More Test Input Generation

Martin Kellogg

Reading Quiz: EvoSuite

Reading Quiz: EvoSuite

Q1: Which of the following does EvoSuite have in common with AFL? (select all that apply)

A. its core algorithm is a genetic algorithm
B. it derives oracles using mutation testing
C. it can operate directly on the bytecode of a project, without the need to recompile the code

Q2: **TRUE** or **FALSE**: EvoSuite treats the String class in Java specially

Reading Quiz: EvoSuite

Q1: Which of the following does EvoSuite have in common with AFL? (select all that apply)

A. its core algorithm is a genetic algorithm
B. it derives oracles using mutation testing
C. it can operate directly on the bytecode of a project, without the need to recompile the code

Q2: **TRUE** or **FALSE**: EvoSuite treats the String class in Java specially

Reading Quiz: EvoSuite

Q1: Which of the following does EvoSuite have in common with AFL? (select all that apply)

A. its core algorithm is a genetic algorithm
B. it derives oracles using mutation testing
C. it can operate directly on the bytecode of a project, without the need to recompile the code

Q2: **TRUE** or **FALSE**: EvoSuite treats the String class in Java specially

More test input generation: agenda

- Other approaches that use random testing
  - “feedback-directed” random testing
  - brief introduction to mutation testing
  - EvoSuite: mutation testing + a genetic algorithm
- Lens of Logic: symbolic execution for test input generation
  - concolic testing

More test input generation: agenda

- Other approaches that use random testing
  - **“feedback-directed” random testing**
  - brief introduction to mutation testing
  - EvoSuite: mutation testing + a genetic algorithm
- Lens of Logic: symbolic execution for test input generation
  - concolic testing

Feedback-directed Random Testing

- fuzzing isn’t the only way to randomly generate tests

Feedback-directed Random Testing

- fuzzing isn't the only way to randomly generate tests
- Randoop (Pacheco et al. 2007) is a good example of an alternative approach: ***feedback-directed*** random testing

Feedback-directed Random Testing

- fuzzing isn't the only way to randomly generate tests
- Randoop (Pacheco et al. 2007) is a good example of an alternative approach: ***feedback-directed*** random testing
  - does **not** use a genetic algorithm

Feedback-directed Random Testing

- fuzzing isn't the only way to randomly generate tests
- Randoop (Pacheco et al. 2007) is a good example of an alternative approach: ***feedback-directed*** random testing
  - does **not** use a genetic algorithm
  - but the core idea is similar: build test inputs **incrementally**
    - that is, new inputs extend existing inputs

Feedback-directed Random Testing

- fuzzing isn't the only way to randomly generate tests
- Randoop (Pacheco et al. 2007) is a good example of an alternative approach: ***feedback-directed*** random testing
  - does **not** use a genetic algorithm
  - but the core idea is similar: build test inputs **incrementally**
    - that is, new inputs extend existing inputs
  - execute each new input **immediately** (but there is no explicit fitness function: it is not designed as a genetic algorithm)

Feedback-directed Random Testing

- fuzzing isn't the only way to randomly generate tests
- Randoop (Pacheco et al. 2007) is a good example of an alternative approach: ***feedback-directed*** random testing
    - does **not** use a genetic algorithm
    - but the core idea is similar: build test inputs **incrementally**
        - that is, new inputs extend existing inputs
    - execute each new input **immediately** (but there is no explicit fitness function: it is not designed as a genetic algorithm)
        - tests are discarded if they do not discover **new states**

Randoop: input/output example

- inputs:

- output:

Randoop: input/output example

- inputs:
  - classes under test (this tool targets Java/OOP)

- output:

Randoop: input/output example

- inputs:
  - classes under test (this tool targets Java/OOP)
  - time limit
- output:

Randoop: input/output example

- inputs:
    - classes under test (this tool targets Java/OOP)
    - time limit
    - a set of contracts to use as oracles
        - e.g., “o.equals(o) == true” or “o.hashCode() never throws”
- output:

Randoop: input/output example

- inputs:
    - classes under test (this tool targets Java/OOP)
    - time limit
    - a set of contracts to use as oracles
        - e.g., “o.equals(o) == true” or “o.hashCode() never throws”
- output:
    - sequences of method calls that cause a contract violation

Randoop: input/output example

- inputs:
  - classes under test (t
  - time limit
  - a set of contracts to
    - e.g., “o.equals(o)
- output:
  - sequences of metho

For example:

```
Map h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
List l = new LinkedList();
l.addFirst(a);
Set t = new TreeSet(l);
Set u =
Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

Randoop: input/output example

- inputs:
  - classes under test (t
  - time limit
  - a set of contracts to
    - e.g., “o.equals(o)
- output:
  - sequences of metho

For example:

```
Map h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
List l = new LinkedList();
l.addFirst(a);
Set t = new TreeSet(l);
Set u =
Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

fails when executed

Randoop: type-directed synthesis

- how does it work?

Randoop: type-directed synthesis

- how does it work?
    - start with a set of seed sequences of size 1 (e.g., `int i = 0;`)

Randoop: type-directed synthesis

- how does it work?
  - start with a set of seed sequences of size 1 (e.g., `int i = 0;`)
  - randomly select a method call $\texttt{m(T}_1\texttt{,...,T}_k\texttt{)/T}_{ret}$ s.t. there is a sequence in the seed pool that ends in all $\texttt{T}_i$ for `1 <= i <= k`

Randoop: type-directed synthesis

- how does it work?
  - start with a set of seed sequences of size 1 (e.g., `int i = 0;`)
  - randomly select a method call $\mathtt{m(T_1, ..., T_k)/T_{ret}}$ s.t. there is a sequence in the seed pool that ends in all $\mathtt{T_i}$ for `1 <= i <= k`
  - for each $\mathtt{T_i}$, choose a sequence $\mathtt{S_i}$ that constructs an object $\mathtt{v_i}$ of type $\mathtt{T_i}$ from the pool

