CP also meets Software Testing

Arnaud Gotlieb

Certus Software V&V Centre
SIMULA RESEARCH LABORATORY
Lysaker, Norway

CP meets CAV Workshop, Turunc, Turkey
A day in June 2012

CERTUS is also a Centre for research-based innovation (SFI)

Host
Simula Research Laboratory

User partners
CISCO Systems Norway
ESITO
FMC Technologies
KONGSBERG Maritime
TOLL customs and excises

Budget
~10 MNOK (1.3 MEUR) per year over a 8-years period

Origin (2011)
Prof. Lionel Briand (now in Luxembourg)

CISCO FMC Technologies esito TOLL CUSTOMS KONGSBERG

Industry-driven research problems in Software Validation & Verification

- Certification and verification of real-time embedded software-systems

- Modelling and testing of highly-configurable software-systems

- Automated testing of data-intensive administrative software-systems

With an increasing usage of Constraint Programming techniques (Finite Domains constraint solving, constraint optimization, MIP, Modelling)

Outline

A. Time-aware test configurations generation with Constraint Programming

B. Testing deadline misses for real-time systems using constraint-based scheduling techniques

C. Extraction of a formally verified constraint solver for the certification of tax computation

Outline

Constraint-based testing (CBT)

Constraint-based program exploration for automatic test data generation

Constraints over Memory Model Variables for testing pointer programs

Conclusions

Constraint-Based Testing (CBT)

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

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

Developed in the context of **code-based testing** and **model-based testing**

Lots of Research works and tools !

CBT: main tools

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, P.V. Hentenryck)
INRIA - Celtique (**Euclide, JAUT** A. Gotlieb, F. Charreteur)
...

Tools with **external** industrial usage :

- **GATEL** (CEA B. Marre, since 2004)
- **Test Designer** (Smartesting B. Legeard, since 2003)
- **PEX** (Microsoft P. de Halleux, N. Tillmann, since 2009)

Tools with **internal** industrial usage :

- **Inka V1** (Dassault A. Gotlieb, B. Botella, in 2001)
- **PathCrawler** (CEA N. Williams, since 2004)
- **SAGE** (Microsoft P. Godefroid, since 2010)

The automatic test data generation problem

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

Reachability problem in infinite-state systems is undecidable in general!

Even when adding bounds,
hard combinatorial problem

f (int x_1, int x_2, int x_3) {
if(x_1 == x_2 && x_2 ==x_3)
*if(x_3==x_1*x_2) ...* ← *}*

Using Random Testing,
Prob{ reack k} = 2 over $2^{32} \times 2^{32} \times 2^{32}$ = $\mathbf{2^{-95}}$ = **0.00000…1**.

Constraint solving techniques are required!

- ✓ Loops (i.e., infinite-state systems) and infeasible paths
- ✓ Pointers, dynamic structures, higher-order computations (virtual calls)
- ✓ Floating-point computations, modular computations

Context of this talk

Code-based testing	(not model-based testing)
Imperative programs (C, ...)	(not Functionnal P., not Logic P., not Object-Oriented P.)
Programs with loops	(i.e., infinite-state systems)
Single-threaded programs	(no concurrent or parallel programs)
Selected location in code	(i.e., reachability problems)

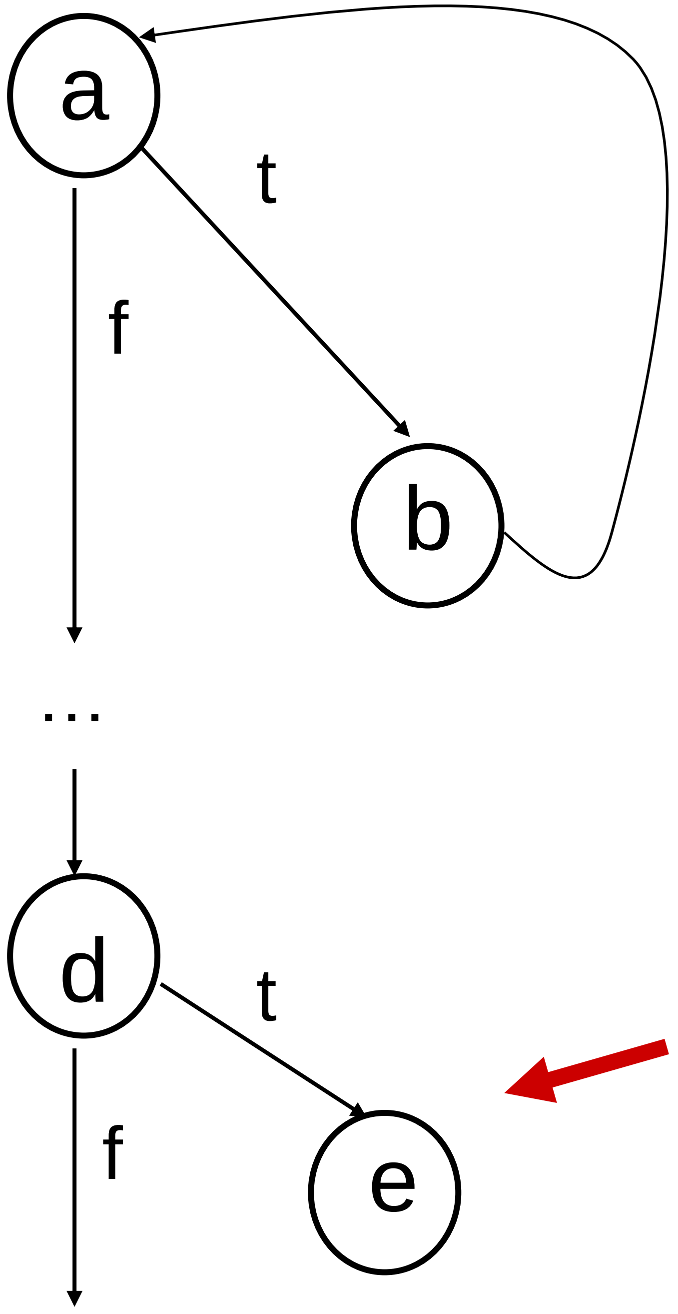
Constraint-based program exploration for automatic test data generation

A reacheability problem

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

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

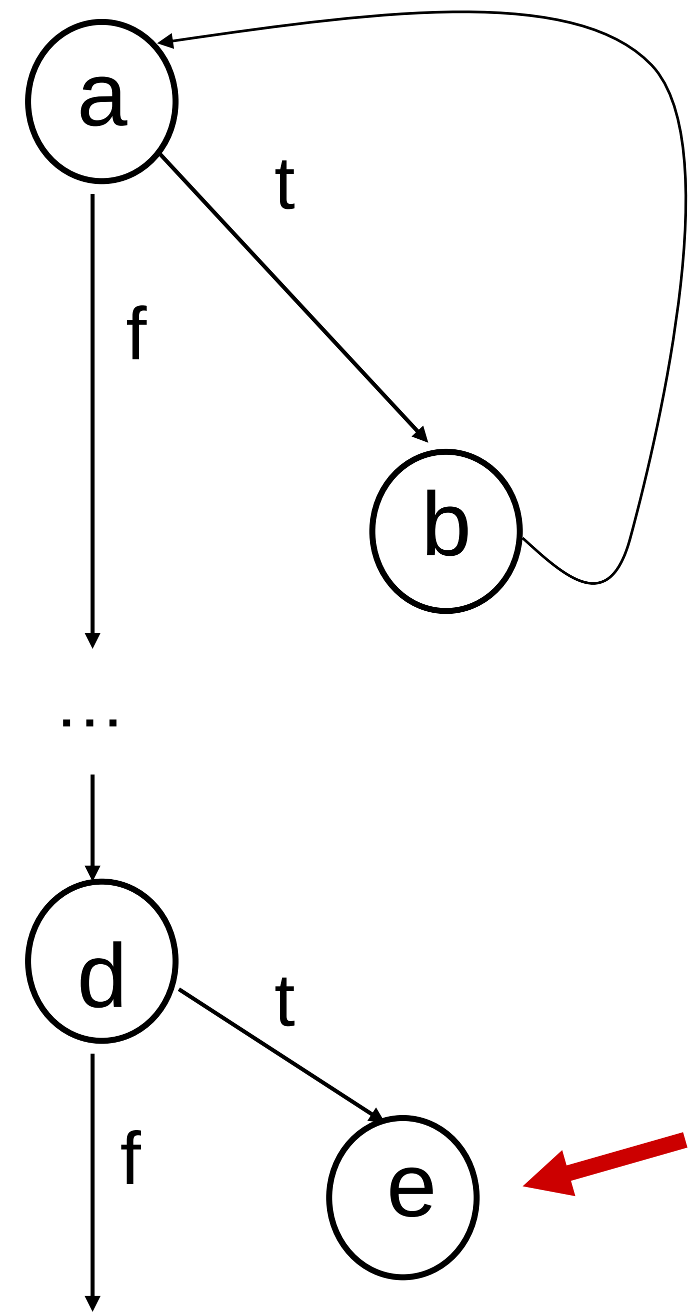
value of i to reach e ?



