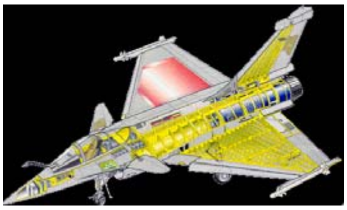
An overview of

Constraint-Based Testing

Arnaud Gotlieb
INRIA Rennes, France

Uppsala University, 05/19/10

Critical software systems

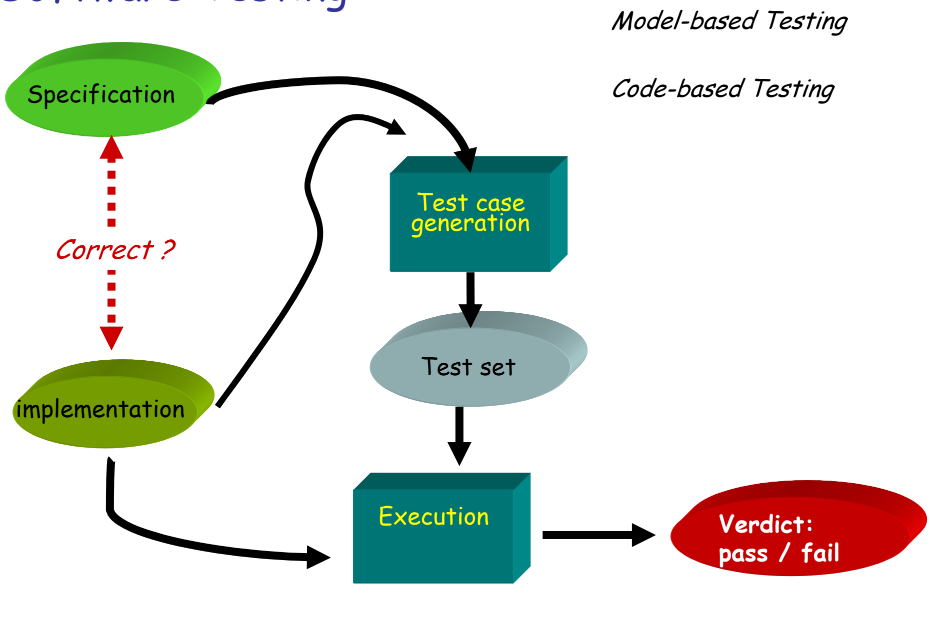


must be thorougly verified !

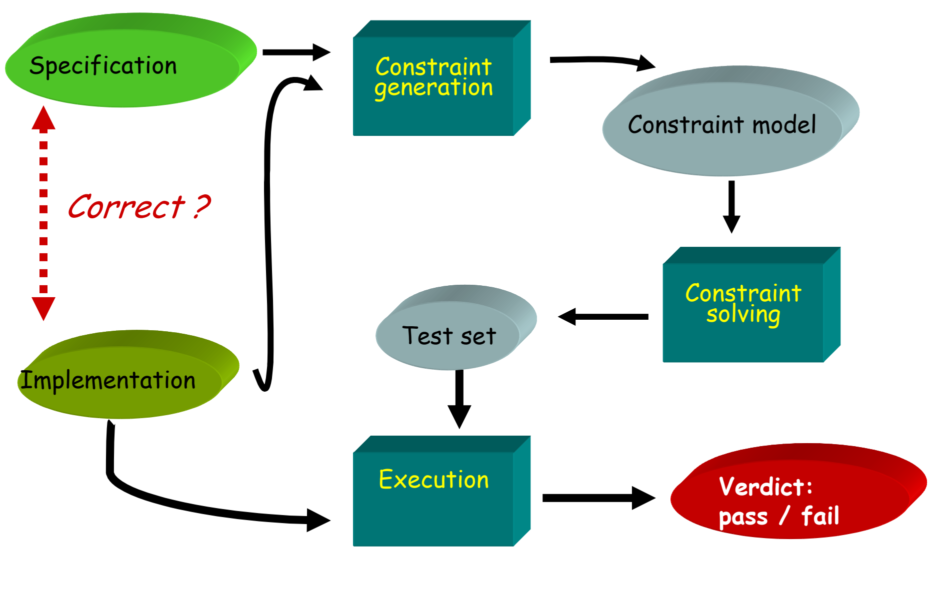
Several (complementary) techniques at the unit level:

program proving
software model-checking
static-analysis based verification
software unit testing

