

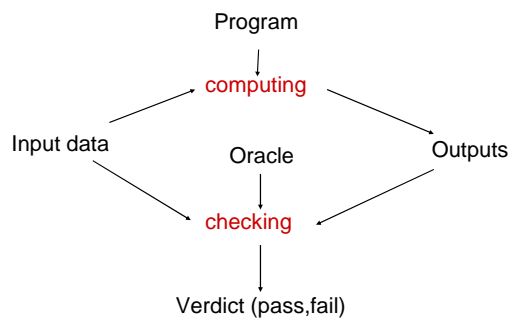
Automatic Test Data Generation with Constraint Logic Programming

Arnaud Gotlieb
Lande project-team
IRISA / INRIA
Rennes, France

Outline

1. The automatic test data generation problem
2. Our *Constraint Logic Programming* framework
3. Exploiting the *CLP* framework to test programs
4. Implementation and experimental results
5. Perspectives

A diagrammatic view of Program Testing

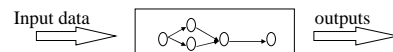


Testing strategies for selecting test data

Functionnal testing : based on specifications



Structural testing : based on source code



Structural testing

- ✓ Test data are selected according to a given model of the source code
- ✓ Coverage techniques (coming from Graph Theory) of
 - Control Flow Graph criteria
 - Def/Use Graph criteria
 - Program Dependence Graph
 - ...
- ✓ Selection can be probabilistic or deterministic

Our concern

Nov. 03

Colorado State University

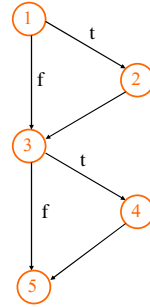
5

Structural testing based on CFG : *all_statements*

```
f( int i )
{
  j := ...
  if( Condition1 )
    j := ...

  if( Condition2 )
    j := ...

  return j
}
```



Nov. 03

Colorado State University

6

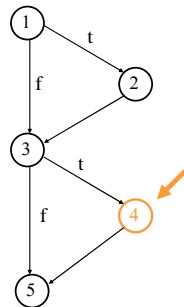
The core problem

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

  if( j > 8 )
    j := 0

  return j
}
```

value of i ?



Nov. 03

Colorado State University

7

Intuition of our approach

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

  if( j > 8 )
    j := 0

  return j
}
```

$j > 8$
 $i > 16 \Rightarrow j \neq 2$
 $i \leq 16 \Rightarrow j = 2 * i$

$i \leq 16, 2 * i > 8$

$5 \leq i \leq 16$

Nov. 03

Colorado State University

8

The Automatic Test Data Generation problem

Undecidable in the general case
(by reduction to the halting problem [Weyuker 79])

Difficulties for classical ad-hoc methods :

✓ Non-feasible paths

✓ Highly combinatorial

$f(\text{int } x_1, \text{int } x_2, \text{int } x_3) \{ \dots \}$

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

Outline

1. The Automatic Test Data Generation problem
2. Our *Constraint Logic Programming* framework
3. Exploiting the CLP framework to test programs
4. Implementation and experimental results
5. Perspectives

Background on *Constraint Logic Programming*

✓ Augments Prolog by adding constraints to the clauses

Elements of syntax (Horn clauses):

facts: $p(t_1, \dots, t_n).$

clauses: $p(t_1, \dots, t_n) :- q_1, \dots, q_m, C_1, \dots, C_k.$

goal: $?- C_1, \dots, C_j, p_1, \dots, p_i, \text{labeling}(X_1, \dots, X_n).$

✓ Solves the goals with the help of a constraint solver

- A solution is an assignment that satisfies all the constraints
- A Prolog's fail is returned when the constraints are shown to be contradictory

Solving the CLP goal

➤ The CLP goal:

?- $C_1, \dots, C_j, p_1, \dots, p_j, \text{labeling}(X_1, \dots, X_n).$

➤ By using classical constraint solving mechanisms:

- * **constraint projectors** – that eliminate inconsistent values
- * **constraint propagation** – iterative algo. that applies projectors
- * **labeling process** – assigns a value and propagates/backtracks according to a given search heuristic