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., $(a\text{-}b)^{14}\text{-}\ldots\text{-}d\text{-}e$

2. Path condition 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 condition solving

   unsatisfiable → FAIL

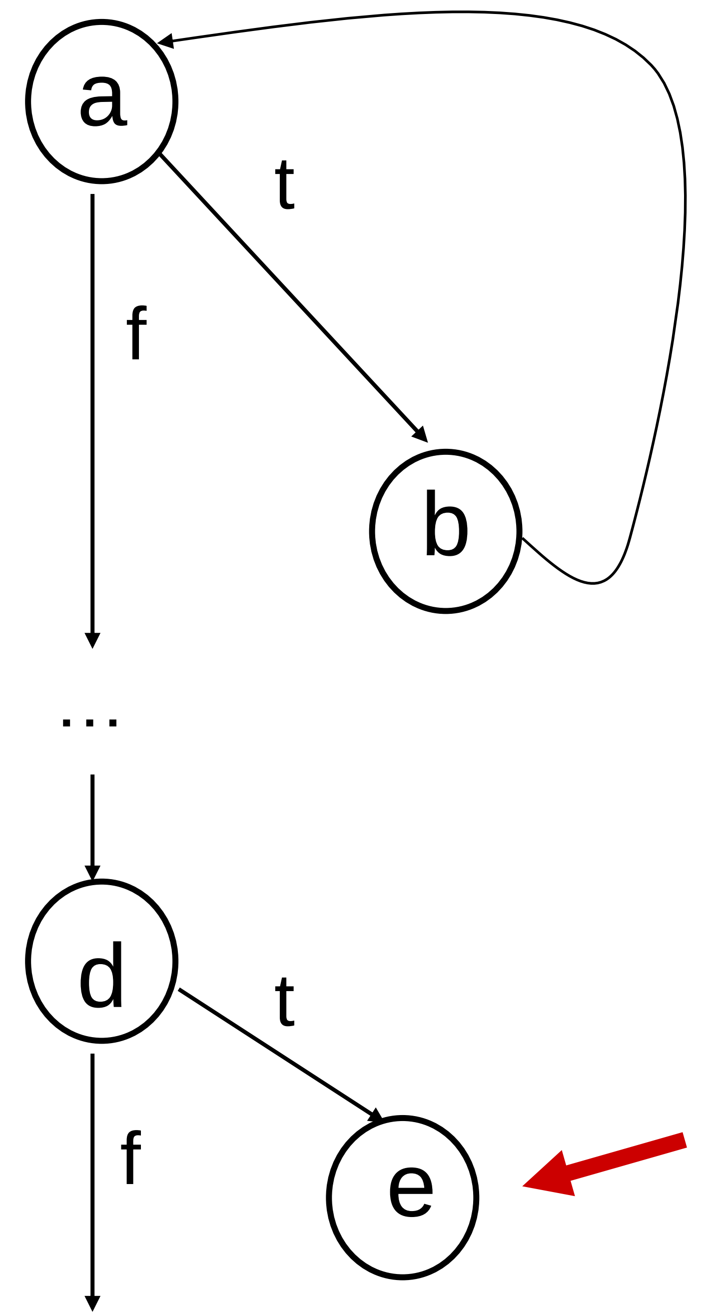
Backtrack !

Even without loops, #paths is exponential with #decisions

Constraint-based program exploration

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

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



1. Constraint model generation

2. Control dependencies generation;
   $j_1=100, \; i_3 \leq 1, \; j_3 > 500$

3. Constraint model solving
   $j_1 \neq j_3$ entailed ➔ unroll the loop 400 times ➔ i_1 in 401 .. 2^{31}-1

No backtrack !

Constraint-based program exploration

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

- Constraint reasoning over control structures

- Requires to build **dedicated constraint solvers**:

  * propagation queue management with priorities

  * specific propagators and meta-constraints

  * structure-aware labelling heuristics
    (Systematic search over finite domains)

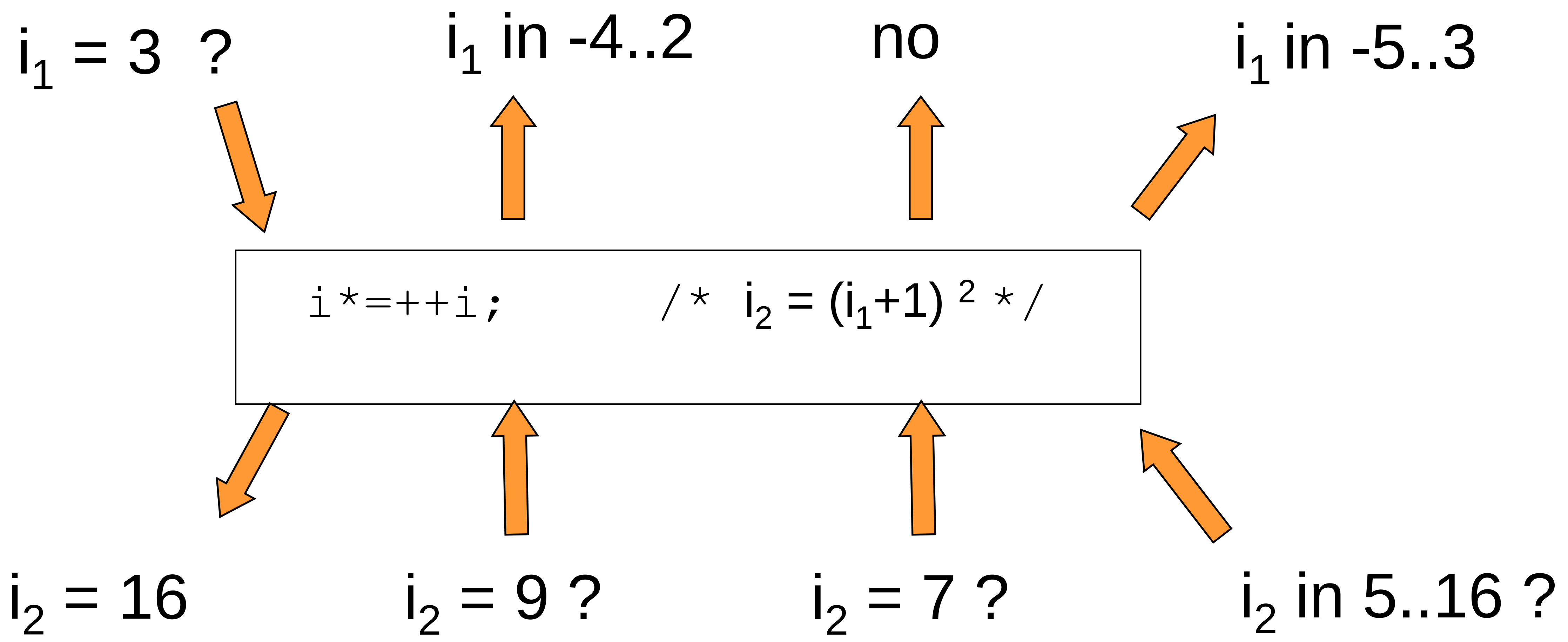
Prototype tools: **Inka** (Gotlieb Botella Rueher ISSTA'98)
Euclide (Gotlieb ICST'09)

