



# Constraint-Based Testing

Arnaud Gottlieb  
Team-Project LANDE  
Summer School TAROT – 2 July 2007

## // 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

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

## Automatic test data generation (1)

Main technical problems:

- ✓ Highly combinatorial problems

Ex:  $f(int\ x_1, int\ x_2, int\ x_3) \rightarrow 2^{96}$  possible test cases

- ✓ Unknown number of iterations in loops, selection of non-feasible paths
- ✓ Pointers, dynamic allocation/deallocation, dynamically allocated structures  
← Aliasing problems, anonymous locations
- ✓ function calls ← mastering the complexity of calls is difficult
- ✓ Floating-point numbers ← floating-point operators cannot be correctly handled with 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:

```

bool UseDigit(int board[81], int digit, int i, int j) {
int x, y;
if(i < 0 || j < 0 || i > 8 || j > 8 || digit < 1 || digit > 9)
return false;
if(board[*9+] != 0) return false;
for(x = 0; x < 9; x++) {
if(board[*9+] == digit || board[*9+x] == digit)
return false; }
for(y = ((i / 3) * 3); y < ((i / 3) * 3) + 3; y++) {
for(x = ((j / 3) * 3); x < ((j / 3) * 3) + 3; x++) {
if(board[*9+x] == digit)
Value of board ?
...

```

3	6	9	7	2	5	1	4	8
8	7	2	4	1	9	5	6	3
5	1	4	8	6	3	7	2	9
9	3	1	2	5	4	8	7	6
7	8	5	6	3	1	2	9	4
4	2	6	9	8	7	3	1	5
6	9	3	1	7	2	4	5	8
2	4	7	5	9	8	6	3	1
1	5	8	3	4	6	9	2	7

## // 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**

Introduced 15 years ago by Offutt and DeMillo in  
(*Constraint-based automatic test data generation* IEEE Trans. Soft. Eng. 1991)

Mainly used in the context of *structural and functional testing*

*By now, not yet recognized as a main ST technique, but lots of current research works !*

## // Constraint-Based Testing (CBT) (2)

CBT is currently developed within several international research labs:

<ul style="list-style-type: none"> <li>IBM Haifa Research Lab</li> <li>CEA - LIST</li> <li>Microsoft Redmond</li> <li>MIT</li> <li>INRIA - Lande</li> <li>Bell Labs, ...</li> </ul>	<ul style="list-style-type: none"> <li>(MUTT, Pex projects)</li> <li>(MulSaw project)</li> <li>(CAT project)</li> </ul>
---	---

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

Commercial tools : IBM X-GEN, LTG

Startup : LEIRIOS (2003)

## 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:

- **Random Testing** (Uniform, Adaptive RT, Statistical structural/functional Testing...)

- **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

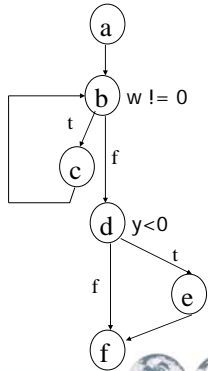
- Select one or several paths → Path selection
- Generate the path conditions → Symbolic evaluation techniques
- Solve the path conditions to generate test data that activate the selected paths

