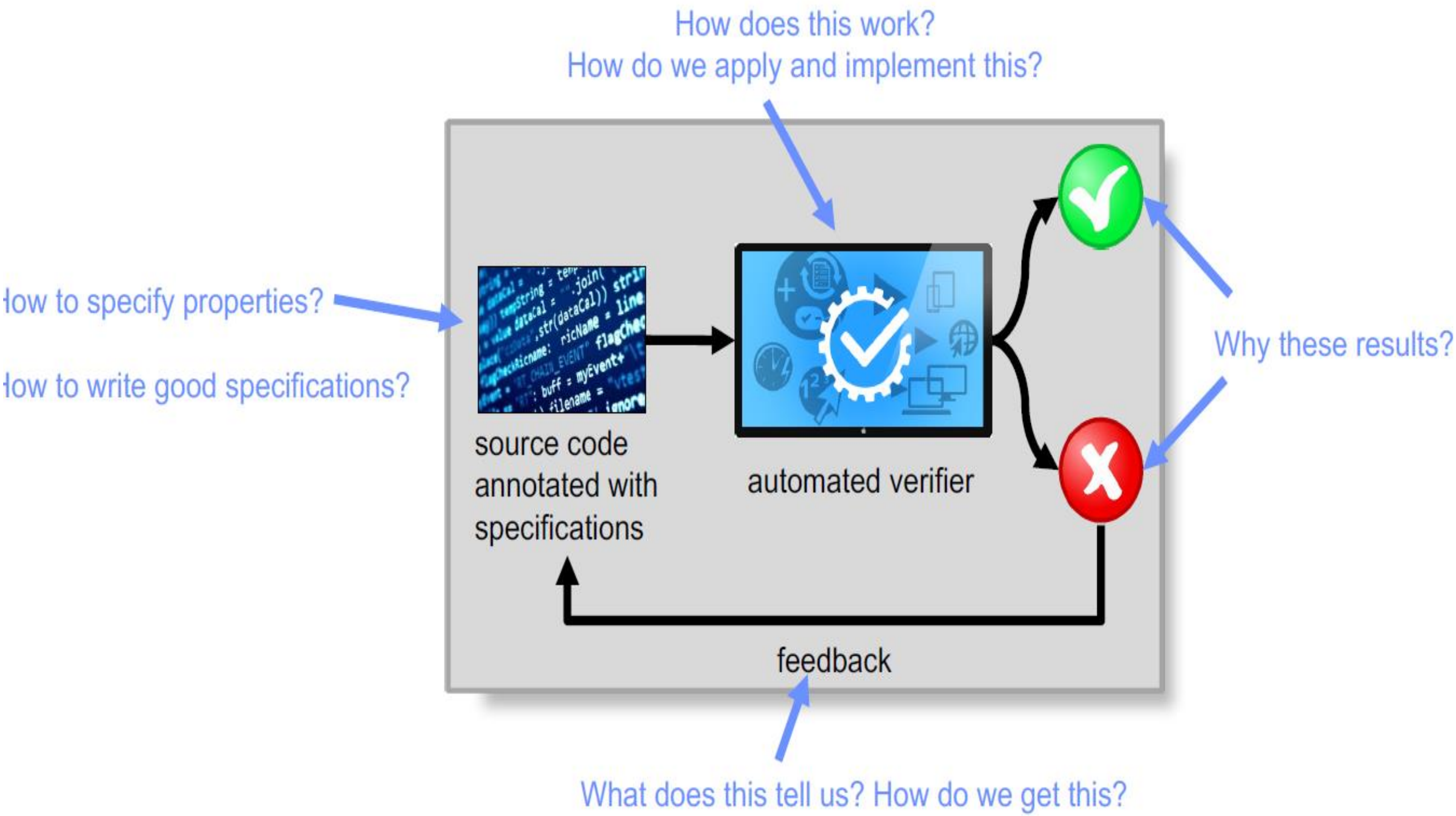
- Strengths:
  - Substantially less effort than interactive verification
  - Integrates into existing development processes
  - More annotations ➔ more correctness guarantees

- Weaknesses:
  - Less expressive than interactive verification
  - May produce false positives (due to undecidability)
  - Still requires effort and expertise

