Software Architecture and Engineering

Automatic Test Case Generation

Peter Müller

Chair of Programming Methodology

Spring Semester 2012

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

9. Automatic Test Case Generation

Test Case Generation

```
int foo( boolean a, boolean b ) {
  int x = 1;
  int y = 1;
  if( a )
    x = 0;
  else
    y = 0;
  if( b )
    return 5 / x;
  else
    return 5 / y;
}
```

```
[ Test ]
public void TestFoo(
       boolean a,
       boolean b )
{
  int res = foo( a, b );
  Assert.IsTrue( res == 5 );
}
```

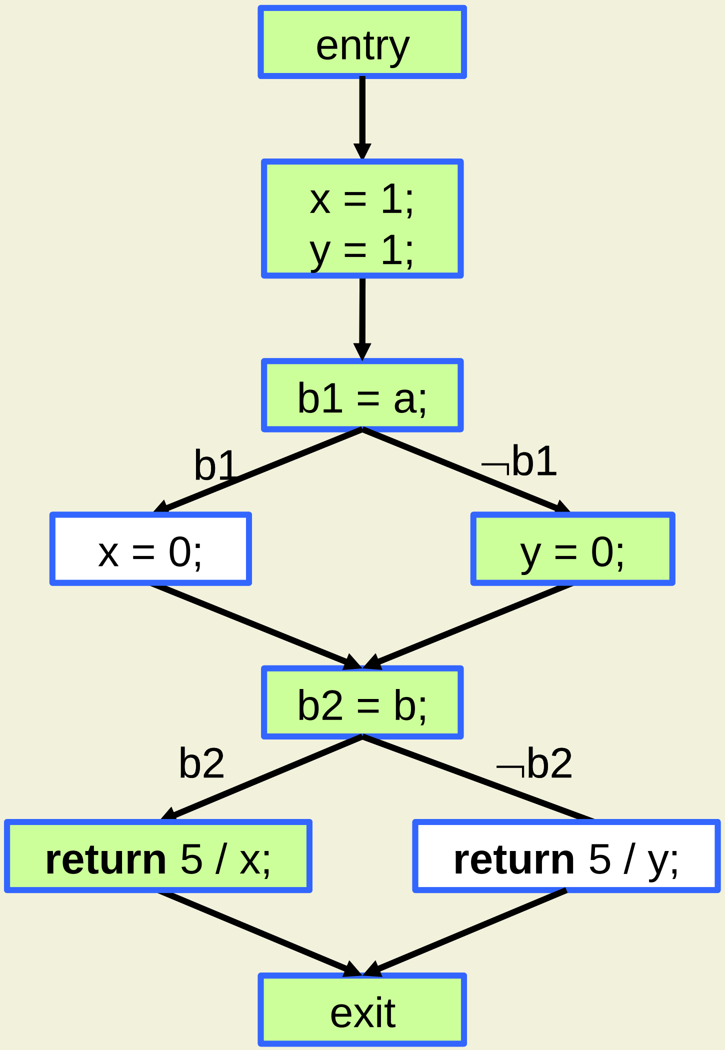
Challenge:
how to determine
test data?

Test driver is
straightforward

Specify test oracle via
parameterized unit test
or assertion

Determining Test Data

- **Choose path** to be tested
  - Based on coverage goals
- **Derive constraints** on inputs from conditions and statements on chosen path
- **Solve constraints** to obtain test inputs



9. Automatic Test Case Generation

9.1 Symbolic Execution

9.2 Concolic Testing

Symbolic Execution

- Symbolic execution **simulates the execution** of a program statically, using **symbolic** rather than concrete **values**
  - Introduce symbolic variables to represent inputs

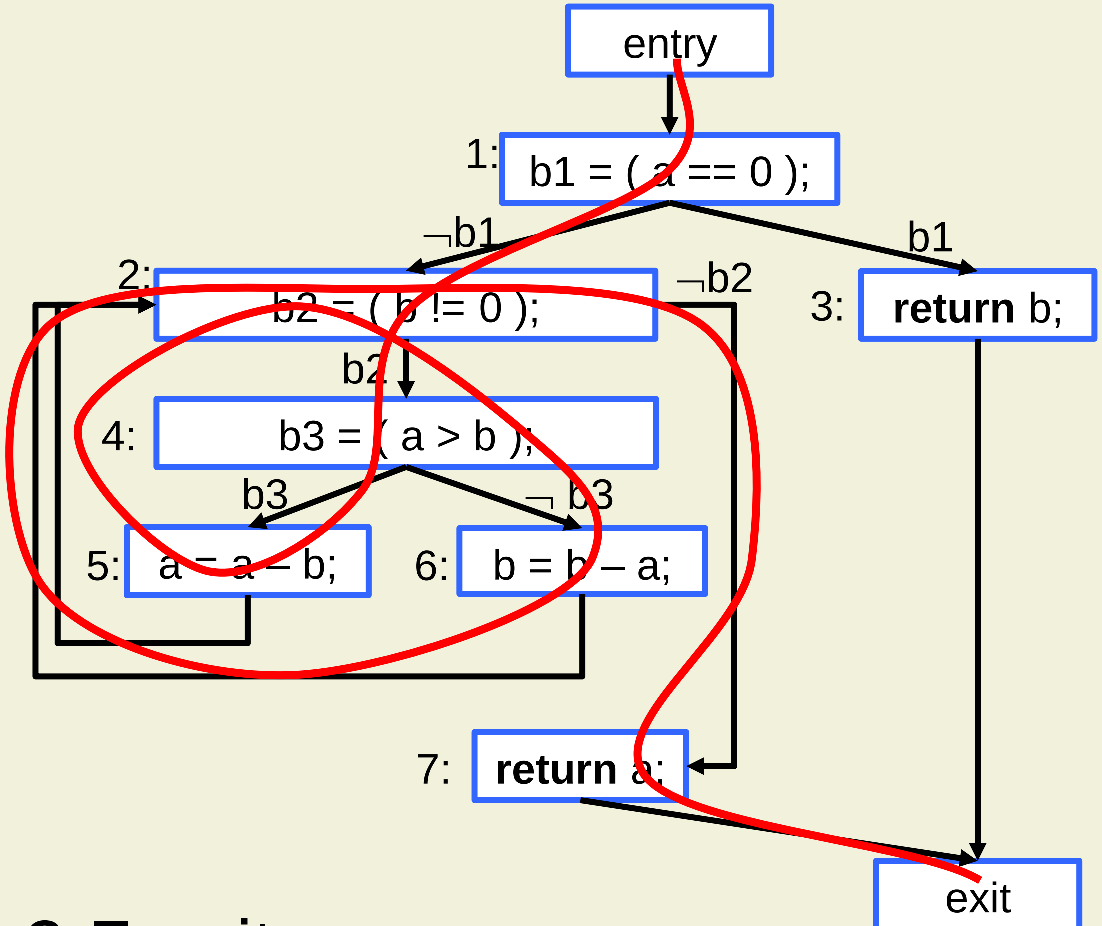
- Symbolic state consists of
  - A prefix of a **path** in the CFG
  - A **symbolic state**, which maps each variable of the program to an **expression over the symbolic variables**
  - A **path condition** (a constraint over the symbolic variables), which holds if and only if the execution takes the current path