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);
}
```



### Path selection on an example

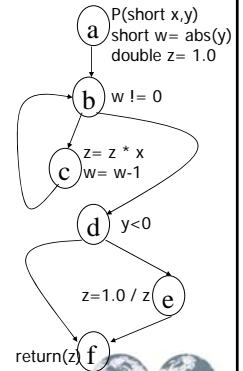
Statement coverage:  
a-b-c-b-d-e-f

Branch coverage:  
a-b-c-b-d-e-f  
a-b-d-f

All-2-paths (at most 2 times in loops):

~~a-b-d-f~~  
~~a-b-d-e-f~~  
...  
a-b-(c-b)<sup>2</sup>-d-e-f

All-paths:  
Impossible



### Problem of non-feasible paths

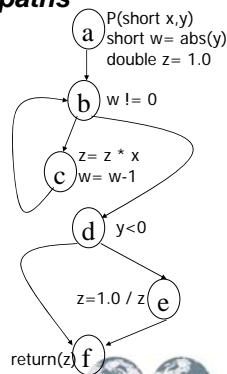
a-b-d-e-f is non-feasible!

(Weyuker 1979)  
Determining whether a 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



### Symbolic Evaluation [King 76, Clarke 76]

- > Three 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.

### Simple forward symbolic execution

Ex.: a-b-(c-b)<sup>2</sup>-d-f with x,y

a:  $w := \text{abs}(Y); z := 1.0;$

b:  $\text{abs}(Y) \neq 0;$

c:  $z := X; w := \text{abs}(Y) - 1;$

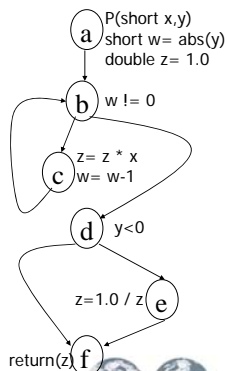
b:  $\text{abs}(Y) - 1 \neq 0;$

c:  $z := X * X; w := \text{abs}(Y) - 2;$

b:  $\text{abs}(Y) - 2 = 0;$

d:  $Y \geq 0$

f:  $\text{return}(X * X)$



### Symbolic state

<Path, State, Path Conditions>

Path =  $n_1 \dots n_j$  is a path of a CFG  
 State =  $\{ \langle v, \phi \rangle \}_{v \in \text{Var}(P)}$  where  $\phi$  is an algebraic expr. over  $x$   
 Path Cond. =  $c_1 \wedge \dots \wedge c_n$  where  $c_k$  is a condition over  $x$

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

**<Path, State, Path Cond> : examples**

```

    graph TD
      a((a)) --> b((b))
      b --> c((c))
      c --> b
      c --> d((d))
      d --> e((e))
      e --> f((f))
      f --> a
  
```

$P(\text{short } x,y)$   
 $\text{short } w = \text{abs}(y)$   
 $\text{double } z = 1.0$

$\langle a, \langle z, 1 \rangle, \langle w, 1 \rangle \rangle, \text{true}$

$\langle a-b-c-b,$   
 $\langle z, X \rangle, \langle w, \text{abs}(Y)-1 \rangle, \text{abs}(Y) \neq 0 \rangle$

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

Node a:  $P(\text{short } x,y)$   
 $\text{short } w = \text{abs}(y)$   
 $\text{double } z = 1.0$   
 Node b:  $w \neq 0$   
 Node c:  $z = z * x$   
 $w = w - 1$   
 Node d:  $y < 0$   
 Node e:  $z = 1.0 / z$   
 Node f:  $\text{return}(z)$

**Computing symbolic states**

- >  $\langle \text{Path}, \text{State}, \text{PC} \rangle$  is computed by induction over each statement of Path
- > When the Path conditions are unsatisfiable then Path is non-feasible and reciprocally

~~ex:  $\langle a-b-d-e-f, \dots \rangle, \text{abs}(Y) = 0 \wedge Y < 0 \rangle$~~

- > Forward vs backward analysis:
  - Forward  $\rightarrow$  interesting when States are needed
  - Backward  $\rightarrow$  saves memory space as states remain implicit

**Backward analysis**

Ex:  $a-b-(c-b)^2-d-f$  with  $X, Y$