Randoop: type-directed synthesis

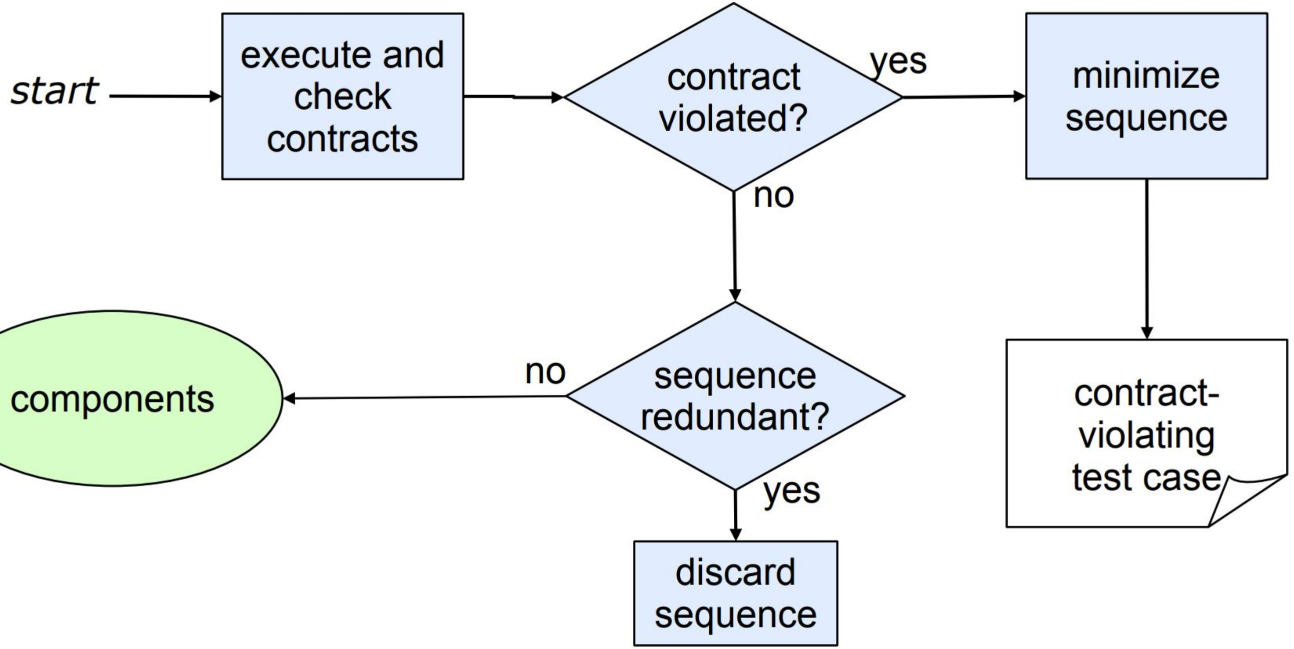
- how does it work?
  - start with a set of seed sequences of size 1 (e.g., `int i = 0;`)
  - randomly select a method call $\texttt{m(T}_1\texttt{,...,T}_k\texttt{)/T}_{ret}$ s.t. there is a sequence in the seed pool that ends in all $\texttt{T}_i$ for `1 <= i <= k`
  - for each $\texttt{T}_i$, choose a sequence $\texttt{S}_i$ that constructs an object $\texttt{v}_i$ of type $\texttt{T}_i$ from the pool
  - create a new sequence:
    - $\texttt{S}_{new} = \texttt{S}_1 \texttt{;} \ldots \texttt{;} \texttt{S}_k \texttt{;} \texttt{T}_{ret}\ \texttt{v}_{new} = \texttt{m(v}_1\texttt{, }\ldots\texttt{, v}_k\texttt{);}$

Randoop: type-directed synthesis

- how does it work?
  - start with a set of seed sequences of size 1 (e.g., `int i = 0;`)
  - randomly select a method call $\texttt{m(T}_1\texttt{,...,T}_k\texttt{)/T}_{ret}$ s.t. there is a sequence in the seed pool that ends in all $\texttt{T}_i$ for `1 <= i <= k`
  - for each $\texttt{T}_i$, choose a sequence $\texttt{S}_i$ that constructs an object $\texttt{v}_i$ of type $\texttt{T}_i$ from the pool
  - create a new sequence:
    - $\texttt{S}_{new}\texttt{ = S}_1\texttt{ ; … ; S}_k\texttt{ ; T}_{ret}\texttt{ v}_{new}\texttt{ = m(v}_1\texttt{, …, v}_k\texttt{);}$
  - classify the new sequence by executing it: may discard, output as a test case, or add it to the pool of sequences

Randoop: classifying sequences

Randoop: classifying sequences



Randoop: redundant sequences

- Randoop discards ***redundant*** sequences

Randoop: redundant sequences

- Randoop discards ***redundant*** sequences
  - during generation, it maintains a set *O* of all objects that it has ever created

Randoop: redundant sequences

- Randoop discards ***redundant*** sequences
  - during generation, it maintains a set O of all objects that it has ever created
  - a sequence is considered redundant if all of the objects created during its execution are members of O (using **`.equals`**)

Randoop: redundant sequences

- Randoop discards ***redundant*** sequences
  - during generation, it maintains a set *O* of all objects that it has ever created
  - a sequence is considered redundant if all of the objects created during its execution are members of *O* (using **`.equals`**)
  - Randoop would work with other reasonable definitions of redundant, too
    - e.g., heap canonicalization

Randoop: in practice

- Randoop has been used to **find real bugs** in e.g., the JDK

Randoop: in practice

- Randoop has been used to **find real bugs** in e.g., the JDK
- It has been **deployed at companies** (e.g., Microsoft)

Randoop: in practice

- Randoop has been used to **find real bugs** in e.g., the JDK
- It has been **deployed at companies** (e.g., Microsoft)
- The tool is **still maintained** (so you could use it yourself)
    - https://randoop.github.io/randoop/

Randoop: in practice

- Randoop has been used to **find real bugs** in e.g., the JDK
- It has been **deployed at companies** (e.g., Microsoft)
- The tool is **still maintained** (so you could use it yourself)
    - https://randoop.github.io/randoop/
- It is commonly used in research papers as a ***baseline***: that is, a method that any new technique is expected to outperform
    - Randoop is fast and easy enough to use that if a new technique cannot outperform it, it's probably not worth using!

More test input generation: agenda

- Other approaches that use random testing
  - “feedback-directed” random testing
  - **brief introduction to mutation testing**
  - EvoSuite: mutation testing + a genetic algorithm
- Lens of Logic: symbolic execution for test input generation
  - concolic testing

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
    - **Intuition**: test whether they can actually find truffles!

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
  - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
    - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds
    - The pig that finds more of the hidden truffles in your backyard is assumed to find more real truffles in the wild

The Lens of Adversity: finding bugs

- Suppose you wanted to evaluate the quality of two truffle-sniffing pigs
  - **Intuition**: test whether they can actually find truffles!
- **Test idea**: hide some truffles in your backyard and see how many each pig finds
  - The pig that finds more of the hidden truffles in your backyard is assumed to find more real truffles in the wild
- Suppose you wanted to evaluate the quality of two bug-finding test suites …

The Lens of Adversity: mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

The Lens of Adversity: mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

- Informally: “You claim your test suite is really great at finding security bugs? Well, I'll just **intentionally add a bug** to my source code and see if your test suite finds it!”

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.
- For mutation testing, defect seeding is typically done automatically (given a model of what human bugs look like)

Mutation testing: mutation operators

Definition: A ***mutation operator*** systematically changes a program point. In mutation testing, the mutation operators are usually modeled on historical human defects.

Mutation testing: mutation operators

Definition: A ***mutation operator*** systematically changes a program point. In mutation testing, the mutation operators are usually modeled on historical human defects.