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 ;` ⟶ $i_2 = (i_1+1)^2$

Using **bound-consistency filtering over finite domains**:



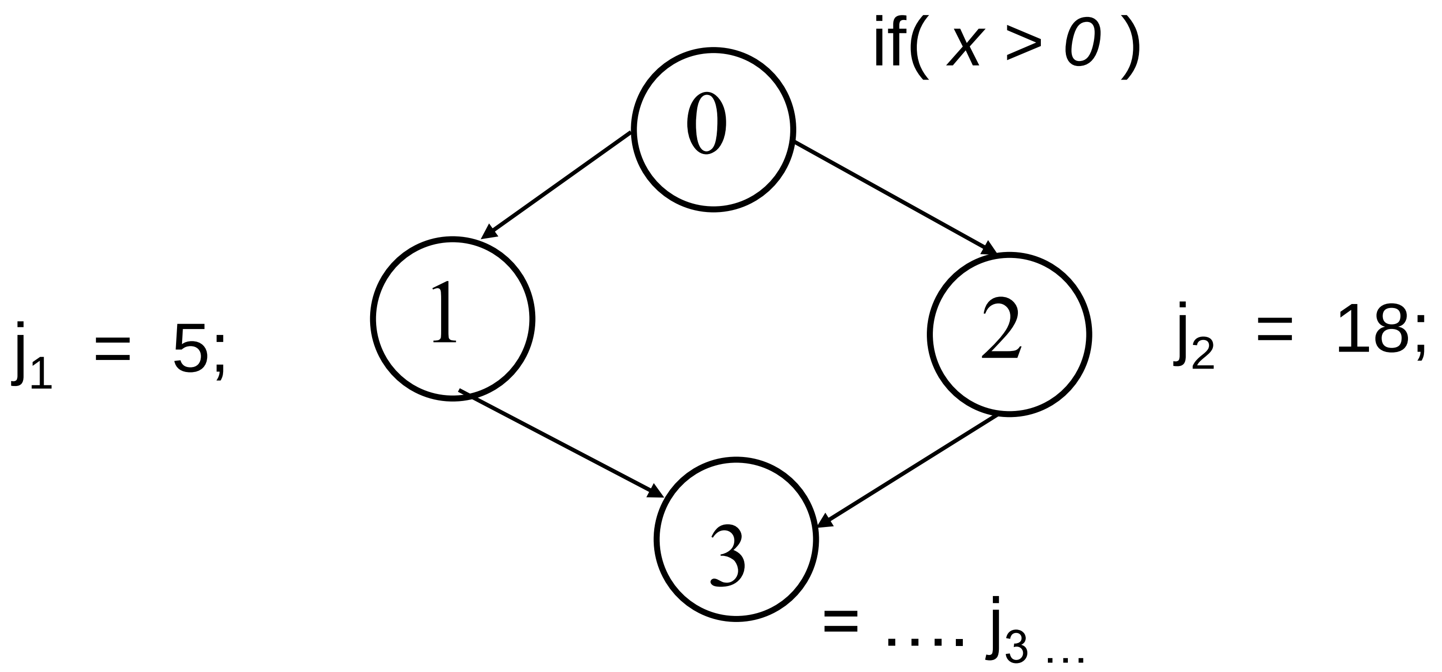
Statements as constraints

- ✓ 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: dedicated meta-constraints
  (interface, awakening conditions and filtering algorithms)

  Conditionnals (SSA) `if D then` C_1; `else` C_2 → ite/6

  Loops (SSA) `while D do C` → w/5

Conditional as meta-constraint: ite/6

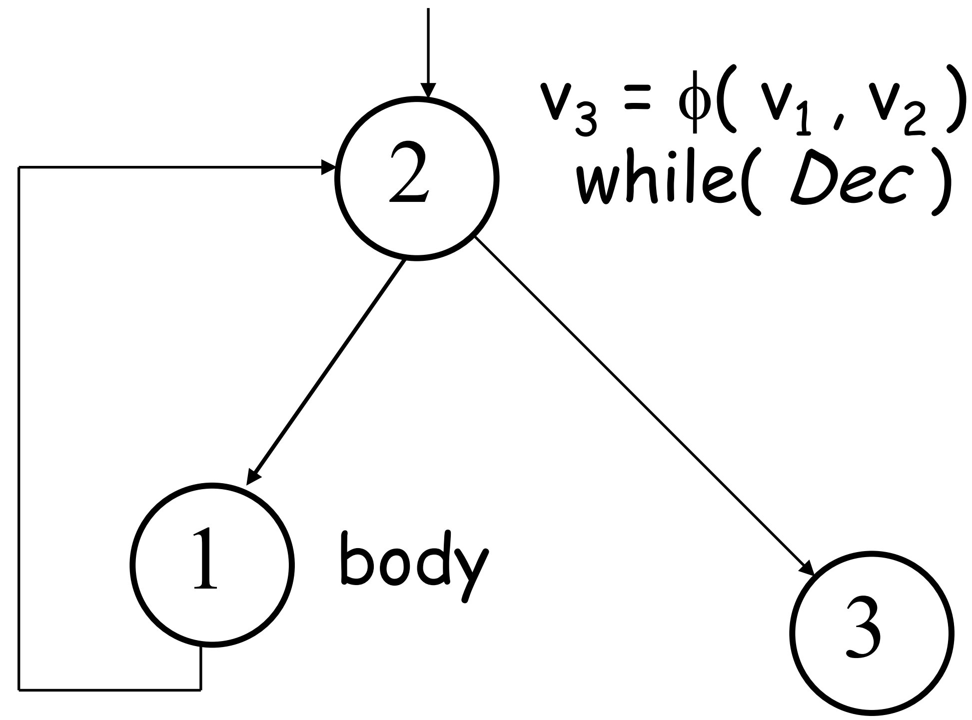


ite($x > 0$, j_1, j_2, j_3, $j_1 = 5$, $j_2 = 18$) iff

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

Implemented as a new global constraint
(interface, awakening conditions, filtering algo.)