```

    graph TD
      a((a)) --> b((b))
      b --> c((c))
      c --> b
      c --> d((d))
      d --> e((e))
      e --> f((f))
      f --> a
  
```

$f, d: Y \geq 0$

$b: Y \geq 0, w = 0$

$c: Y \geq 0, w-1 = 0$

$b: Y \geq 0, w-1 = 0, w \neq 0$

$c: Y \geq 0, w-2 = 0, w-1 \neq 0$

$b: Y \geq 0, w-2 = 0, w-1 \neq 0, w \neq 0$

$a: Y \geq 0, \text{abs}(Y)-2 = 0,$   
 $\text{abs}(Y)-1 \neq 0, \text{abs}(Y) \neq 0$

Node a:  $P(\text{short } x,y)$   
 $\text{short } w = \text{abs}(y)$   
 $\text{double } z = 1.0$   
 Node b:  $w \neq 0$   
 Node c:  $z = z * x$   
 $w = w - 1$   
 Node d:  $y < 0$   
 Node e:  $z = 1.0 / z$   
 Node f:  $\text{return}(z)$

**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 != 0 )
{
  z = z*x ;
  w = w-1 ;
}
  
```

$z_1 = 1.0$   
 $w_1 = \text{abs}(Y)$

$z_{k+1} = z_k * X$   
 $w_{k+1} = w_k - 1$

- > Exponential in time and memory space

**An example**

```

    graph TD
      Root["<z1, 1.0>, <w1, abs(Y)>"]
      Root --> B["abs(Y) = 0"]
      Root --> C["abs(Y) != 0"]
      B --> B1["Y < 0"]
      B --> B2["Y >= 0"]
      B1 --> B1_fail["fail"]
      B2 --> B2_state["<z, 1.0>"]
      C --> C1["<z_{k+1}, z_k * X>, <w_{k+1}, w_k - 1>"]
      C1 --> C2["w_{k+1} = 0"]
      C2 --> C2_1["Y < 0"]
      C2 --> C2_2["Y >= 0"]
      C2_1 --> C2_1_state["<z, 1/z_{k+1}>"]
      C2_2 --> C2_2_state["<z, z_{k+1}>"]
      C --> C_graph["Control flow graph (a-f)"]
      C_graph --> C_graph_return["return(z) f"]
  
```

$\langle z_1, 1.0 \rangle,$   
 $\langle w_1, \text{abs}(Y) \rangle$

$\text{abs}(Y) = 0$

$\text{abs}(Y) \neq 0$

$\langle z_{k+1}, z_k * X \rangle,$   
 $\langle w_{k+1}, w_k - 1 \rangle$

$w_{k+1} = 0$

$Y < 0$

$Y \geq 0$

$\langle z, 1/z_{k+1} \rangle$

$\langle z, z_{k+1} \rangle$

Node a:  $P(\text{short } x,y)$   
 $\text{short } w = \text{abs}(y)$   
 $\text{double } z = 1.0$   
 Node b:  $w \neq 0$   
 Node c:  $z = z * x$   
 $w = w - 1$   
 Node d:  $y < 0$   
 Node e:  $z = 1.0 / z$   
 Node f:  $\text{return}(z)$

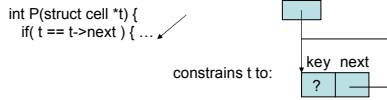
### Problems for symbolic evaluation techniques

- Number of iterations in loops must be selected prior to any symbolic execution
- Arrays
 

```
int P(int i) {
    A[47] = 18;
    A[29] = 46;
    if ( A[i] < 6 {
```

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



### Problems for path-oriented test data generation

- Non-feasible paths and symbolic execution problems (as discussed earlier)
- Handling loops ( manual vs automatic path selection)

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

→ Reaching d implies the selection of a single path among 2<sup>100</sup> possible paths

## Constraint generation

- Path-oriented test data generation

- Goal-oriented test data generation

## 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* (Gottlieb 2000), *GATEL* (Marre 2000)

## A constraint model of imperative programs

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

$$i := i + 1 \longrightarrow i_2 = i_1 + 1$$

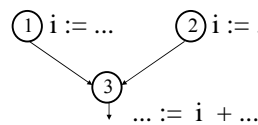
→ *Static Single Assignment (SSA)* form (Cytron 1991) or *single assignment language*