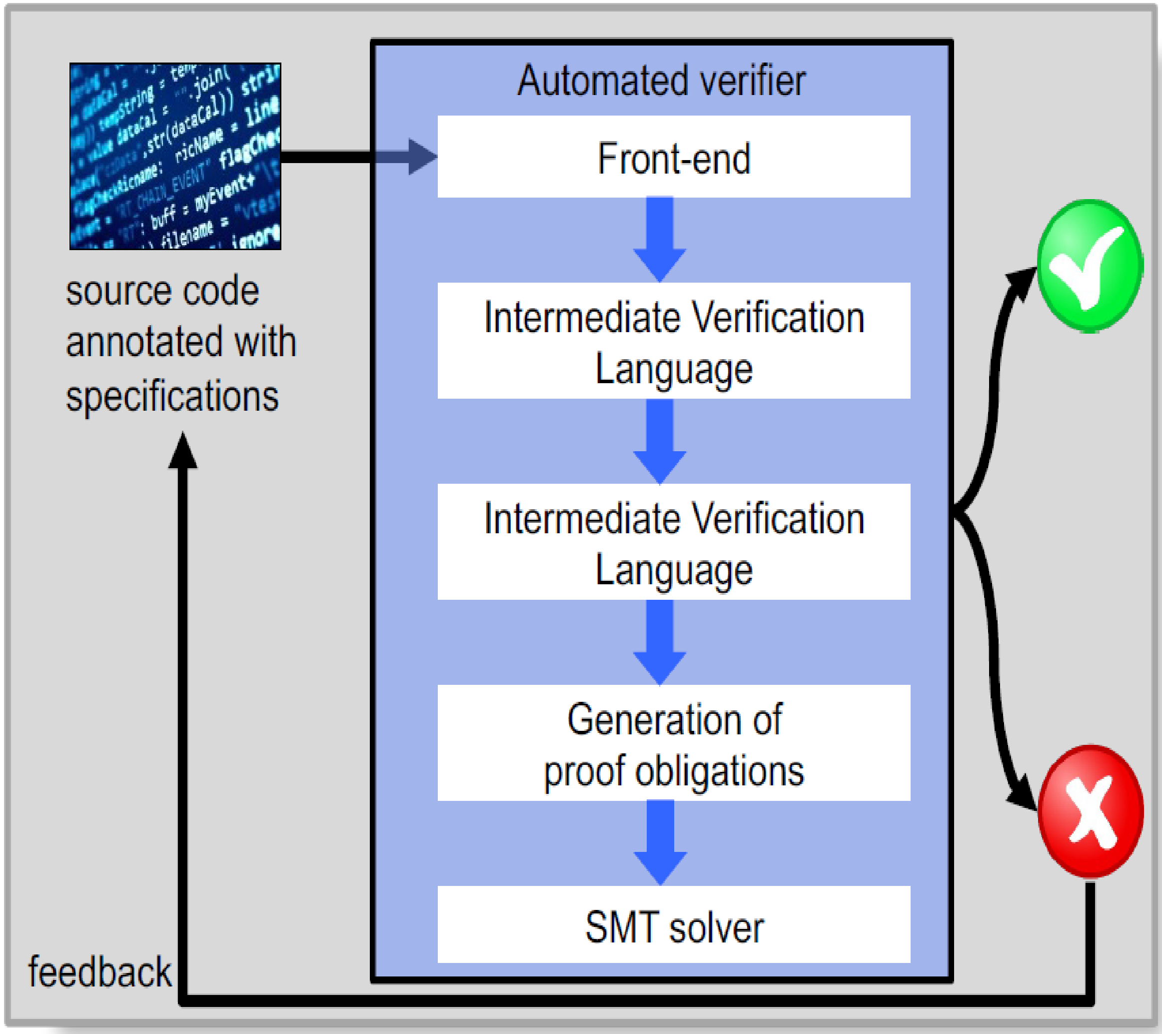
Outline

1. Why Program Verification?
2. Course Overview
3. Starting with Viper

Course objectives

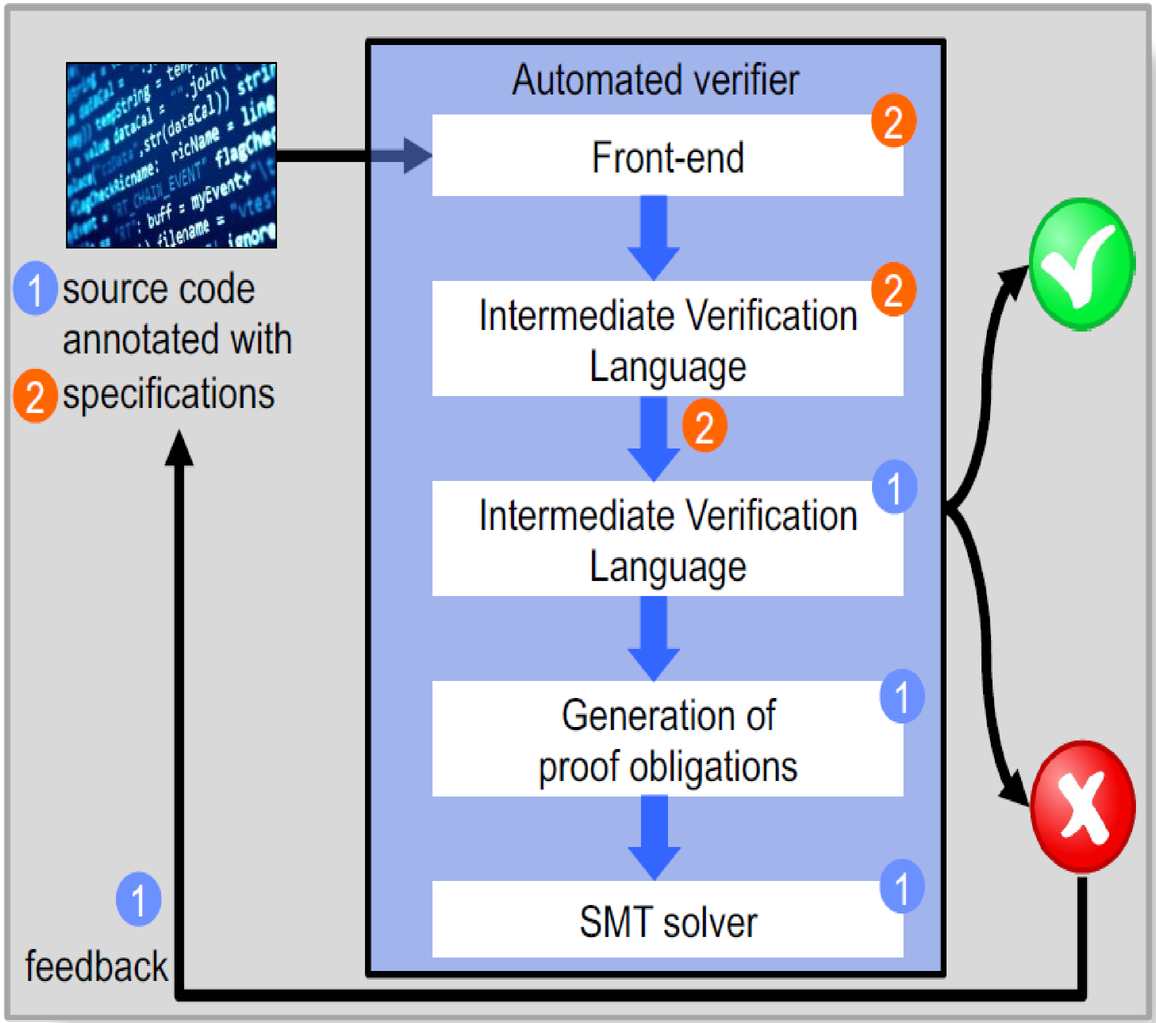


Architecture of automated program verifiers



- Automated verifiers are often implemented as a tool stack
- Stepwise compilation of programs into logical formulas (and back for error reporting)
- Each transformation deals with one verification problem
- Requirements:
  - reasoning principles
  - verification methodologies
  - engineering practices

