So many different test objectives...

But how to create the tests?

Test Data Generation
Given a function and a location we want to reach, how do we derive inputs to the function that lead the control flow to the desired statement?

We are still looking at the problem of deriving input data that will lead execution to some particular point in the control flow that is interesting for testing reasons. Today, we will consider constraint based testing, which allows us to reason about the precise conditions under which a test goal is satisfied, and allows us to deduce test data satisfying the test goals.
Looking at the control flow graph of the `cgi_decode` example, we see that in order to reach node E, several conditions have to hold - conditions on variables that are altered during the program execution.

**Constraint-based Testing**

- **Constraint generation**
  Extract a constraint system from the program and a testing objective

- **Constraints on inputs**
  If inputs satisfy constraints, then testing objective will be satisfied

- **Constraint solving:**
  Solve the constraint system to generate test data

- **Static analysis aims at finding runtime errors** (e.g. division-by-zero, overflows, ...) at compile time

- **CBT aims at finding functional faults** (e.g. P returns 3 while 2 was expected)
- Model-checking tools explore paths of software models for proving properties
- CBT looks only for counter-examples

- Dynamic analysis approaches extract likely invariants
- CBT exploits symbolic reasoning to find counter-examples to given properties

Overview

The idea of constraint-based testing is to transform a program and a test goal for that program to a constraint system...
Overview

Constraint system: $(a > b \land c = 2) \lor (a > 5 \land b > a) \lor (a < b \land b < 5 \land a > 0)$

Constraint solver

Test data:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
</tr>
</tbody>
</table>

...and then use a constraint solver to automatically derive a solution to the constraint system - which is our test data.

Constraint Solving

What is a constraint?

- A constraint is a condition that a solution to an optimization problem must satisfy
- $x > 5, y < 10$
- $a > b \land b > 10$
- **Constraint satisfaction:** Finding value assignments to variables such that constraints are satisfied

Relevant questions: Does the constraint system (CS) have a solution? Can we generate a solution to CS? Can we generate the best solution to CS?
Relevant Questions

- Does the constraint system (CS) have a solution?  
  To decide whether the testing objective is reachable or not
- Can we generate a solution to CS?  
  Test data generation
- Can we generate the best solution to CS?  
  Test data generation that optimizes a cost function

Constraint solving

- Computational domain, constraint language results from the choice of programs and properties to be considered
- Booleans - Boolean formula (A&&B&&C)||(...) 
- Integers 
- Bounded Integers 
- Rationals 
- Reals 
- Floating-point numbers

Decidability and complexities

<table>
<thead>
<tr>
<th></th>
<th>Boolean Formula</th>
<th>Linear constraints</th>
<th>Polynomial constraints</th>
<th>Non-linear constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booleans</td>
<td>2-SAT in P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3-SAT is NP-complete</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0-1 programming is NP-complete</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>?</td>
<td></td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Bounded integers</td>
<td>-</td>
<td>NP-complete</td>
<td>NP-complete</td>
<td>NP-complete</td>
</tr>
<tr>
<td>Integers</td>
<td>-</td>
<td>Integer programming is NP complete</td>
<td>Undecidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>Rationals and reals</td>
<td>-</td>
<td>Linear programming in P</td>
<td>Nonlinear programming is NP-complete</td>
<td>Undecidable</td>
</tr>
</tbody>
</table>

The landscape of complexities for different domains (rows) and types of constraints (columns) looks scary - in practice, however, we can handle anything involving non-linear constraints quite well using heuristics.
Decision procedures (best practices)

<table>
<thead>
<tr>
<th>Boolean Formula</th>
<th>Linear constraints</th>
<th>Polynomial constraints</th>
<th>Non-linear constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Booleans</td>
<td>Davis &amp; Putnam (DPLL)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounded integers</td>
<td>Cooper's procedure for Oresburger algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integers</td>
<td>Constraint satisfaction</td>
<td>Constraint satisfaction</td>
<td>Constraint satisfaction</td>
</tr>
<tr>
<td>Rationals and reals</td>
<td>Simplex Fourier Elimination</td>
<td>Groebner basis (Buchberger alg)</td>
<td>Interval propagation</td>
</tr>
</tbody>
</table>