Software Testing



Constraint-Based Testing



Constraint-Based Testing (CBT)

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

Introduced 20 years ago by Offut and DeMillo in
(Constraint-based automatic test data generation IEEE TSE 1991)

Mainly used in the context of code-based testing with *code coverage* objectives, for finding *functional faults*

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

CBT: main tools

Microsoft Research	(**SAGE/PEX** P.Godefroid, P. de Halleux, N. Tillmann)
CEA - List	(**Osmose** S. Bardin P.Herrmann)
Univ. of Madrid	(**PET** M. Gomez-Zamalloa, E. Albert, G. Puebla)
Univ. of Stanford	(**EXE** D. Engler, C. Cadar, P. Guo)
Univ. of Nice Sophia-Antipolis	(**CPBPV** M. Rueher, H. Collavizza)
INRIA - Celtique	(**Euclide** A. Gotlieb, T. Denmat, F. Charreteur)
…	

Main CBT tools (industrial usage) :

PEX	(Microsoft	P. de Halleux, N. Tillmann)
InKa	(Dassault	A. Gotlieb, B. Botella**)**,
GATEL	(CEA	B. Marre**)**,
PathCrawler	(CEA	N. Williams**)**

The automatic test data generation problem

Given a location k in a program under test, generate a test input that reaches k

Undecidable in general, but ad-hoc methods exist

- ✓ Highly combinatorial $f\,(int\ x_1, int\ x_2, int\ x_3)\ \{ \dots \}$

2^{32} possibilities $\times$ 2^{32} possibilities $\times$ 2^{32} possibilities = **2^{96} possibilities**

- ✓ Loops and non-feasible paths
- ✓ Modular integer and floating-point computations
- ✓ Pointers, dynamic structures, function calls, …

Context of the presentation:

A single-threaded ANSI C function (infinite-state system)
selected location in code (reachability problems)

CBT: Pros/Cons

Pros: Handling control and data structures is essential in automatic software test data generation (i.e., SAT-solving doesn't work in that context !)

Improves significantly code-coverage
(as constraints capture hard-to-reach test objectives)

Fully automated test data generation methods

Cons: No semantics description, no formal proof → correction is not a priority !

Unsatisfiability detection has to be improved (to avoid costly labelling)

Still have to confirm that techniques and tools can scale to the testing of large-sized applications

Outline

- Introduction
- Path-oriented exploration
- Constraint-based exploration
- Further work

Path-oriented test data generation

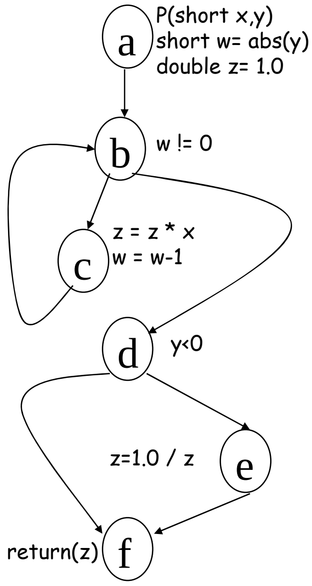
- Select one or several paths → **Path selection** step
- Generate the path conditions → **Symbolic evaluation** techniques
- Solve the path conditions to generate test data that activate the selected paths → **Constraint solving**

Test objectives: generating a test suite that covers a given testing criterion (all-statements, all-paths…) or a test data that raise a safety or security problem (assertion violation, buffer overflow, …)