Symbolic Execution Algorithm

- Symbolic execution can be described by an operational semantics that operates on symbolic states
- Key operations
  - Expressions: Evaluation yields **an expression**
  - Branches: Add label of target basic block to path prefix and conjoin branching-condition to path condition
  - Assignments: Update the symbolic state
- We look at heap data structures later

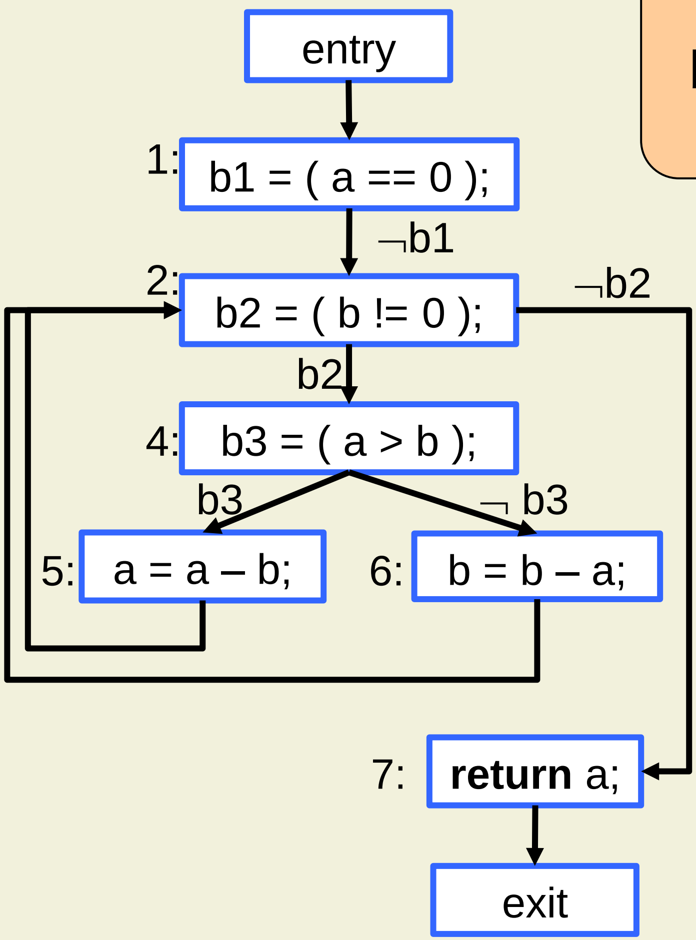
Example

```
int gcd( int a, int b ) {
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```



- Test the path entry-1-2-4-5-2-4-6-2-7-exit

Example: Constraint Generation



Next basic block to be executed

State changes

Additional conditions (conjuncts)

Path	State	Path Condition
1	a→A, b→B	true
2	b1→(A = 0)	A ≠ 0
4	b2→(B ≠ 0)	B ≠ 0
5	b3→(A > B)	A > B
2	a→(A – B)	true
4	b2→(B ≠ 0)	B ≠ 0
6	b3→(A – B > B)	¬(A – B > B)
2	b→(B – (A – B))	true
7	b2→(B – (A – B) ≠ 0)	¬(B – (A – B) ≠ 0)

Example: Constraint Solution

Path Condition
true
$A \neq 0$
$B \neq 0$
$A > B$
true
$B \neq 0$
$\neg(A - B > B)$
true
$\neg(B - (A - B) \neq 0)$

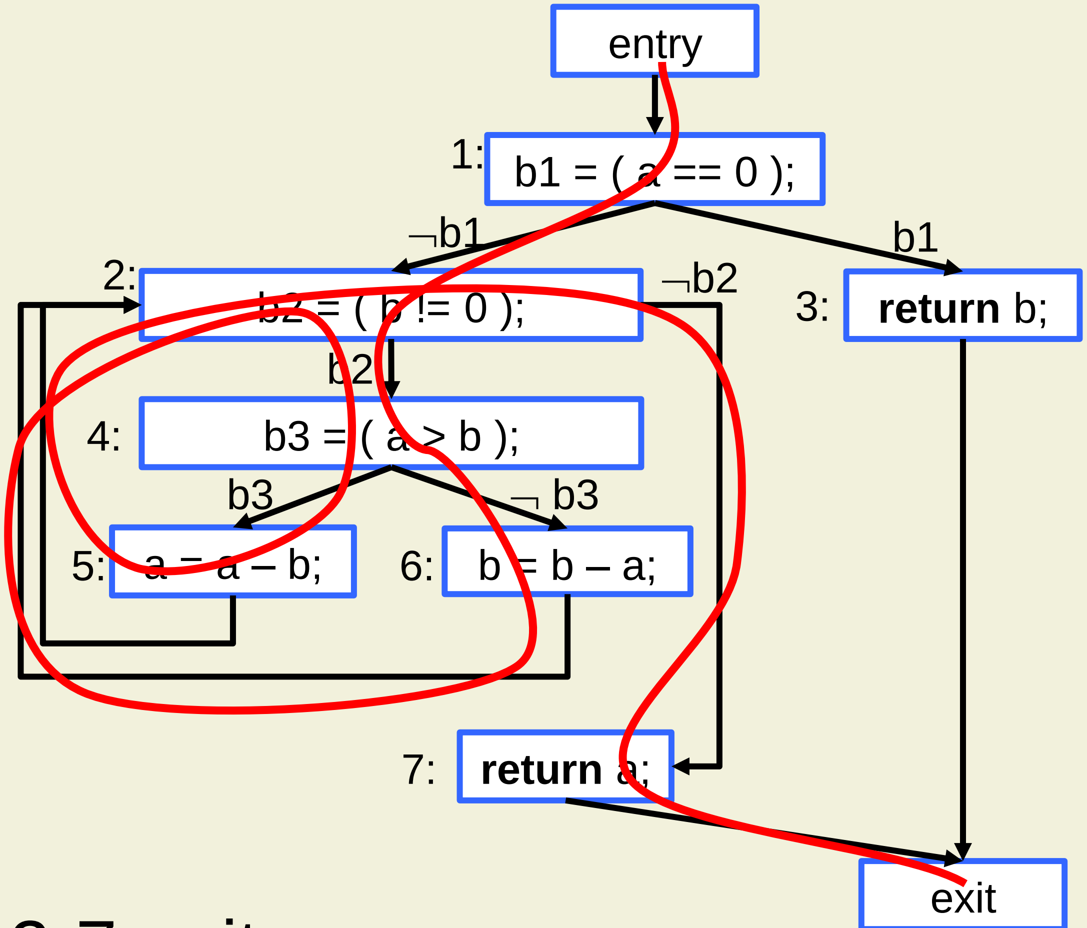
- Simplifying the path condition yields:
  $A \neq 0 \land B \neq 0 \land A > B \land A = 2 \times B$

- Possible solutions are for instance
  - $A = 2, B = 1$
  - $A = 4, B = 2$
- All solutions test the chosen path