## SSA form

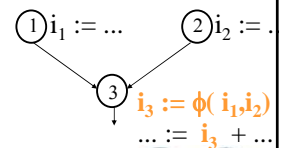
Each use of a variable refers to a single definition

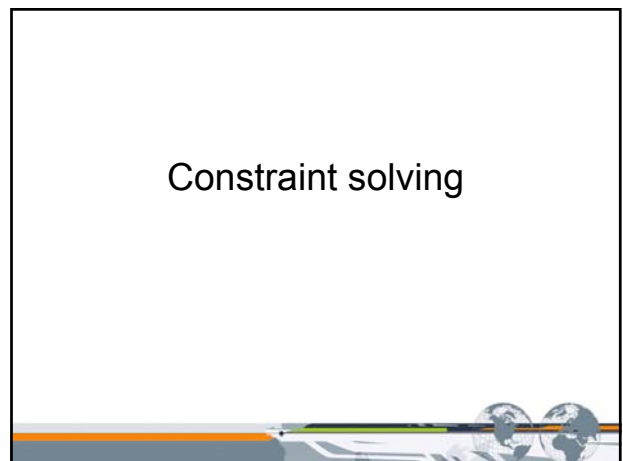
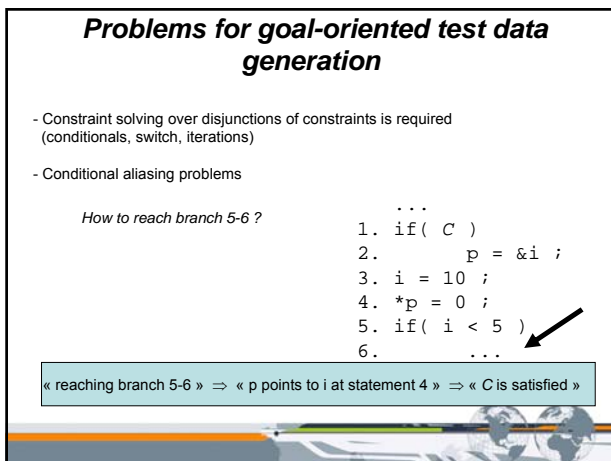
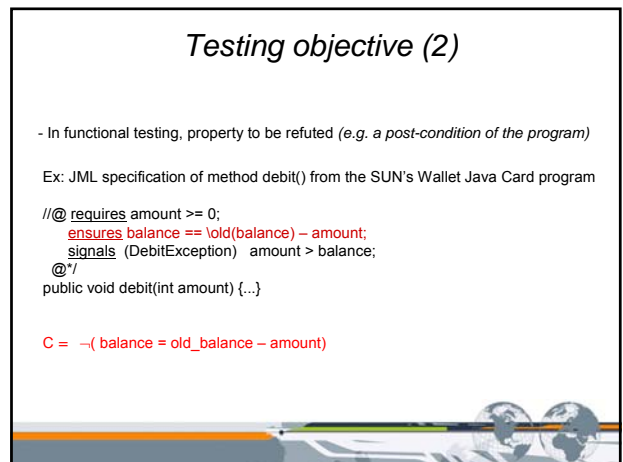
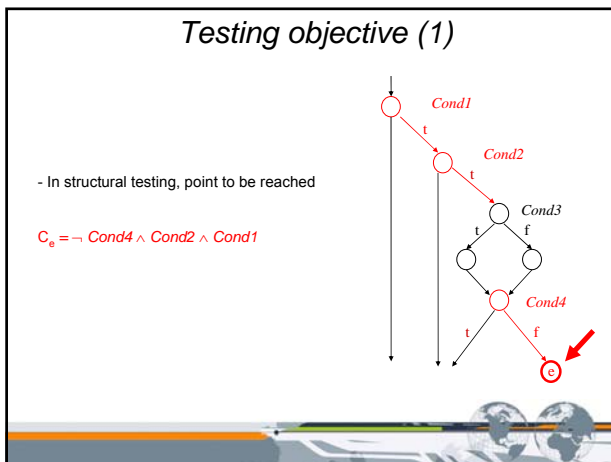
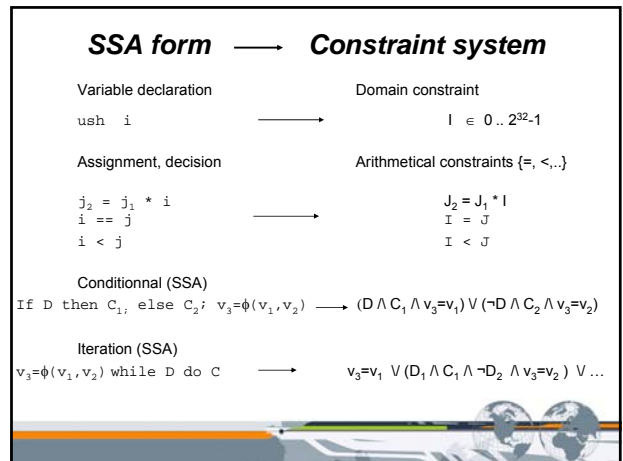
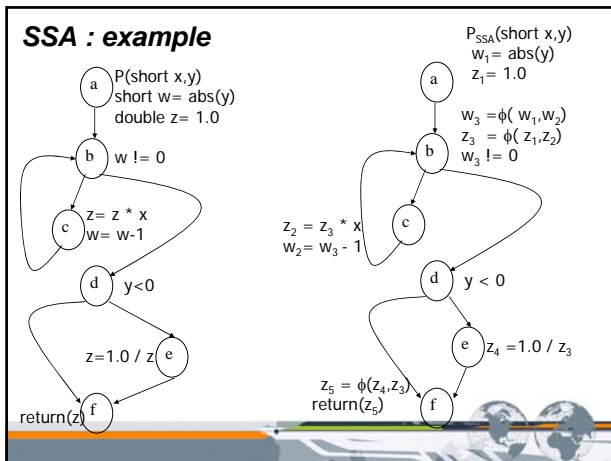
```
x := x + y;      x1 := x0 + y0;
y := x - y;      y1 := x1 - y0;
X := x - y;      x2 := x1 - y1;
```