- Example mutations:
    - `if (a < b)` → `if (a <= b)`
    - `if (a == b)` → `if (a != b)`
    - `a = b + c` → `a = b - c`
    - `f(); g();` → `g(); f();`
    - `x = y` → `x = z`

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Definition: The ***order*** of a mutant is the number of mutation operators applied.

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Definition: The ***order*** of a mutant is the number of mutation operators applied.

```
// original                        // 2nd-order mutant
if (a < b):                    if (a <= b):
x = a + b           →              x = a - b
print(x)                       print(x)
```

•

Mutation testing: killing mutants

Definition: A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.

•

Mutation testing: killing mutants

Definition: A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.

- test suites that kill more mutants are generally considered better

•

Mutation testing: killing mutants

Definition: A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.

- test suites that kill more mutants are generally considered better
- (sorry for all the vocabulary, but it's necessary to understand how EvoSuite works)

Mutation testing: more to come!

- My intention today is to give you a high-level idea of how mutation testing works
  - because EvoSuite (which you’ll use for HW4) relies on it

Mutation testing: more to come!

- My intention today is to give you a high-level idea of how mutation testing works
  - because EvoSuite (which you'll use for HW4) relies on it
- We will discuss mutation testing in much more detail in two weeks
  - and you'll get a chance to try your hand at it in HW6

More test input generation: agenda

- Other approaches that use random testing
  - “feedback-directed” random testing
  - brief introduction to mutation testing
  - **EvoSuite: mutation testing + a genetic algorithm**
- Lens of Logic: symbolic execution for test input generation
  - concolic testing

EvoSuite: core idea

- much like AFL or other fuzzers, EvoSuite uses a **genetic algorithm** to evolve better tests

EvoSuite: core idea

- much like AFL or other fuzzers, EvoSuite uses a **genetic algorithm** to evolve better tests
  - however, EvoSuite views the **test suite** as the individual

EvoSuite: core idea

- much like AFL or other fuzzers, EvoSuite uses a **genetic algorithm** to evolve better tests
  - however, EvoSuite views the **test suite** as the individual
    - individual tests are themselves “chromosomes”

EvoSuite: core idea

- much like AFL or other fuzzers, EvoSuite uses a **genetic algorithm** to evolve better tests
  - however, EvoSuite views the **test suite** as the individual
    - individual tests are themselves “chromosomes”
    - the whole population is made up of many test suites

EvoSuite: core idea

- much like AFL or other fuzzers, EvoSuite uses a **genetic algorithm** to evolve better tests
  - however, EvoSuite views the **test suite** as the individual
    - individual tests are themselves "chromosomes"
    - the whole population is made up of many test suites
  - this simplifies crossover/parentage: just add/remove tests

EvoSuite: core idea

- much like AFL or other fuzzers, EvoSuite uses a **genetic algorithm** to evolve better tests
  - however, EvoSuite views the **test suite** as the individual
    - individual tests are themselves “chromosomes”
    - the whole population is made up of many test suites
  - this simplifies crossover/parentage: just add/remove tests
- EvoSuite also uses mutation testing to produce **oracles**

EvoSuite: core idea

- much like AFL or other fuzzers, EvoSuite uses a **genetic algorithm** to evolve better tests
  - however, EvoSuite views the **test suite** as the individual
    - individual tests are themselves "chromosomes"
    - the whole population is made up of many test suites
  - this simplifies crossover/parentage: just add/remove tests
- EvoSuite also uses mutation testing to produce **oracles**
  - **key idea**: assertions that **kill mutants** make good oracles
    - we'll come back to this idea next week

EvoSuite vs. AFL

EvoSuite vs. AFL

- EvoSuite emphasizes producing **human-readable** tests

EvoSuite vs. AFL

- EvoSuite emphasizes producing **human-readable** tests
- EvoSuite's developers actually expect you to look at the tests that it produces, and to use them for **regression testing**

EvoSuite vs. AFL

- EvoSuite emphasizes producing **human-readable** tests
- EvoSuite's developers actually expect you to look at the tests that it produces, and to use them for **regression testing**
- by contrast, AFL is looking to **find bugs**
  - leads to test inputs that aren't easy to understand!

EvoSuite: HW4 thoughts

- HW4 asks you to use EvoSuite to generate test suites for a Java library
- As you do, consider how EvoSuite:

EvoSuite: HW4 thoughts

- HW4 asks you to use EvoSuite to generate test suites for a Java library
- As you do, consider how EvoSuite:
  - differs from AFL as deployed in HW3

EvoSuite: HW4 thoughts

- HW4 asks you to use EvoSuite to generate test suites for a Java library
- As you do, consider how EvoSuite:
  - differs from AFL as deployed in HW3
  - compares to the sort of tests that you might write by hand

EvoSuite: HW4 thoughts

- HW4 asks you to use EvoSuite to generate test suites for a Java library
- As you do, consider how EvoSuite:
  - differs from AFL as deployed in HW3
  - compares to the sort of tests that you might write by hand
  - does it achieve its goal of creating useful regression suites?

More test input generation: agenda

- Other approaches that use random testing
  - “feedback-directed” random testing
  - brief introduction to mutation testing
  - EvoSuite: mutation testing + a genetic algorithm
- **Lens of Logic: symbolic execution for test input generation**
  - concolic testing

Symbolic Execution

- we've seen coverage used as a fitness function for a fuzzer

Symbolic Execution

- we've seen coverage used as a fitness function for a fuzzer
  - but what if we just try to figure out **which inputs would improve coverage** directly?

Symbolic Execution

- we've seen coverage used as a fitness function for a fuzzer
  - but what if we just try to figure out **which inputs would improve coverage** directly?
- this is the key idea behind using **symbolic execution** to generate test inputs that improve coverage

Symbolic Execution

Definition: ***symbolic execution*** abstractly runs the target program while computing a **formula** for each variable

Symbolic Execution

Definition: ***symbolic execution*** abstractly runs the target program while computing a **formula** for each variable

- effectively, use math to figure out which values of each variable will cause the program to take particular paths

Symbolic Execution

Definition: ***symbolic execution*** abstractly runs the target program while computing a **formula** for each variable

- effectively, use math to figure out which values of each variable will cause the program to take particular paths
- our plan: choose an uncovered bit of code, and then symbolically execute **backwards** from there to figure out what values the input variables would need to take on in order to cover the code

Symbolic Execution

Definition: ***symbolic execution*** abstractly runs the target program while computing a **formula** for each variable

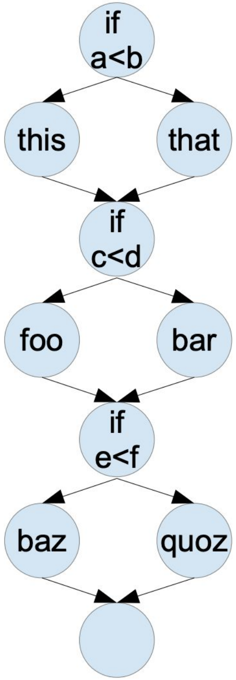
- effectively, use math to figure out which values of each variable will cause the program to take particular paths
- our plan: choose an uncovered bit of code, and then symbolically execute **backwards** from there to figure out what values the input variables would need to take on in order to cover the code
    - this is the **Lens of Logic** again, but applied in a different way