Loop as meta-constraint: w/5



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

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

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

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

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

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

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

$i = 23,\ j_1 = 100$?

no

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

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 constraint-based exploration

✓ Special meta-constraints implementation for ite and w

By construction, w is unfolded only when necessary
but w may NOT terminate !
→ only a **semi-correct** test data generation procedure

✓ Join is implemented using ***Abstract Interpretation*** operators
(e.g., interval-based union, weak-join operator, widening in **Euclide**)

✓ Special propagators based on linear-based relaxations
Using **Linear Programming over rationals** (i.e., Q_polyhedra)

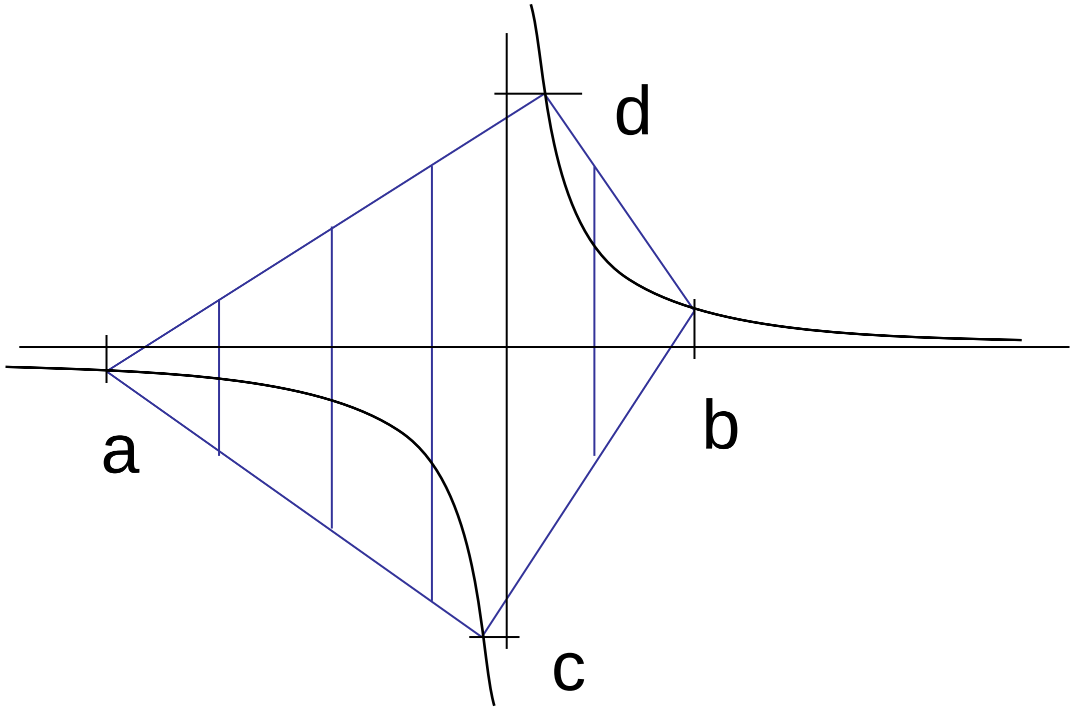
Abstraction-based relaxations

Abstraction-based relaxations

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

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

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



→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*)

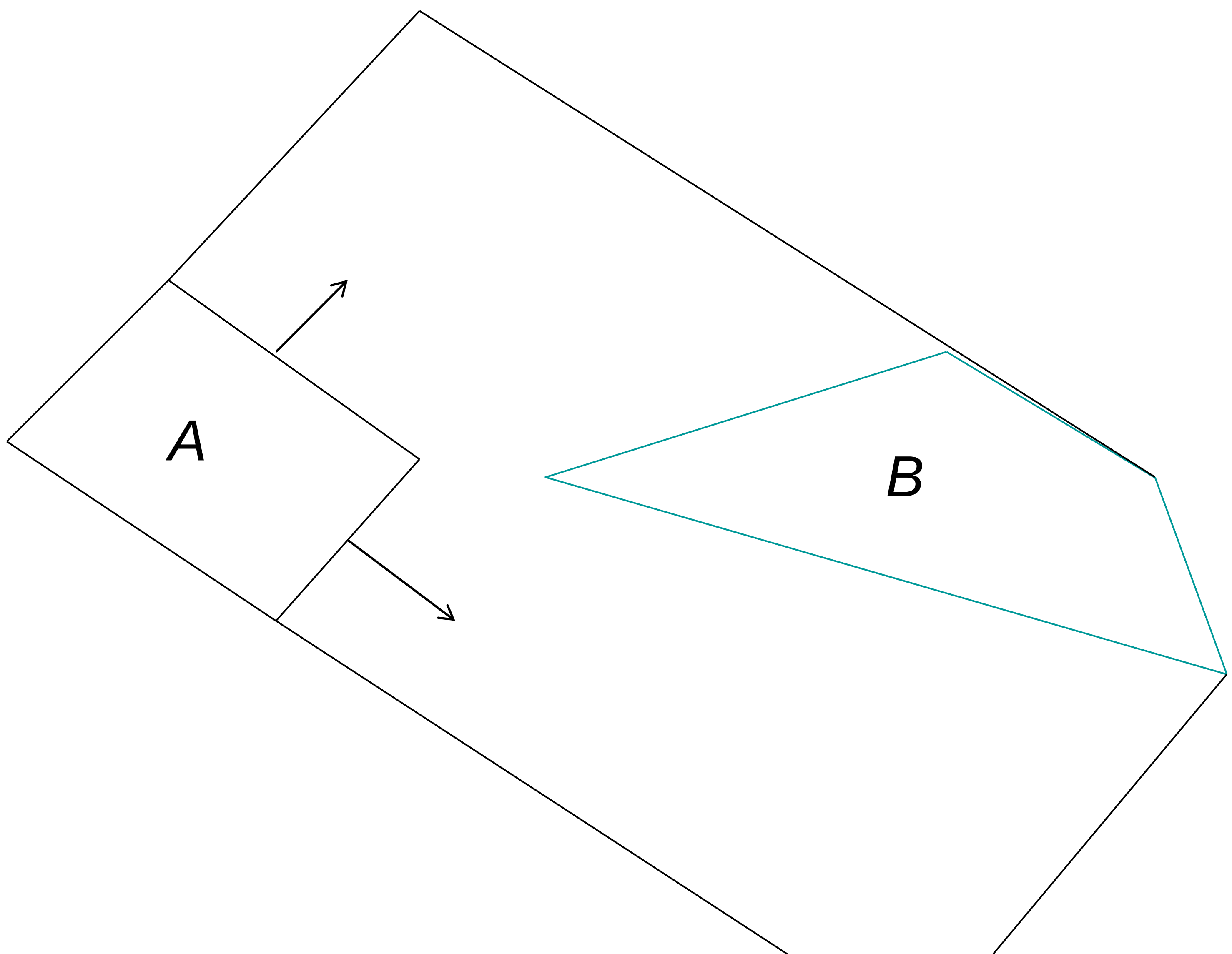
→ Q-Polyhedra in **Euclide**, implementing **Dynamic Linear Relaxation,** propagation queue with priorities

Abstraction-based relaxations: weak-join operator

(Sankaranarayanan et al. VMCAI'06)