Main CBT tools: **ATGen** (Meudec 2001), **EXE** (Cadar et al. 2006)

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

all-statement coverage:

a-b-c-b-d-e-f

All-branches 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

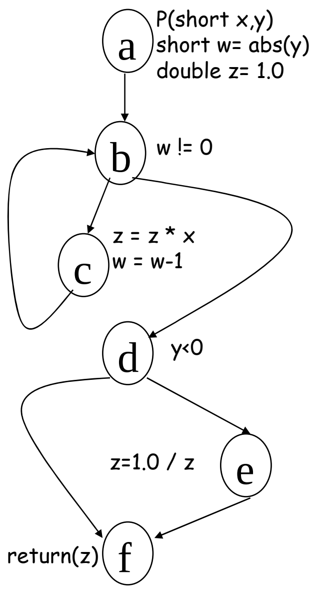
~~a-b-d-e-f~~

...

a-b-(c-b)^2-d-e-f

all-paths:

Impossible



Path condition generation

Symbolic state: ‹Path, State, Path Conditions›

Path = n_i-..-n_j is a path expression of the CFG
State = $\langle v_i, \varphi_i \rangle_{v \in Var(P)}$ where φ_i is an algebraic expression over ***X***
Path Cond. = c_1,..,c_n where c_i is a condition over ***X***

X denotes symbolic variables associated to the program inputs and `Var(P)` denotes internal variables

Symbolic execution

<u>Ex :</u> a-b-(c-b)^2-d-f with X,Y

‹a, ‹z,1.›, ‹w,abs(Y)›, true ›

‹a-b, ‹z,1.›, ‹w,abs(Y)›, abs(Y) != 0 ›

‹a-b-c, ‹z,X›, ‹w,abs(Y)-1›, abs(Y) != 0 ›

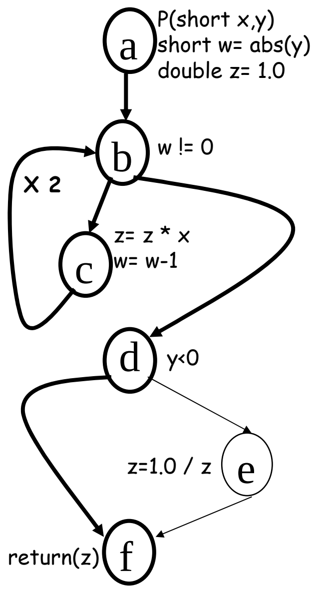
‹a-b-c-b, ‹z,X.›, ‹w,abs(Y)-1›, abs(Y) != 0, abs(Y)-1 != 0 ›

‹ a-b-c-b-c, ‹z,X^2›, ‹w,abs(Y)-2›, abs(Y) != 0, abs(Y)-1 != 0 ›

‹a-b-(c-b)^2, ‹z,X^2›, ‹w,abs(Y)-2›, abs(Y) != 0, abs(Y) != 1, abs(Y)-2 = 0 ›

‹a-b-(c-b)^2-d, ‹z,X^2›, ‹w,abs(Y)-2›, abs(Y) != 0, abs(Y) != 1, abs(Y) = 2, Y $\geq$ 0 ›

‹a-b-(c-b)^2-d-f, ‹z,X^2›, ‹w,0›, Y=2 ›



Computing symbolic states

- <Path, State, PC> is computed by induction over each statement of Path

- When the Path conditions are unsatisfiable then Path is non-feasible and reciprocally (i.e., symbolic execution captures the concrete semantics)

  ex : <a-~~b-d-e~~-f, {…}, **abs(Y)=0** $\wedge$ **Y<0** >

- Forward vs backward analysis:

  Forward → interesting when states are needed
  Backward → saves memory space, as complete states are not computed

