Introduction: Software Testing and Quality Assurance

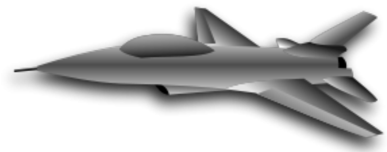
Software Testing, Quality Assurance, and Maintenance

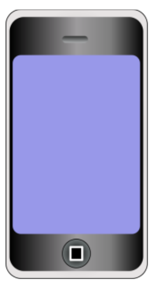
Winter 2020

Prof. Arie Gurfinkel

UNIVERSITY OF WATERLOO

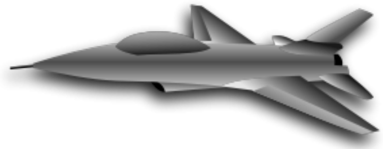
Software is Everywhere






Software is Everywhere



"Software easily rates as the most poorly constructed, unreliable, and least maintainable technological artifacts invented by man"

Paul Strassman, former CIO of Xerox

UNIVERSITY OF WATERLOO

Infamous Software Disasters

Between 1985 and 1987, **Therac-25** gave patients massive overdoses of radiation, approximately 100 times the intended dose. Three patients died as a direct consequence.

On February 25, 1991, during the Gulf War, an American **Patriot Missile** battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.

On June 4, 1996 an unmanned **Ariane 5** rocket launched by the European Space Agency forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million.

http://www5.in.tum.de/~huckle/bugse.html

UNIVERSITY OF WATERLOO

Recent Software Disasters

BUSINESS

FAA Finds New Software Problem in Boeing's 737 MAX

Plane maker agrees to address the problem and believes it can be fixed with a software tweak

By Andrew Tangel and Andy Pasztor

Updated June 26, 2019 9:55 pm ET

PRINT TEXT

Boeing Co. and federal regulators said they have identified a new software problem on the 737 MAX, further delaying the process of returning the troubled jet to service.

Opinion
Technology

The millennium bug was real - and 20 years later we face the same threats

Martyn Thomas

Tue 31 Dec 2019 09.00 GMT

The Y2K problem is now seen as a bit of a joke - but only a fool would be complacent about the vulnerability of IT systems

https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html#slide1

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/

UNIVERSITY OF WATERLOO

Why so many bugs?

Software Engineering is very complex

- Complicated algorithms
- Many interconnected components
- Legacy systems
- Huge programming APIs
- …

Software Engineers need better tools to deal with this complexity!

What Software Engineers Need Are …

Tools that give better confidence than *ad-hoc* testing while remaining easy to use

And at the same time, are

- … fully automatic
- … (reasonably) easy to use
- … provide (measurable) guarantees
- … come with guidelines and methodologies to apply effectively
- … apply to real software systems

UNIVERSITY OF WATERLOO

Testing

Software validation the “old-fashioned” way:

- Create a test suite (set of test cases)
- Run the test suite
- Fix the software if test suite fails
- Ship the software if test suite passes

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

Edsger W. Dijkstra

Very hard to test the portion inside the "if" statement!

```
x = read();
if (hash(x) == 10) {
    ...
}
```

Hypothetical program

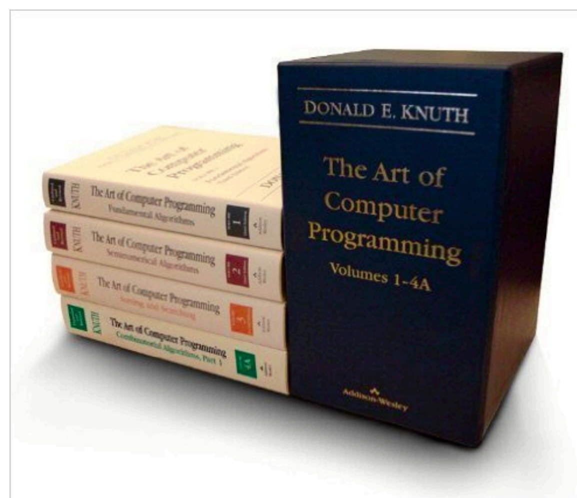
UNIVERSITY OF WATERLOO

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald Knuth

You can only verify what you have specified.

