

## INKA: Ten years after the first ideas

Arnaud Gotlieb  
Lande project team  
IRISA-INRIA

(In collaboration with Bernard Botella  
from Thales Aerospace)

### INKA: A constraint-based white-box testing tool

- Automated white-box testing:  
Test data selection based on a code-based objective  
(reach a statement, a branch, a path, a def/use pair, etc.)  
or a coverage criterion  
(*all\_statements*, *all\_decisions*, MC/DC, etc.)
- A constraint-based approach:  
Converts the test objective and the program under test into a constraint model and exploits constraint programming techniques to find the test data

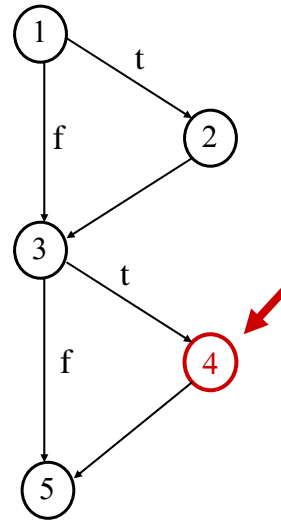
### A trivial example

```

int f( int i )
{
1a.  j = 2
1b.  if( i ≤ 16 )
2.    j = j * i
3.    if( j > 8 )
4.      j = 0
5.  return j
}

```

Find a value of  $i$   
such as  
statement 4  
is executed ?



### Intuition of our approach

```

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

  if( j > 8 )
    j = 0

  return j
}

```

$$\begin{aligned}
 &j > 8 \\
 &i > 16 \Rightarrow j = 2 \\
 &i \leq 16 \Rightarrow j = 2 * i
 \end{aligned}$$

---


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

---


$$5 \leq i \leq 16$$

## The Automatic Test Data Generation problem (Undecidable in the general case)

### Difficulties for classical ad-hoc methods :

✓ Non-feasible paths

✓ 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**

✓ Floating-point computations

✓ Pointers, dynamic structures, function calls, etc.

## INKA: history

1995 -- Start of the Research works (Start of my PhD thesis) in Thales

1996-- "Automatic Test Data Generation using CLP" ICSSEA'96 -1<sup>st</sup> publication

1998-- **Prototype tool for C--** , first experimental results over a (small) realtime embedded program (parts of the BCE – Avion Banc d'Essai Rafale)

2000–02 *RNTL Inka* (Thales, Axlog, 3 academics labs I3S, LIFC, LSR)

2002-- **INKA V1** (Principal investigator: me) – (Validation: SA RTOS Rafale)

2003–06 *RNTL Danocops* (Thales, Axlog, 3 academics labs I3S, LIFC, LSR)  
*ACI V3F project* (floating-point numbers)  
( INRIA Cassis, Coprin, Lande/Vertecs, CEA, Thales)

2005 – **INKA V2** in progress (Principal investigator: B. Botella from Thales)

2006 – (Re-)start of Research works in the INRIA's Lande team

## INKA V2: current scope

### **LANGUAGE: a small subset of the C/C++ language**

All ANSI Integer data types (`bool`, `char`, `short`, `long`, `unsigned short`, ...)  
All control structures and almost all operators  
Some IEEE-754 Floating-point data type (`float`, `double`)  
Structures and «arrays/strings» with restrictions  
Function calls and some method calls (including static operator/method overloading)  
Pointers and references (including dynamic allocation and deallocation)  
Inheritance and some static type casting (implicit and explicit)  
Namespace

### **But:**

No `break/continue/goto`, No exceptions  
No unconstrained pointer arithmetic  
No function pointers, no «`const`», no (`void *`) (2<sup>nd</sup> order programming),  
No dynamic C++ features (virtual method, templates, dyn. op. overloading, `typeid`,...)  
No library function/method calls, no external calls  
No function with an unknown number of parameters  
No memory type casting (ex: `int * → float *`): requires a bit-to-bit reinterpretation of the physical memory, no fields of bits