```
int gcd( int a, int b ) {
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```

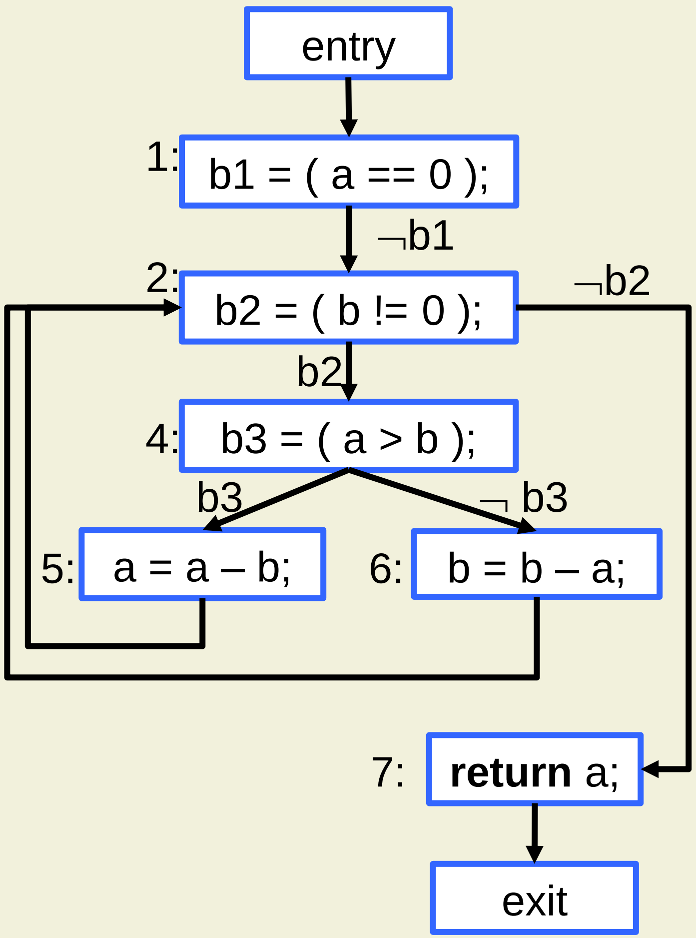
Example: Another Path

```
int gcd( int a, int b ) {
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```



- Test the path
  entry-1-2-4-6-2-4-5-2-7-exit

Example: Constraint Generation



Path	State	Path Condition
1	a→A, b→B	true
2	b1→(A = 0)	A ≠ 0
4	b2→(B ≠ 0)	B ≠ 0
6	b3→(A > B)	¬(A > B)
2	b→(B – A)	true
4	b2→(B – A ≠ 0)	B – A ≠ 0
5	b3→(A > B – A)	A > B – A
2	a→(A – (B – A))	true
7	b2→((B – A) ≠ 0)	¬((B – A) ≠ 0)

Example: Constraint Solution

Path Condition
true
$A \neq 0$
$B \neq 0$
$\neg(A > B)$
true
$B - A \neq 0$
$A > B - A$
true
$\neg((B - A) \neq 0)$

- The path condition is unsatisfiable: $B - A \neq 0 \wedge \neg((B - A) \neq 0)$
- The chosen path is not feasible
  - There is no input that will execute this path
- Infeasible paths do not necessarily indicate dead code