At the junction nodes



ϕ\_functions





## 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:** **Constraint language (quantifier-free formula):**

Booleans	Boolean formula $(A \wedge B \wedge \neg C) \vee (\neg A \wedge B \wedge C)$
Integers	
Bounded integers	Conjunction of linear constraints $(X \geq Y, 5X - Y \leq 0)$
Rationals	
Reals	Polynomial constraints $(3 * X^2 + Y - 1 > 0, 4X^3 - Y + 8 = 0)$
Floating-point numbers	Non-linear constraints $(Z = \text{gcd}(X, Y) \vee Z = X \text{ div } Y)$
...	...

## Decidability and complexities

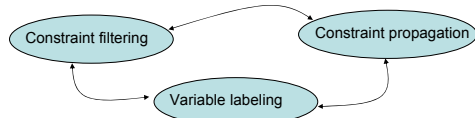
	Boolean formula	Linear constraints	Polynomial constraints	Non-linear constraints
Booleans	2-SAT in P 3-SAT is NP_complete	0-1 Programming is NP_complete	?	?
Bounded integers	--	NP-complete	NP-complete	NP-complete
Integers	--	Integer Programming is NP_complete	Undecidable	Undecidable
Rationals and reals	--	Linear Programming in P	Nonlinear Programming is NP_complete	Undecidable