Solving CLP goals : example

$p(X,Y,Z) :- X \text{ in } 1..4, Y \text{ in } 3..8, Z \text{ in } 0..5.$

?- $Z = X * Y, p(X,Y,Z), \text{labeling}(X,Y,Z)$

$X \leftarrow Z/Y$ leads to $X = 1$
 $Y \leftarrow Z/X$ leads to $Y \text{ in } 3..5$
 $Z \leftarrow X * Y$ leads to $Z \text{ in } 3..5$

Solutions = $\{(1,3,3), (1,3,4), (1,3,5), (1,4,3), (1,4,4), (1,4,5), (1,5,3), (1,5,4), (1,5,5)\}$

Our CLP framework

✓ Principle: inductive translation of each statement into a constraint

✓ Requires first to rename the variables

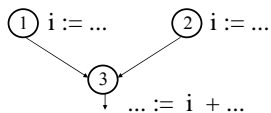
$$i := i+1 \rightarrow I_2 = I_1+1$$

✓ Adopted solution: using the **Static Single Assignment** form [Cytron et al. 89]

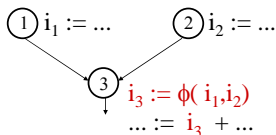
Static Single Assignment form (SSA)

Each use of a variable is associated with a single definition

Difficulty to the junction nodes



Use of ϕ _functions



Translating SSA statements into constraints

Declarations \rightarrow Domain constraints ($X \in 0..2^{32}-1$)

Assignments, decisions \rightarrow Basic constraints $\{=, <, \dots\}$

$$j_2 := j_1 * i \quad J_2 = J_1 * I$$

Array access $v_1 := a_0[i_0] \rightarrow$ Combinator $\text{element}(I_0, A_0, V_1)$

$$? \text{-element}(I_0, [5, 9, 3], V_1), V_1 < 6.$$

$$I_0 \in \{0, 2\}, V_1 \in \{3, 5\}$$

Conditionals and iterations \rightarrow Specific combinators $\text{ite}/6 \text{ w}/5$

Définition d'opérateur dans CLP(FD)

☞ Store de contraintes (*noté* σ) :
conjonction de contraintes à satisfaire

☞ Contrainte-gardée : $C_1 \rightarrow C_2$

- Si C_1 est impliquée par σ alors C_2 est ajoutée à σ
- Si $\neg C_1$ est impliquée par σ alors $C_1 \rightarrow C_2$ est retirée de σ
- Sinon suspension

☞ Test d'implication : C_1 est impliquée si $\sigma \wedge \neg C_1$ est inconsistant (basé sur des consistances locales)

Nov. 03

Colorado State University

17

Conditional : defining the combinator `ite/6`

```

if( cond )
  { then_part }
else { else_part }
v3 =  $\phi(v_1, v_2)$ ;
    
```

`ite(C_cond, v1, v2, v3, C_then, C_else) :-`

$C_{cond} \rightarrow C_{then} \wedge v_3=v_1,$

$\neg C_{cond} \rightarrow C_{else} \wedge v_3=v_2,$

$\neg(C_{cond} \wedge C_{then} \wedge v_3=v_1) \rightarrow \neg C_{cond} \wedge C_{else} \wedge v_3=v_2,$

$\neg(\neg C_{cond} \wedge C_{else} \wedge v_3=v_2) \rightarrow C_{cond} \wedge C_{then} \wedge v_3=v_1.$

Nov. 03

Colorado State University

18

Iteration : defining the combinator `w/5`

```

v3 =  $\phi(v_1, v_2)$ ;
while( cond )
  { body }
    
```

`w(C_cond, v1, v2, v3, C_body) :-`

$C_{cond} \rightarrow C_{body} \wedge w(C_{cond}, v_2, v_4, v_3, C_{body}),$

$\neg C_{cond} \rightarrow v_3=v_1,$

$\neg(C_{cond} \wedge C_{body}) \rightarrow \neg C_{cond} \wedge v_3=v_1,$

$\neg(\neg C_{cond} \wedge v_3=v_1) \rightarrow C_{cond} \wedge C_{body} \wedge w(C_{cond}, v_2, v_4, v_3, C_{body}).$

Nov. 03

Colorado State University

19

Features of combinator `w/5`

- ✓ Managed by the solver as a global constraint (its consistency is iteratively checked)
- ✓ By construction, `w/5` is unfolded only when necessary
- ✓ **Limitation : `w/5` may not terminate !**