```
int gcd( int a, int b ) {
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```

Constraints

- Constraints can be classified according to the **domain of the variables** and the **relations between the variables**
- Common domains
  - Booleans, bounded integers, integers, rationals, set, sequences, functions, etc.
- Common relations
  - Linear constraints, polynomial constraints, etc.
- Generated constraints depend on the programming language and the UUT

Constraint Solving

- For many of the interesting constraints, constraint solving is **NP-complete**
  - E.g., linear constraints over bounded integers
- For some classes, constraint solving is **undecidable**
  - E.g., Non-linear constraints over rationals
- Nevertheless, useful tools with **powerful heuristics** exist
  - E.g., SMT (Satisfiability Modulo Theories) solvers

Applications of Symbolic Execution

- Enumerate all paths to achieve a **given coverage**
  - E.g., branch and loop coverage
  - Generate inputs for each path

- Enumerate paths until a **timeout** is reached
  - Limits in particular the number of loop iterations

- **Active bug searching**
  - Attempt to create inputs that trigger an error

Bug Searching: Example

```
boolean isPalindrome( int[ ] a ) {
  int j;
  if( a == null ) throw new NullPointerException( );
  j = a.length – 1;
  for( int i = 0; i < j; i++ ) {
    if( a == null ) throw new NullPointerException( );
    if( i < 0 || a.length <= i ) throw new IndexOutOfBoundsException( );
    if( j < 0 || a.length <= j ) throw new IndexOutOfBoundsException( );
    if( a[ i ] != a[ j ] )
      return false;
    j--;
  }
  return true;
}
```

Attempt to generate inputs that execute exceptional paths

If the exceptional path is not feasible then the error cannot occur

Bug Searching and Oracles

```
int abs( int x ) {
  if( x < 0 )     return –x;
  else            return x;
}
```

```
[ Test ]
public void TestAbs( int x ) {
  int res = abs( x );
  Assert.IsTrue( 0 <= res );
}
```

- CFG contains two paths with path conditions x < 0 and 0 <= x
- Covering both paths does not necessarily detect the bug

Bug Searching and Oracles (cont’d)

```
[ Test ]
public void TestAbs( int x ) {
  int res = abs( x );
  Assert.IsTrue( 0 <= res );
}
```

```
int abs( int x )
  ensures 0 <= res;
{
  if( x < 0 )     return –x;
  else            return x;
}
```

```
int abs( int x ) {
  if ( x < 0 )
    if ( 0 <= –x )
      return –x;
    else
      throw new ContractException( … );
  else
    if ( 0 <= x )
      return x;
    else
      throw new ContractException( … );
}
```

Bug Searching and Oracles (cont'd)

```
int abs( int x ) {
  if ( x < 0 )
    if ( 0 <= –x )
      return –x;
    else
      throw new ContractException( … );
  else
    if ( 0 <= x )
      return x;
    else
      throw new ContractException( … );
}
```

- Instrumented method contains four paths
- Constraints
  - $x < 0 \wedge 0 <= -x$: correct
  - $x < 0 \wedge \neg 0 <= -x$: error
  - $\neg x < 0 \wedge 0 <= x$: correct
  - $\neg x < 0 \wedge \neg 0 <= x$: infeasible
- Instrumentation causes symbolic execution to actively attempt to violate the test oracle

Undesired Test Cases

```
class SavingsAccount {
  int balance;
  // invariant: 0 <= balance
  void deposit( int a )
  {
    balance = balance + a;
    assert 0 <= balance;
  }
}
```

Check invariant as part of the oracle

```
class SavingsAccount {
  int balance;
  // invariant: 0 <= balance
  void deposit( int a )
  {
    balance = balance + a;
    if( ! ( 0 <= balance ) )
      throw new Exception( );
  }
}
```

- Symbolic execution will find test cases that violate the assertion

balance	a
0	-5
-5	0
Integer.MAX_VALUE	1

Assume Statements

- Assume statements introduce constraints that are trusted by the symbolic execution
- Useful for
  - Preconditions
  - Invariants
  - Properties of called methods
  - Information about the environment

```
class SavingsAccount {
  int balance;
  // invariant: 0 <= balance
  void deposit( int a )
  {
    assume 0 <= balance;
    assume 0 <= a;
    balance = balance + a;
    assume 0 <= balance;
    if( ! ( 0 <= balance ) )
      throw new Exceptio
  }
}
```