Testing is still important, but can we make it less impromptu?



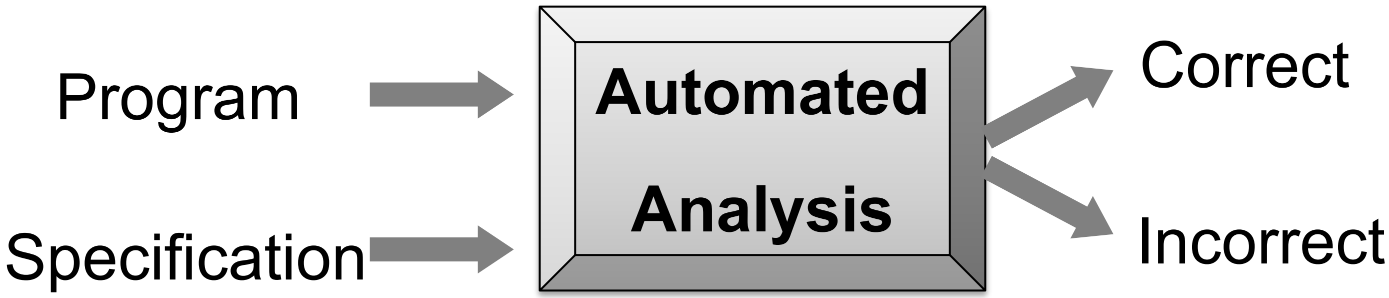
Verification / Quality Assurance

Verification: formally prove that a computing system satisfies its specifications

- **Rigor**: well established mathematical foundations
- **Exhaustiveness**: considers all possible behaviors of the system, i.e., finds all errors
- **Automation**: uses computers to build reliable computers

Formal Methods: general area of research related to program specification and verification

Ultimate Goal: Static Program Verification



Reasoning statically about behavior of a program without executing it

- compile-time analysis
- exhaustive, considers all possible executions under all possible environments and inputs

The *algorithmic* discovery of *properties* of program by *inspection* of the *source text*

Manna and Pnueli

Also known as *static analysis*, *program verification*, *formal methods*, etc.

UNIVERSITY OF WATERLOO

Turing, 1936: “undecidable”

Undecidability

A problem is undecidable if there does not exists a Turing machine that can solve it

