Software Testing

Software Testing is a cognitively complex task
- Requires code or spec. understanding
- Program's input space is usually unbounded
- Complex software yields to complex bugs
- Oracles have to be defined

Not easily amenable to automation
- Automatic test data generation is undecidable in the general case
- Exploring the input space yields to combinatorial explosion
- Automated oracles are usually not available

Thesis: Constraint reasoning can help as it has demonstrated high potential to address hard combinatorial problems with great flexibility

Automatic test data generation (1)

- Highly combinatorial problems
- Unknown number of iterations in loops, selection of non-feasible paths
- Pointers, dynamic allocation/deallocation, dynamically allocated structures
- Aliasing problems, anonymous locations
- Function calls
- Masterizing the complexity of calls is difficult
- Floating-point numbers
- Floating-point operators cannot be correctly handled with operators over rationals/reals

Automatic test data generation (2)

Given a program and a testing objective, finding a test data meeting the objective is like solving a Sudoku-problem:

Constraint-Based Testing (CBT) (1)

Constraint-Based Testing (CBT) is the process of generating test cases against a testing objective by using constraint solving techniques


Mainly used in the context of structural and functional testing

Constraint-Based Testing (CBT) (2)

CBT is currently developed within several international research labs:
- IBM Haifa Research Lab
- CEA - LIST
- Microsoft Redmond
- MIT
- INRIA - Lande
- INRIA - Lande (MUTT, Pex projects)
- INRIA - Lande (MulSaw project)
- Bell Labs, ...

CBT tools (academic): InKa, GATEL, PathCrawler, Autofocus, DART/CUTE, ...

Commercial tools: IBM X-GEN, LTG

How CBT relates to other bug-finding techniques?

- **Static analysis** aims at finding runtime errors (e.g., division-by-zero, overflows, …) at compile-time while CBT aims at finding functional faults (e.g., P returns 3 while 2 was expected) at runtime.
- **Model-checking tools** explores paths of software models for proving properties while CBT looks only for counter-examples.
- **Dynamic analysis approaches** extract likely invariants while CBT exploits symbolic reasoning to find counter-examples to given properties.

How CBT relates to other test data generation techniques?

- Other test cases generation techniques include:
  - **Dynamic methods** (program executions, Korel’s method, binary search, …)
  - **Evolutionary techniques** (Genetic Algorithms, search-based methods, …)

By combining symbolic reasoning and numerical inference, CBT exploits program structure and data to refine the test case generation process and differs so from «blind» techniques that attempt to reach the testing objective by trials.

Plan

- Introduction
- Foundations of CBT
- Advanced CBT techniques and tools
- Conclusions

CBT is a two-stage process

- **Constraint generation**: Extract a constraint system from the program and a testing objective
- **Constraint solving**: Solve the constraint system to generate test data

Constraint generation

- Path-oriented test data generation
- Goal-oriented test data generation

Path-oriented test data 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 testing criterion (all-statements, all-branches, all-defs, all-uses, all-k-paths…)

Main CBT tools: **ATGen** (Meudec 2001), **PathCrawler** (Williams 2005), **DART/CUTE** (Godefroid/Sen 2005)
Path selection on an example

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);
}

Problem of non-feasible paths

\(a-b-d-e-f\) is non-feasible!

\(\text{Problem of non-feasible paths}\) (Weyuker 1979)

Determining whether an element is reachable or not is undecidable in the general case

Sketch of proof: Reduce to the Turing's halting machine problem

- Non-feasible paths are ubiquitous in imperative programs
- Non-feasible paths can be selected during the path selection process

Simple forward symbolic execution

Ex: \(a-b-(c-b)^2-d-f\) with \(x, y\)

\(\begin{align*}
  a: & \quad w := \text{abs}(y); \quad z := 1.0; \\
  b: & \quad \text{abs}(y) \neq 0; \\
  c: & \quad z := X; \quad w := \text{abs}(y) - 1; \\
  b: & \quad \text{abs}(y) - 1 \neq 0; \\
  c: & \quad z := X \times X; \quad w := \text{abs}(y) - 2; \\
  b: & \quad \text{abs}(y) - 2 = 0; \\
  d: & \quad X \geq 0 \\
  f: & \quad \text{return}(X \times X)
\end{align*}\)

Symbolic Evaluation \([\text{King 76, Clarke 76}]\)

 três path-oriented techniques

- Simple symbolic execution (forward and backward)
- Dynamic symbolic evaluation
- Global symbolic evaluation

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 and so on.