Invariant

Precondition

No overflow

Assert vs. Assume

Assert

- Checked at run time
- Checked by symbolic execution (bug search)
- Confirm that condition holds

```
int div( int p ) {
  int x = p – 1;
  assert x != 0;
  return 5 / x;
}
```

Assume

- Checked at run time
- Trusted by symbolic execution
- Express properties that are justified elsewhere

```
int div( int p ) {
  int x = p – 1;
  assume x != 0;
  return 5 / x;
}
```

Symbolic Execution with Assumptions

- We encode path conditions as sequences of elements of the form
  - brn(P) for a **branching condition** P (if, while, etc.)
  - asm(P) for an **assumption** P
  - During symbolic execution, branching statements and assume statements **append an element** to the sequence
- The path condition encoded by a sequence s is

$$\bigwedge_{(\, brn(\, P\,) \in s \,\wedge\, asm(\, P\,) \in s\,)} P$$

  - brn(P) and asm(P) are treated identically in the path condition, but we will use them differently later

SE with Assumptions: Example

```
int div( int p ) {
  int x = p – 1;
  if( x == 0 )
    throw new Exception( );
  return 5 / x;
}
```

- Symbolic execution produces [brn(P–1=0)]
  - Path condition is $P-1=0$
  - Test fails for p=1

```
int div( int p ) {
  int x = p – 1;
  assume x != 0;
  if( x == 0 )
    throw new Exception( );
  return 5 / x;
}
```

- Symbolic execution produces [asm(P–1≠0), brn(P–1=0)]
  - Path condition is $P-1 \neq 0 \wedge P-1=0$
  - Path is infeasible

Heaps Data Structures

- Conditions and oracles may depend on heap values
- Symbolic states must track values of heap locations
- Path conditions may include constraints over heap locations

```
class Account {
  int balance;
  void transfer( Account to, int a )
    requires to != null;
    ensures this.balance ==
              old( this.balance ) – a;
  {
    this.balance = this.balance – a;
    to.balance = to.balance + a;
  }
}
```

Symbolic Heaps

- The heap memory can be modeled as an updatable map per field f
  - $Heap_f$: Reference $\rightarrow$ Value
- Heap operations
  - Field read:
    select: Heap $\times$ Reference $\rightarrow$ Value
  - Field update:
    store: Heap $\times$ Reference $\times$ Value $\rightarrow$ Heap
- Axioms
  - $\forall$H,r,v.select(store(H, r, v), r) = v
  - $\forall$H,r,r',v.r $\neq$ r' $\Rightarrow$ select(store(H, r, v), r') = select(H, r')

Symbolic Heaps: Example

Symolic state:
this→T0, to→T1, a→A, heap→H0

Symbolic heap: heap→H1
H1≡store(H0, T0, select(H0, T0) – A)

Symbolic heap: heap→H2
H2≡store(H1, T1, select(H1, T1) + A)

```
class Account {
  int balance;
  void transfer( Account to, int a)
    requires to != null;
    ensures this.balance ==
              old( this.balance ) – a;
  {
    this.balance = this.balance – a;
    to.balance = to.balance + a;
  }
}
```

- Path condition for postcondition includes select(H2, T0) = select(H0, T0) – A

Limitations: Method Calls

- Results of method calls may be
  - Assigned to variables: how to update symbolic state?
  - Used as conditions: how to record path condition?

```
Seq prefix( Seq s, int n ) {
  if ( s.length( ) < n )  return s;
  else                    …
}
```

```
void transfer( Account to, int a) {
  checkAvailability( a );
  this.balance = this.balance – a;
  to.balance = to.balance + a;
}
```

- Methods may have side effects
  - How to update the symbolic heap?

- Perform interprocedural symbolic execution
  - Issues: scalability, modularity

Limitations: Constraint Solving

```
void roots( double a, double b, double c ) {
  double q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    double r = Math.sqrt( q );
    x1 = (-b + r) / (2 * a);
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    x1 = -b / (2 * a);
  } else {
    numRoots = 0;
  }
}
```

- Nonlinear constraint $b^2 - 4ac > 0 \wedge a \neq 0$ over reals is difficult to solve
- Without a solution, no test data will be provided
  - Neither for the other branches