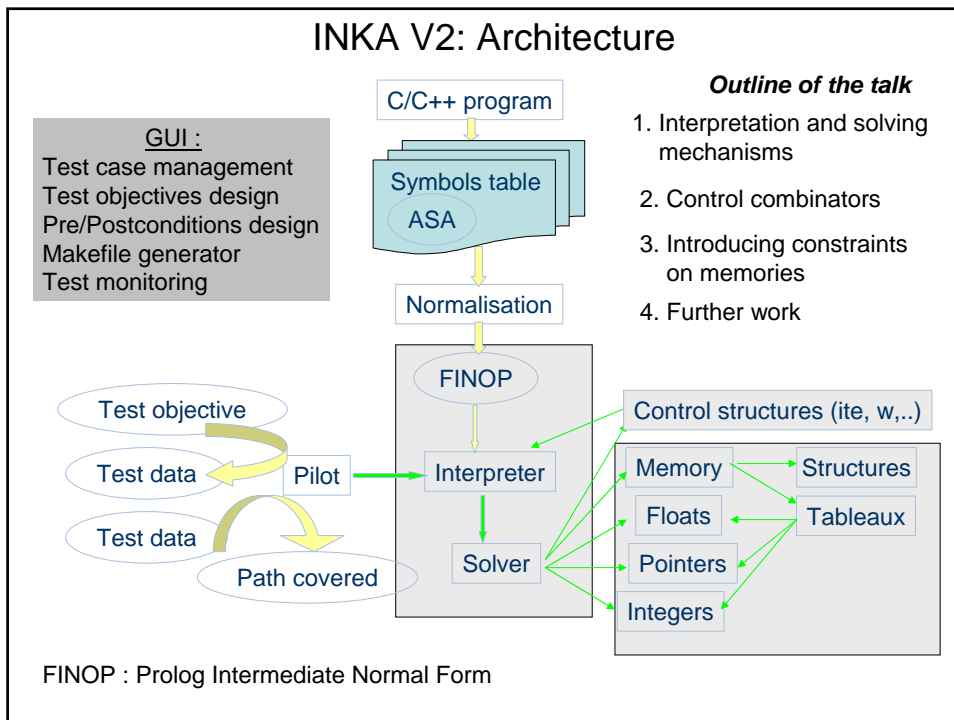
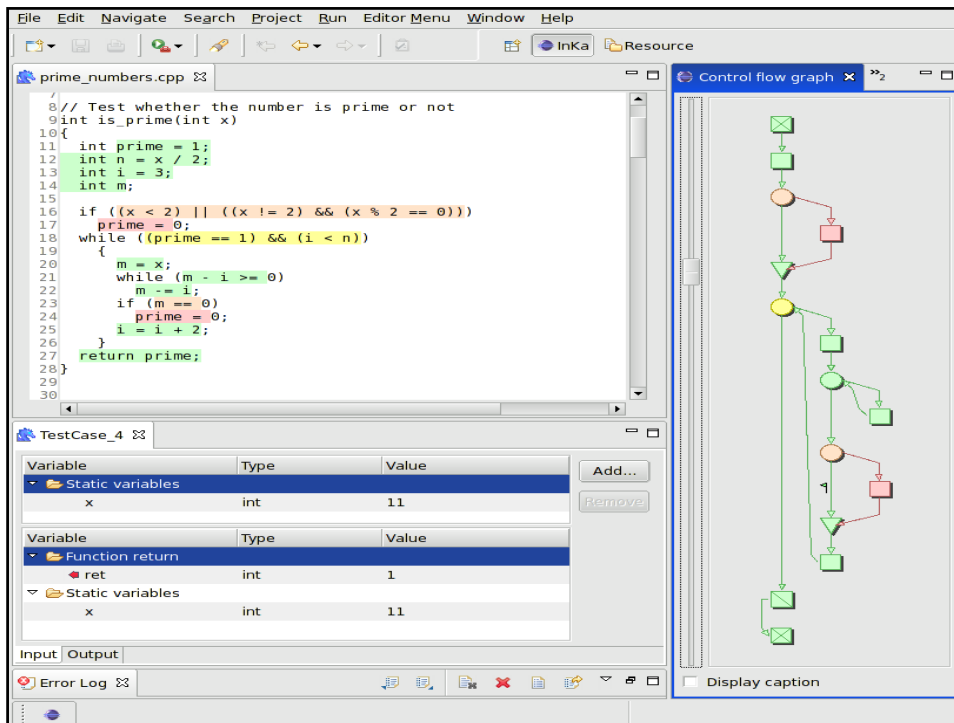
## INKA V2: features

### **MAIN:**

- Coverage monitoring (on the constraint model) = simulated execution
  - Automatic test data generation for a given block of statements or decision
- Automatic test data generation for *All\_statements*, *All\_decisions*, and *MC/DC*

### **ADDITIONAL:**

- Automatic detection of (some) non-feasible block or decision
- Partial verification via automatic refutation of (numeric) properties



## INKA V2: FINOP

- FINOP (Prolog Intermediate Normal Form)
  - Small instructions set (i.e. arrays → pointers)
  - New temporary variables (complex statements are broken)
  - Reification of control statements (decision variables only)
  - Explicit allocations and casting (définition/undéfinition)
  - Numbering of control statements, allocations, function calls, etc.
  