## Some decision procedures (best practices)

	Boolean formula	Linear constraints	Polynomial constraints	Non-linear constraints
Booleans	Davis & Putnam (DPLL)			
Integers		-Cooper's procedure for Presburger arith. - Shostak's algo.		
Bounded integers		Constraint satisfaction	Constraint satisfaction	Constraint satisfaction
Rationals and reals		-Simplex - Fourier Elimination	Gröbner basis (Buchberger alg.)	Interval propagation

## 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:



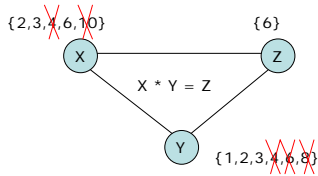
## Constraint filtering (1)

- 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)

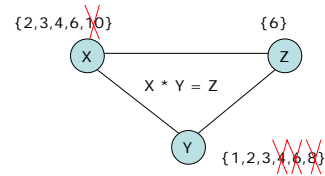
**Domain consistency** (for each value in  $D_x$ , find a support in  $D_z$  and  $D_y$ )



→ 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  $D_y$ )



→ Does not depend on the size of domains !

### Constraint Propagation (1)

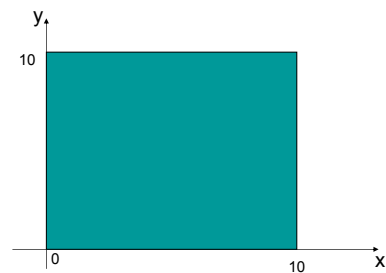
- Propagates prunings throughout the constraint system
- Implemented as a fixpoint algorithm:

```

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') ;
    
```

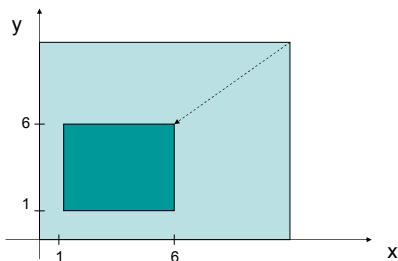
Ex:  $X, Y$  in  $0..10$ ,  $X * Y = 6$ ,  $X + Y = 5$ .

### Constraint propagation (2)



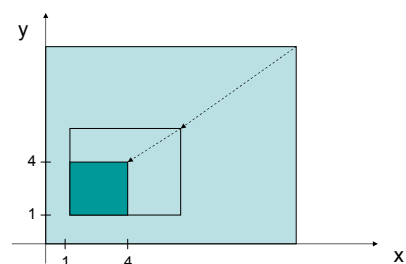
Ex :  $X, Y$  in  $0..10$ ,  $X * Y = 6$ ,  $X + Y = 5$ .

### Constraint propagation (3)



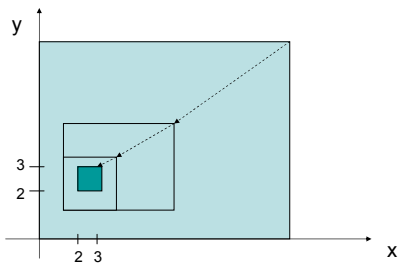
Ex :  $X, Y$  in  $0..10$ .  $X * Y = 6$ ,  $X + Y = 5$ .

### Constraint propagation (4)



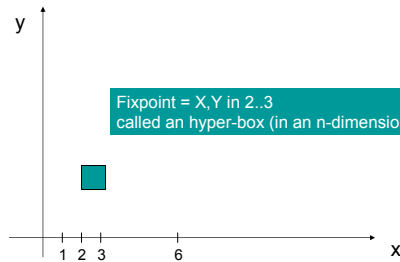
Ex :  $X, Y$  in  $0..10$ .  $X * Y = 6$ ,  $X + Y = 5$ .

