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

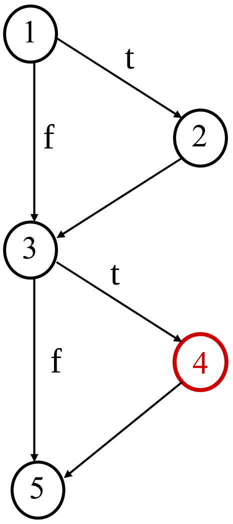
A trivial example

```
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
 }
```

$j > 8$

$i > 16 \Rightarrow \cancel{j = 2}$

$i \leq 16 \Rightarrow j = 2 * i$

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

$5 \leq i \leq 16$

The Automatic Test Data Generation problem

(Undecidable in the general case)

Difficulties for classical ad-hoc methods :

- ✓ Non-feasible paths
- ✓ Highly combinatorial $f\,(int\ \ x_1,\ \ int\ \ x_2,\ \ int\ \ x_3)\ \ \{\ ...\ \}$

2^{32} possibilities × 2^{32} possibilities × 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 -1st 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 *`) (2nd 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

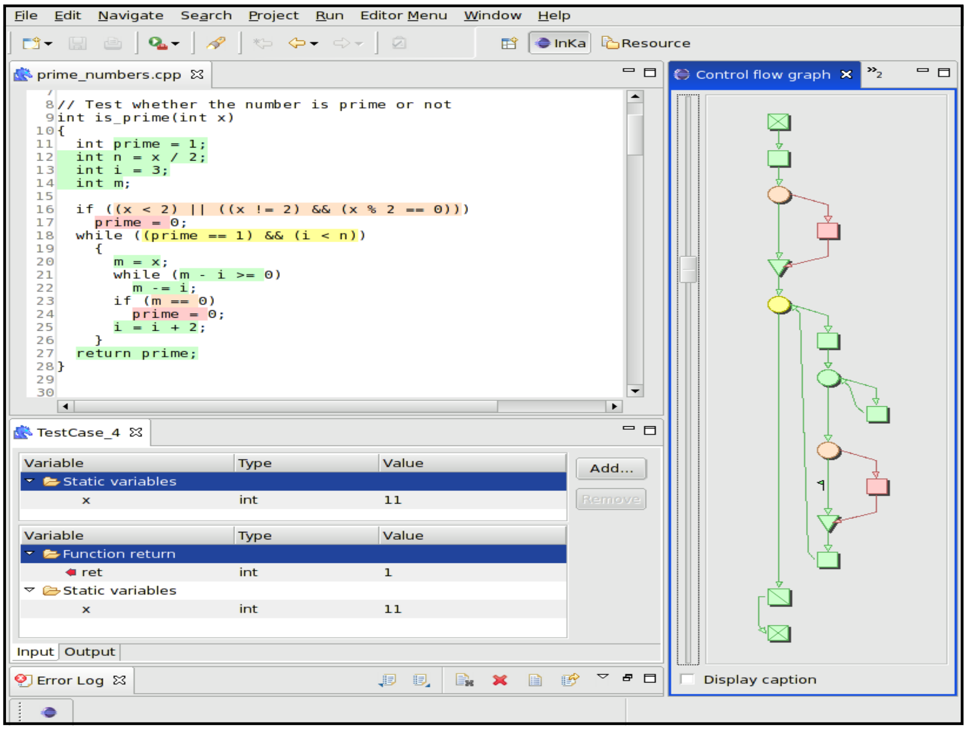
- Automatic test data generation for a given block of statements or decision

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



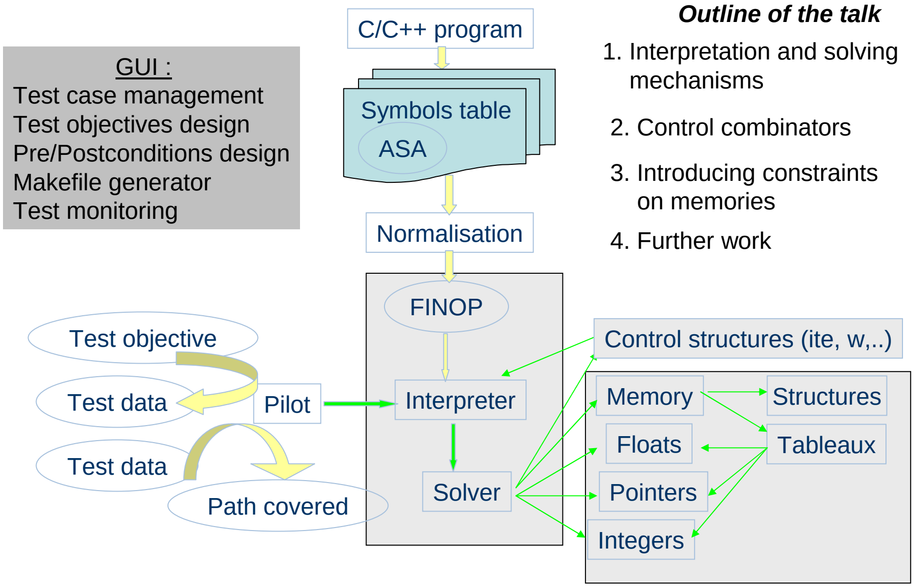
```
// Test whether the number is prime or not
int is_prime(int x)
{
  int prime = 1;
  int n = x / 2;
  int i = 3;
  int m;

  if ((x < 2) || ((x != 2) && (x % 2 == 0)))
    prime = 0;
  while ((prime == 1) && (i < n))
    {
      m = x;
      while (m - i >= 0)
        m -= i;
      if (m == 0)
        prime = 0;
      i = i + 2;
    }
  return prime;
}
```