Nov. 03

Colorado State University

20

Results interpretation

A solution of the CLP goal is found -->
A test datum that solve the goal

The query is contradictory (Prolog's fail) -->
A proof that there is no test datum

Interrupting prematurely the solving process (w/5) -->
A failure of our approach

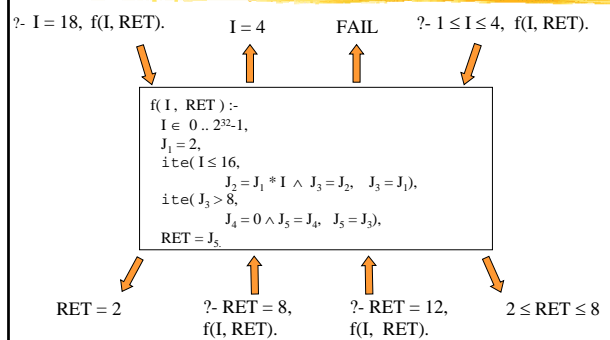
Consider again our example: SSA form

<pre>f(int i) { j := 2 if(i ≤ 16) j := j * i if(j > 8) j := 0 return j }</pre>	<p>→</p>	<pre>f(int i) { j₁ := 2 if(i ≤ 16) j₂ := j₁ * i j₃ := φ(j₁, j₂) if(j₃ > 8) j₄ := 0 j₅ := φ(j₃, j₄) return j₅ }</pre>
---	----------	---

Example : building a CLP program

<pre>f(int i) { j₁ := 2 if(i ≤ 16) j₂ := j₁ * i j₃ := φ(j₁, j₂) if(j₃ > 8) j₄ := 0 j₅ := φ(j₃, j₄) return j₅ }</pre>	<p>→</p>	<pre>f(I, RET) :- 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.</pre>
---	----------	--

Example : solving some CLP goals



Outline

1. The Automatic Test Data Generation problem
2. Our *Constraint Logic Programming* framework
3. Exploiting the CLP framework to test programs
 - w.r.t. the *all_nodes* criterion
 - w.r.t. Metamorphic- and Symmetry- relations violations
4. Implementation and experimental results
5. Related works and perspectives

Nov. 03

Colorado State University

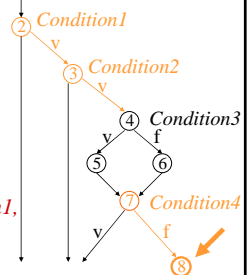
25

Reaching a selected node

Using the **Control-Dependencies**
[Ferrante *et al.* TOPLAS'87]

A CLP goal to solve :

?- \neg Condition4, Condition2, Condition1,
p(Input, Internals, Output)



Nov. 03

Colorado State University

26

Results interpretation

- A solution of the CLP goal is found -->
a test datum that sensitizes the selected node
- The query is contradictory (Prolog's fail) -->
the selected node is unreachable
- Interrupting prematurely the solving process (w/5) -->
A failure of our approach

Nov. 03

Colorado State University

27

Example : solving automatically the motivating problem

$5 \leq I \leq 16$ \longrightarrow Test datum: $I = 10$

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

```
f( [I], [J5], RET ) :-
  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.
```

\longleftarrow ?- $J_3 > 8$,
 $f([I], [J_3], RET)$

Nov. 03

Colorado State University

28

Outline

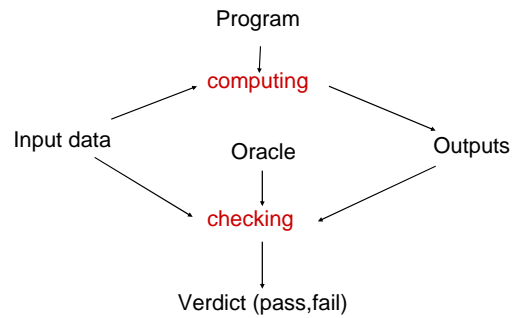
1. The Automatic Test Data Generation problem
2. A *Constraint Logic Programming* framework
3. Exploiting the framework to test programs
 - w.r.t. the *all_nodes* criterion
 - w.r.t. Metamorphic- and Symmetry- relations violations
4. Implementation and experimental results
5. Related works and perspectives

Nov. 03

Colorado State University

29

A diagrammatic view of Program Testing



Nov. 03

Colorado State University

30

Metamorphic Testing [Chen et al. 98]