9. Automatic Test Case Generation

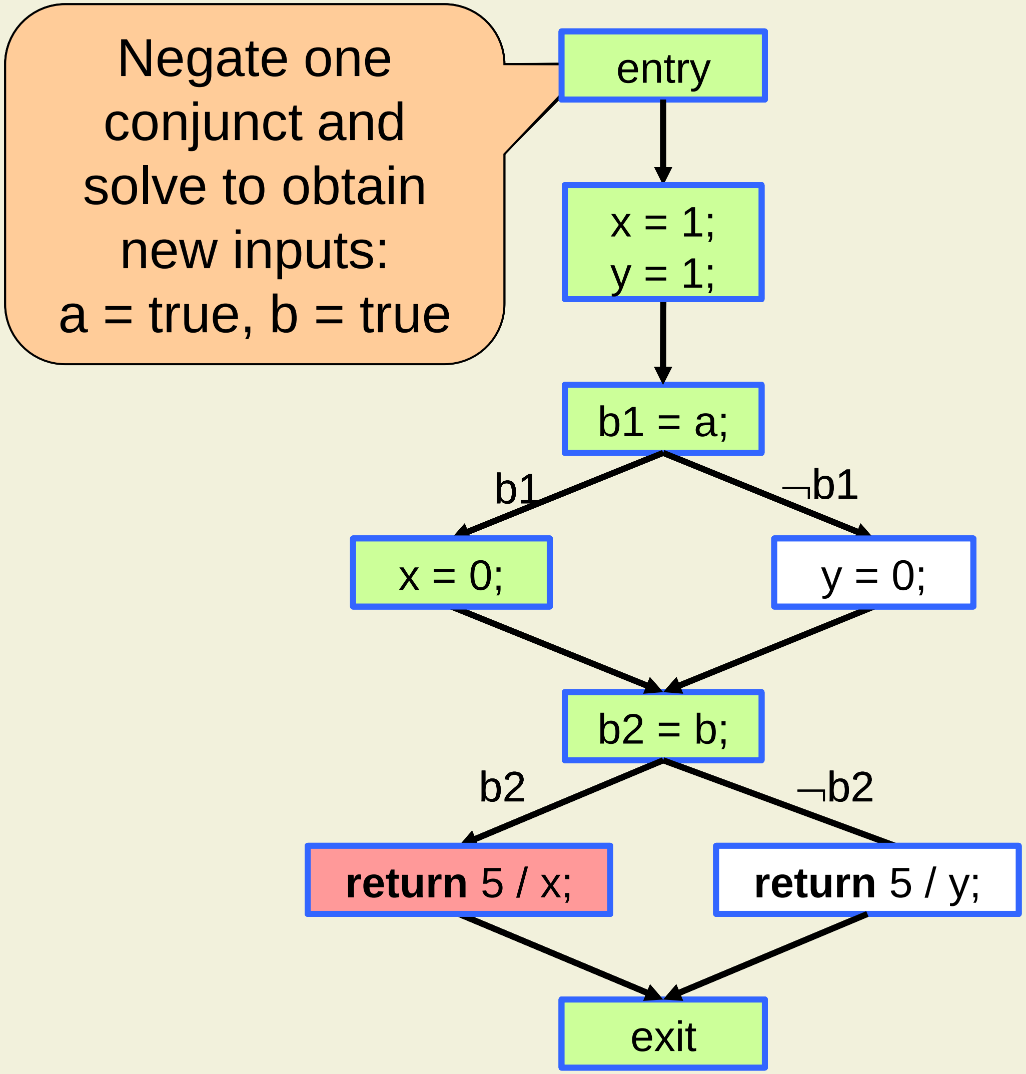
9.1 Symbolic Execution

9.2 Concolic Testing

Approach

- Combine concrete program executions with symbolic execution

- Use values from concrete executions when constraint solving fails



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

(Simplified) Procedure

- Attempt to explore all paths whose path condition starts with a given prefix

```
procedure explore( Seq<Condition> prefix ) is
  values              := solve( prefix );
  if solution is available then
    path              := executeConcrete( values );
    pathCondition     := executeSymbolic( path );
    extension         := pathCondition[ |prefix| … ];
    foreach non-empty prefix p of extension do
      if p = p’ ○ [ brn(c) ] for some p’,c then
        explore( prefix ○ p’ ○ [ brn(¬c) ] );
      end
    end
  end
end
```

Obtain concrete values for prefix

Obtain path condition without the given prefix

For each additional branching condition, explore negation

Testing starts with a call explore([ ])

Example: Roots

```
void roots( int a, int b, int c ) {
  int q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    int r = Math.sqrt( q );
    if( a == 0 ) throw new DivisionByZeroException( );
    x1 = (-b + r) / (2 * a);
    if( a == 0 ) throw new DivisionByZeroException( );
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    if( a == 0 ) throw new DivisionByZeroException( );
    x1 = -b / (2 * a);
  } else { numRoots = 0; }
}
```

Example: Step 1

```
void roots( int a, int b, int c ) {
  int q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    int r = Math.sqrt( q );
    if( a == 0 ) abort;
    x1 = (-b + r) / (2 * a);
    if( a == 0 ) abort;
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    if( a == 0 ) abort;
    x1 = -b / (2 * a);
  } else { numRoots = 0; }
}
```

- Start by calling explore([ ])
- Solving path condition for [ ] yields arbitrary values, e.g., A=1, B=2, C=3
- Test passes and yields path condition (Q ≡ B×B – 4×A×C) [$\neg$(Q>0 $\wedge$ A≠0), $\neg$(Q=0)]
- Recurs on [$\neg$(Q>0 $\wedge$ A≠0), Q=0] and [Q>0 $\wedge$ A≠0]

We omit the brn since there are no assumptions in this example

Example: Step 2

```
void roots( int a, int b, int c ) {
  int q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    int r = Math.sqrt( q );
    if( a == 0 ) abort;
    x1 = (-b + r) / (2 * a);
    if( a == 0 ) abort;
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    if( a == 0 ) abort;
    x1 = -b / (2 * a);
  } else { numRoots = 0; }
}
```

- Solving $\neg(Q>0 \land A\neq0) \land Q=0$ yields e.g. A=0, B=0, C=0
- Test fails and yields path condition $[\neg(Q>0 \land A\neq0), Q=0, A=0]$
- Recurs on $[\neg(Q>0 \land A\neq0), Q=0, A\neq0]$

Example: Step 3

```
void roots( int a, int b, int c ) {
  int q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    int r = Math.sqrt( q );
    if( a == 0 ) abort;
    x1 = (-b + r) / (2 * a);
    if( a == 0 ) abort;
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    if( a == 0 ) abort;
    x1 = -b / (2 * a);
  } else { numRoots = 0; }
}
```