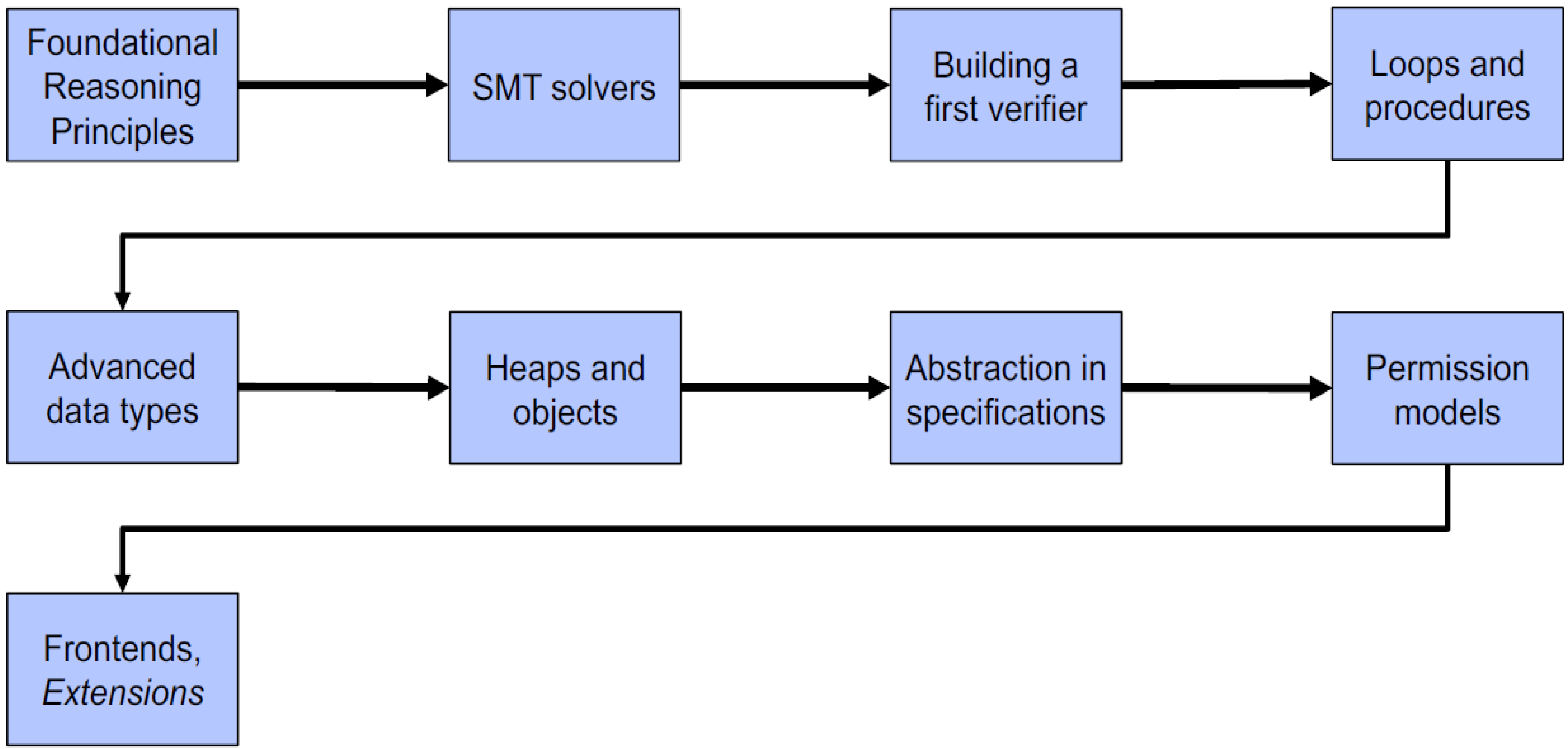
Roadmap



1. We learn how to build and use a verification tool for a small programming language
   - Core reasoning principles
   - Generation of proof obligations
   - Working with SMT solvers
   - Error reporting

2. We extend the language by advanced features
   - Verification challenges
   - Advanced reasoning and specification principles
   - Automation via encoding to lower levels