- > Principle: exploiting **user-defined** relations over input test data and the outputs, when no oracle is available
- > Program p intended to compute a function f , test set $\{I_1, \dots, I_n\}_{n>1}$, a metamorphic relation is :
$$r(I_1, \dots, I_n) \Rightarrow r_f(p(I_1), \dots, p(I_n))$$
- > Only a necessary condition for the correctness of p w.r.t. f

Nov. 03

Colorado State University

31

Example of Metamorphic-Relation : a symmetry relation

P : a program that implements the *gcd* of 2 integers

Problem: $P(1309, 693) = ?$

Trivial symmetry : $\forall u, \forall v, \text{gcd}(u, v) = \text{gcd}(v, u)$

Hence, if $P(1309, 693) \neq P(693, 1309)$
then **verdict = fail**

Nov. 03

Colorado State University

32

How to automate the generation of test data that violate a metamorphic relation ?

Problem statement^(*):

find $\{I_1, \dots, I_n\}_{n>1}$ such as: $r(I_1, \dots, I_n) \wedge \neg r_f(p(I_1), \dots, p(I_n))$

➤ The CLP goal:

?- $r(I_1, \dots, I_n),$
 $p(I_1, O_1), \dots, p(I_n, O_n),$
 $\neg r_f(O_1, \dots, O_n),$
 $\text{labeling}(I_1, \dots, I_n).$

(*) likely to be undecidable in the general case !

Nov. 03

Colorado State University

33

Language of metamorphic-relations

$r(I_1, \dots, I_n) \Rightarrow r_f(p(I_1), \dots, p(I_n))$ where $n>1$

In our framework, r and r_f are n -ary relations built with

- * integer constants
- * symbolic input values
- * $\{+, -, *, \text{div}, \text{mod}\}$
- * $\{=, \neq, >, \geq, <, \leq, \neg, \wedge, \vee\}$

➔ language closed under negation

Nov. 03

Colorado State University

34

Results interpretation and limits

➤ Results interpretation:

- a solution $\{I_1, \dots, I_n\}_{n>1}$ is found ➔ **MR is violated**
- inconsistency of the goal (Prolog's fail)
➔ **MR is proved to be satisfied (within the finite domains) !**
- interrupting prematurely the solving process
➔ **failure of the method**

➤ Impossible to know which inputs among $\{I_1, \dots, I_n\}_{n>1}$ are responsible of the MR violation

Nov. 03

Colorado State University

35

Outline

1. The Automatic Test Data Generation problem
2. Our *Constraint Logic Programming* framework
3. Exploiting the CLP framework to test programs
4. Implementation and experimental results
5. Perspectives

Nov. 03

Colorado State University

36

InKa : history

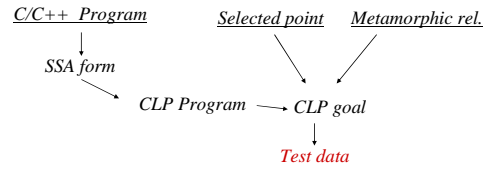
- 1995 -- Start of the Research works (Start of my PhD thesis)
- 1996 -- Automatic Test Data Generation using CLP -- first publication
- 1998 -- **Prototype tool for C--**, first experimental results
- 2000 -- *InKa project* : Grant from the French Ministry of Industry
Partners : Thales AS, Axlog, 3 academics labs I3S, LIFC, LSR
- 2002 -- **InKa 1.0 for (a non-trivial subset of) C/C++**
- 2003 -- Automated Metamorphic Testing + Symmetric Testing
V3F (floating-point numbers) : Research Grant --
Partners : INRIA Cassis, Coprin, Lande, CEA

Nov. 03

Colorado State University

37

The InKa tool



✓ Implemented in SICStus Prolog, Java and Tcl/Tk, SSA form generated by an single-pass algorithm [Brandis & Mössenbock 1994]

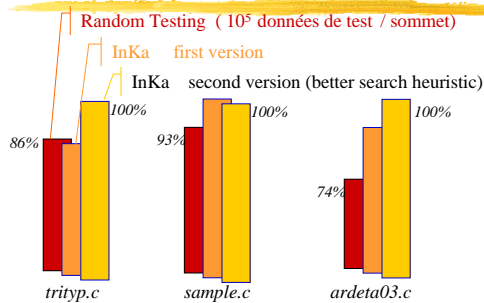
✓ Non-trivial subset of C/C++ : structured programs only, structures, 34 operators over 42, integer data types, pointers to named locations, function calls without call_by_reference, ...