### Constraint propagation (5)



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

### Constraint propagation (6)

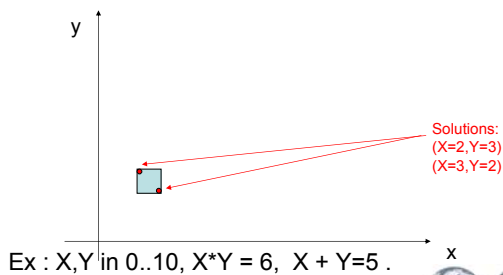


Fixpoint =  $X, Y$  in  $2..3$   
called an hyper-box (in an n-dimensions space)

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  $X = v$



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

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

### 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

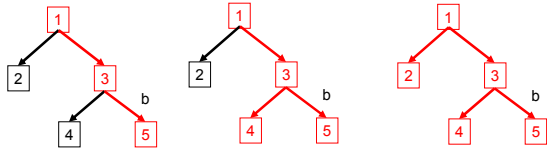
### Plan

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

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

- N. Williams, B. Marre, P. Mouy and M. Roger. PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis. In Proc. Dependable Computing - EDCC 2005, Budapest, Hungary, April 2005  
<http://www-list.cea.fr/labs/fr/LSL/test/pathcrawler/index.html>
- 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 dynamic symbolic exec. over the corresp. **feasible path**

Try to solve the CS where the last constraint is refuted

Backtrack and try to solve the remaining CS

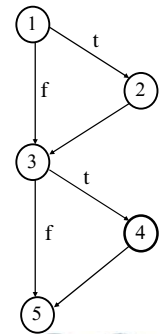
**If unsatisfiable then path is non-feasible and problem comes from the last constraint**  
**Else coverage of feasible path is improved**

### An example (1)

```
f( int i )
{
  j = 2;
  if( i ≤ 16 )
    j = j * i;

  if( j > 8 )
    j = 0;

  return j;
}
```



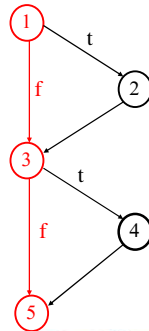
### An example (2)

```
f( int i )
{
  j = 2;
  if( i ≤ 16 )
    j = j * i;

  if( j > 8 )
    j = 0;

  return j;
}
```

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



### An example (3)

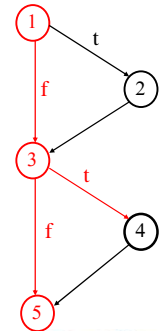
```
f( int i )
{
  j = 2;
  if( i ≤ 16 )
    j = j * i;

  if( j > 8 )
    j = 0;

  return j;
}
```

Try to solve  
 $j_1 = 2$   
 $i > 16$   
 $j_1 > 8$

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



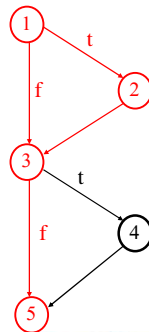
### An example (4)

```
f( int i )
{
  j = 2;
  if( i ≤ 16 )
    j = j * i;

  if( j > 8 )
    j = 0;

  return j;
}
```

Backtrack and try to solve  
 $j_1 = 2$   
 $i <= 16$   
 →  $(i = 2)$  -- Path 1-2-3-5



### An example (5)

```
f( int i )
{
  j = 2;
  if( i ≤ 16 )
    j = j * i;

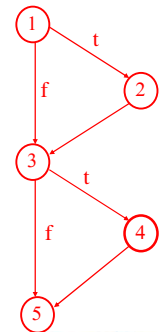
  if( j > 8 )
    j = 0;

  return j;
}
```