Tentative course outline



Outline

1. Why Program Verification?
2. Course Overview
3. Starting with Viper

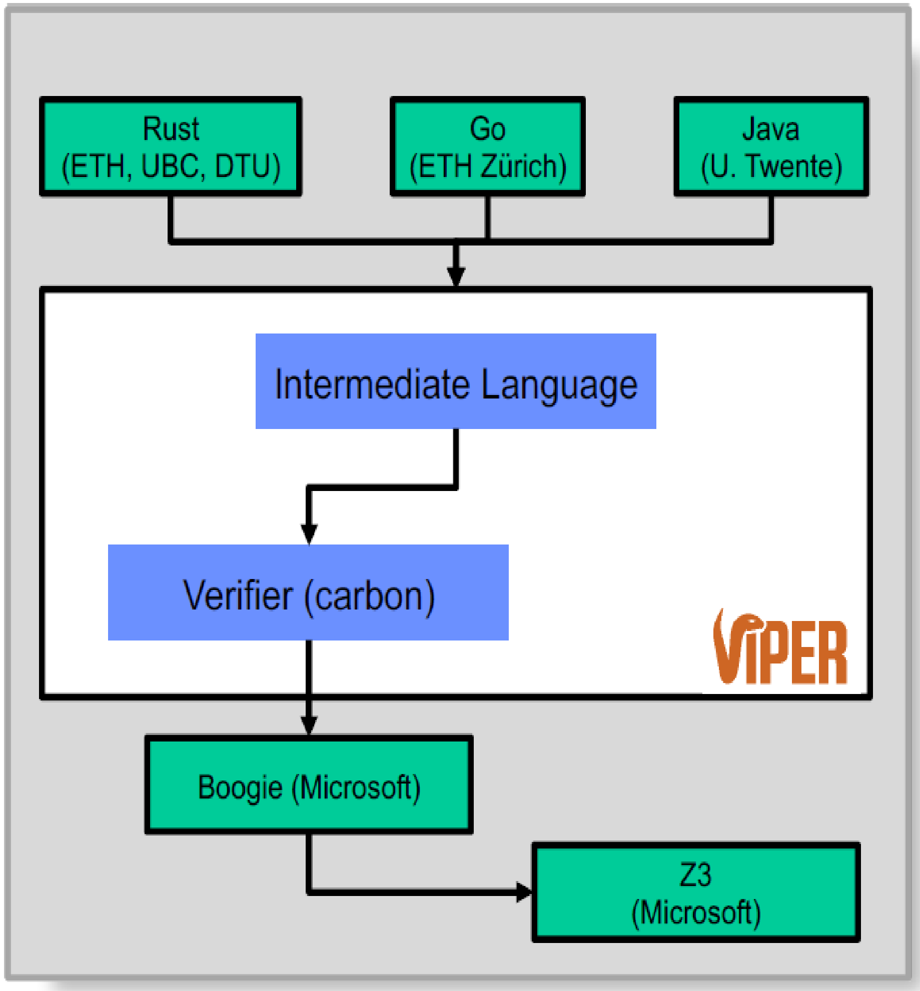
The Viper Verification Framework

- Viper language
  - Models verification problems
  - Some statements are not executable

- Two verification backends
  - Carbon (close to what you will build)
  - Silicon

- **For now:** Programming language
  with a built-in verifier

- **Later:** Automate new methodologies



Viper methods

```
method triple(x: Int) returns (r: Int)
{
   r := 3 * x
}
```

Viper methods

all Viper statements are put in **methods**

read-only **input** parameters

```
method triple(x: Int, flag: Bool)
  returns (r: Int)
{
    if (flag) {
        r := 3 * x
    } else {
        var y: Int
        y := x + x
        r := x + y
    }
}
```

read-write **output** parameters

no explicit return statements

local variable declaration

Assertions

```
method triple(x: Int, flag: Bool)
  returns (r: Int)
{
    if (flag) {
        r := 3 * x
        assert r > 0
    } else {
        var y: Int
        y := x + x
        r := x + y
        assert r == 3 * x
    }
}
```

- `assert` expr tests if expr evaluates to `true`
  - Yes: no effect
  - No: runtime error
- Testing: no assertion error for *chosen* inputs
- Verification: no assertion error for *all* inputs

Which assertions hold?

Assertions

```
method triple(x: Int, flag: Bool)
  returns (r: Int)
{
    if (flag) {
        r := 3 * x
        assert r > 0
    } else {
        var y: Int
        y := x + x
        r := x + y
        assert r == 3 * x
    }
}
```

Postconditions

```
method triple(x: Int) returns (r: Int)
   ensures r == 3 * x
{
  var y: Int
  y := x + x
  r := x + y
}

method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

- Postconditions specify how returned outputs are related to inputs
  - Default: `true`

Postconditions

```
method triple(x: Int) returns (r: Int)
   ensures r == 3 * x
{
  var y: Int
  y := x + x
  r := x + y
}
```

check: r == 3 * x

```
method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

learn: z == 3 * 7

- Postconditions specify how returned outputs are related to inputs
  - Default: `true`
- *Checked* against implementation for all possible parameters
- *Guaranteed* to hold after method calls for supplied parameters

Alternative Implementation

```
method triple(x: Int) returns (r: Int)
   ensures r == 3 * x
{
    r := x / 2
    r := 6 * r
}

method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

x = 7

x = 3

x = 18

- Some implementations do not work for arbitrary inputs
- A **precondition** filters out undesirable inputs

Preconditions

```
method triple(x: Int) returns (r: Int)
   requires x % 2 == 0
   ensures r == 3 * x
{
  r := x / 2
  r := 6 * r
}

method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

- Preconditions specify on what inputs a method can be called
  - Default: `true`

Preconditions

```
method triple(x: Int) returns (r: Int)
   requires x % 2 == 0
   ensures r == 3 * x
{
  r := x / 2
  r := 6 * r
}

method client() {
  var z: Int
  z := triple(7)
  assert z == 21
}
```

r == 3 * x for even x

7 % 2 == 1

- Preconditions specify on what inputs a method can be called
  - Default: `true`
- *Guaranteed* at the beginning of method implementation
- *Checked* before method calls for supplied parameters

Contracts

A method **contract** consist of the method's

- name,
- input and output parameters, and
- pre- and postconditions.