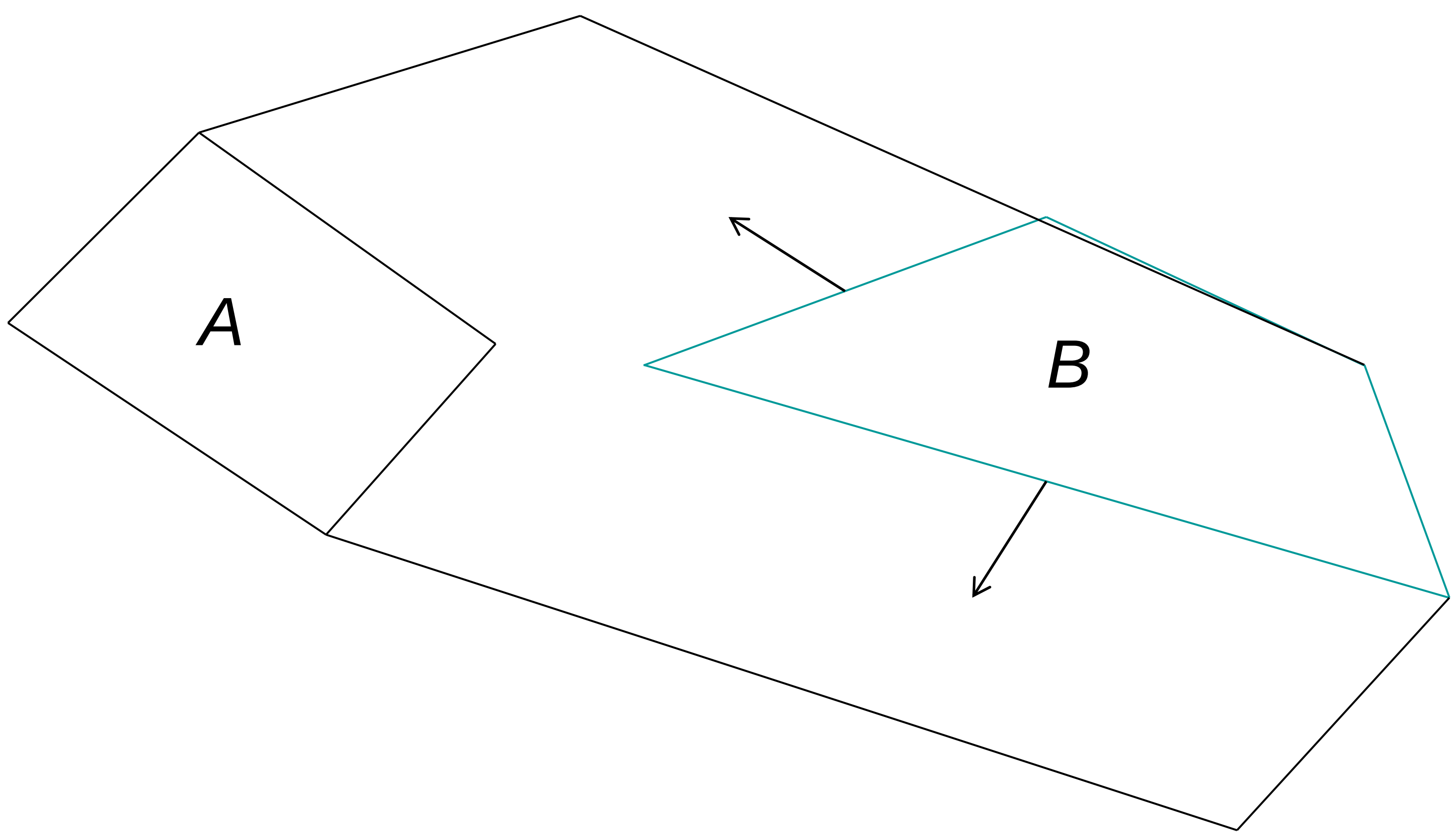
Join operations can be realized by convex hull, but usually too costly !

In Euclide, we took advantage of the weak-join of Q_polyhedra
(based on simplex calculations)



Abstraction-based relaxations: weak-join operator

(Sankaranarayanan et al. VMCAI'06)



Abstraction-based relaxations: weak-join operator

(Sankaranarayanan et al. VMCAI'06)

Weak_join operator

The disjunction: $\{g_1^i(x) \geq c_1^i\}_{i \in I} \vee \{g_2^i(x) \geq c_2^i\}_{i \in I}$

$x = (x_1, .., x_n)$, where $x_i \in Z$

Weak_join: $\alpha_1 \quad = \text{Minimize} \quad g_1^1(x)$ subject to $\{g_2^i(x)\}_{i \in I}$

...

$\alpha_p \quad = \text{Minimize} \quad g_1^{card(I)}(x)$ subject to $\{g_2^i(x)\}_{i \in I}$

$\alpha_{p+1} \quad = \text{Minimize} \quad g_2^1(x)$ subject to $\{g_1^i(x)\}_{i \in I}$

...

$\alpha_{2p} \quad = \text{Minimize} \quad g_2^{card(I)}(x)$ subject to $\{g_1^i(x)\}_{i \in I}$

$g_1^1(x) \geq Min(\alpha_1, c_1^1),$

...

$g_2^{card(I)}(x) \geq Min(\alpha_{2p}, c_2^{card(I)})$

Constraint-based program exploration

- Handles loops in constraint-based test data generation, without bounding the number of iterations ;

- Useful for reaching a particular uncovered location in the code (complement an existing test set generated by « systematic » path-exploration)

- Use of the global constraint interface in clpfd to implement w, or dedicated solver (propagation queue management)

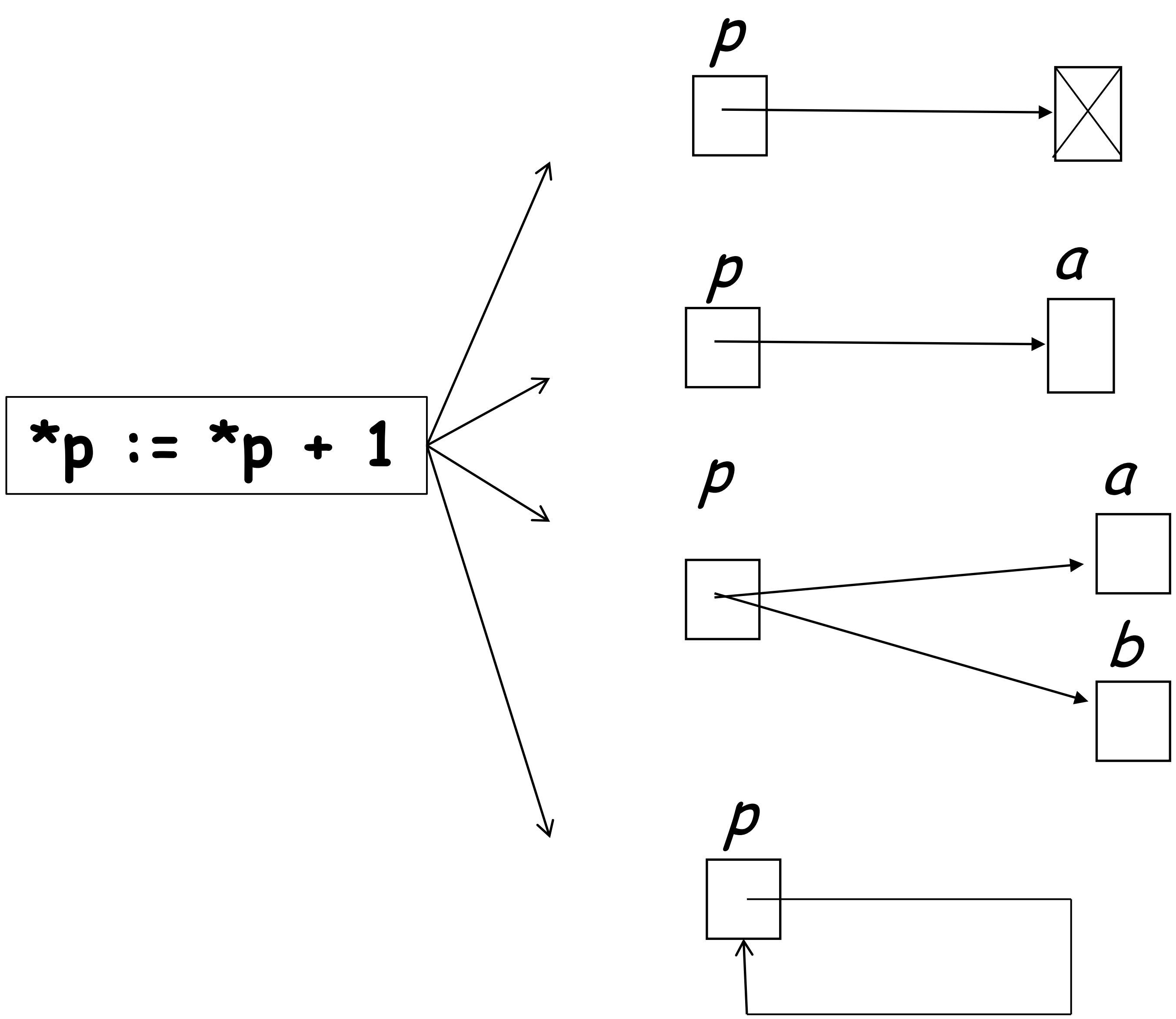
- May not terminate, timeout needed!