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
   if a < b: this
   else: that
   if c < d: foo
   else: bar
   if e < f: baz
   else: quoz
```

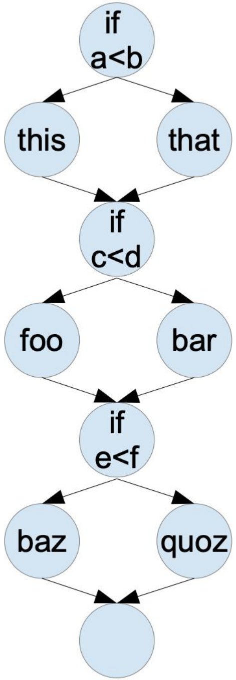
Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```



Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

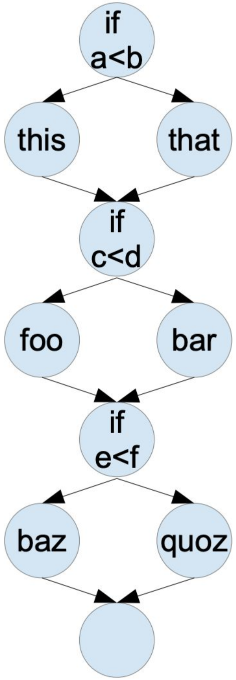


How would you choose inputs that **maximize**:

- **line** coverage?

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

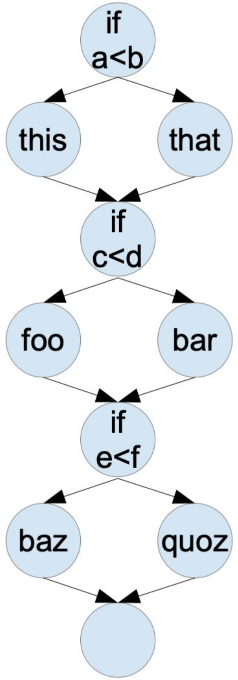


How would you choose inputs that **maximize**:

- **line** coverage?
- **branch** coverage?

Lens of Logic: maximize coverage

```
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

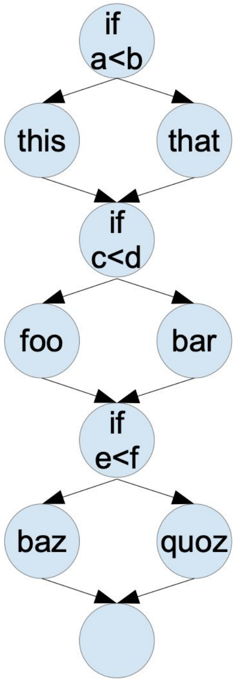


How would you choose inputs that **maximize**:

- **line** coverage?
- **branch** coverage?
- **path** coverage?

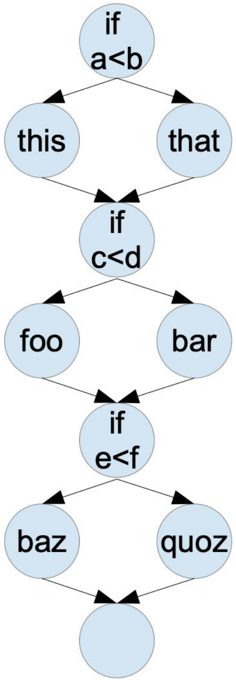
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …



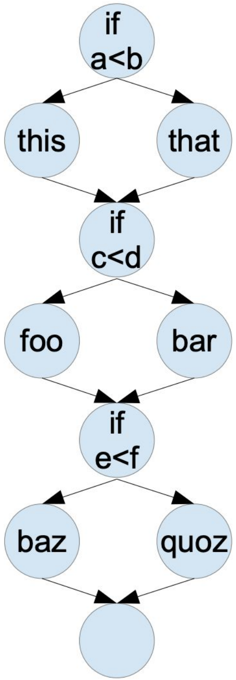
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges



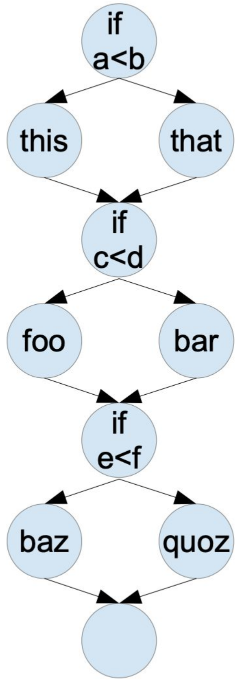
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements ...
- There are **2N** branch edges
    - Which you could cover in 2 tests!



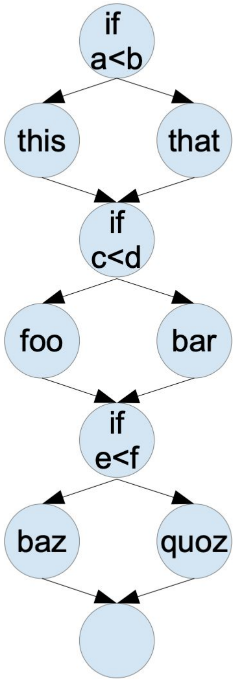
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right



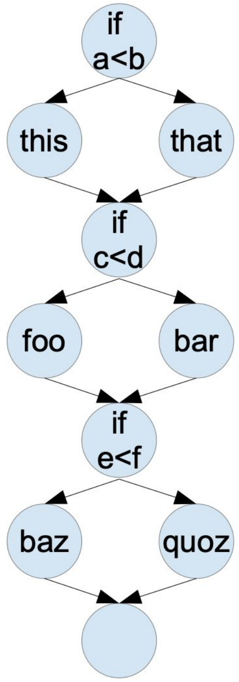
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements ...
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **2^N paths**



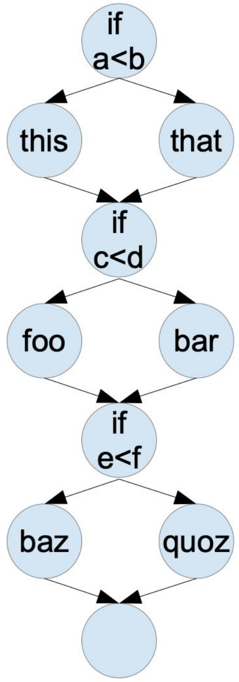
Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements ...
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **2^N paths**
  - You need **2^N tests** to cover them



Lens of Logic: maximize coverage

- If you have **N** sequential (or serial) if statements …
- There are **2N** branch edges
  - Which you could cover in 2 tests!
    - One always goes left, one always right
- But there are **2^N paths**
  - You need **2^N tests** to cover them
- Recall that path coverage **subsumes** branch coverage



Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path

Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
  - If we could do that, branch/statement/etc coverage is easy

Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
    - If we could do that, branch/statement/etc coverage is easy
- **Key idea**: solve this problem with **math**

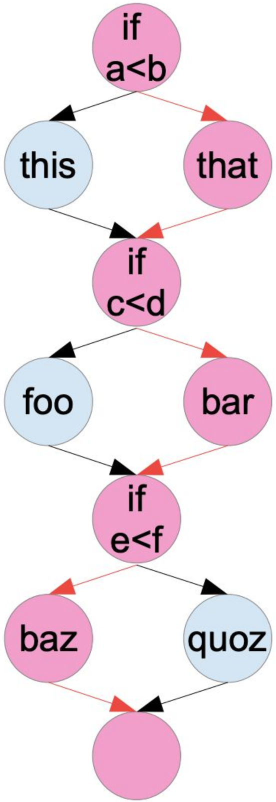
Lens of Logic: maximize coverage

- Consider generating test inputs to cover a path
  - If we could do that, branch/statement/etc coverage is easy
- **Key idea**: solve this problem with **math**

Definition: a ***path predicate*** (or ***path condition***, or ***path constraint***) is a boolean formula over program variables that is true when the program executes the given path

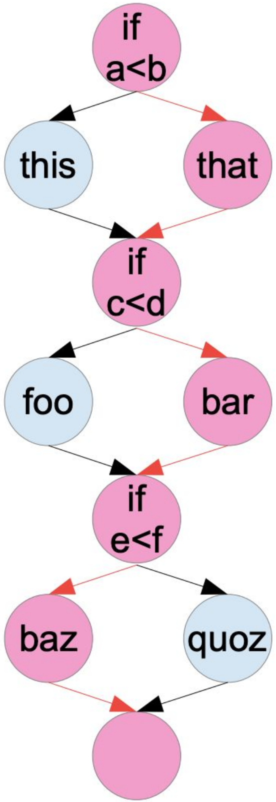
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?



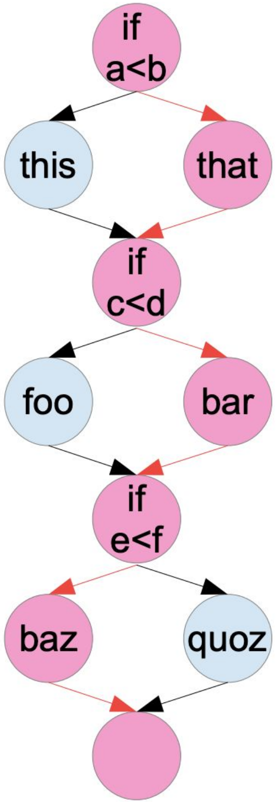
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?
  - `a >= b && c >= d && e < f`



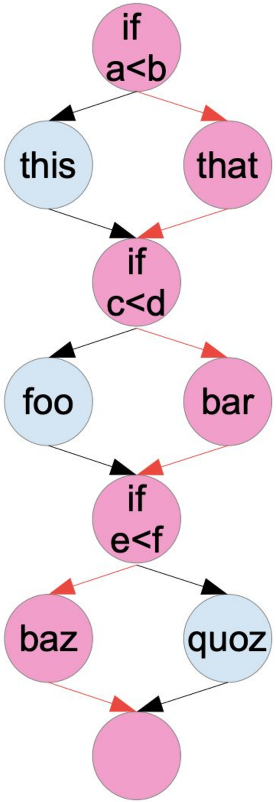
Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?
  - `a >= b && c >= d && e < f`
- When the path predicate is true, control flow will follow the given path



Lens of Logic: path predicate example

- Consider the highlighted (in **pink**) path
  - i.e., “false, false, true”
- What is its path predicate?
  - `a >= b && c >= d && e < f`
- When the path predicate is true, control flow will follow the given path
- So, given a path predicate, how do we choose a test input that covers the path?



Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

- What is a satisfying assignment for
  - `a >= b && c >= d && e < f` ?

Lens of Logic: solving path predicates

Definition: A ***satisfying assignment*** is a mapping from variables to values that makes a predicate true.

- What is a satisfying assignment for
  - `a >= b && c >= d && e < f` ?
    - `a=5, b=4, c=3, d=2, e=1, f=2`
    - `a=0, b=0, c=0, d=0, e=0, f=1`
    - … many more

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
  - Option 1: **ask humans**
    - labor-intensive, slow, expensive, etc.

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
    - Option 1: **ask humans**
        - labor-intensive, slow, expensive, etc.
    - Option 2: repeatedly **guess randomly**
        - works surprisingly well (when answers are **not sparse**)

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
  - Option 1: **ask humans**
    - labor-intensive, slow, expensive, etc.
  - Option 2: repeatedly **guess randomly**
    - works surprisingly well (when answers are **not sparse**)
  - Option 3: use an ***automated theorem prover***
    - cf. Wolfram Alpha, MatLab, Mathematica, Z3, etc.
    - works very well for a **restricted class of equations** (e.g., linear but not arbitrary polynomials, etc.; more detail in week 14)

Lens of Logic: solving path predicates

- How do we find satisfying assignments in general?
  - Option 1: **ask humans**
    - labor-intensive, slow, expensive, etc.
  - Option 2: repeatedly **guess randomly**
    - works surprisingly well (when answers are **not sparse**)
  - Option 3: use an ***automated theorem prover***
    - cf. Wolfram Alpha, MatLab, Mathematica, Z3, etc.
    - works very well for a **restricted class of equations** (e.g., linear but not arbitrary polynomials, etc.; more detail in week 14)

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
    - A solution is a satisfying assignment of values to input variables → those are your test input

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
    - A solution is a satisfying assignment of values to input variables → those are your test input
    - None found? Dead code, tough predicate, etc.

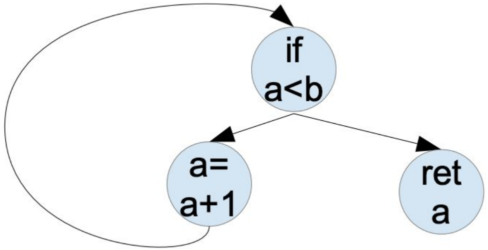
Lens of Logic: enumerating paths

- What could **go wrong** with enumerating paths in a method?

Lens of Logic: enumerating paths

- What could **go wrong** with enumerating paths in a method?
- There could be **infinitely many**!

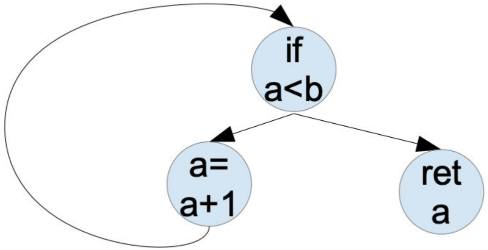
```
while a < b:
  a = a + 1
return a
```



Lens of Logic: enumerating paths

- What could **go wrong** with enumerating paths in a method?
- There could be **infinitely many**!

```
while a < b:
  a = a + 1
return a
```



- One path corresponds to executing the loop once, another to twice, another to three times, etc.