- INTERPRETER
  - Adds memory references to each interpreted statement
  - Adds a test objective to each interpreted statement
  - Computes the path followed within a control structure

NB: *if\_then\_else*, *while\_do*, ... contain FINOP statements that are interpreted during statement evaluation

### Parts of the C FINOP's grammar:

```

Structure ::= defstruct(Ident, [ { Descr_type }* ] )
Program ::= { Function | Method }*
Function ::= fct(Ident, { Formal_parameters }*, Formal_return, Body)
Body ::= [ { Statement }* ]

Statement ::= Declaration | Control | Alloc | Dealloc | Assign | Assert | Sp_call
Declaration ::= def(UID, Var, VName, Type, { Expression }*)
Control ::= ite(UID, Var, Body, Body) | w(UID, Var, Body) | dow(UID, Var, Body)
           | sw(UID, Var, (Label)*, (Body)*, Body)
Alloc ::= new(UID, Ident, BloclD, Type, Expression)
Dealloc ::= delete(UID, Ident) | undef(UID, Var)
Assign ::= assign(UID, Ident, Expression)
Assert ::= test(UID, LogicalExp)
Sp_call ::= fct_call(UID, Ident, { Ident }*, Ident)

LogicalExp ::= Var | LogicalExp && LogicalExp, | LogicalExp || LogicalExp
Expression ::= Ident | &(Ident) | Un_op(SimpleExpr) | Bin_op(SimpleExpr, SimpleExpr)
Type ::= entier(Tint(size)) | flottant(Tfloat(size)) | structure(TName) | pointer(Type)
Tint ::= char | int | short | long | uint | ushort | ulong | uchar | bool
Tfloat ::= float | double
Ident ::= Var | *Var | Var->Nfield | Var.Nfield
Un_op ::= ! | - | ~ Bin_op ::= == | != | <= | < | >= | > | & | '|' | ^ | && | || | + | - | / | % | *

```

# INKA V2: several collaborating solvers

- Constraint library over the pointers domain
  - Constraint library over the floats domain
  - Constraint library over structures accesses and updates (struct + class)
- 
- Constraint library over memories, over tableaux from the memory
- 