Foundations of the approach **(Gotlieb Botella Rueher ISSTA'98,SEN'98,CL'00)**
Abstraction-based relaxation **(Denmat Gotlieb Ducassé ISSRE'07)**
Global constraint w, extended with widenning **(Denmat Gotlieb Ducassé CP'07)**
Euclide: A Constraint-based testing platform for C **(Gotlieb ICST'09)**
Application on the TCAS case study **(Gotlieb KER Journal 2012)**

Constraints over Memory Model Variables for testing pointer programs

Constraints over memory models: aliasing problems

How to apply constraint-based reasoning over statement like *p := *p+1 ?



Then **fail** or **exception**

Then $\mathbf{a_2 = a_1+1}$

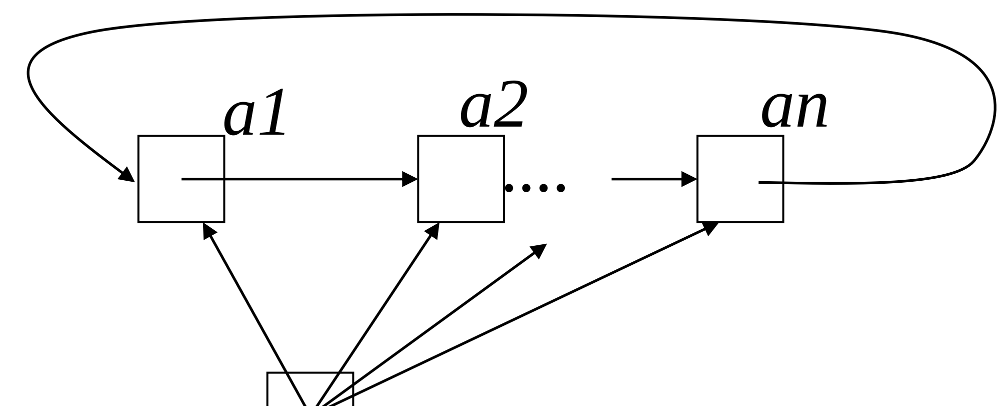
Then $\mathbf{a_2 = a_1+1}$ or $\mathbf{b_2 = b_1+1}$

Then $\mathbf{p_2 = p_1+1}$, meaning that p now refers to the next memory location

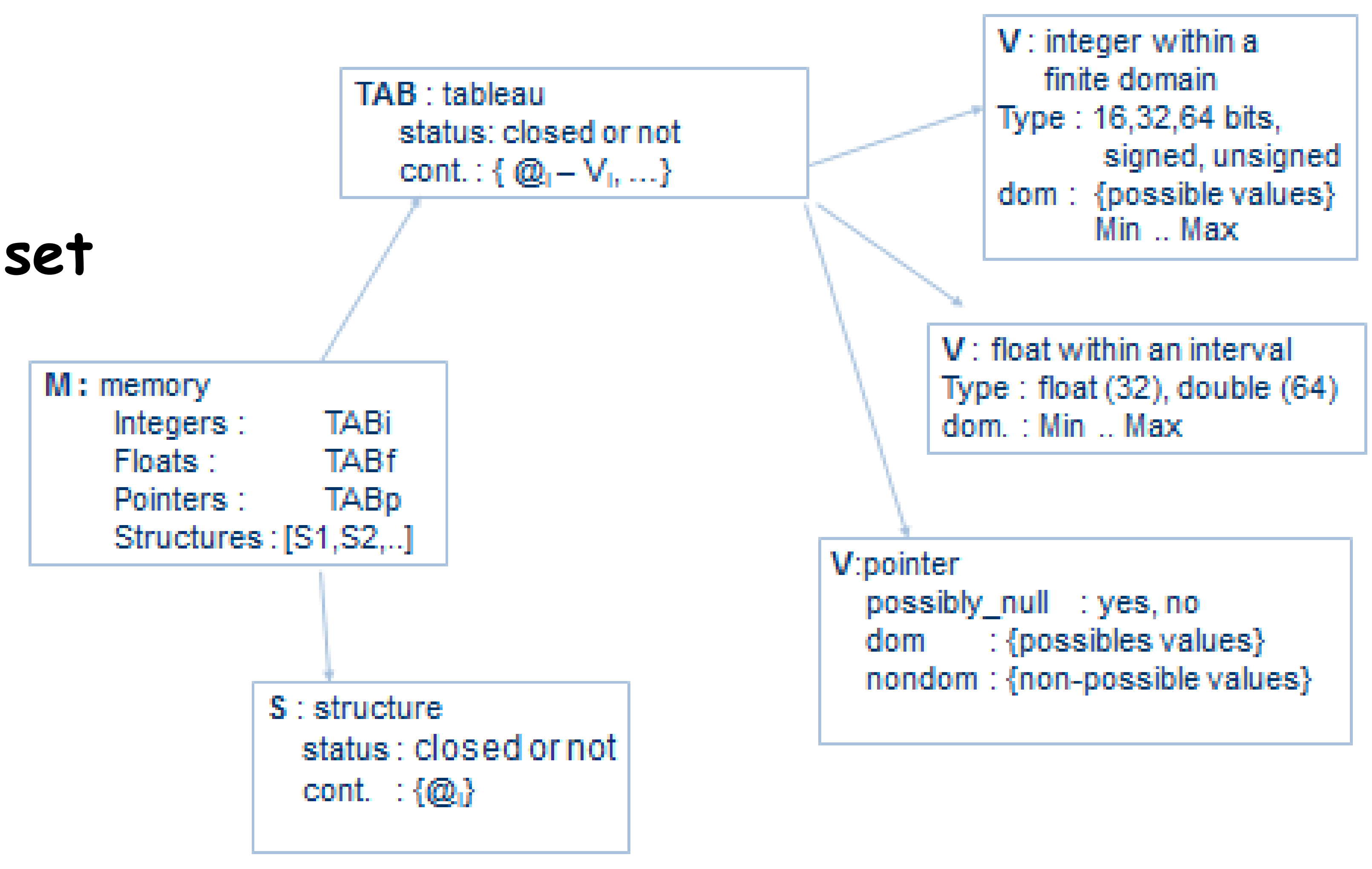
Our propositions

How to represent abstract memories and to reason on them ?

1) Constraint reasoning over
Memory, as a set of graphs
(Gotlieb et al., ASE'05, IST 2007)



2) Constraint reasoning over
Memory, as a structured set of unbounded arrays
(Charreteur et al., JSS 2009)



Weaknesses of our first memory model

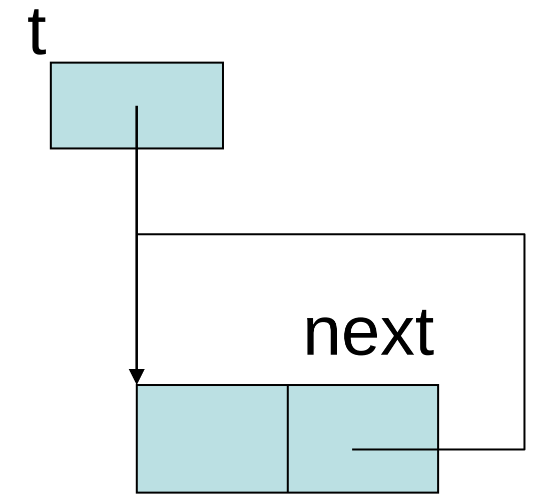
- Requires a preliminary points-to analysis that may be too imprecise when dynamic (de-)allocation is involved

- Pointers as function inputs, can point to anything on the heap

- Some conditions may constrain the shape of dynamic data structures. How to handle this in a constraint solver ?

constrains t to

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



Memory, as a structured set of unbounded arrays

TAB : *tableau*
status: closed or not
cont. : { $@_i - V_i$, …}

V : *integer within a finite domain*
Type : 16,32,64 bits, signed, unsigned
dom : {possible values}
Min .. Max

V : *float within an interval*
Type : float (32), double (64)
dom. : Min .. Max

M : *memory*
Integers : TABi
Floats : TABf
Pointers : TABp
Structures : [S1,S2,..]