Backward analysis

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

f,d: **Y** $\geq$**0**

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

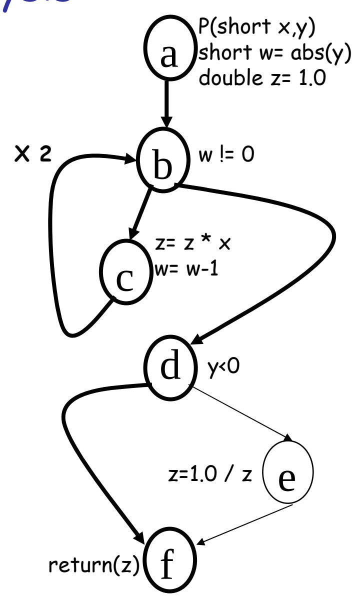
c: **Y** $\geq$**0**, w-1 = 0

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

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

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

a: **Y** $\geq$**0**, **abs(Y)-2 = 0,**
abs(Y)-1 != 0, abs(Y) != 0



Problems for symbolic evaluation techniques

→ Combinatorial explosion of paths (heuristics are needed to explore the search space)

→ Pointer and array aliasing problems

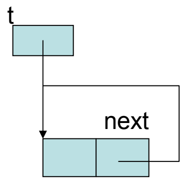
```
int P(int * p, int a) {
        if ( *p  !=  a ) { ...
```

if *p and a are aliased (i.e., p==&a) then the request is unsatisfiable!

→ Symbolic execution constrains the shape of dynamically allocated objects

```
int P(struct cell * t) {
  if( t == t->next ) { …
```

constrains t to:



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

Dynamic symbolic evaluation

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

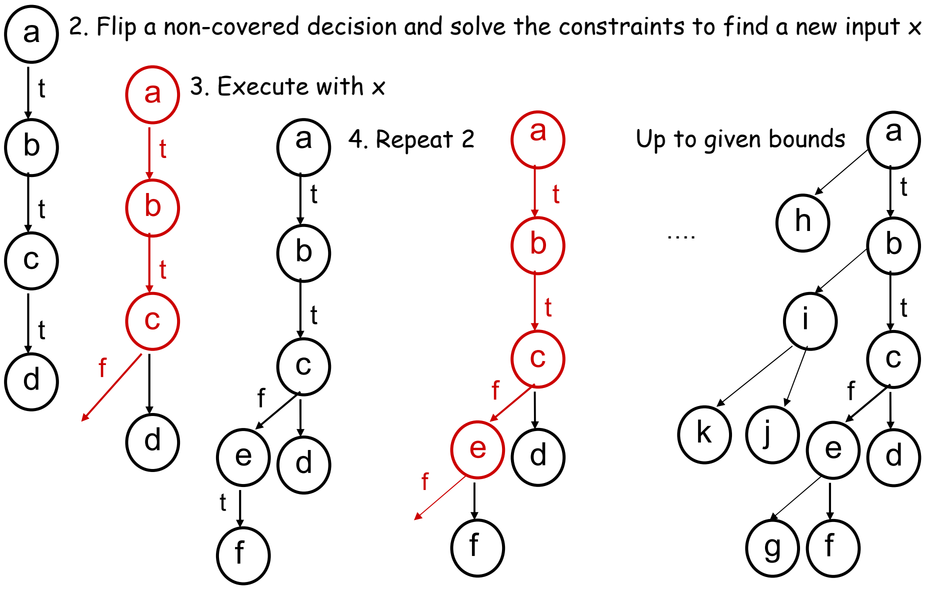
Main CBT tools:

PathCrawler (Williams et al. 2005),
DART/CUTE (Godefroid/Sen et al. 2005),
PEX (Tillman et al. Microsoft 2008), **SAGE** (Godefroid et al.2008)

Concolic execution

1. Draw an input at random, execute it and record path conditions
2. Flip a non-covered decision and solve the constraints to find a new input x
3. Execute with x
4. Repeat 2

Up to given bounds



Constraint solving in symbolic evaluation

- Mixed Integer Linear Programming approaches
  (i.e., simplex + Fourier's elimination + branch-and-bound)