- i.e., not solvable by a computer program

The halting problem

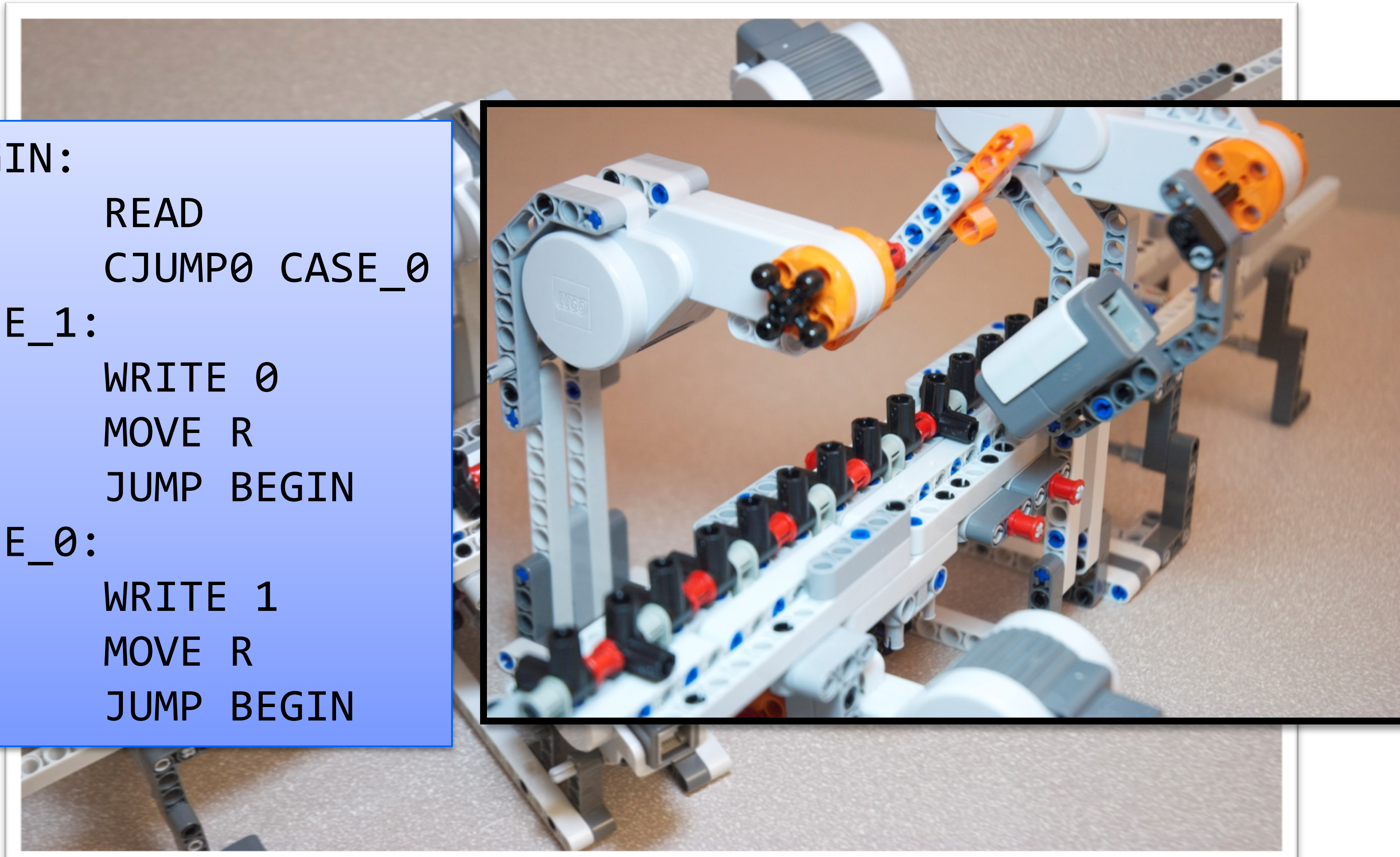
- does a program P terminates on input I
- proved undecidable by Alan Turing in 1936
- https://en.wikipedia.org/wiki/Halting_problem

Rice's Theorem

- for any non-trivial property of partial functions, no general and effective method can decide whether an algorithm computes a partial function with that property
- in practice, this means that there is no machine that can always decide whether the language of a given Turing machine has a particular nontrivial property
- https://en.wikipedia.org/wiki/Rice%27s_theorem

LEGO Turing Machine

```
BEGIN:
        READ
        CJUMP0 CASE_0
CASE_1:
        WRITE 0
        MOVE R
        JUMP BEGIN
CASE_0:
        WRITE 1
        MOVE R
        JUMP BEGIN
```



by Soonho Kong. See http://www.cs.cmu.edu/~soonhok for building instructions.