Symbolic state

\(<\text{Path}, \text{State}, \text{Path Conditions}>\)

\(\begin{align*}
  \text{Path} &= n_1 \ldots n_m \text{ is a path of a CFG} \\
  \text{State} &= \{(v, \phi)\}_{v \in \text{var}} \quad \text{where } \phi \text{ is an algebraic expr. over } x \\
  \text{Path Cond.} &= c_1 \ldots c_n \quad \text{where } c_i \text{ is a condition over } x
\end{align*}\)

\(x\) denotes symbolic variables associated to the inputs of program \(P\) and \(\text{Var}(P)\) denotes internal variables
Computing symbolic states

- \langle \text{Path, State, PC} \rangle \text{ is computed by induction over each statement of Path}
- When the Path conditions are unsatisfiable, then Path is non-feasible and reciprocally
- Forward \rightarrow interesting when States are needed
- Backward \rightarrow saves memory space as states remain implicit

Backward analysis

- \langle a-b-(c-b)^2-d-f \rangle \text{ with } X,Y
- a: Y \geq 0
- b: Y \geq 0, w = 0
- c: Y \geq 0, w-1 = 0
- d: Y \geq 0, w-2 = 0, w-1 \neq 0
- c: Y \geq 0, w-2 = 0
- b: Y \geq 0, w-1 \neq 0, w \neq 0
- a: Y \geq 0, abs(T) \neq 0, abs(Y) = 0
- b: Y \geq 0, abs(T) \neq 0
- c: Y \geq 0, abs(T) = 0
- d: Y \geq 0, abs(T) = 0

Dynamic symbolic evaluation

- Symbolic execution of a concrete execution (also called concolic execution)
- By using input values, feasible paths only are (automatically) selected
- Implemented by instrumenting each statement of P

Global symbolic evaluation

- Execution tree that represents symbolically all the paths
- Requires to build a loop analysis:
  ```
  while ( w \neq 0 )
  {
      z = z*x ;
      w = w-1 ;
  }
  ```
- Exponential in time and memory space

An example
Problems for symbolic evaluation techniques
- Number of iterations in loops must be selected prior to any symbolic execution
- Arrays

```c
int P(int i) {

A[47], A[29] et A[i] are seen as a single variable A.
Note that interesting solutions exist for this problem (Coen-Porisini & De Paoli 1993)
```
- Symbolic execution constrains the shape of dynamically allocated objects

```c
int P(struct cell *t) {
    if( t == t->next ) {
        constrains t to:  

key  next
```

Problems for path-oriented test data generation
- Non-feasible paths and symbolic execution problems (as discussed earlier)
- Handling loops (manual vs automatic path selection)

```c
i = 0; b. while( i < 100 ) {
    c. if( i == 50 )
        d. ...
    e. i++ ;}
```

→ Reaching d implies the selection of a single path among 2^{100} possible paths

Goal-oriented test data generation

A three-step process:
- Generate a constraint model of the whole program
- Choose a goal: point to be reached or property to be refuted
- Generate a test data that respects the model and satisfies the goal

Useful for generating test data that reach a single testing objective (reach a statement or a branch, find a counter-example to a property, etc.)

Main tools: InKa (Gotlieb 2000), GATEL (Marre 2000)

Constraint generation
- Path-oriented test data generation
- Goal-oriented test data generation

Constraint model of imperative programs

Viewing an assignment statement as a relation requires to rename the variables

```c
i := i + 1  ----> i_2 := i_1 + 1
```

→ Static Single Assignment (SSA) form (Cytron 1991) or single assignment language

SSA form

Each use of a variable refers to a single definition

- At the junction nodes
  - i := i + ...  ----> i_2 := i_1 + ...
  - i_1 := i_2 + ...  ----> i_3 := i_1 + ...