Nov. 03

Colorado State University

38

Experiments : covering *all* statements



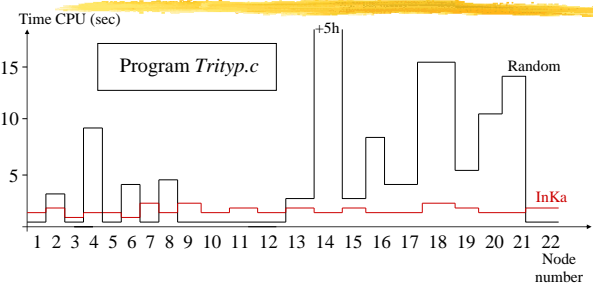
Max 10sec / node - Sparc 5 Solaris 2.5

Nov. 03

Colorado State University

39

Time needed to generate a single test datum



☞ Successful over difficult node to reach

Nov. 03

Colorado State University

40

First experiments for Methamorphic Testing

- ✓ 3 classical programs
 - bsearch (the non-trivial example)
 - GetMid (median of 3 integers)
 - Trityp (triangle classification)
- ✓ Faults injected by manual mutation -- 37 mutants
- ✓ Typical consequences of injected faults :
 - missing path error
 - non-feasible path
 - infinite paths

First experimental results

programs	Metamorphic--relation	Mutation score	Average time to kill a mutant
bsearch	$I_1=(a, e) \Rightarrow p(I_1)=p(I_2)$ $I_2=(f(a), f(e))$	1 / 2	3.1 sec
GetMid	$I_2=\pi(I_1) \Rightarrow p(I_1)=p(I_2)$ \forall permutation $\pi \in S_3$	2 / 2	0.3 sec
Trityp	$I_1=(u, v, w) \Rightarrow p(I_1)=p(I_2)$ $I_2=(2u, 2v, 2w)$	5 / 33	14.3 sec
Trityp	$I_2=\pi(I_1) \Rightarrow p(I_1)=p(I_2)$ \forall permutation $\pi \in S_3$	23 / 33	3.8 sec

(CPU time on 1.8GHz Pentium 4)

Outline

1. The Automatic Test Data Generation problem
2. A *Constraint Logic Programming* framework
3. Exploiting the framework to test programs
4. Implementation and experimental results
5. Perspectives

Perspectives

- Automatic test data generation for programs with floating-point variables

if (X + 16.0 == 16.0 && X > 0.0)

...

doesn't have any solution over the reals but has many over the floats (with a near rounding mode) !

- Fault localization using Symmetric Testing:
 - finding statements where equivalent states diverge

References

- ✓ Automatic Structural Test Data Generation with CLP
 « **Automatic Test Data Generation using Constraint Solving Techniques** »
 A. Gotlieb, B. Botella, M. Rueher *ISSTA'98*, also SEN vol. 2, 1998.
- « **A CLP Framework for Computing Structural Test Data** »
 A. Gotlieb, B. Botella, M. Rueher, *Comp. Logic CL'00 also in LNAI 1861*, 2000
- ✓ Floating-point numbers
 « **Solving constraints over floating-point numbers** »
 C. Michel, M. Rueher, Y. Lebbah *CP'2001*, also in LNCS 2239
- ✓ Metamorphic Testing, Symmetric Testing
 « **Automated Metamorphic Testing** » A. Gotlieb, B. Botella *COMPSAC'03*
- « **Exploiting Symmetries to Test Programs** » A. Gotlieb, *ISSRE'03*

Nov. 03

Colorado State University

45

Related works

- ☞ **Statistical Structural testing**
 [Thévenod-Fosse 91, Gouraud et al. ASE'01]
 → probabilistic generation
- ☞ **Constraint-Based Automatic Test Data Generation**
 [DeMillo & Offutt TSE'91, Meudec STVR 01, Sy & Deville ESEC-FSE'03]
 → based on symbolic execution, path selection is required
- ☞ **Dynamic method**
 [Korel TSE'90, Gupta et al. ASE'99]
 [Michael et al. TSE'01, Gupta ICSE'03]
 → based on program executions

Nov. 03

Colorado State University

46

Interest of the approach

```

f( int i )
{
  j1 := 2
  if( i ≤ 16 )
    j2 := j1 * i
    j3 := φ(j1, j2)
  if( j3 > 8 )
    ...