Living with Undecidability

"Algorithms" that occasionally diverge

Limit programs that can be analyzed

- finite-state, loop-free

Partial (unsound) verification

Testing **Sym Exec**

- analyze only some executions up-to a fixed number of steps

Incomplete verification / Abstraction

- analyze a superset of program executions

Automated Verification

Programmer Assistance

- annotations, pre-, post-conditions, inductive invariants

Deductive Verification

Formal Software Analysis

J. McCarthy, "*A basis for mathematical theory of computation*", 1963.

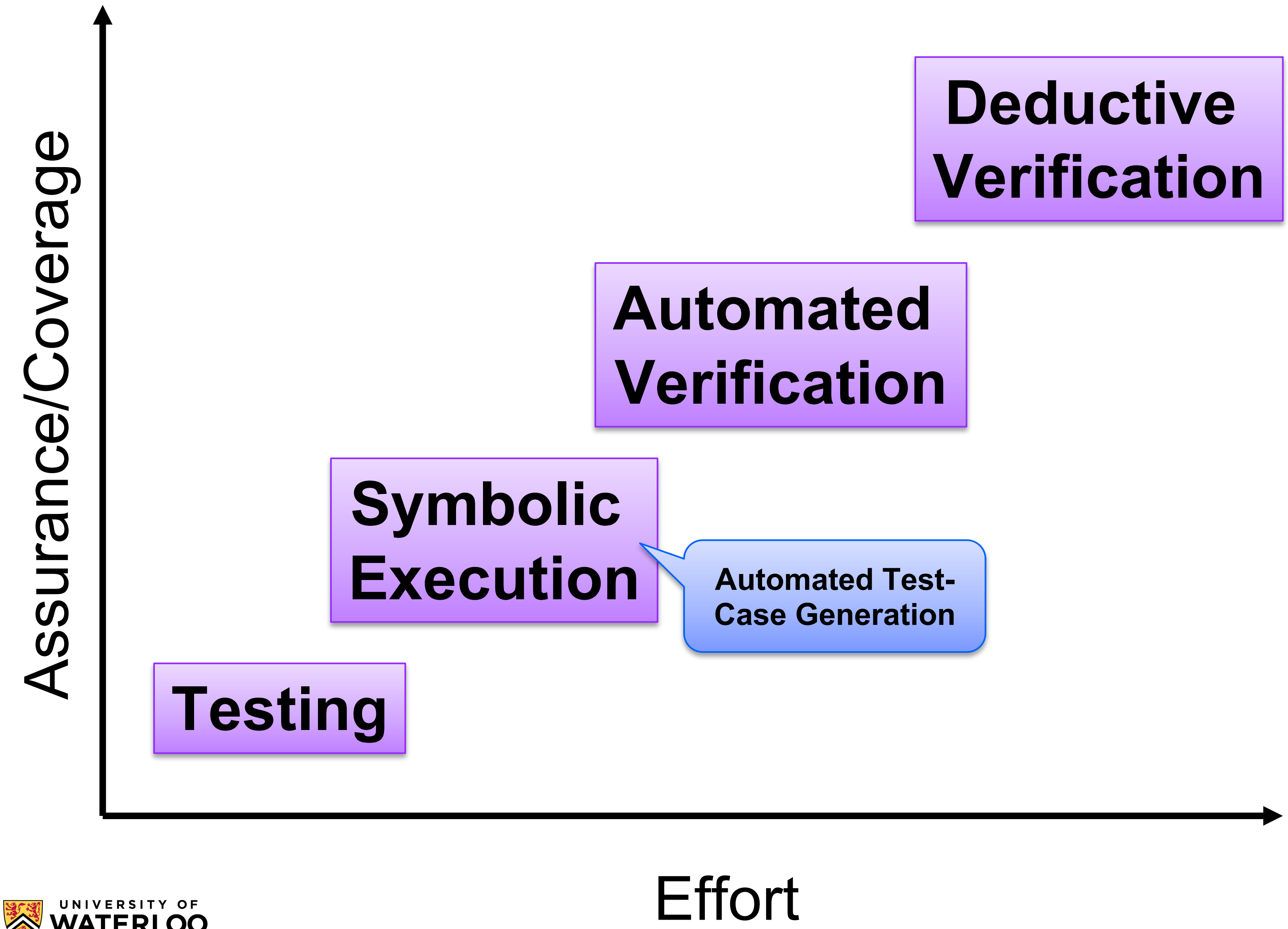
P. Naur, "*Proof of algorithms by general snapshots*", 1966.

R. W. Floyd, "*Assigning meaning to programs*", 1967.

C.A.R Hoare, "*An axiomatic basis for computer programming*", 1969.

E. W. Dijkstra: "*Guarded Commands, Nondeterminacy and Formal derivation* ", 1975.

(User) Effort vs (Verification) Assurance



Why are Testing and Verification Necessary

Why Test?

Why Verify?