**All these constraints are managed through a common propagation queue (called Agenda) that propagates the constraints**

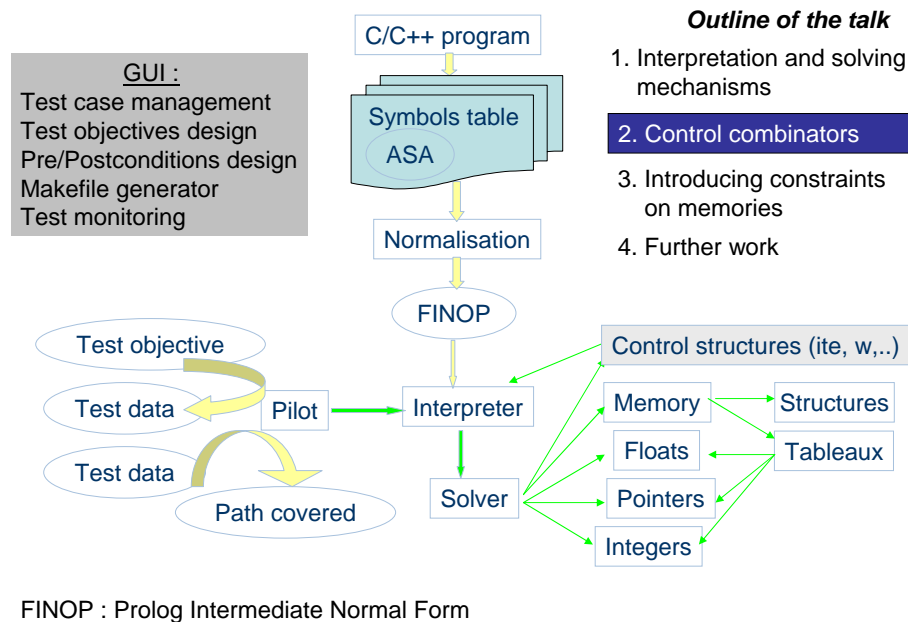
---

- Collaborates with `clp(fd)` for constraints on integers
- 

## Implementation:

- Environment (agenda, ctrs\_network, timeout, K\_flag, epsilon, ...)
- Variables (sort, associated\_ctrs, variables\_of\_the\_neighborhood, ...)
- Constraints (relation, in\_agenda, ignore, associated\_vars, ...)

## INKA V2: Architecture



## 2. Control combinators

☞ Set of guarded-constraints with « don't care » non-determinism  
 $\{ C_1 \rightarrow C'_1, \dots, C_n \rightarrow C'_n \}$

☞ Operational semantic:  
 (with store  $\sigma$ : conjunction of domain constraints)

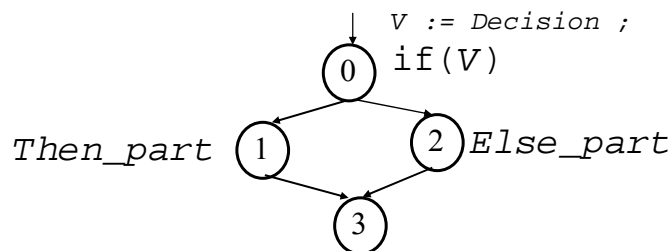
-If  $C_i$  is entailed by  $\sigma$  then  $C'_i$  is pushed on the propagation queue  
 and  $\{ C_j \rightarrow C'_j \}_{j \neq i}$  are all removed from the queue

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

-Else  $C_i \rightarrow C'_i$  is suspended and would be awakened whenever at least one of its  
 variable domains is modified

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

## Conditional: the combinator *ite*



$\text{ite}(V, C_{\text{THEN}}, C_{\text{ELSE}}, M_{\text{IN}}, M_{\text{OUT}}) :-$

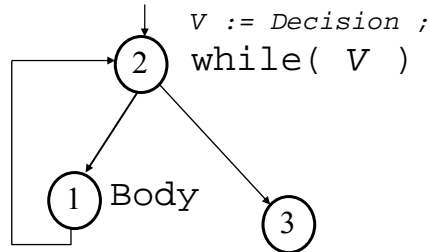
- $V=1 \rightarrow C_{\text{THEN}} \wedge M_{\text{OUT}} = M_{\text{THEN}}$
- $V=0 \rightarrow C_{\text{ELSE}} \wedge M_{\text{OUT}} = M_{\text{ELSE}}$