Contracts must be upheld by method calls and implementations.

```
method triple(x: Int) returns (r: Int)
   requires x % 2 == 0
   ensures r == 3 * x
{
  // implementation
  r := x / 2
  r := 6 * r
}
```

```
method client()
{
  triple(7)
  // violates precond.
}
```

Underspecification

```
method triple(x: Int) returns (r: Int)
   requires x > 3
   ensures r > x
{
   r := 3 * x
}
```

- Implementation details are often irrelevant
- Contracts may
  - require more than an implementation needs
  - ensure less than an implementation gives

Give another contract implementation.

Underspecification

```
method triple(x: Int) returns (r: Int)
   requires x > 3
   ensures r > x
{
   r := 3 * x
}
```

```
method triple(x: Int) returns (r: Int)
   requires x > 3
   ensures r > x
{
   r := x + 1
}
```

- Implementation details are often irrelevant
- Contracts may
  - require more than an implementation needs
  - ensure less than an implementation gives

Give another contract implementation.

Verifying Method Calls

```
method triple(x: Int) returns (r: Int)
   requires x > 0
   ensures r > x
{
   r := 3 * x
}

method client() {
  var z: Int
  z := triple(7)
  assert z > 5
  assert z == 21
}
```

What is happening here?

Verifying Method Calls

```
method triple(x: Int) returns (r: Int)
   requires x > 0
   ensures r > x
{
   r := 3 * x
}

method client() {
  var z: Int
  z := triple(7)
  assert z > 5
  assert z == 21
}
```

correct; unclear without looking at implementation

Modular Verification

- Inspect method contracts
- Do *not* inspect method implementations
- *Design decision*

What are pros and cons of using modular verification?

Verifying Method Calls

```
method triple(x: Int) returns (r: Int)
   requires x > 0
   ensures r > x
{
   r := 3 * x
}

method client() {
  var z: Int
  z := triple(7)
  assert z > 5
  assert z == 21
}
```

correct; unclear without looking at implementation

Modular Verification

- Inspect method contracts
- Do *not* inspect method implementations
- *Design decision*

Pros:

- Avoid client re-verification if implementation changes
- Respects the *information hiding* principle (encapsulation)
- Handling of recursion

Cons:

- False negatives (*incompleteness*)
- Need to write more contracts

Abstract Methods

```
method triple(x: Int) returns (r: Int)
   ensures r == 3 * x
```

```
method isqrt(x: Int) returns (r: Int)
    requires x >= 0
    ensures x >= r * r
    ensures x < (r+1) * (r+1)
```

```
method foo(a: Int) returns (b: Int)
    requires a > 0
    ensures b > a
{
    b := isqrt(a)
    b := triple(a)
}
```

- Contracts without Implementations
  - abstract from hard-to-verify code
  - abstract from unknown implementation
- Verification and good software engineering facilitate each other
  - *Incremental development* by refinement
  - Contracts become simpler if every method has a *single responsibility*
  - Avoid premature optimizations

More abstract methods

```
method unsound(x: Int)
  returns (r: Int)
  ensures r != r


method test() {
  var a: Int
  a := unsound(17)
  assert 2 != 2
}
```

More abstract methods

```
method unsound(x: Int)
  returns (r: Int)
  ensures r != r


method test() {
  var a: Int
  a := unsound(17)
  assert 2 != 2
}
```

- **Trusted code base:** code that is not checked by the verifier
- Danger of *unsoundness*: trusted inconsistencies may cause false positives
- Requires separate correctness arguments
- Methods are trusted until implemented

Wrap-up: Informal Overview

- Specification mechanisms
  - Assertions
  - Pre- and postconditions
  - Underspecification

- Using an automated verifier
  - Modular reasoning with contracts
  - Abstract methods
  - Soundness and completeness issues
  - Trusted code base

- Verification and good software engineering facilitate each other
  - Information hiding, single responsibility principle
  - Incremental development