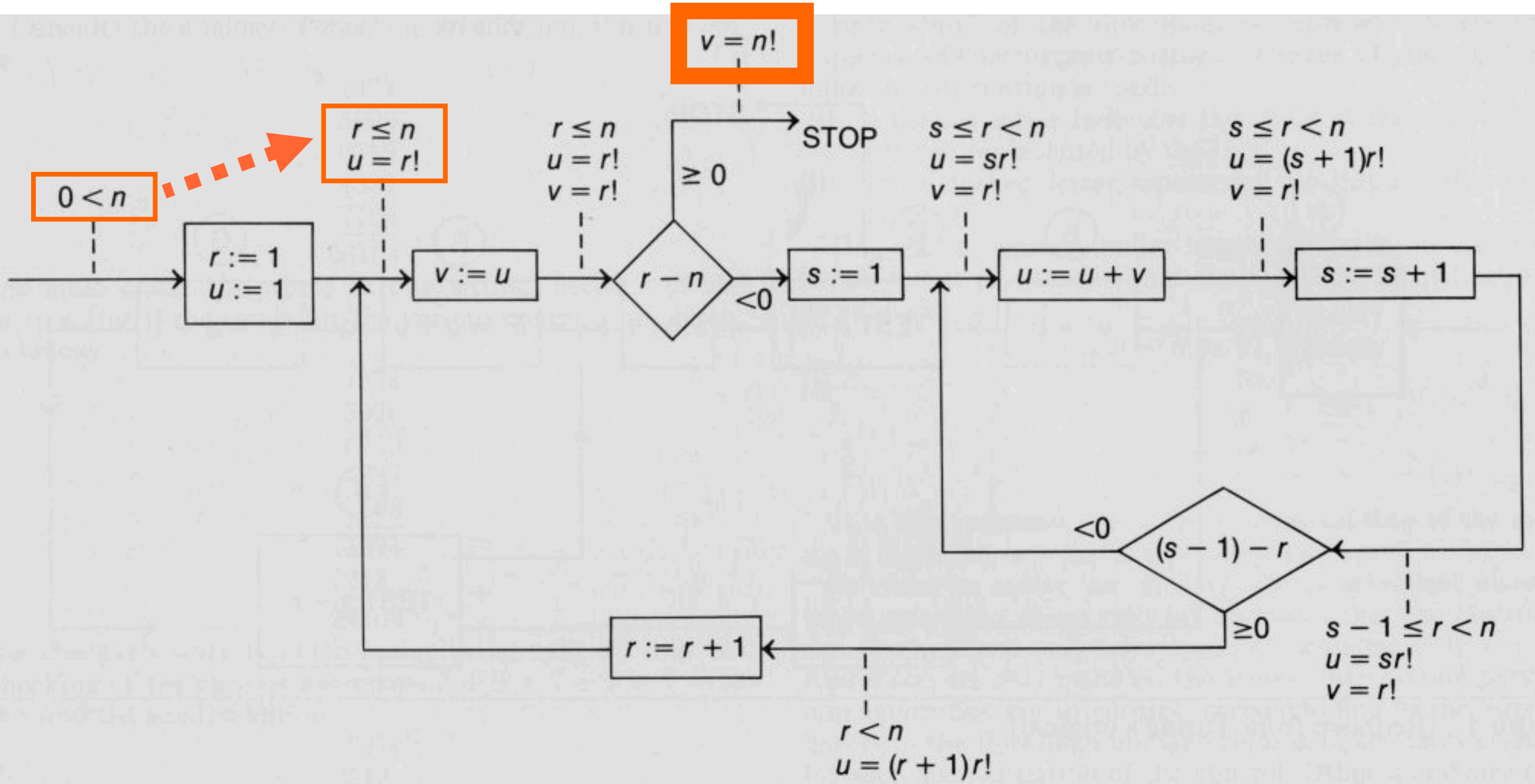
What is Verification? How is it different from Testing?

Turing, 1949

Alan M. Turing. "Checking a large routine", 1949

How can one check a routine in the sense of making sure that it is right?

programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.



```
method factorial_turing (n: int) returns (v: int)
{
  var r := 1;
  var u := 1;

  while (true)
  {
    v := u;
    if (r - n ≥ 0)
    { return v; }
    var s := 1;
    while (true)
    {
      u := u + v;
      s := s + 1;
      if ((s - (r + 1)) ≥ 0)
      {break;}
    }
    r := r + 1;
  }
}
```

```
method factorial (n: int) returns (v:int)


{
  v := 1;
  if (n == 1) { return v; }
  var i := 2;
  while (i <= n)


  {
    v := i * v;
    i := i + 1;
  }
  return v;
}
```

```
method factorial (n: int) returns (v:int)
  requires n >= 0;
  ensures v = fact(n);
{
  v := 1;
  if (n <= 1) { return v; }
  var i := 2;
  while (i <= n)
    invariant i <= n + 1
    invariant v = fact(i - 1)
  {
    v := i * v;
    i := i + 1;
  }
  return v;
}
```

Specification

Inductive Invariant

Proving inductive invariants

The main step is to show that the invariant is preserved by one execution of the loop

```
assume(i <= n + 1);
assume(v == fact(i - 1));
assume(i <= n);
v := i * v;
i := i + 1;
assert(i <= n + 1);
assert(v == fact(i - 1));
```

Correctness of a loop-free program can (often) be decided by a Theorem Prover or a Satisfiability Modulo Theory (SMT) solver.