- φ functions
  - i_1 := φ( i_1, i_2 )
  - i_3 := i_1 + ...
### Testing objective (1)

- In structural testing, point to be reached

\[ C_2 = \neg \text{Cond4} \land \text{Cond2} \land \text{Cond1} \]

### Testing objective (2)

- In functional testing, property to be refuted (e.g. a post-condition of the program)

Ex: JML specification of method debit() from the SUN’s Wallet Java Card program

```java
//@ requires amount >= 0;
ensures balance == \old(balance) – amount;
signals (DebitException)   amount > balance;
@*/
public void debit(int amount) {...}
```

- Constraint solving over disjunctions of constraints is required
  - (conditionals, switch, iterations)

### Problems for goal-oriented test data generation

- Conditional aliasing problems

  How to reach branch 5-6 ?

  1. if( C )
  2. \( p = &i \)
  3. \( i = 10 \)
  4. \(*p = 0 \)
  5. if( i < 5 )
  6. ...
Constraint solving for test data generation (1)

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 for test data generation (2)

- (computational domain, constraint language) resulting from the choice of programs and properties to be considered
  
  **Computational domain:**
  - Booleans
  - Integers
  - Bounded integers
  - Rationals
  - Reals
  - Floating-point numbers

  **Constraint language (quantifier-free formula):**
  - Boolean formula
  - Combination of linear constraints
  - Polynomial constraints
  - Non-linear constraints

Decidability and complexities

<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>2-SAT in P</td>
<td>0:1 Programming is NP_complete</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>3-SAT is NP_complete</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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>Davis &amp; Putnam (DPLL)</td>
<td>Boolean formula (A ∧ B ∧ ¬C) ∨ (¬A ∧ B ∧ C)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounded integers</td>
<td>Constraint satisfaction</td>
<td>Constraint satisfaction</td>
<td>Constraint satisfaction</td>
</tr>
<tr>
<td>Rationals and reals</td>
<td>Simplex</td>
<td>Fourier Elimination</td>
<td>Gröbner basis (Buchberger alg.)</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 (1)

- Given a single constraint, filter the domains of variable by removing inconsistent values
- Depends on a level of consistency to be achieved
  - Domain consistency
  - Bound consistency
  - and many more, not discussed here!

Ex: X in {2, 3, 4, 6, 10}, Y in {1, 2, 3, 4, 6, 8}, Z in {6}.

X * Y = Z
Constraint filtering (2)

Domain consistency (for each value in $D_X$, find a support in $D_Z$ and $D_Y$)

$\{2,3,6,10\} \times \{1,2,3,4,6,8\} \rightarrow \{1,2,3,4,6,8\}$

→ ideal but costly to compute when domains are large

Constraint filtering (3)

Bound consistency (for each bound in $D_X$, find a support in $D_Z$ and $D_Y$)

$\{2,3,4,6,10\} \times \{1,2,3,4,6,8\} \rightarrow \{1,2,3,4,6,8\}$

→ Does not depend on the size of domains!

Constraint Propagation (1)

• Propagates prunings throughout the constraint system

• Implemented as a fixpoint algorithm:

```plaintext
Agenda := C ;
while( Agenda ≠ ∅ )
    c := POP(Agenda) ;
    D' := narrow(c,D) ;          % Filtering
    if( D' ≠ D )
        Agenda := Agenda ∪ {c' ∈ C / vars(c') ∩ vars(c) ≠ ∅}
    D := D' ;
return(D') ;
```

Ex: $X,Y$ in $0..10$, $X \cdot Y = 6$, $X + Y=5$.

Constraint Propagation (2)

Ex: $X,Y$ in $0..10$, $X \cdot Y = 6$, $X + Y=5$.

Constraint Propagation (3)

Ex: $X,Y$ in $0..10$, $X \cdot Y = 6$, $X + Y=5$.