Constraint Satisfaction

- A constraint system involves a set of variables \( V \), a set of finite domains \( D \), and a set of constraints \( C \).
- A solution is an assignment of \( V \) to values in \( D \) that satisfies \( C \).
- A constraint system is unsatisfiable when it has no solutions.
- Constraint satisfaction involves 3 interleaved processes:
  - Constraint filtering
  - Constraint propagation
  - Variable labeling

Constraint filtering

- Given a single constraint, filter the domains of variables by removing inconsistent values.
- Depends on a level of consistency to be achieved:
  - Domain consistency - bound consistency - and many more.
- Example:
  - \( X \) in \{2,3,4,6,10\}, \( Y \) in \{1,2,3,4,6,8\}, \( Z \) in \{6\}, \( X*Y=Z \)
Domain Consistency

For each value in $D_x$, find a support in $D_z$ and $D_y$

{2,3,4,6,10}  
\[ X \]
\[ X \times Y = Z \]
\[ Y \]
\[ \{1,2,3,4,6,8\} \]

Bound Consistency

For each bound in $D_x$, find a support in $D_z$ and $D_y$

{2,3,4,6,10}  
\[ X \]
\[ X \times Y = Z \]
\[ Y \]
\[ \{1,2,3,4,6,8\} \]

Constraint Propagation

- Propagates prunings throughout the constraint system
- Implemented as a fixpoint algorithm:

\[
\text{Agenda := } C;\\
\text{while(!Agenda.isEmpty()) } \{
\text{ c := POP(Agenda); }\\
\text{ D' := narrow(c,D); }\\
\text{ if(D' != D) }\\
\text{ Agenda := Agenda } \cup \{c' \text{ in C | } \text{vars(c')} \cap \text{vars(c)}!=\emptyset \}\\
\text{ D := D' }\\
\text{ return D';}
\]
**Constraint Propagation**

\[ X, Y \text{ in } 0..10, X \times Y = 6, X+Y = 5 \]

**Constraint Propagation**

\[ X, Y \text{ in } 0..10, X \times Y = 6, X+Y = 5 \]

**Constraint Propagation**

\[ X, Y \text{ in } 0..10, X \times Y = 6, X+Y = 5 \]
X, Y in 0..10, \(X \times Y = 6\), \(X + Y = 5\)

Fixpoint = X, Y in 2..3 called a hyper-box (in an n-dimensional space)

X, Y in 0..10, \(X \times Y = 6\), \(X + Y = 5\)

When heuristics for selecting values and variables are complete, labeling is a decision procedure for constraint satisfaction. But, it is also the costly part of it (NP-complete) while constraint filtering and propagation are polynomial in the number of constraints (and values in domains). Routinely in applications, constraint satisfaction handles thousands of constraints and variables.
Selection Heuristics

- **Leftmost**
  Select the leftmost variable in the list

- **First-fail**
  Select the variable with the smallest domain

- **Most-constrained**
  Select the var that has the most constraints suspended on it

- And many more

Variable Labeling

- When heuristics for selecting values and variables are complete, labeling is a decision procedure for constraint satisfaction

- But, it is also the costly part of it (NP-complete) while constraint filtering and propagation are polynomial in the number of constraints (and values in domains)

- Routinely in applications, constraint satisfaction handles thousands of constraints and variables

Satisfiability Modulo Theory (SMT)

- To decide the satisfiability of formulas with respect to decidable background theories.
  \[ \Phi ::= A | \neg \Phi | \Phi \land \Phi \]

- Numerous applications including test data generation

- Used in PEX, through Z3 the SMT-solver of Microsoft
Satisfiability Modulo Theories (SMT)

- Example theories:
  - R: theory of rationals
    SR = \{≤, +, -, 0, 1\}
  - L: theory of lists
    SL = \{=, hd, tl, nil, cons\}
  - E: theory of equality
    SE: uninterpreted functions and predicate symbols

- Problem:
  Is \( x \leq y \land y \leq x + \text{hd}(\text{cons}(0, \text{nil})) \land P(h(x) - h(y)) \land \neg P(0) \) satisfiable in R.L.E?