Backtrack and try to solve  
 $j_1 = 2$   
 $i <= 16$   
 $j_2 = j_1 * i$

$j_2 > 8$   
 →  $(i = 10)$  -- Path 1-2-3-4-5

All-paths covered with three test Data  $(i = 15448, i = 2, i = 10)$



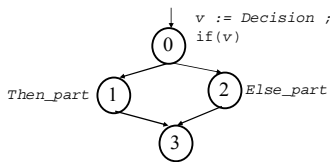
### The PathCrawler method: discussion

- 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



ite(V, C<sub>THEN</sub>, C<sub>ELSE</sub>, M<sub>IN</sub>, M<sub>OUT</sub>) :-

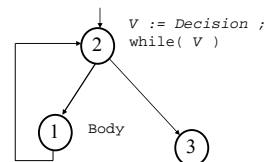
- V=1 → C<sub>THEN</sub> ∧ M<sub>OUT</sub> = M<sub>THEN</sub>
- V=0 → C<sub>ELSE</sub> ∧ M<sub>OUT</sub> = M<sub>ELSE</sub>

$$\neg(V=1 \wedge C_{\text{THEN}} \wedge M_{\text{OUT}} = M_{\text{THEN}}) \rightarrow V=0 \wedge C_{\text{ELSE}} \wedge M_{\text{OUT}} = M_{\text{ELSE}}$$

$$\neg(V=0 \wedge C_{\text{ELSE}} \wedge M_{\text{OUT}} = M_{\text{ELSE}}) \rightarrow V=1 \wedge C_{\text{THEN}} \wedge M_{\text{OUT}} = M_{\text{THEN}}$$

$$M_{\text{OUT}} := \text{Proj}(\text{OUT}, M_{\text{THEN}} \cup M_{\text{ELSE}}) \quad M_{\text{IN}} := \text{Proj}(\text{IN}, M_{\text{THEN}} \cup M_{\text{ELSE}})$$

### Iteration: the combinator w



w(V, C<sub>BODY</sub>, M<sub>IN</sub>, M<sub>OUT</sub>) :-

- V=1 → C<sub>BODY</sub> ∧ w(V, C<sub>BODY</sub>, M<sub>BODY</sub>, M<sub>OUT</sub>)
- V=0 → M<sub>OUT</sub> = M<sub>IN</sub>

$$\neg(V=1 \wedge C_{\text{BODY}}) \rightarrow V=0 \wedge M_{\text{OUT}} = M_{\text{IN}}$$

$$\neg(V=0 \wedge M_{\text{OUT}} = M_{\text{IN}}) \rightarrow V=1 \wedge C_{\text{BODY}} \wedge w(V, C_{\text{BODY}}, M_{\text{BODY}}, M_{\text{OUT}})$$

$$M_{\text{OUT}} := \text{Proj}(\text{OUT}, M_{\text{Before}} \nabla M_{\text{Body}}) \quad M_{\text{IN}} := \text{Proj}(\text{IN}, M_{\text{Before}} \nabla M_{\text{Body}})$$

Where  $\nabla$  stands for the widening operator

### Example (1)

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

```
I ∈ 0 .. 232-1,
J1 = 2,
ite( I ≤ 16, J2 = J1 * I ∧ J3 = J2, J3 = J1 ),
ite( J3 > 8, J4 = 0 ∧ J5 = J4, J5 = J3 ),
RET = J5
```

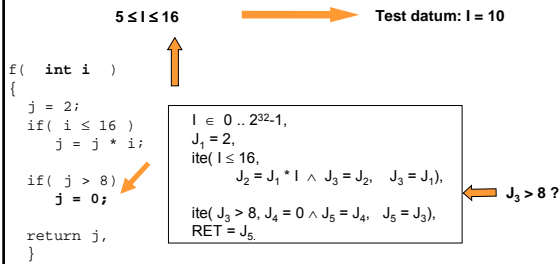
### Example (2)

I = 18 ?    I = 4    Impossible    1 ≤ I ≤ 4 ?

```
I ∈ 0 .. 232-1,
J1 = 2,
ite( I ≤ 16,
  J2 = J1 * I ∧ J3 = J2, J3 = J1 ),
ite( J3 > 8, J4 = 0 ∧ J5 = J4, J5 = J3 ),
RET = J5
```

RET = 2    RET = 8 ?    RET = 12 ?    2 ≤ RET ≤ 8

### Example (3)



### 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 stuck 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