Constraint Propagation (4)

Ex: $X,Y$ in $0..10$, $X \cdot Y = 6$, $X + Y=5$. 
Constraint propagation (5)

Ex: X,Y in 0..10, X*Y = 6, X + Y = 5.

Constraint propagation (6)

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

Ex: X,Y in 0..10, X*Y = 6, X + Y = 5.

Variable labeling (1)

- Select a value v from the domain of X and propagates X = v

Ex: X,Y in 0..10, X*Y = 6, X + Y = 5.

Solutions:
(X=2, Y=3)
(X=3, Y=2)

Variable labeling (2)

- Heuristics for selecting variables and values
  - 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, not discussed here!

- 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

Plan

- Introduction
- Foundations of CBT
  - Advanced CBT techniques and tools
  - Conclusions

On-the-fly selection of feasible paths: The PathCrawler method

- Addresses the problem of non-feasible paths in path-oriented test data generator for C programs
- A randomized algo. based on dynamic symbolic execution and constraint satisfaction over finite domains (a proprietary constraint library in Eclipse Prolog)
- An important emerging idea in CBT: papers in PLDI (Godefroid et al. 2005), ESEC/FSE (Sen et al. 2005), POPL (Godefroid 2007), ...
The idea

Generate a random input and try to solve the CS where the last constraint is refuted.

Try to solve the remaining CS if unsatisfiable.

Backtrack and try to solve the remaining CS if unsatisfiable.

If unsatisfiable then path is non-feasible and problem comes from the last constraint.

Else coverage of feasible paths is improved.

An example (1)

f( int i )
{ j = 2;
  if( i <= 16 )
    j = j * i;
  if( j > 8)
    j = 0;
  return j;
}

An example (2)

Random input generation:
(i = 15448)  
→ Path 1-3-5

An example (3)

Try to solve:

j_1 = 2
i > 16
j_1 > 8

Unsatisfiable, therefore Path 1-3-4 is non-feasible.

An example (4)

Backtrack and try to solve:

j_2 = j_1 * i
i <= 16
→ (i = 2) → Path 1-2-3-5

An example (5)

Bactrack and try to solve:

j_2 = j_1
j_2 = j_3
→ (i = 10) → Path 1-2-3-4-5

All-paths covered with three test data (i = 15448, i = 2, i = 10)
The PathCrawler method: discussion

- Requires to bound the number of iterations in loops
  → suitable for automatic test data generation for the All-k-paths criterion
- Performance of the method depends on the first initial random input
- Numerous extensions to handle pointers as input parameters, logical decisions, function calls, bit-to-bit operations

InKa: a goal-oriented test data generator based on constraint combinators

- Automatic Test Data Generation using Constraint Solving Techniques
  A. Gotlieb, B. Botella, M. Rueher
  ACM International Symposium on Software Testing and Analysis (ISSTA'98), Clearwater Beach, FL, USA, March 1998
- Addresses the problem of loops, non-feasible paths, floating-point numbers and pointer aliasing in goal-oriented test data generator for C programs
- A deterministic algo. based on SSA, Constraint combinators over finite domains
  (built over SICStus clp(fd) constraint library)
- More than 10 years of Research, three prototype tools, fifteen published papers

Conditional: the combinator `ite`

\[
\text{ite}(V, C\text{\text{THEN}}, C\text{\text{ELSE}}, \text{MIN}, M\text{\text{OUT}}) \leftarrow
\begin{array}{ll}
V=1 & \rightarrow \text{C\text{\text{THEN}}} \land M\text{\text{OUT}} = \text{M\text{\text{THEN}}} \\
V=0 & \rightarrow \text{C\text{\text{ELSE}}} \land M\text{\text{OUT}} = \text{M\text{\text{ELSE}}} \\
\neg (V=1 \land \text{C\text{\text{THEN}}} \land M\text{\text{OUT}} = \text{M\text{\text{THEN}}} ) & \rightarrow \neg (V=0 \land \text{C\text{\text{ELSE}}} \land M\text{\text{OUT}} = \text{M\text{\text{ELSE}}} )
\end{array}
\]

\[
\text{M\text{\text{OUT}}} := \text{Proj(OUT, M\text{\text{THEN}} \cup M\text{\text{ELSE}})} \\
\text{MIN} := \text{Proj(IN, M\text{\text{THEN}} \cup M\text{\text{ELSE}})}
\]