---

Z3 is one of the most powerful SMT solvers currently available. In this video, the authors of Z3 briefly describe constraint solving and demonstrate how to use a constraint solver via API calls. The video is available online at http://channel9.msdn.com/posts/Peli/The-Z3-Constraint-Solver/
Path-Oriented Testing

Path-Oriented Generation

- Select one or several paths - Path selection
- Generate the path conditions - symbolic evaluation techniques
- Solve the path conditions to generate test data that activate the selected paths
- Useful for generating a test suite that covers a given test criterion (all statements, all branches, all defs, all uses, ...)

The first step in a path oriented test generation technique is to select which path we want to execute in our test case.

Path Selection

double P(short x, short y) {
    short w = abs(y);
    double z = 1.0;
    while(w != 0) {
        z = z * x;
        w = w - 1;
    }
    if(y < 0)
        z = 1.0 / z;
    return z;
}

double P(short x, short y) {
    short w = abs(y);
    double z = 1.0;
    while(w != 0) {
        z = z * x;
        w = w - 1;
    }
    if(y < 0)
        z = 1.0 / z;
    return z;
}
double P

double P(short x, short y) {
    short w = abs(y);
    double z = 1.0;

    while(w != 0) {
        z = z * x
        w = w - 1

        if(y < 0)
            z = 1.0 / z
    }

    return z;
}
double P

double P(short x, short y) {
    short w = abs(y);
    double z = 1.0;
    while(w != 0) {
        z = z * x
        w = w - 1
        if(y < 0)
            z = 1.0 / z
    }
    return z;
}
double P

double P(short x, short y) {
    short w = abs(y);
    double z = 1.0;
    while(w != 0) {
        z = z * x
        w = w - 1
        if(y < 0)
            z = 1.0 / z
    }
    return z;
}

Infeasible Paths

Determining whether an element is reachable or not is undecidable in the general case
Weyuker, 1979
Infeasible Paths

- Determining whether an element is reachable or not is undecidable in the general case
- Infeasible paths are ubiquitous in imperative programs
- Infeasible paths can be selected during the path selection process

We have encountered the infeasible path problem before - infeasible DU pairs are another example instance of the same problem.

The equivalent mutant problem can also be reduced to the infeasible path problem.
Symbolic Evaluation

- Three path-oriented techniques:
  1. Simple symbolic execution (forward and backward)
  2. Dynamic symbolic execution
  3. Global symbolic execution
- Exploits algebraic expressions over symbolic inputs to represent internal states of variables
- Application in software testing, compiler optimization, specialization, parallel computing, model-checking, program proving, etc.

Simple forward symbolic execution
A-B-C-B-C-B-D-F with X,Y

- A: \( w := \text{abs}(Y); z := 1.0; \)
- B: \( \text{abs}(Y) \neq 0 \)
- C: \( z := X; w := \text{abs}(Y) - 1; \)
- B: \( \text{abs}(Y) - 1 \neq 0 \)
- C: \( z := X \times X; w := \text{abs}(Y) - 2; \)
- B: \( \text{abs}(Y) - 2 = 0 \)
- D: \( Y \geq 0 \)
- F: return \((X \times X)\);

Given a path, we can derive constraints by symbolically executing the path either in a forward or backward fashion.

Symbolic State

- \(<\text{Path, State, Path Conditions}>\)
- Path = \( n_1 \cdot \ldots \cdot n_l \) is a path of a CFG
- State = \( \{ <v, \varphi> \}_{v \in \text{Var}(P)} \) where \( \varphi \) is an algebraic expression over \( x \)
- Path Condition = \( c_1 \land \ldots \land c_n \) where \( c_i \) is a condition over \( x \)
- \( x \) denotes symbolic variables associated to the inputs of program \( P \) and \( \text{Var}(P) \) denotes internal variables

During symbolic execution we maintain a symbolic execution of the execution. If we encounter a condition along the execution then the path conditions are updated, if we encounter assignments, then the state expressions are updated.
double P(short x, short y) {
    short w = abs(y);
    double z = 1.0;
    while(w != 0) {
        z = z * x;
        w = w - 1;
        if(y < 0)
            z = 1.0 / z;
    }
    return z; }

• A: w := abs(Y); z := 1.0;
• B: abs(Y) != 0
• C: z := X; w := abs(Y) - 1;
• B: abs(Y) - 1 != 0
• C: z := X * X; w := abs(Y) - 2;
• B: abs(Y) - 2 = 0
• D: Y >= 0
• F: return (X*X);