Variable	Type	Value
Static variables		
x	int	11

Variable	Type	Value
Function return		
ret	int	1
Static variables		
x	int	11

INKA V2: Architecture



GUI :
Test case management
Test objectives design
Pre/Postconditions design
Makefile generator
Test monitoring

C/C++ program

Symbols table
ASA

Normalisation

FINOP

Interpreter

Solver

Test objective
Test data
Pilot
Test data
Path covered

Control structures (ite, w,..)

Memory
Structures
Floats
Tableaux
Pointers
Integers

Outline of the talk

1. Interpretation and solving mechanisms
2. Control combinators
3. Introducing constraints on memories
4. Further work

FINOP : Prolog Intermediate Normal Form

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, BlocId, 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

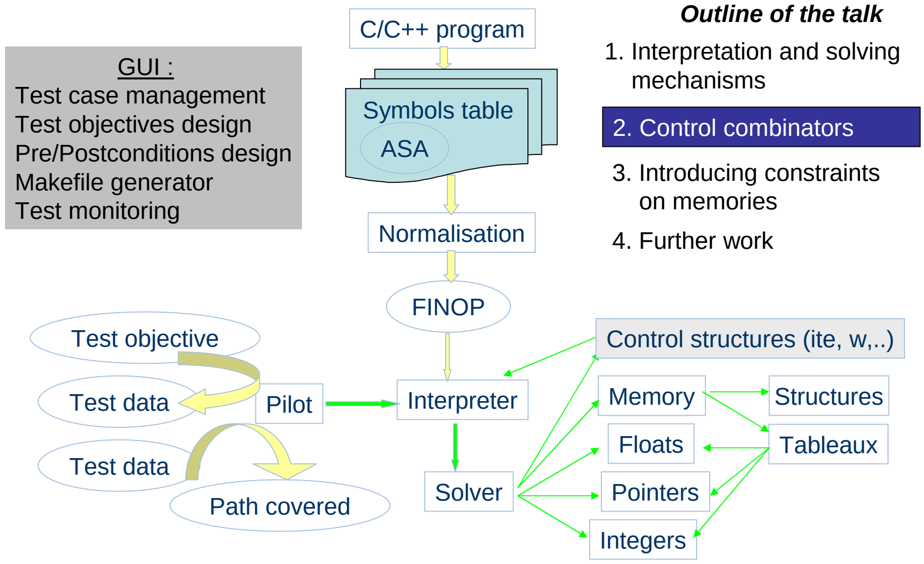
---

- 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



Outline of the talk

1. Interpretation and solving mechanisms
2. Control combinators
3. Introducing constraints on memories
4. Further work

FINOP : Prolog Intermediate Normal Form

2. Control combinators

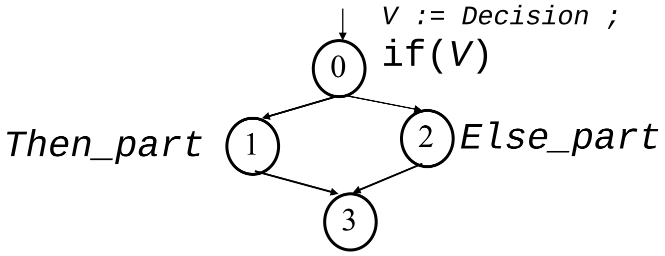
☞ Set of guarded-constraints with « don't care » non-determinism
$\{ C_1 \rightarrow C'_1 , \ldots, C_n \rightarrow C'_n \}$

☞ Operational semantic:
(with store σ: conjunction of domain constraints)

-If C_i is entailed by σ 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 by σ then only $C_i \rightarrow C'_i$ is removed
-Else $C_i \rightarrow C'_i$ is suspended and would awaked whenever at least one of its variable domains is modified