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

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

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

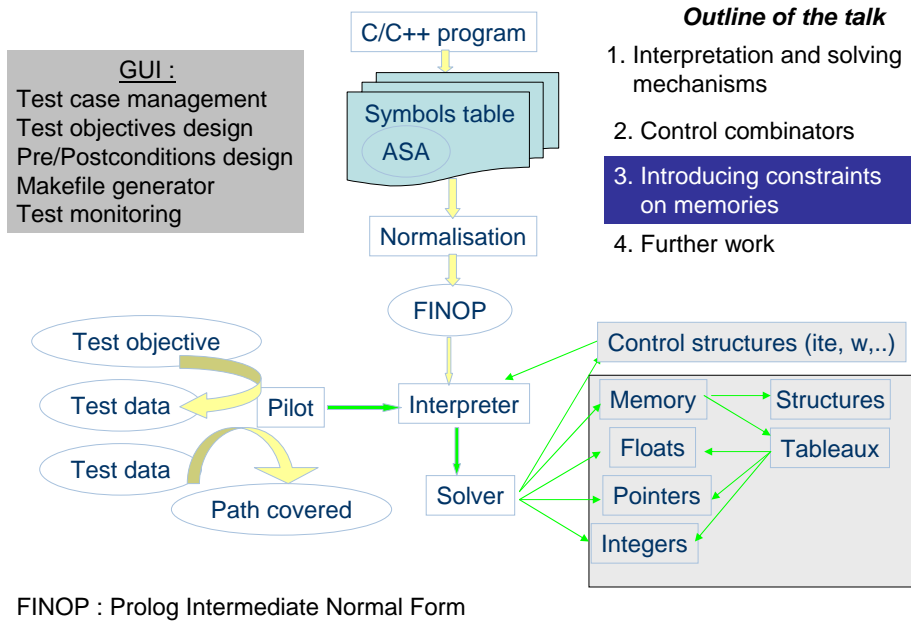
## Iteration: the combinator w



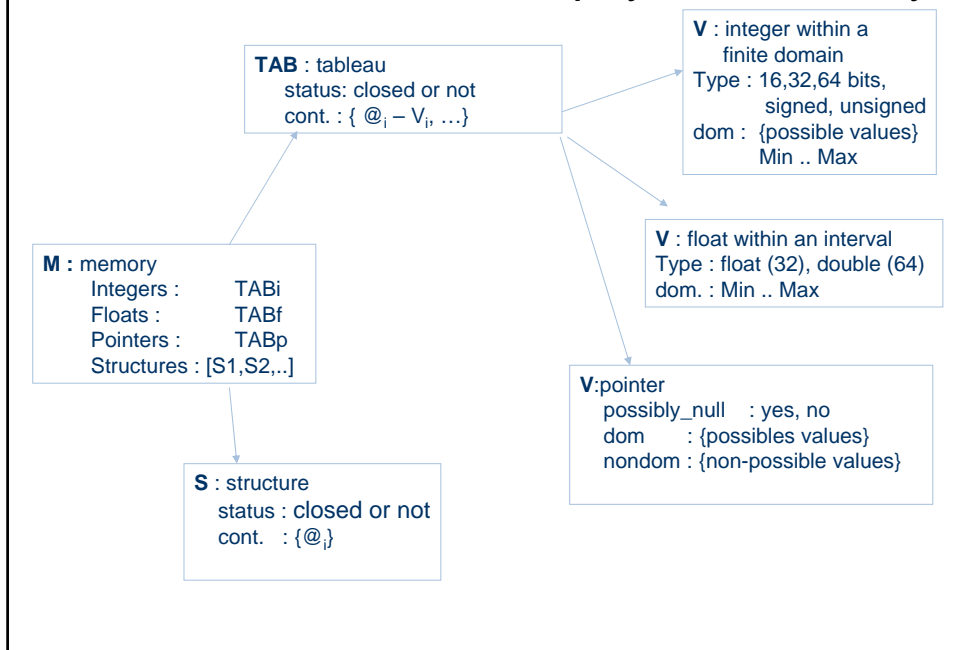
$w(V, C_{\text{BODY}}, M_{\text{IN}}, M_{\text{OUT}}) :-$