Proving inductive invariants

The main step is to show that the invariant is preserved by one execution of the loop

```
(i0 <= n0+1)      &&
(v0 == (i0-1)!)   &&
(i0 <= n0)        &&
(v1 = i0 * v0)    &&
(i1 = i0 + 1)
➔
((i1 <= n0+1)     &&
(v1 == (i1-1)!))
```

```
assume(i <= n+1);
assume(v == fact(i-1));
assume(i <= n);
v := i*v;
i := i+1;

assert(i<=n+1);
assert(v == fact(i-1));
```

Correctness of a loop-free program can (often) be decided by a Theorem Prover or a Satisfiability Modulo Theory (SMT) solver.

Available Tools

Testing

- many tools actively used in industry. We will use Python **unittest**

Symbolic Execution / Automated Test-Case Generation

- mostly academic tools with emerging industrial applications
- KLEE, S2E, jDART, Pex (now Microsoft IntelliTest)

Automated Verification

- built into compilers, may lightweight static analyzers
  - clang analyzer, Facebook Infer, Coverity, …
- academic pushing the coverage/automation boundary
  - SeaHorn (my tool), JayHorn, CPAChecker, SMACK, T2, …

(Automated) Deductive Verification

- academic, still rather hard to use, we'll experience in class ☺
- Dafny/Boogie (Microsoft), Viper, Why3, KeY, ...

Key Challenges

Testing

- Coverage

Symbolic Execution and Automated Verification

- Scalability

Deductive Verification

- Usability

Common Challenge

- Specification / Oracle

Calendar Description

Software Testing, Quality Assurance and Maintenance

Introduces students to systematic testing of software systems. Software verification, reviews, metrics, quality assurance, and prediction of software reliability and availability. Related management issues.

Topics Covered in the Course

Foundations

- syntax, semantics, abstract syntax trees, visitors, control flow graphs

Testing

- coverage: structural, dataflow, and logic

Symbolic Execution / Automated Test-Case Generation

- using SMT solvers, constraints, path conditions, exploration strategies
- building a (toy) symbolic execution engine

Deductive Verification

- Hoare Logic, weakest pre-condition calculus, verification condition generation
- verifying algorithm using Dafny, building a small verification engine

Automated Verification

- (basics of) software model checking

Frequently Asked Questions

Is this course practical?

Is this course easy / hard?

What knowledge from the course is applicable to a developer?

Is it a compilers course?

Is it a logic course?

Do I have to attend the lectures?

What are most useful skills learned in the course?

- Foundations of testing and verification
- State-of-the-art tools and technique to automate testing and reasoning
- Understanding the difference between wishful thinking (I hope it works) and a strong argument (I know it works, here is why…)

A little about me

2007, PhD University of Toronto

2006-2016, Principle Researcher at Software Engineering Institute, Carnegie Mellon University

Sep 2016, Associate Professor, University of Waterloo

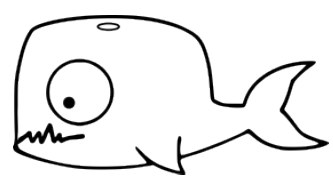
వింత

UFO

FrankenBit

SPACER

Avy



SeaHorn

Interests and Tools

Interests

- Software Model Checking, Program Verification, Decision Procedures, Abstract Interpretation, SMT, Horn Clauses, …

Active Tools

- SeaHorn – Algorithmic Logic-Based Verification framework for C
- AVY – Hardware Model Checker with Interpolating PDR
- SPACER – Horn Clause Solver based on Z3 GPDR
- for more, see http://arieg.bitbucket.org/tools.html

Current Work

- parametric symbolic reachability – verifying safety properties of parametric systems
- automated verification of C
- …