Iteration: the combinator `w`

\[
w(V, C\text{\text{BODY}}, M\text{\text{OUT}}) \leftarrow
\begin{array}{ll}
V=1 & \rightarrow V=1 \land C\text{\text{BODY}} \\
V=0 & \rightarrow V=0 \land M\text{\text{OUT}} = \text{MIN} \\
\neg (V=1 \land C\text{\text{BODY}} ) & \rightarrow V=0 \land M\text{\text{OUT}} = \text{MIN} \\
\neg (V=0 \land C\text{\text{BODY}} ) & \rightarrow V=1 \land M\text{\text{OUT}} = \text{MIN}
\end{array}
\]

Where `V` stands for the widening operator

Example (1)

\[
f(\text{int} \ i) \\
| i = 2; \ \\
| \text{if}(i < 16) \\
| \quad j = j * i; \\
| \text{if}(j > 8) \\
| \quad j = 0; \\
| \text{return} \ j;
\]

Example (2)

\[
\begin{array}{l}
i = 0 \ldots 2^{n}-1, \ \\
J_i = 2, \ \\
\text{for}(i < 16), \ J_i = J_{i-1} \land J_i = J_{i+1}; \ J_i = 0, \\
\text{if}(J_i > 8) \ \\
\quad J_i = 0; \ J_i = 0 \land J_i = J_{i+1}; \ \\
\text{return} \ J_i
\end{array}
\]
Example (3)

```c
f(int i)
{
    j = 2;
    if (i ≤ 16)
        j = j * i;
    if (j > 8)
        j = 0;
    return j;
}
```

Test datum: \( i = 10 \)

\( i \leq 16 \) \( \Rightarrow \) \( j = 2 \times 10 = 20 \)
\( j > 8 \) \( \Rightarrow \) \( j = 0 \)
\( j = 0 \)

\( i = 0 \) \( \Rightarrow \) 
\( j = 2 \times 0 = 0 \)
\( j = 0 \)

\( j = 0 \)

\( \text{RET} = j \)

The InKa method: discussion

- Handles nicely conditionals and loops
- First introduction of constraint satisfaction techniques in structural software testing
- Suitable for generating test data in front of a single testing objective (to complete an existing test suite)
- Can be stucked on infinite loops (time-out required)

Extensions (in the Lande team-project)

- Handling conditional pointer aliasing problems
  - Constraint-based test data generation for programs with stack-directed C pointer programs
    Gotlieb Denmat Botella COMPSAC’05, ASE’05, IST’07
- Handling floating-point numbers in constraint-based structural test data generation
  - Botella Gotlieb Michel STVR 2006
  - Floating-point computations cannot be handled by constraints over the rationals or reals due to distinct semantics (e.g. + over the floats cannot be correctly handled with + over the reals)
  - Tool: FPSE (Floating-Point Symbolic Execution)
- Efficient handling of conditional and iterations based on Abstract Interpretation techniques
  - Denmat Gotlieb Ducassé CP 2007

Plan

- Introduction
- Foundations of CBT
- Advanced CBT techniques and tools
  - Conclusions

CBT

- Emerging concept in automatic test data generation techniques based
- Based on constraint generation and constraint solving
  - (Linear Programming, constraint satisfaction)
- Exploited in structural and functional testing of imperative programs, model-based testing and hardware verification
- Mature tools (academic and industrial) already exist

Open questions

- How to improve constraint generation
  - to facilitate the constraint solving step
  - to handle dynamically allocated data structures
  - to handle efficiently function calls (modular analysis) and virtual calls in OO Programming
- How to improve constraint satisfaction
  - to handle loops
  - to handle efficiently floating-point computations
  - to deal with disjunctive constraint systems