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, **approximate** instead

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:

Lens of Logic: enumerating paths: approximation

- **Key idea**: don’t enumerate all paths, approximate instead
- Typical Approximations:
    - Consider only acyclic paths (corresponds to taking each loop zero times or one time)

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, approximate instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times

Lens of Logic: enumerating paths: approximation

- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times
  - Enumerate paths breadth-first or depth-first and **stop after *k*** paths have been enumerated

Lens of Logic: enumerating paths: approximation

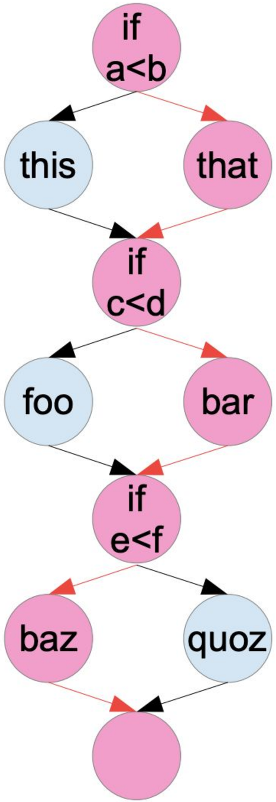
- **Key idea**: don't enumerate all paths, **approximate** instead
- Typical Approximations:
  - Consider only **acyclic** paths (corresponds to taking each loop zero times or one time)
  - Consider only taking each loop **at most *k*** times
  - Enumerate paths breadth-first or depth-first and **stop after *k*** paths have been enumerated
  - Concretely execute the program and see what it does (we'll come back to this later when we discuss **concolic testing**)

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
    - A solution is a satisfying assignment of values to input variables → those are your test input
    - None found? Dead code, tough predicate, etc.

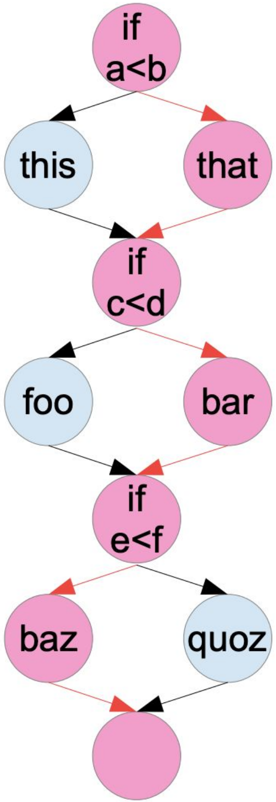
Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?



Lens of Logic: collecting path predicates

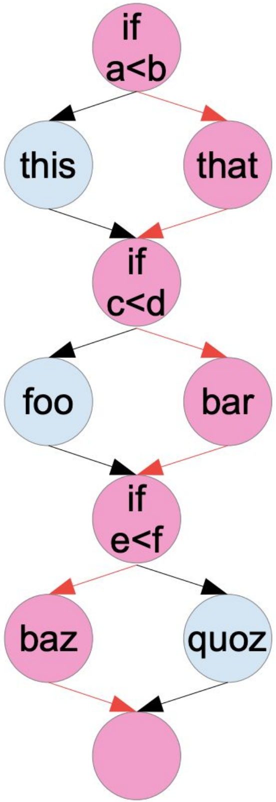
- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?
  - The path predicate may not be **expressible** in terms of the inputs we control



Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** the path predicate?
  - The path predicate may not be **expressible** in terms of the inputs we control

```
foo(a,b):
  str1 = read_from_url("abc.com")
  str2 = read_from_url("xyz.com")
  if (str1 == str2): bar()
```

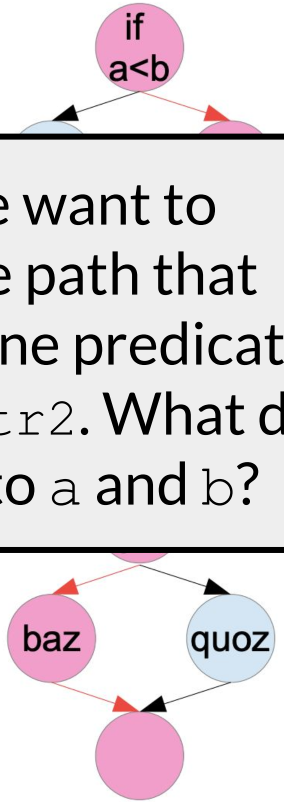


Lens of Logic: collecting path predicates

- Now we have a path through the program
- What could go wrong with **collecting** th
  predicate?
  - The path predicate may not be **expr**
    terms of the inputs we control

Suppose we want to exercise the path that calls `bar`. One predicate is `str1==str2`. What do you assign to `a` and `b`?

```
foo(a,b):
  str1 = read_from_url("abc.com")
  str2 = read_from_url("xyz.com")
  if (str1 == str2): bar()
```



Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?

Lens of Logic: path predicate woes

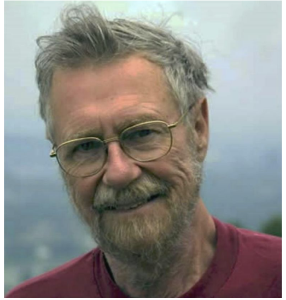
- When we can’t solve for a path predicate, what can we do?
    - **Ignore the problem** (i.e., don’t generate a test)

Lens of Logic: path predicate woes

- When we can’t solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don’t generate a test)
- Remember, testing can show the presence of bugs, but not their absence

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence



Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
    - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
    - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can
  - Ask the solver to find a solution in terms of the input variables

Lens of Logic: path predicate woes

- When we can't solve for a path predicate, what can we do?
  - **Ignore the problem** (i.e., don't generate a test)
- Remember, testing can show the presence of bugs, but not their absence → **no guarantee** either way
- So, we make a **best effort**:
  - Collect the path predicates as best we can
  - Ask the solver to find a solution in terms of the input variables
  - If it can't (because the math is too hard, we don't control the input, etc.), we give up

Lens of Logic: test input generation plan

- Consider generating high-branch-coverage tests for a method:
- **Enumerate** “all” paths in the method
- For each path, **collect** the path predicate
- For each path predicate, **solve** it
  - A solution is a satisfying assignment of values to input variables → those are your test input
  - None found? Dead code, tough predicate, etc.

Lens of Logic: test input generation plan

- Recall: we want to automatically generate **test cases**

Lens of Logic: test input generation plan

- Recall: we want to automatically generate **test cases**
- We have an approach that works well in practice:
  - **Enumerate** some paths
  - **Extract** their path constraints
  - **Solve** those path constraints

Symbolic execution in practice

- symbolic execution was invented in the 1970s
  - but theorem provers of the time could **rarely solve predicates**, and the available hardware could enumerate **few paths**
- modern **SMT solvers** can handle the first problem, while second problem is less relevant due to **Moore's Law**

Aside: SMT solvers

- the path predicates I've used as examples today have mostly been **boolean formulas**

Aside: SMT solvers

- the path predicates I've used as examples today have mostly been **boolean formulas**
  - how hard is it to check if a boolean formula is **satisfiable**?