  CLP(R,Q) in **ATGen (Meudec 2001)**
  lpsolve in **DART/CUTE (Godefroid/Sen et al. 2005)**

- SMT-solving (= SAT + Theories)

  STP in **EXE (Cadar et al. 2006)**,
  Z3 in **PEX (Tillmann and de Halleux 2008)**

- Constraint Programming techniques (constraint propagation and labelling)

  Colibri in **PathCrawler (Williams et al. 2005)**
  Disolver in ***SAGE*** (Godefroid et al. 2008)

Outline

- Introduction
- ➔ Path-oriented exploration
- Constraint-based exploration
- Further work

Constraint-based program exploration

- Based on a constraint model of the whole program
  (i.e., each statement is seen as a relation between two memory states)

- Constraint reasoning over control structures

- Requires to build ***dedicated constraint solvers***:
  * propagation queue management with priorities
  * specific propagators and global constraints
  * structure-aware labelling heuristics

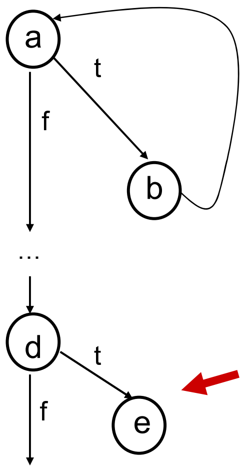
Main CBT tools: **InKa (**Dassault A. Gotlieb, B. Botella**)**,
GATEL (CEA B.Marre**)**,
Euclide (INRIA A. Gotlieb)

A reacheability problem

```
 f(  int i  )
  {
a.     j = 100;
       while( i > 1)
b.         { j++ ; i-- ;}

       …
d.     if( j > 500)
e.         …
```

value of i to reach e ?



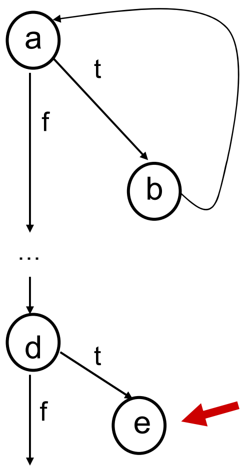
Path-oriented exploration

```
 f(  int i  )
  {
a.     j = 100;
       while( i > 1)
b.         { j++ ; i-- ;}

       …
d.     if( j > 500)
e.         …
```

1. Path selection
   e.g., $(a\text{-}b)^{14}$-…-d-e
2. *Path conditions* generation (via symbolic exec.)
   $j_1=100,\ i_1>1,\ j_2=j_1+1,\ i_2=i_1-1,\ i_2>1, \ldots, j_{15}>500$
3. *Path conditions* solving
   unsatisfiable → FAIL

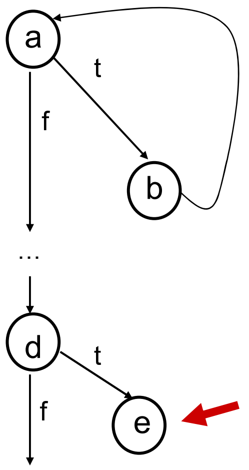
Backtrack !



Constraint-based exploration

```
 f(  int i  )
  {
a.     j = 100;
       while( i > 1)
b.         { j++ ; i-- ;}

       …
d.     if( j > 500)
e.         …
```



1. *Constraint model* generation (through SSA)
2. *Control dependencies* generation;
   $j_1=100,\ i_3 \le 1,\ j_3 > 500$
3. *Constraint model* solving
   $j_1 \neq j_3$ entailed ➔ unroll the loop 400 times ➔ i_1 in $401 .. 2^{31}-1$

No backtrack !