- Solving $\neg(Q>0 \wedge A\neq0) \wedge Q=0 \wedge A\neq0$ yields e.g. A=1, B=0, C=0
- Test passes and yields path condition $[\ \neg(Q>0 \wedge A\neq0), Q=0, A\neq0\ ]$
- No further recursion because path condition is identical to prefix

Example: Step 4

```
void roots( int a, int b, int c ) {
  int q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    int r = Math.sqrt( q );
    if( a == 0 ) abort;
    x1 = (-b + r) / (2 * a);
    if( a == 0 ) abort;
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    if( a == 0 ) abort;
    x1 = -b / (2 * a);
  } else { numRoots = 0; }
}
```

- Solving
  $Q>0 \wedge A \neq 0$
  yields e.g. A=-1, B=3, C=-1
- Test passes and yields path condition
  $[\ (Q>0 \wedge A \neq 0),\ A \neq 0,\ A \neq 0\ ]$
- Recurs on
  $[\ (Q>0 \wedge A \neq 0),\ A=0\ ]$ and
  $[\ (Q>0 \wedge A \neq 0),\ A \neq 0,\ A=0\ ]$

Example: Step 5

```
void roots( int a, int b, int c ) {
  int q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    int r = Math.sqrt( q );
    if( a == 0 ) abort;
    x1 = (-b + r) / (2 * a);
    if( a == 0 ) abort;
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    if( a == 0 ) abort;
    x1 = -b / (2 * a);
  } else { numRoots = 0; }
}
```

- Solving
  $(Q>0 \wedge A \neq 0) \wedge A=0$
  yields no result
- The path is infeasible, that is, this division by zero cannot occur
- Analogous for the condition
  $(Q>0 \wedge A \neq 0) \wedge A \neq 0 \wedge A=0$

GCD Example: Step 1

```
int gcd( int a, int b ) {
  assume 0 <= a;
  assume 0 <= b;
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```

- Start by calling explore([ ])
- Solving path condition for [ ] yields arbitrary values, e.g., A=0, B=0
- Test passes and yields path condition [asm(0<=A), asm(0<=B), brn(A=0)]
- Recurs on [asm(0<=A), asm(0<=B), brn(A≠0)]

GCD Example: Step 2

```
int gcd( int a, int b ) {
  assume 0 <= a;
  assume 0 <= b;
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```

- Solving
  $0 <= A \wedge 0 <= B \wedge A \neq 0$
  yields e.g. A=1, B=0
- Test passes and yields path condition
  [asm(0<=A), asm(0<=B), brn($\neg$A=0), brn($\neg$B≠0)]
- Recurs on
  [asm(0<=A), asm(0<=B), brn($\neg$A=0), brn(B≠0)]

GCD Example: Step 3

```
int gcd( int a, int b ) {
  assume 0 <= a;
  assume 0 <= b;
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```

- Solving
  $0<=A \wedge 0<=B \wedge A\neq 0 \wedge B\neq 0$
  yields e.g. A=1, B=1
- Test passes and yields path condition
  [asm(0<=A), asm(0<=B), brn(¬A=0), brn(B≠0), brn(¬A>B), brn(¬B–A≠0)]
- Recurs on
  […, brn(¬A>B), brn(B–A≠0)]
  and […, brn(A>B)]

GCD Example: Step 4

```
int gcd( int a, int b ) {
  assume 0 <= a;
  assume 0 <= b;
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```

- Solving
  $0<=A \land 0<=B \land A\neq 0 \land B\neq 0 \land \neg A>B \land B–A\neq 0$
  yields e.g. A=1, B=2
- Test passes and yields path condition
  [asm(0<=A), asm(0<=B), brn(¬A=0), brn(B≠0), brn(¬A>B), brn(B–A≠0), brn(¬A>B–A), brn(¬B–A–A≠0)]
- Recurs on […, brn(B–A–A≠0)]

GCD Example: Step 5

```
int gcd( int a, int b ) {
  assume 0 <= a;
  assume 0 <= b;
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```

- Solving
  $0<=A \wedge 0<=B \wedge A\neq 0 \wedge B\neq 0 \wedge \neg A>B \wedge B-A\neq 0 \wedge \neg A>B-A \wedge B-A-A\neq 0$
  yields e.g. A=1, B=3
- Test passes and yields path condition
  […, brn($\neg$(B–A–A–A≠0))]
- **Every additional recursion tests one more loop iteration**

GCD Example: Observations

```
int gcd( int a, int b ) {
  assume 0 <= a;
  assume 0 <= b;
  if( a == 0 )
    return b;
  while( b != 0 ) {
    if( a > b )
      a = a – b;
    else
      b = b – a;
  }
  return a;
}
```

- The simplified concolic testing procedure **might not terminate**
  - Not surprising; for loops, we generally cannot statically enumerate all paths
- Some branches might not be covered because the algorithm gets stuck exploring a loop before exploring the branch

Improved Procedure: Termination

```
procedure explore( Seq<Condition> prefix ) is
  values            := solve( prefix );
  if solution is available then
    path            := executeConcrete( values );
    pathCondition   := executeSymbolic( path );
    extension       := pathCondition[ |prefix| … ];
    foreach non-empty prefix p of extension do
      if p = p’ ○ [ brn(c) ] for some p’,c then
        explore( prefix ○ p’ ○ [ brn(¬c) ] );
      end
    end
  end
end
```

Bound number of loop iterations or use a time-out

Method Calls in Path Conditions

- Results of method calls may be used as conditions