Aside: SMT solvers

- the path predicates I've used as examples today have mostly been **boolean formulas**
  - how hard is it to check if a boolean formula is **satisfiable**?
    - boolean satisfiability is **the classic NP-complete problem**

Aside: SMT solvers

- the path predicates I've used as examples today have mostly been **boolean formulas**
  - how hard is it to check if a boolean formula is **satisfiable**?
    - boolean satisfiability is **the classic NP-complete problem**
- in practice, path predicates also include other kinds of expressions besides booleans
  - e.g., linear arithmetic, checking whether a pointer is null, etc.

Aside: SMT solvers

- the path predicates I've used as examples today have mostly been **boolean formulas**
  - how hard is it to check if a boolean formula is **satisfiable**?
    - boolean satisfiability is **the classic NP-complete problem**
- in practice, path predicates also include other kinds of expressions besides booleans
  - e.g., linear arithmetic, checking whether a pointer is null, etc.
- an SMT solver is a **generalization** of a SAT solver that uses ***theories*** of other kinds of expressions to handle real programs
  - we'll come back to SMT solvers in week 14

Aside: SMT solvers

- the path predicates I've used as examples today have mostly been **boolean formulas**
  - how hard is it to check if a boolean
    - boolean satisfiability is **the clas**
- in practice, path predicates also include besides booleans
  - e.g., linear arithmetic, checking whe
- an SMT solver is a **generalization** of a SAT solver that uses ***theories*** of other kinds of expressions to handle real programs
  - we'll come back to SMT solvers in week 14

Modern SMT solvers (e.g., Z3, cvc5) are **extraordinarily effective** at solving most instances (millions or billions of clauses in < 30s)

Symbolic execution in practice

- symbolic execution was invented in the 1970s
  - but theorem provers of the time could **rarely solve predicates**, and the available hardware could enumerate **few paths**
- modern **SMT solvers** can handle the first problem, while second problem is less relevant due to **Moore's Law**

Aside: Moore’s Law

Aside: Moore’s Law

- Moore’s Law says that the available computing power for a given price **increases exponentially**

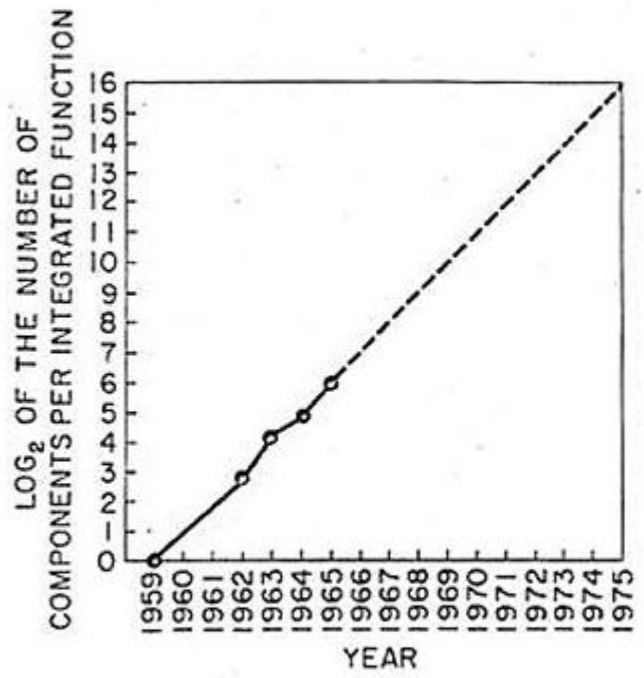


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Aside: Moore’s Law

- Moore’s Law says that the available computing power for a given price **increases exponentially**
- **not actually a law**, but an observation that has been true basically since the invention of the transistor

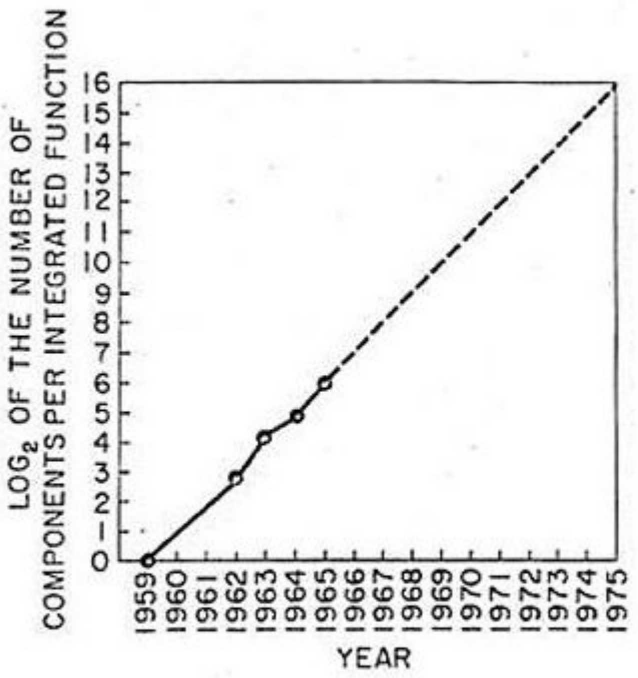


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Aside: Moore’s Law

- Moore’s Law says that the available computing power for a given price **increases exponentially**
- **not actually a law**, but an observation that has been true basically since the invention of the transistor
    - but might not last forever

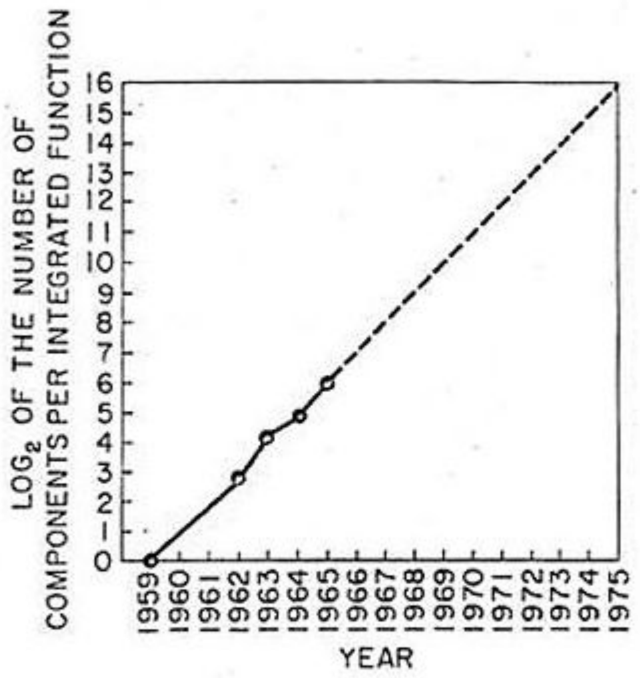


Fig. 2 Number of components per Integrated function for minimum cost per component extrapolated vs time.