Simple forward symbolic execution
A-B-C-B-C-B-D-F with X,Y

Path conditions

A, {<z,⊥>, <w,⊥>}, true>
A-B-C-B-C-B-D-F,
{<z,X>, <w,abs(Y)-1>},
abs(Y) != 0>
A-B-C-B-C-B-D-F,
{<z,X^2>, <w,abs(Y)-2>},
(abs(Y)!=0) · (abs(Y)!=1) · (abs(Y) =2) · (Y>=0)>

Path, State, Path Conditions>
Computing Symbolic States

- \(<\text{Path,State,PC}>\) is computed by induction over each statement of Path
- When the path conditions are unsatisfiable then Path is infeasible
- Example: \(<\text{A-B-D-E-F,..., abs}(Y)=0 \&\& Y<0>\>
- Forward vs backward analysis:
  - Forward: Interesting when states are needed
  - Backward: Saves memory space as states remain implicit

Example

Classify triangle by the length of the sides

- Equilateral
- Isosceles
- Scalene

```java
int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }
    return 3; // scalene
}
```
The triangle example does not change the state along the execution, so symbolic execution reduces to collecting path conditions.

Backward analysis, the state is not explicit as on forward execution (which means this needs less memory). We simply maintain the set of path conditions, and whenever we encounter a state update, we apply this update to the path conditions collected so far.
Example

Classify triangle by the length of the sides

Equilateral Isosceles Scalene

```java
int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }
    return 3; // scalene
}
```
Implementing Symbolic Execution

- Transformation approach
  Transform to program that operates only on symbolic values
- Instrumentation approach
- Customized runtime approach

Transformation Approach

- Transform the program to another program that operates on symbolic values such that execution of the transformed program is equivalent to symbolic execution of the original program
**Instrumentation Approach**

- callback hooks are inserted in the program such that symbolic execution is done in background during normal execution of program
- easy to implement for C

**Customized runtime approach**

- Customize the runtime (e.g. JVM) to support symbolic execution
- Java PathFinder (NASA)
- Applicable to Java, .NET

**Limitations**

- Limited by the power of constraint solver
  No non-linear or complex constraints
- Does not scale when number of paths is large
- Source code or equivalent (Bytecode) is required for precise symbolic execution
- Infeasible path problem
Goal-oriented testing

- A three step process:
  1. Generate a constraint model of the whole program
  2. Choose a goal: point to be reached or property to be refuted
  3. Generate test data that respects the model and satisfies the goal
- Useful for generating test data that reaches a single testing objective (reach a statement or a branch, find a counter-example to a property, etc.)

A constraint model of imperative programs

- Viewing an assignment statement as a relation requires to rename the variables
- $i := i + 1 \rightarrow i_2 = i_1 + 1$
- Static Single Assignment (SSA) form or single assignment language

Destructive assignment to variables makes it necessary to rename variables for representation in a logic system.
SSA form

- Each use of a variable refers to a single definition

\[
\begin{align*}
x & := x + y; & x_1 & := x_0 + y_0; \\
y & := x - y; & y_1 & := x_1 - y_0; \\
x & := x - y; & x_2 & := x_1 - y_1;
\end{align*}
\]

\[\phi\] Functions

At join points in the control flow, we need to add phi functions that represent a choice of the values of the two branches. In an IF condition the phi function is simply added after the if and else branches, but for loops we need to add the phi function before loop condition.
From SSA to a Constraint System

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Domain constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int i</td>
<td>$i \in 0 \ldots 2^{32}-1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assignment, decision</th>
<th>Arithmetical constraints {=, &lt;, ...}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j_2 = j_1 \ast i$</td>
<td>$j_2 = j_1 \ast i$</td>
</tr>
<tr>
<td>$i == j$</td>
<td>$i = j$</td>
</tr>
<tr>
<td>$i &lt; j$</td>
<td>$i &lt; j$</td>
</tr>
</tbody>
</table>

From SSA to a Constraint System

Conditional (SSA)

if $D$ then $C_1$ else $C_2$
$v_3 := \phi(v_1, v_2)$

$(D \land C_1 \land v_3 = v_1) \lor (\neg D \land C_2 \land v_3 = v_2)$

Iteration (SSA)

$v_3 := \phi(v_1, v_2)$
while $D$ do $C$
$v_3 = v_3 \lor (D_1 \land C_1 \land \neg D_2 \land v_3 = v_2) \lor \ldots$

Conversion of the power function to SSA.