```

```

graph TD
    A[Node 4 selected] --> B[Combinator ite  
(3rd guarded-constraint)]
    B --> C[1-3-4 non-feasible]
    C --> D[1-2-3-4 to be executed]
    D --> E[Reduce the search  
space by filtering]
    E --> F[Labeling  
process]

```

```

graph TD
    1((1)) -- v --> 2((2))
    1 -- f --> 3((3))
    3 -- v --> 4((4))
    3 -- f --> 4

```

Nov. 03

Colorado State University

47

Functional testing

Specification :
 returns the product
 of i by j
 (i = 0, j = 0) --> 0
 (i = 10, j = 100) --> 1000
 ...
 --> OK

```

prod(int i, int j )
{
  int k ;
  if( i == 2 )
    k := i << 1 ;
  else
    ( Faire i fois l 'addition de j )
  return k ;
}

```

Nov. 03

Colorado State University

48

Structural testing is indispensable !

Specification :

returns the product of i by j

(i = 0, j = 0) --> 0

(i = 10, j = 100) --> 1000

Undetected fault by functional testing

patch -> `k := j << 1`

```
prod(int i, int j)
{
  int k ;
  if( i==2 )
    k := i << 1 ;
  else
    (faire i fois l'addition de j)
  return k ;
}
```

Rappels : famille d'algorithmes de filtrage AC-n

Entrées : σ store de contraintes, D domaines des variables

Sorties : D' domaines filtrés, partiellement consistants

```
Agenda :=  $\sigma$  ;
while( Agenda  $\neq \emptyset$  )
  c := POP(Agenda) ;
  D' := narrow(c,D) ;      % Filtrage
  if( D'  $\neq D$  )
    Agenda := Agenda  $\cup$  {c'  $\in \sigma$  / vars(c')  $\cap$  vars(c)  $\neq \emptyset$ }
  D := D' ;
return(D') ;
```

Programmation par Contraintes

`x in 1..4, y in 3..8, z in 0..5,`
`z = x * y.`

`x <-- z / y` entraîne `x = 1`
`y <-- z / x` entraîne `y in 3 .. 5`
`z <-- x * y` entraîne `z in 3 .. 5`

Solutions = {(1,3,3), (1,3,4), (1,3,5),
(1,4,3), (1,4,4), (1,4,5),
(1,5,3), (1,5,4), (1,5,5)}

Importance des conditions dans les gardes

```
f( int i )
{
  j1 := 3
  if( i < 0 )
    j2 := i
    j3 :=  $\Phi(j_1, j_2)$ 
  if( j3 > 2 )
    ...
```

- $i < 0 \rightarrow j_2 = i \wedge j_3 = j_2$
- $i \geq 0 \rightarrow j_3 = j_1$
- $\neg(i < 0 \wedge j_2 = i \wedge j_3 = j_2) \rightarrow i \geq 0 \wedge j_3 = j_1$
- $\neg(i \geq 0 \wedge j_3 = j_1) \rightarrow i < 0 \wedge j_2 = 0 \wedge j_3 = j_2$

$S = \{j_3 > 2\}$

$S \wedge \{j_2 = i, j_3 = j_2\}$ localement consistant
par contre,

$S \wedge \{i < 0, j_2 = i, j_3 = j_2\}$ inconsistent
Et la garde est impliquée

Problème posé par les flottants

$16.0 + X = 16.0$ avec $X > 0$

Pas de solution sur les réels

mais a des solutions sur les flottants (arrondi : near)

$X^2 = 2$ avec $X > 0$

Pas de solution sur les flottants (near)

mais ... a une solution sur les réels

un solveur sur les flottants

[Michel, Rueher CP 01]

Problème posé par les pointeurs

```
...
4.  *p := 0
5.  if( i > 1 )
6.  ...
```

1) Si $p \rightarrow i$ alors $i := 0$ et 4-5-6 non-exécutable

2) Si $p \not\rightarrow i$ alors $i > 1$ est possible

Absence d'infos : instruction 4 peut modifier n'importe quelle variable

Comment traduire *p ?