V:*pointer*
possibly_null : yes, no
dom : {possibles values}
nondom : {non-possible values}

S : *structure*
status : closed or not
cont. : {$@_i$}

Introducing constraints on memories

- Memories = unknowns representing states (sets of pairs Adress-Value)
- Relations on these unknowns, constraint reasonning on these unknowns

C program	*Constraints store*
i = i + 1 --------->	load_elt(@i, I_1, M_1) $I_2 = I_1 + 1$ store_elt(@i, I_2, M_1, M_2)
*p = 3 ---------->	load_elt(@p, P_1, M_2) $DP_1 = 3$ store_elt(P_1, DP_1, M_2, M_3)
j = i + 2 ---------->	load_elt(@i, I_3, M_3) $J_1 = I_3 + 2$ store_elt(@j, J_1, M_3, M_4)

Constraints on memories

- `new_elt(TYPE, X, V_INIT,  M0, M1, ENV)`
- `delete_elt(TYPE, X, M0, M1, ENV)`

- `load_elt(TYPE, X, VALUE, M, ENV)`
- `store_elt(TYPE, X, VALUE, M0, M1, ENV)`

- `M1 = M2` **/* Useful in control structures */**
  - `closed(M)`

  /* Useful to closed the memory during final search */

store_elt(P,V,M1,M2)

Store_elt

M1 :
Status : not closed
Includes :
i – Vi
j – Vj
k – Vk
…

M2 :
Status : not closed
Includes :
i – Vi’
j – Vj’
k – Vk’
…

P :
Domain pointer
{i,j}

V:
Domain Integer
1.. 5

store_elt(P,V,M1,M2)