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

## INKA V2: Architecture



## An abstract model of the physical memory



## INKA V2: Introducing constraints on memories

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

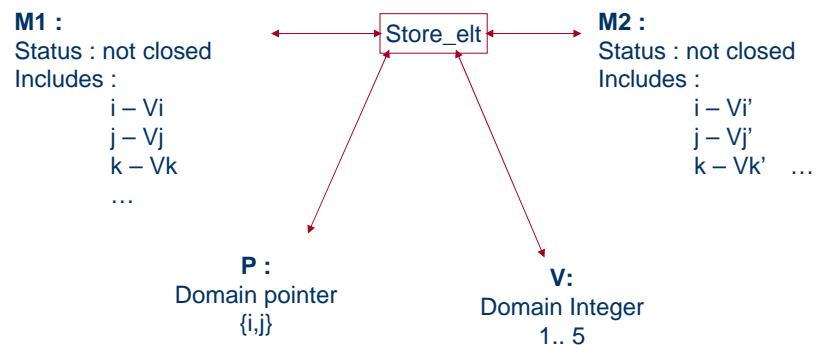
C program

Constraints store

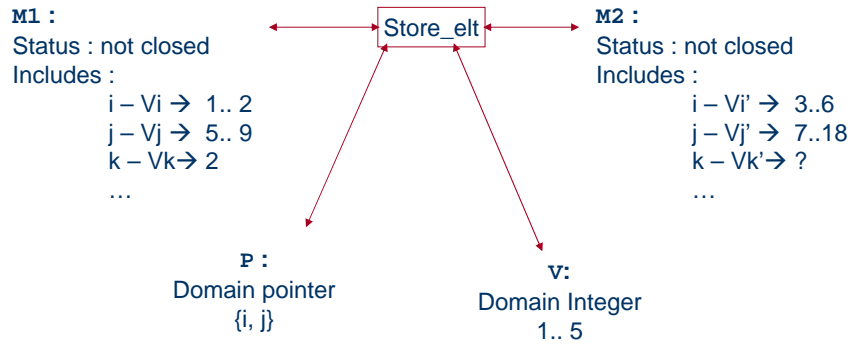
## Constraints on memories

- `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)`



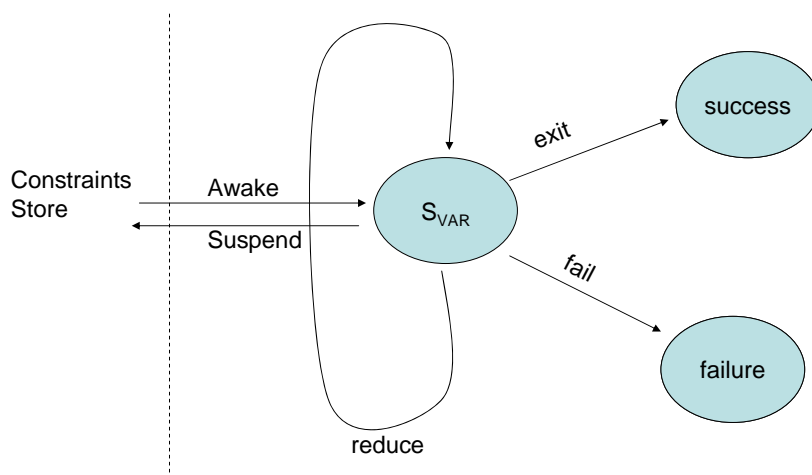
## store\_elt(P,V,M1,M2)