```
Seq prefix( Seq s, int n ) {
  if ( s.length( ) < n )  return s;
  else                    …
}
```

- For simple methods, interprocedural symbolic execution may produce an expression for the result
  - For example, getters
- For non-trivial methods, the result **cannot be summarized automatically**
  - For example, iteration would typically require quantified expressions

Calls in Path Conditions: Encoding

- Idea: Replace expressions that cannot be encoded as constraints by their concrete values
  - Adapt definition of procedure executeSymbolic

```
Seq prefix( Seq s, int n ) {
  if ( s.length( ) < n )  return s;
  else                    …
}
```

Solver could choose for instance S=[e], N=1

- Execute, e.g., with S=[ ], N=0
- The observed path condition is ¬(S.length( ) < N)
- In the encoding, replace call by concrete result to obtain [brn(¬0<N)]
- Recurs on [brn(0<N)]

Calls in Path Conditions: Encoding (cont'd)

- Using a concrete method result makes sense only as long as the method arguments remain the same
  - Fix method arguments by adding assumptions

```
Seq prefix( Seq s, int n ) {
  if ( s.length( ) < n )  return s;
  else                    …
}
```

- Execute, e.g., with S=[ ], N=0
- The observed path condition is ¬(S.length( ) < N)
- Encode path condition as [asm(S=[ ]), brn(¬0<N)]
- Recurs on [asm(S=[ ]), brn(0<N)]

Solver could choose for instance S=[ ], N=1

Method Calls in Path Conditions (cont'd)

- Replacing calls by concrete values may lead to constraints on other inputs that can be solved

```
Seq prefix( Seq s, int n ) {
  if ( s.length( ) < n )  return s;
  else                    …
}
```

Path condition:
[asm(S=[ ]), brn(0<N)]

- However, when method results are constrained by values other than symbolic values, the resulting constraints do not provide useful conditions

```
void sort( Seq s ) {
  if ( s.length( ) == 0 )      return;
  else                         …
}
```

Path condition:
[asm(S=[ ]), brn(0=0)]

```
void copy( Seq from, Seq to ) {
  if ( from.length( ) != to.length( ) ) throw …;
  else …
}
```

Path condition:
[asm(F=[ ]), asm(T=[ ]), brn(0≠0)]

Reminder: Limitations: Constraint Solving

```
void roots( double a, double b, double c ) {
  double q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    double r = Math.sqrt( q );
    x1 = (-b + r) / (2 * a);
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    x1 = -b / (2 * a);
  } else {
    numRoots = 0;
  }
}
```

- Nonlinear constraint $b^2 - 4ac > 0 \wedge a \neq 0$ over reals is difficult to solve
- Without a solution, no test data will be provided
  - Neither for the other branches

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

Improved Procedure: Simplifying Constraints

```
procedure explore( Seq<Condition> prefix ) is
  values             := solve( prefix );
  if solution is available then
    path             := executeConcrete( values );
    pathCondition    := executeSymbolic( path );
    extension        := pathCondition[ |prefix| … ];
    foreach non-empty prefix p of extension do
      if p = p’ ○ [ brn(c) ] for some p’,c then
        explore( prefix ○ p’ ○ [ brn(¬c) ] );
      end
    end
  end
end
```

When solver returns “unknown”, simplify constraint by replacing symbolic variable by a concrete value and retry solving

Example: Step 1

```
void roots( double a,
            double b,
            double c ) {
  double q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    // ignore
  } else if( q == 0 ) {
    // ignore
  } else {
    // ignore
  }
}
```

- Solving path condition for [ ] yields arbitrary values, e.g., A=1, B=2, C=3
- Test yields path condition ($Q \equiv B \times B - 4 \times A \times C$) [brn($\neg$($Q>0 \wedge A \neq 0$)), brn($\neg Q=0$)]
- Recurs on [brn($\neg(Q>0 \wedge A \neq 0)$), brn($Q=0$)] and [brn($Q>0 \wedge A \neq 0$)]

Example: Step 2

```
void roots(   double a,
              double b,
              double c ) {
  double q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    // ignore
  } else if( q == 0 ) {
    // ignore
  } else {
    // ignore
  }
}
```

- Solving $\neg(Q>0 \wedge A\neq0) \wedge Q=0$ yields, e.g., A=0, B=0, C=0
- Test yields path condition [brn($\neg(Q>0 \wedge A\neq0)$), brn(Q=0)]

Example: Step 3

```
void roots( double a,
            double b,
            double c ) {
  double q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    // ignore
  } else if( q == 0 ) {
    // ignore
  } else {
    // ignore
  }
}
```

- Solving $Q>0 \wedge A\neq 0$ (for reals) yields unknown
- Replace one variable with a concrete value from the parent run via an assumption
  [ass(A=1), brn($Q>0 \wedge A\neq 0$)]
- Now solver yields e.g.
  A=1, B=0, C= –1/4

Simplifying Constraints with Concrete Values

- Replacing a variable with a concrete value
  - Might **reduce the complexity** (e.g., replacing A in A×B>0 goes from non-linear to linear arithmetic)
  - **Reduces the size** (number of free variables)
- Strategy does not necessarily succeed
  - New constraint might be unsatisfiable (e.g., A=0 on previous slide)
  - Solver might still yield “unknown”
- Use heuristics to identify set of variables to be replaced

Summary

- Automatic test case generation reduces the test effort tremendously
  - Achieve very high code coverage
  - Find test data
  - Actively search bugs
- Large effort remains
  - Writing oracles and assertions
  - Writing preconditions as assumptions
  - Writing factories to create test data (objects)
- Very active research area