☞ Approche naïve

Pointeur déréférencé (*p) \rightarrow Une variable logique

☞ Notre approche :

Pointeur déréférencé (*p) \rightarrow Les variables (logiques)
« pointables » par p

☞ Premiers résultats : pointeurs sur des zones nommées
de un et plusieurs niveaux

Caractéristiques

Pointeur déréférencé (*p) \rightarrow Les variables (logiques)
« pointables » par p

☞ Utilisation d'une analyse de pointeurs particulière :

« **Points-to** » analyse

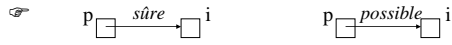
[Emami et al. PLDI 94]

☞ Extension de la forme SSA pour les pointeurs :

Forme Pointeur-SSA

☞ Définition d'un opérateur spécifique dans CLP sur les
domaines finis

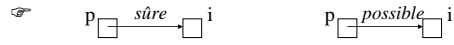
« Points-to » analyse



Analyse sensible au flot

Code	Analyse sensible au flot
1. p := &i	
2. if(...)	p -> possible i
3. p := &j	
4. ...	p -> possible j

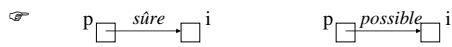
« Points-to » analyse



Analyse sensible au flot

Code	Analyse sensible au flot
1. p := &i	
2. if(...)	p -> sûre i
3. p := &j	
4. ...	p -> possible j

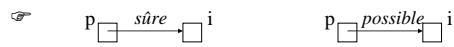
« Points-to » analyse



Analyse sensible au flot

Code	Analyse sensible au flot
1. p := &i	
2. if(...)	
3. p := &j	p -> sûre j
4. ...	

« Points-to » analyse



Analyse sensible au flot

Code	Analyse sensible au flot
1. p := &i	
2. if(...)	
3. p := &j	p -> possible i
4. ...	p -> possible j

Forme Pointeur_SSA

- Utilisation d'une double numérotation pour *p
- Introduction de ϕ_d fonctions (associées aux relations de pointage)

<p>4. *p := 0</p>	<p>$(*p_0)_1 := 0$</p>
	<p>$p \xrightarrow{\text{possible}} i$ d'où $i_1 := \phi_d(0, i_0)$</p>
	<p>$p \xrightarrow{\text{possible}} j$ d'où $j_1 := \phi_d(0, j_0)$</p>
<p>5. if(i > 1)</p>	<p>if($i_1 > 1$)</p>
<p>6. ...</p>	<p>...</p>

Traduction des ϕ_d fonctions : opérateur dans CLP

- Utilisation des contrainte-gardées

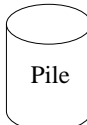
$p \xrightarrow{\text{possible}} i$ d'où $i_1 := \phi_d(0, i_0)$

$p = \&i \rightarrow i_1 = 0$
 $p \neq \&i \rightarrow i_1 = i_0$

- Limitation : non extensible aux structures de données dynamiques
- Autre piste : Pointeur déréférencé (*p) \rightarrow T[T[&p]]

Restrictions principales de notre approche

- Programmes non structurés (goto)
- Test objet

 <p>Pile</p>	<pre>private : int nb_elements ; public : boolean est_vide() ; int get_nb() ; void empiler(element e) ; void depiler() ;</pre>	<pre>... if(Pile.get_nb() > 2) ... </pre>
--	---	--

Difficulté liée aux chemins non-exécutables

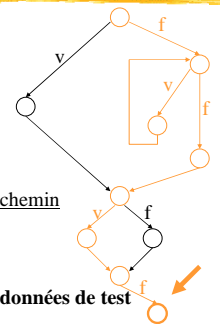
Instructions de boucle

\rightarrow **infinité de chemins** dans le **graphe**

Approche classique : choix au préalable d'un chemin

éventuellement **non-exécutable**

\rightarrow **échec de la génération automatique de données de test**



Two fundamental problems

Notations:

Program P intended to compute a function F
over a input domain D

1. How to select $\{X_1, \dots, X_n\}$ from D , to test P w.r.t. F ?
= **The test data selection problem**
2. How to verify that $P(X_i) = F(X_i)$ for all i ?
= **The oracle problem**