Assignment as Constraint

Viewing an assignment as a relation requires to normalize expressions and rename variables (through single assignment languages, e.g., SSA)

`i*=++i ;` ⟶ $i_2 = (i_1+1)^2$

$i_1 = 3$? i_1 in -4..2   no i_1 in -5..3

```
i*=++i;       /* i2 =(i1+1)2  */
```

$i_2 = 16$ $i_2 = 9$? $i_2 = 7$? i_2 in 5..16 ?

Statements as (global) constraints

- ✓ Type declaration: `signed long x;` → x in $-2^{31}..2^{31}-1$
- ✓ Assignments: `i*=++i ;` → $i_2 = (i_1+1)^2$
- ✓ Memory and array accesses and updates:
  `v=A[i]` (or `p=Mem[&p]`) → variations of element/3
- ✓ Control structures and function calls: dedicated global constraints

  Conditionals (SSA) `if D then` C_1; `else` C_2; $v_3=\phi(v_1,v_2)$ → ite/6

  Loops (SSA) $v_3=\phi(v_1,v_2)$ `while D do C` → w/5

  Function calls (SSA) `f(x1, .., xn)` → sp_call/2

a_0 in 25..75
a_1 in 50..100
a_2 in 0..5
v in 10..300

$\boxed{\text{element}(i, [a_0,a_1,a_2], v)}$

i in 0..1
v in 25..100

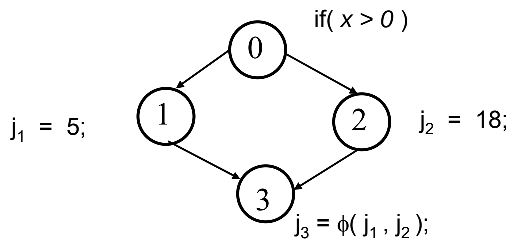
Global constraint definition

- Interface: set of variables of the relation
- Awakening conditions (X becomes valued, domain of X is pruned, …)
- Filtering algorithm (performed when awaked)

can be defined with a set of guarded-constraints

$C_1 \rightarrow C'_1$, …, $C_n \rightarrow C'_n$

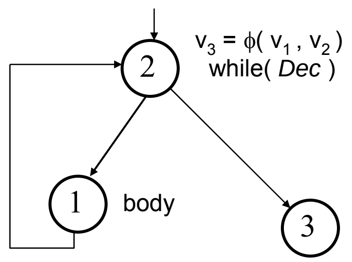
Conditional as global constraint: ite/6



ite($x > 0$, j_1, j_2, j_3, $j_1 = 5$, $j_2 = 18$) iff

- $x > 0 \rightarrow j_1 = 5 \wedge j_3 = j_1$
- $\neg(x > 0) \rightarrow j_2 = 18 \wedge j_3 = j_2$
- $\neg(x > 0 \wedge j_1 = 5 \wedge j_3 = j_1) \rightarrow \neg(x > 0) \wedge j_2 = 18 \wedge j_3 = j_2$
- $\neg(\neg(x > 0) \wedge j_3 = j_2) \rightarrow x > 0 \wedge j_1 = 5 \wedge j_3 = j_1$
- Join($x > 0 \wedge j_1 = 5 \wedge j_3 = j_1$, $\neg(x > 0) \wedge j_1 = 18 \wedge j_3 = j_2$)

Loop as global constraint: w/5



w(Dec, V_1, V_2, V_3, body) iff

- $Dec_{V3\leftarrow V1} \rightarrow body_{V3\leftarrow V1} \wedge$ **w**$(Dec, v_2, v_{new}, V_3, body_{V2\leftarrow Vnew})$
- $\neg Dec_{V3\leftarrow V1} \rightarrow V_3 = V_1$
- $\neg(Dec_{V3\leftarrow V1} \wedge body_{V3\leftarrow V1}) \rightarrow \neg Dec_{V3\leftarrow V1} \wedge V_3 = V_1$
- $\neg(\neg Dec_{V3\leftarrow V1} \wedge V_3 = V_1) \rightarrow Dec_{V3\leftarrow V1} \wedge body_{V3\leftarrow V1} \wedge$ **w**$(Dec, V_2, V_{new}, V_3, body_{V2\leftarrow Vnew})$
- join($Dec_{V3\leftarrow V1} \wedge body_{V3\leftarrow V1} \wedge$ **w**$(Dec, V_2, V_{new}, V_3, body_{V2\leftarrow Vnew})$, $\neg Dec_{V3\leftarrow V1} \wedge V_3 = V_1$)

```
f(  int i  ) {
  j = 100;
  while( i > 1)
   { j++ ; i-- ;}
  …
  if( j > 500)
    …
```