Assignment and comparison have equal operations due to SSA.

For conditionals and loops we need to get rid of the phi functions when converting to a constraint system. For a conditional this gives us a disjunction of the two possible outcomes of the condition (note that $v_3$ is assigned within this disjunction). Loops need to be unrolled.
To reach a certain point in the control flow the control dependencies need to be satisfied. For the constraint system, we add all the control dependent conditions.
double P(short x, short y) {
    short w = abs(y);
    double z = 1.0;
    while(w != 0) {
        w = w - 1;
        double z = 1.0; // This line is repeated
        double P(short x, short y) {
            return z;
        }
    }
    return z;
}

w1 := 0 ∧ w2 := 0

w1 := abs(y) ∧
z1 := 1.0 ∧
z2 := z1 ∗ x ∧
w3 := w1 - 1 ∧
z3 := z2 ∗ x ∧
w4 := w2 - 1 ∧
((w4 = w2 ∧ z4 = z2 ∧ w2 = 0) ∨
(w4 = w3 ∧ z4 = z3 ∧ w2 = 0)) ∧
((w1 = 0 ∧ z5 = z1 ∧ w5 = w1) ∨
(w1 = 0 ∧ z5 = z4 ∧ w5 = w4)) ∧
z6 := 1.0 / z3 ∧
((y >= 0 ∧ z7 = z5) ∨ (y < 0 ∧ z7 =
z5)) ∧
w1 := 0 ∧ w2 := 0
Example

Classify triangle by the length of the sides

Equilateral  Isosceles  Scalene

```cpp
int triangle(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (!(a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }
    return 3;  // scalene
}
```
int triangle(int a, int b, int c) {
    int r = 3; // scalene
    if (a <= 0 || b <= 0 || c <= 0) {
        r = 4; // invalid
    }
    if (!((a + b > c && a + c > b && b + c > a))) {
        r = 4; // invalid
    }
    if (a == b && b == c) {
        r = 1; // equilateral
    } else if (a == b || b == c || a == c) {
        r = 2; // isosceles
    }
    return r;
}

int triangle(int a, int b, int c) {
    int r0 = 3; // scalene
    if (a <= 0 || b <= 0 || c <= 0) {
        r1 = 4; // invalid
    } else {
        if (!((a + b > c && a + c > b && b + c > a))) {
            r2 = 4; // invalid
        } else {
            if (a == b && b == c) {
                r3 = 1; // equilateral
            } else {
                if (a == b || b == c || a == c) {
                    r4 = 2; // isosceles
                }
                r5 = f(r4, r0);
            }
            r6 = f(r5, r3);
        }
        r7 = f(r6, r2);
    }
    r8 = f(r7, r1);
    return r8;
}

int triangle(int a, int b, int c) {
    int r0 = 3; // scalene
    if (a <= 0 || b <= 0 || c <= 0) {
        r1 = 4; // invalid
    } else {
        if (!((a + b > c && a + c > b && b + c > a))) {
            r2 = 4; // invalid
        } else {
            if (a == b && b == c) {
                r3 = 1; // equilateral
            } else {
                if (a == b || b == c || a == c) {
                    r4 = 2; // isosceles
                }
                r5 = f(r4, r0);
            }
            r6 = f(r5, r3);
        }
        r7 = f(r6, r2);
    }
    r8 = f(r7, r1);
    return r8;
}
int triangle(int a, int b, int c) {
    int r0 = 3; // scalene
    if (a <= 0 || b <= 0 || c <= 0) {
        r1 = 4; // invalid
    } else {
        if (!(a + b > c && a + c > b && b + c > a)) {
            r2 = 4; // invalid
        } else {
            if (a == b && b == c) {
                r3 = 1; // equilateral
            } else {
                if (a == b || b == c || a == c) {
                    r4 = 2; // isosceles
                } else {
                    r5 = ϕ(r4, r0);
                }
                r6 = ϕ(r5, r3);
            }
            r7 = ϕ(r6, r2);
        }
        r8 = ϕ(r7, r1);
    }
    return r8;
}