Aside: Moore's Law

- Moore's Law says that the available computing power for a given price **increases exponentially**
- **not actually a law**, but an observation that has been true basically since the invention of the transistor
  - but might not last forever
- **implication**: amount of computing power available today is huge compared to the 1970s

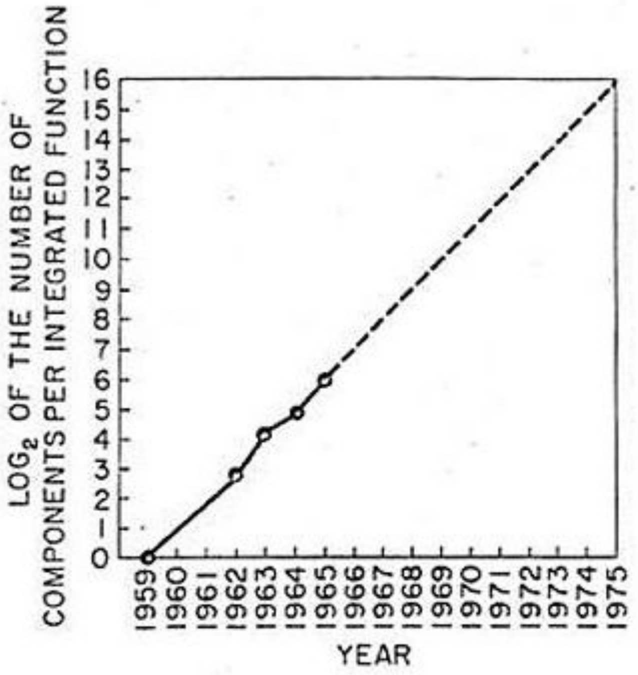


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

Symbolic execution in practice

- symbolic execution was invented in the 1970s
  - but theorem provers of the time could **rarely solve predicates**, and the available hardware could enumerate **few paths**
- modern **SMT solvers** can handle the first problem, while second problem is less relevant due to **Moore's Law**
- **implication**: symbolic execution has been **widely deployed in industry** since the early 2000s
  - e.g., PREfix (Microsoft), Coverity, KLEE

Symbolic execution: pros and cons

Symbolic execution: pros and cons

- the biggest strength of symbolic execution is that it produces **no false positives**

Symbolic execution: pros and cons

- the biggest strength of symbolic execution is that it produces **no false positives**
  - that is, every test it generates really does lead to a violation of whatever policy it is enforcing (e.g., really leads to a crash)

Symbolic execution: pros and cons

- the biggest strength of symbolic execution is that it produces **no false positives**
  - that is, every test it generates really does lead to a violation of whatever policy it is enforcing (e.g., really leads to a crash)
- there are two serious downsides:

Symbolic execution: pros and cons

- the biggest strength of symbolic execution is that it produces **no false positives**
  - that is, every test it generates really does lead to a violation of whatever policy it is enforcing (e.g., really leads to a crash)
- there are two serious downsides:
  - it is **expensive** (recall: it relies on solving an NP-complete problem repeatedly!)

Symbolic execution: pros and cons

- the biggest strength of symbolic execution is that it produces **no false positives**
  - that is, every test it generates really does lead to a violation of whatever policy it is enforcing (e.g., really leads to a crash)
- there are two serious downsides:
  - it is **expensive** (recall: it relies on solving an NP-complete problem repeatedly!)
  - it **cannot cover** many parts of programs (recall: solving path predicates is NP-complete, so solvers sometimes fail!)

More test input generation: agenda

- Other approaches that use random testing
  - “feedback-directed” random testing
  - brief introduction to mutation testing
  - EvoSuite: mutation testing + a genetic algorithm
- Lens of Logic: symbolic execution for test input generation
  - **concolic testing**

Limits of Symbolic Execution

- however, symbolic execution has **serious limitations**
  - for example, consider the function to the right:

```
testme(int x, int y) {
  if (bbox(x) == y) {
    ERROR;
  } else {
    // OK
  }
}
```

Limits of Symbolic Execution

- however, symbolic execution has **serious limitations**
  - for example, consider the function to the right:
- if `bbox(x)` is **uninterpretable**, then symbolic execution cannot determine if the ERROR statement is reachable

```
testme(int x, int y) {
  if (bbox(x) == y) {
    ERROR;
  } else {
    // OK
  }
}
```

Limits of Symbolic Execution

- however, symbolic execution has **serious limitations**
  - for example, consider the function to the right:
- if `bbox(x)` is **uninterpretable**, then symbolic execution cannot determine if the ERROR statement is reachable

```
testme(int x, int y) {
  if (bbox(x) == y) {
    ERROR;
  } else {
    // OK
  }
}
```

Key question: how could we get around this limitation?

Concolic Testing

Definition: ***concolic testing*** combines concrete execution of the program (via other test generation techniques) with symbolic execution

Concolic Testing

Definition: ***concolic testing*** combines concrete execution of the program (via other test generation techniques) with symbolic execution

- "concolic" is a portmanteau of "***conc***rete" and "symb***olic***"

Concolic Testing

Definition: ***concolic testing*** combines concrete execution of the program (via other test generation techniques) with symbolic execution

- “concolic” is a portmanteau of “***conc***rete” and “symb***olic***”
- **key idea**: when symbolic execution gets stuck, actually execute the program and record what values the uninterpretable code **actually produces**

Concolic Testing: example

- symbolic execution determines that bbox(x) is **uninterpretable**

```
testme(int x, int y) {
  if (bbox(x) == y) {
    ERROR;
  } else {
    // OK
  }
}
```

Concolic Testing: example

- symbolic execution determines that bbox(x) is **uninterpretable**
- choose a random value of x, then **execute** the program

```
testme(int x, int y) {
  if (bbox(x) == y) {
    ERROR;
  } else {
    // OK
  }
}
```

Concolic Testing: example

- symbolic execution determines that bbox(x) is **uninterpretable**
- choose a random value of x, then **execute** the program
- replace the call to bbox(x) with **whatever it returned** on the concrete execution

```
testme(int x, int y) {
  if (bbox(x) == y) {
    ERROR;
  } else {
    // OK
  }
}
```

Concolic Testing: example

- symbolic execution determines that bbox(x) is **uninterpretable**
- choose a random value of x, then **execute** the program
- replace the call to bbox(x) with **whatever it returned** on the concrete execution
- let symbolic execution **solve for y**

```
testme(int x, int y) {
  if (bbox(x) == y) {
    ERROR;
  } else {
    // OK
  }
}
```

HW4 in-class

- in today's in-class/homework, you'll run EvoSuite
    - in general, students usually report that this assignment is easier and less time-consuming than HW3
    - however, there are **two full length papers** to read for next week
        - so you'll have plenty to do this week :)