w(Dec, V_1, V_2, V_3, body) :-

- $Dec_{V3 \leftarrow V1} \rightarrow body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew})$
- $\neg Dec_{V3 \leftarrow V1} \rightarrow v_3 = v_1$
- $\neg(Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1}) \rightarrow \neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1$
- $\neg(\neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1) \rightarrow$
  $Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew})$
- $join(Dec_{V3 \leftarrow V1} \wedge body_{V3 \leftarrow V1} \wedge \mathbf{w}(Dec, v_2, v_{new}, v_3, body_{V2 \leftarrow Vnew},$
  $\neg Dec_{V3 \leftarrow V1} \wedge v_3 = v_1)$

$i = 23, j_1 = 100$?

no

i in $401..2^{31}-1$

$\boxed{\mathbf{w}(i_3 > 1, (i,j_1), (i_2,j_2), (i_3,j_3),\ j_2 = j_3 + 1 \wedge i_2 = i_3 - 1)}$

$i_3 = 1, j_3 = 122$

$i_3 = 10$?

$j_1 = 100$,
$j_3 > 500$?

Features of the w relation

- ✓ It can be nested into other relation (e.g., nested loops w($cond_1$, v_1,v_2,v_3, w($cond_2$, …))
- ✓ Managed by the solver as any other *constraint* (its consistency is iteratively checked, awakening conditions, success/failure/suspension)
- ✓ By construction, w is unfolded only when necessary but
  w may NOT terminate !
- ✓ Join is implemented using *Abstract Interpretation* operators (interval union, weak-join, widening)

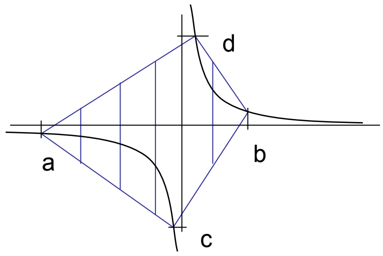
(Gotlieb et al. CL'2000, Denmat et al. CP'2006)

Abstraction-based relaxations

→ During constraint propagation, constraints can be relaxed in Abstract Domains (e.g., Q-Polyhedra)

$Z = X * Y, \quad X \text{ in } a..b, Y \text{ in } c..d$

$$\Leftrightarrow \{ \; Z - Ya - Xc + ac \geq 0, \; Xd - Z - ad + aY \geq 0, \; bY - bc - Z + Xc \geq 0, \; bd - bY - Xd + Z \geq 0, \; a \leq X \leq b, c \leq Y \leq d \}$$



→To benefit from specialized algorithm (e.g., simplex for linear constraints) and capture global states of the constraint system

→ Require safe/correct over-approximation (to preserve property such as: *if the Q-Polyhedra is void then the constraint system is unsatisfiable*)

→ Q-Polyhedra in **Euclide** (Gotlieb ICST'09), Difference constraints in **Gatel**, Congruences domain in **IBM ILOG Jsolver** (Leconte CSTVA'06) and now **Gatel**

Outline

- Introduction
- Path-oriented exploration
- → Constraint-based exploration
- Further work

CBT (summary)

- Emerging concept in code-based automatic test data generation
- Two main approaches:

  Path-oriented test data generation *vs* constraint-based exploration

- Constraint solving:

  - Linear programming
  - SMT-solvers
  - Constraint Programming techniques with *abstraction-based relaxations*

- Mature tools (academic and industrial) already exist but application to real-sized applications still have to be demonstrated

Further work

- In constraint generation:

  - to handle complex data structures and type casting → advanced memory models (as complex as those used in automated program proving)

  - to handle efficiently function calls (modular analysis) and virtual calls in OO Programming (Thesis of F. Charreteur Mar. 2010, JAUT tool)

  - to deal with multi-threaded programs

- In constraint solving:

  - to improve the handling of modular integer and floating-point constraint solving

  - loops with abstraction-based relaxation → widening techniques

  - exploit parallelism to boost program exploration (in both path-oriented and constraint-based exploration)

Many thanks for your attention !

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

- Software **model-checking tools** explores a bounded boolean structure of the program in order to prove properties or find counter-examples

  while CBT uses global constraints to capture the structure

- **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.

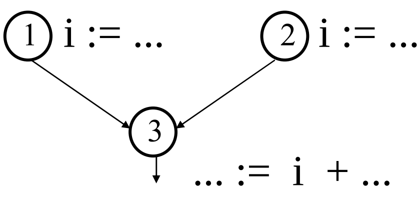
SSA form

Each use of a variable refers to a single definition

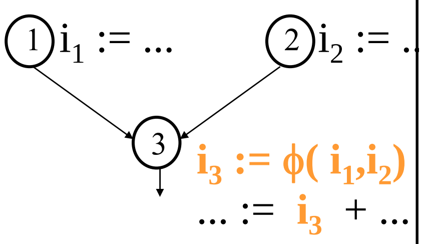
```
x := x + y;
y := x - y;
X := x - y;
```

$x_1 := x_0 + y_0;$
$y_1 := x_1 - y_0;$
$x_2 := x_1 - y_1;$

At the junction nodes



ϕ_functions



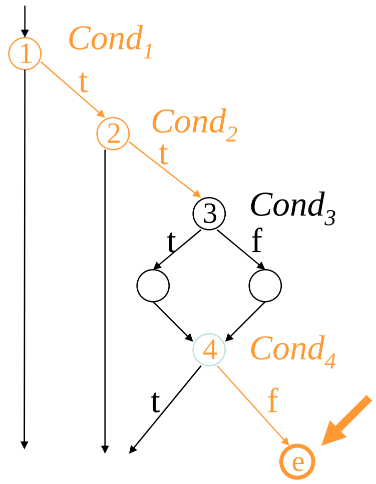
The reach directive

- Static control dependencies analysis over structured programs
- Implemented as a network of boolean constraints

$v_1 \Leftrightarrow cond_1$, $v_2 \Leftrightarrow cond_2$,
$v_3 \Leftrightarrow cond_3$, $v_4 \Leftrightarrow cond_4$,

$v_2 \Rightarrow v_1$, $v_3 \Rightarrow v_2$, $\neg v_3 \Rightarrow v_2$
$v_4 \Rightarrow v_2$, $\neg v_4 \Rightarrow v_2$

reach(e) ➔ v_4 = false



Global constraint definition

- Interface: set of variables of the relation
- Awakening conditions (X becomes valued, domain of X is pruned, …)
- Filtering algorithm (performed when awaked)

can be defined with a set of guarded-constraints $C_1 \rightarrow C'_1$, …, $C_n \rightarrow C'_n$

☞ Operational semantics of $C_i \rightarrow C'_i$ w.r.t. a constraint store :

-If C_i is entailed then C'_i is pushed on the propagation queue
and $\{C_j \rightarrow C'_j\}_{\forall j}$ are all removed from the queue

-If C_i is disentailed then only $C_i \rightarrow C'_i$ is removed

-Else $C_i \rightarrow C'_i$ is suspended and could be awaked when global ctr resumes

☞ Detection of entailment:
C_i is entailed by σ if $\sigma \wedge \neg C_i$ is inconsistent