M1 :
Status : not closed
Includes :
i – Vi → 1.. 2
j – Vj → 5.. 9
k – Vk → 2
…

Store_elt

M2 :
Status : not closed
Includes :
i – Vi’ → 3..6
j – Vj’ → 7..18
k – Vk’ → ?
…

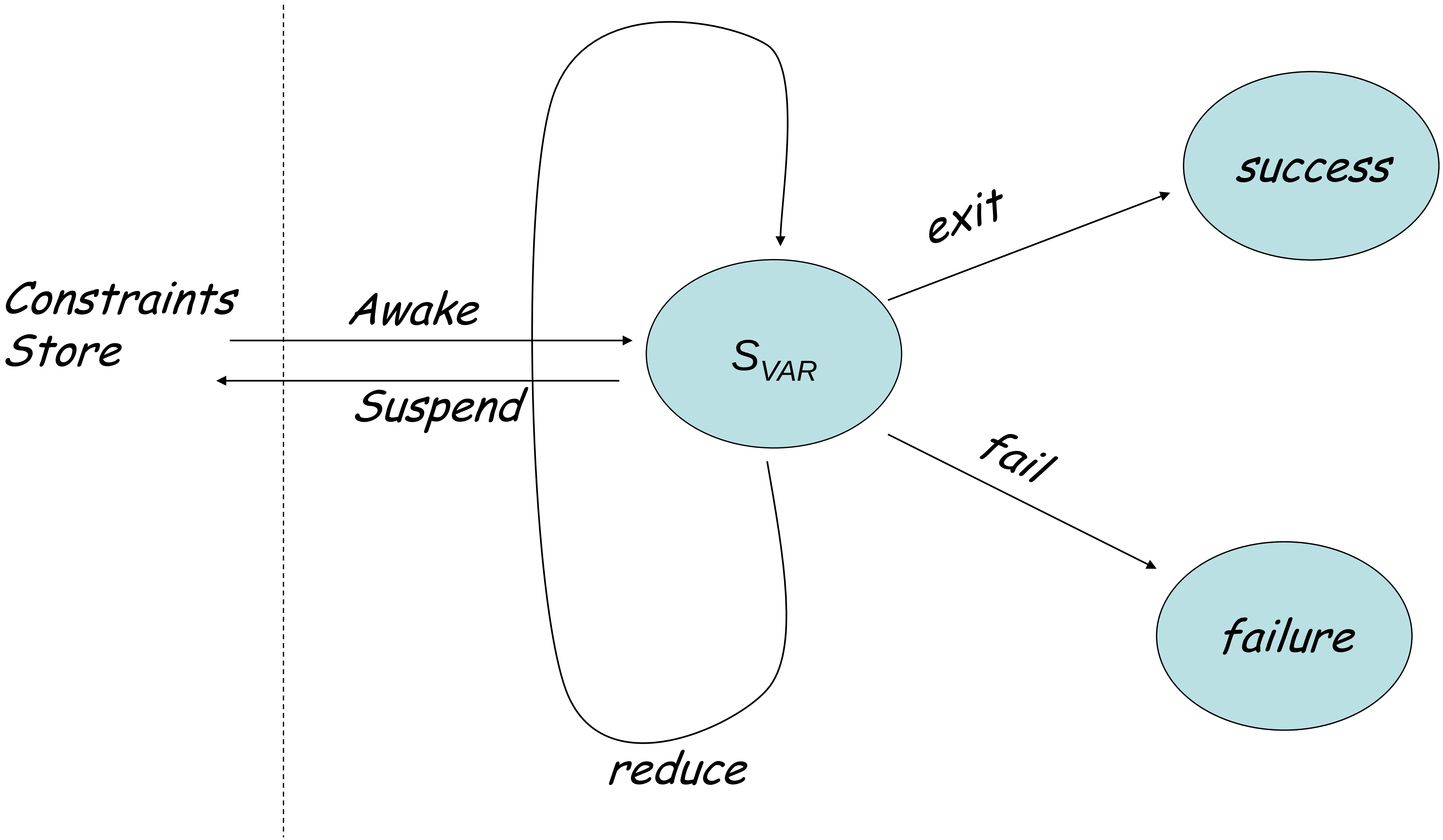
P :
Domain pointer
{i, j}

V:
Domain Integer
1.. 5

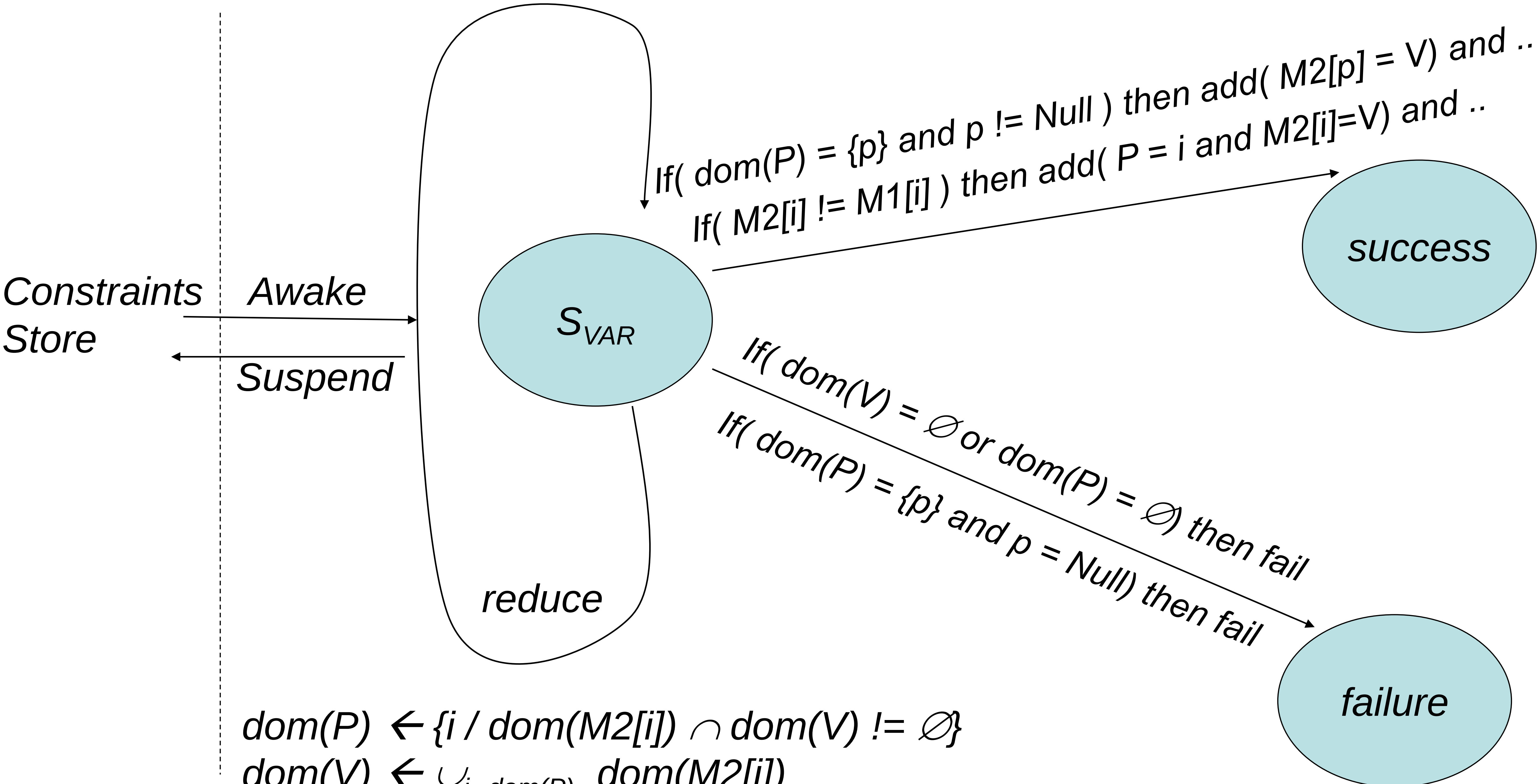
Automatic deductions after the constraint propagation step :

P = i, V = Vi’ in 3..5, Vj = Vj’ in 7..9, Vk = Vk’ =2

Model for the definition of a new constraint



store_elt(P,V,M1,M2)



$dom(P) \leftarrow \{i \,/\, dom(M2[i]) \cap dom(V) \mathrel{!=} \emptyset\}$

$dom(V) \leftarrow \cup_{i \in dom(P)}\ dom(M2[i])$

$dom(M1[i]) \leftarrow dom(M2[i]) \cap dom(M1[i])$ if($i \notin dom(P)$)

$dom(M2[i]) \leftarrow dom(M1[i]) \cap dom(M2[i])$ if($i \notin dom(P)$)

$dom(M2[i]) \leftarrow dom(M1[i] \cup dom(V))$ otherwise

Conclusions

What was left apart in my talk

- **Constraints over floating-point variables**: FPSE Solver
  (Botella Gotlieb Michel STVR 2006, Carlier Gotlieb ICTAI'11)

- Constraints over modular integers (Gotlieb Leconte Marre ModRef'10)

- Constraints over memory models for Java Bytecode (i.e., with **inhritance** and **virtual method calls**) (Charreteur Gotlieb ISSRE'10)

- **Uniform random generation** of test data in path testing
  (Gotlieb Petit CP'07, JSS'10)

- Explanation-based generalization of **infeasible paths** in *Dynamic Symbolic Execution* (Delahaye Botella Gotlieb ICST'10, TSE in rev)

Applications & Systems

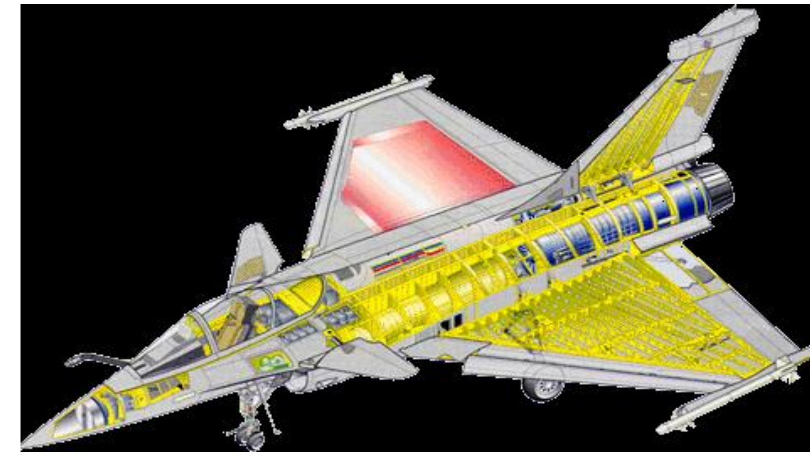
- Applications to the testing of critical embedded software

  - BCE ABE Rafale (2001)
  - Java Card (2004-2005)
  - TCAS SIR (2008)
  - TCAS unmaned planes (2011)

- Development of 4 Research prototype tools :

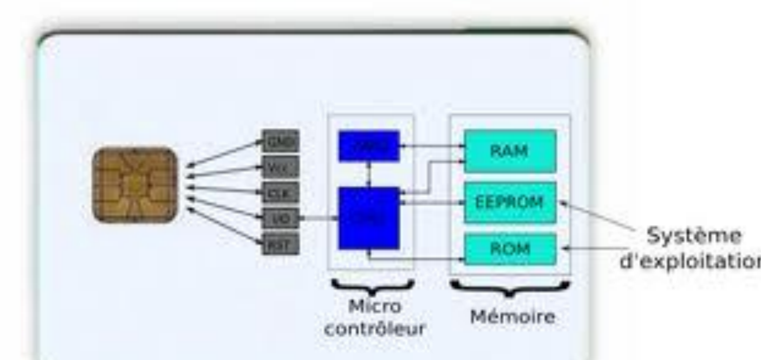
  Inka, Euclide, PRT and FPSE
  (more than 45KLOC Prolog, Java, C, Tcl/Tk)

- Research projects: INKA, DANOCOPS, CASTLES, ACI V3F, ANR CAT/U3CAT, ANR CAVERN...



BCE Rafale – Dassault Electronics

TCAS - Airbus



Java Card - Oberthur

Conclusions

- Emerging concept in code- and model-based software testing
- Constraint Programming techniques offers:

  - **Global constraint design**

  - **disjunctive** constraint programs in a constructive way.

  - Time-aware optimization through branch&bound is given for free

  - but **unsatisfiability detection has to be improved** **(e.g.,** by combining techniques SMT/CP)

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

Further work

- **Array constraint solving. (More global reasonning are required!)**

  A combined SMT/CP approach for solving constraints with arrays and arithmetics. Constraint solver CCFD and large experimental validation over random formulas.

  joint work with S. Bardin from CEA

- Improving constraint-reasoning over function calls, modelling function calls as global constraints

- Dedicated labelling search, exploiting the structure of the programme

- PhD students

  Tristan Denmat,
  Matthieu Petit,
  Florence Charreteur,
  Mickael Delahaye,
  Nadjib Lazaar,
  Aymeric Hervieu

Thank you!

- Post-doc

  Sandrine Gouraud, Pierre Rousseau, Matthieu Carlier

- Co-authors

  Olivier Lhomme, Michel Rueher, Claude Michel, Yahia Lebbah, Michel Leconte, Mireille Ducassé, Bernard Botella, Patrick Taillibert, Franck Calvet, Bruno Marre , Benjamin Blanc, Frédéric Dadeau, Nicky Williams, Catherine Dubois, Patrick Bernard, Matthieu Wattel, Benoit Baudry, Sébastien Bardin, Lionel Briand