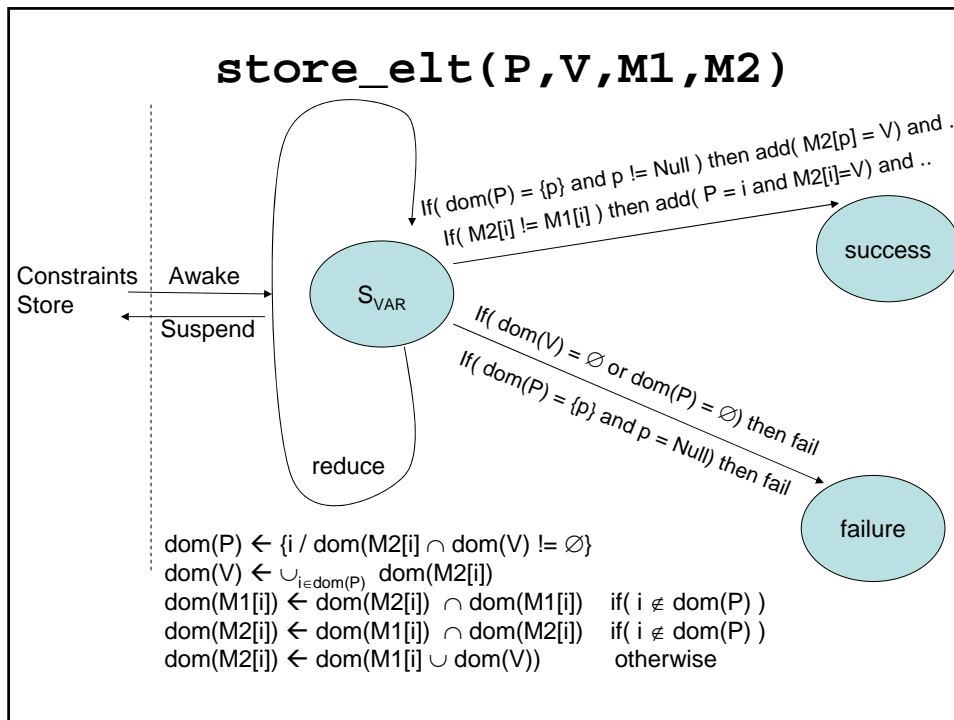


**Automatic deductions after the constraint propagation step :**

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

## Metamodel for the definition of a new constraint





## Constraints over the integers domain

- **new**(X, TYPE, ENV)
  - **affiche**(X)
  - **get\_domaine**(X, DOM)
  - **set\_domaine**(X, DOM)
  - **empty\_domaine**(DOM)
  - **intersection\_domaine**(D1, D2, DI)
  - **union\_domaine**(X, LY, DX)
- 
- **affect**(X, Y)
  - **affect(const(ATOME), X)**
  - **affect(in(Min, Max), X)**
  - **affect('-', X, Y)**
  - **affect('~', X, Y)**
  - **affect('!', X, Y)**
  - **affect(conv(double, long), A, R)**
  - **affect(conv(float, long), A, R)**
  - **affect(conv(signed, unsigned), A, R)**
  - **affect('==', X, Y, B)**
  - **affect('!=', X, Y, B)**
  - **affect('<=', X, Y, B)**
  - **affect('<', X, Y, B)**
  - **affect('>=', X, Y, B)**
  - **affect('>', X, Y, B)**
  - **affect('&', X, Y, Z)**
  - **affect('|', X, Y, Z)**
  - **affect('^', X, Y, Z)**
  - **affect('&&', X, Y, Z)**
  - **affect('||', X, Y, Z)**
  - **affect('+', X, Y, Z)**
  - **affect('-', X, Y, Z)**
  - **affect('/', X, Y, Z)**
  - **affect('%', X, Y, Z)**
  - **affect('\*', X, Y, Z)**
  - **affect('>>', X, Y, Z)**
  - **affect('<<', X, Y, Z)**

## Constraints over the floats domain

- **new**(X,T,ENV)
  - **get\_domaine**(X,I..S)
  - **set\_domaine**(X,I..S)
  - **empty\_domaine**(DOM)
  - **intersection\_domaine**(D1, D2, DI)
  - **union\_domaine**(X, LY, DX)
- 
- |  |                               |
|--|-------------------------------|
| • <b>affect</b> (A ,R)                             | • <b>affect</b> ('+' ,A,B,R)  |
| • <b>affect</b> ( <b>const</b> (ATOM) ,R)          | • <b>affect</b> ('-' ,A,B,R)  |
| • <b>affect</b> ( <b>in</b> (Binf,Bsup) ,R)        | • <b>affect</b> ('**' ,A,B,R) |
| • <b>affect</b> ('-' ,A,R)                         | • <b>affect</b> ('/' ,A,B,R)  |
| • <b>affect</b> ( <b>conv</b> (float,double) ,A,R) | • <b>affect</b> ('=<' ,A,B,R) |
| • <b>affect</b> ( <b>conv</b> (double,float) ,A,R) | • <b>affect</b> ('<' ,A,B,R)  |
| • <b>affect</b> ( <b>conv</b> (double,long) ,A,R)  | • <b>affect</b> ('>=' ,A,B,R) |
| • <b>affect</b> ( <b>conv</b> (long,double) ,A,R)  | • <b>affect</b> ('>' ,A,B,R)  |
| • <b>affect</b> ( <b>conv</b> (float,long) ,A,R)   | • <b>affect</b> ('==' ,A,B,R) |
| • <b>affect</b> ( <b>conv</b> (long,float) ,A,R)   | • <b>affect</b> ('!=' ,A,B,R) |

## Constraints over the pointers domain

- **new**(X, TYPE, ENV)
  - **affiche**(X)
  - **get\_domaine**(X, DOM)
  - **set\_domaine**(X, DOM)
  - **empty\_domaine**(DOM)
  - **intersection\_domaine**(D1, D2, DI)
  - **union\_domaine**(X, LY, DX)
- 
- |   |   |
|---|---|
| • <b>affect</b> (X ,Y)                  | • <b>affect</b> ('>' , X,Y,B)   |
| • <b>affect</b> ( <b>in</b> (DOM), X)   | • <b>affect</b> ('+' , X,Y,Z) X: pointer, Y:integer, Z:pointer                                      |
| • <b>affect</b> ( <b>const</b> ('0'),X) | • <b>affect</b> ('-' , X,Y,Z) X: pointer, Y:integer, Z: pointer or X pointer, Y pointer, Z: integer |
| • <b>affect</b> ('==' , X,Y,B)          |   |
| • <b>affect</b> ('!=' , X,Y,B)          |   |
| • <b>affect</b> ('=<' , X,Y,B)          |   |
| • <b>affect</b> ('<' , X,Y,B)           |   |
| • <b>affect</b> ('>=' , X,Y,B)          |   |

## Constraints on structures

- **new(s, TYPE\_S, ENV)** Declaration of a structure variable S with void contents and not-closed status
- **new\_s(TYPE\_S, X, S0, S1, ENV)** Definition (allocation) of a reference X over an object of type structure or class TYPE\_S.
- **delete\_s(TYPE, X, S0, S1, ENV)** Deallocation of the object referenced by X
- **access\_s(TYPE, X, Champ, VALUE, S, ENV)** Access to the field « Champ » of the object referenced by X of type structure or class S.
- **S0 = S1** Equality between structures
- **closed(S)**

## INKA V2: Labelling

- Labelling on the contents of the input memory
  - Giving a « shape » to the input memory (pointed objects, structures)
  - Giving values to basic variables and valid references to pointers
- *Labelling on the test objective (reaching a selected element within a loop)*
- *Labelling on paths of the control flow graph*

## Why giving a « shape » to the input memory ?

- The function under test contains pointers or objects including references as inputs: `int f( int *p, struct cell *t)`

- Reaching the selected element requires objects or structures to be created first: `x = t->next->next ;`

-----  
labelling on the possible « shapes » of the input memory

Tries first to avoid creating anything else in the memory and then instantiates all the tableaux

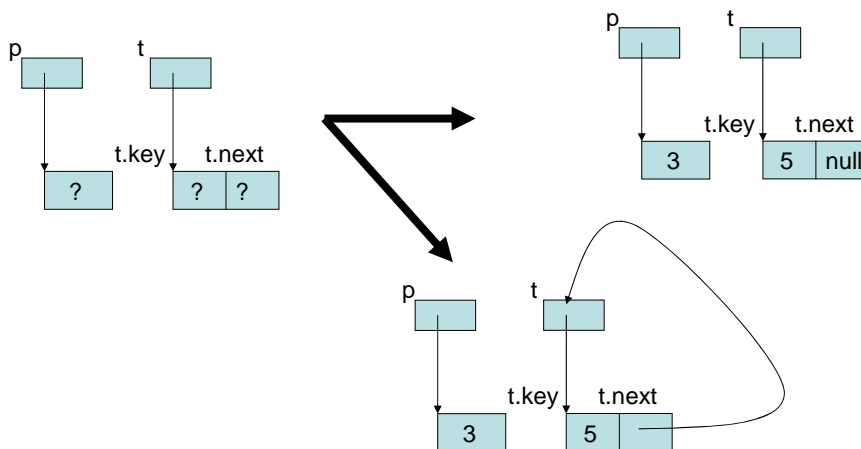


The input memory is then closed, propagating so its shape to any other memories of the program

## Why giving a « shape » to the input memory

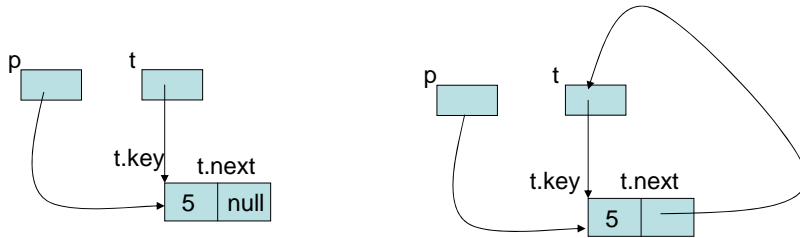
→ labelling on the possible « shapes » of the input memory

- Creates a single object for each pointer and instantiates all the tableaux



## Labelling on the shapes

- Unbounded, requires strong stopping criteria
- Strategy that can produce more objects than strictly required:



haven't been tried !

***DEMO !***

## Current work

- Function calls modeled as constraint combinators  
→ to avoid introducing tones of constraints within the agenda  
(current work of Florence Charreteur)
- Improving constraint refutation by building a solver that combines the advantages of finite domain constraint solving and a linear constraints solver (PPL, clp(Q))  
(PhD thesis of Tristan Denmat)
- Integrating static analyses during constraint resolution. In particular, points-to analyses will reduce the negative effects of defining a statement as a relation between two memories

## Perspectives

- Dealing with (unconstrained) pointer arithmetic. Exploiting dynamic analyses to define a view of the physical memory
- INKA for Java bytecode: constraints over the stack of operands (accesses and updates), virtual methods calls (bytecode `invokevirtual`)
- *Foundations of constraint-based testing: relational semantics*