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

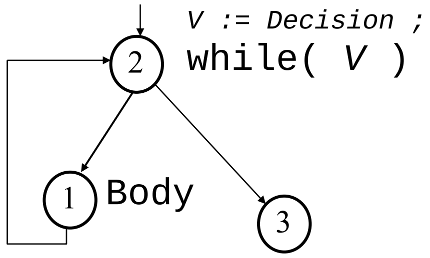
Conditional: the combinator `ite`



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

- V=1 → $C_{THEN} \wedge M_{OUT} = M_{THEN}$
- V=0 → $C_{ELSE} \wedge M_{OUT} = M_{ELSE}$
- $\neg(V=1 \wedge C_{THEN} \wedge M_{OUT} = M_{THEN}) \rightarrow V=0 \wedge C_{ELSE} \wedge M_{OUT} = M_{ELSE}$
- $\neg(V=0 \wedge C_{ELSE} \wedge M_{OUT} = M_{ELSE}) \rightarrow V=1 \wedge C_{THEN} \wedge M_{OUT} = M_{THEN}$
- $M_{OUT} := Proj(OUT, M_{THEN} \cup M_{ELSE})$ $M_{IN} := Proj(IN, M_{THEN} \cup M_{ELSE})$

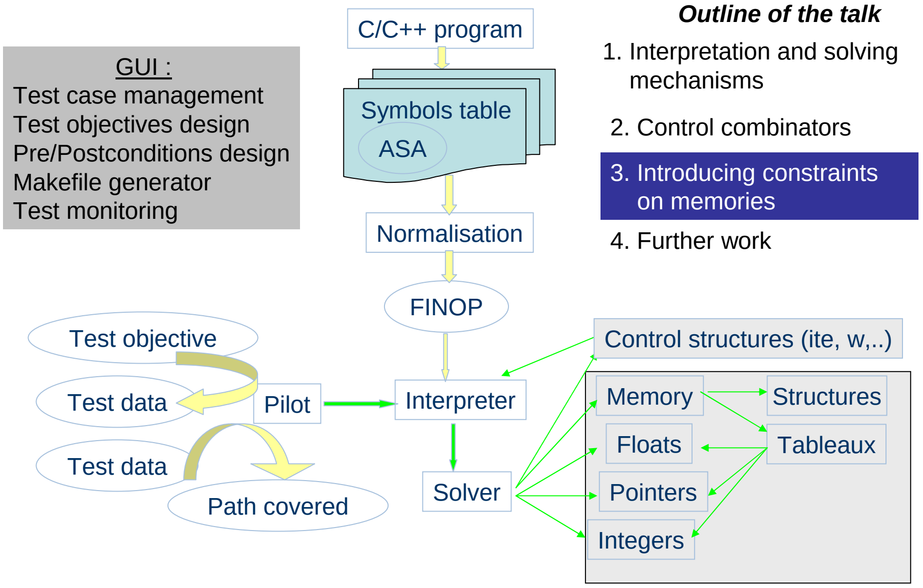
Iteration: the combinator **w**



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

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

INKA V2: Architecture



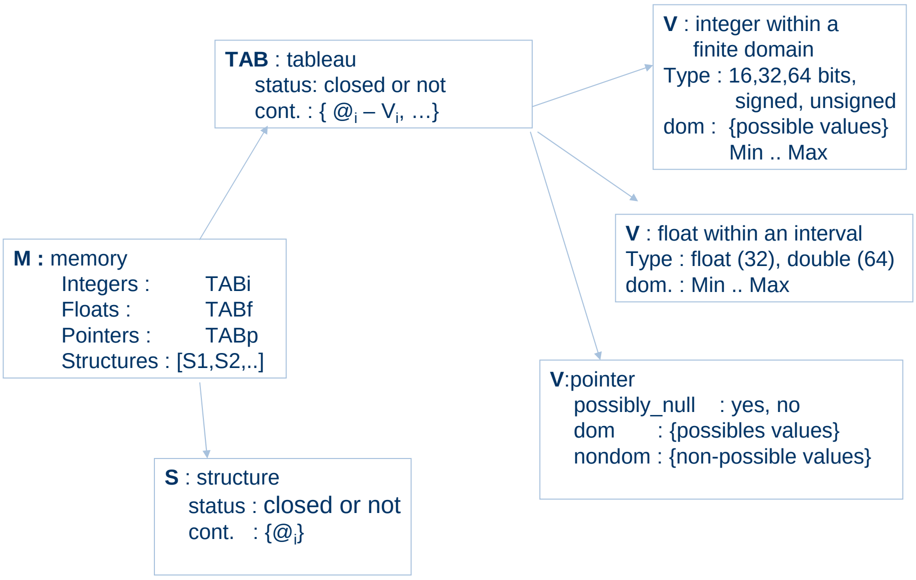
GUI :
Test case management
Test objectives design
Pre/Postconditions design
Makefile generator
Test monitoring

Outline of the talk

1. Interpretation and solving mechanisms
2. Control combinators
3. Introducing constraints on memories
4. Further work

FINOP : Prolog Intermediate Normal Form

An abstract model of the physical memory



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. : {$@_i$}

INKA V2: Introducing constraints on memories

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

C program *<u>Constraints store</u>*

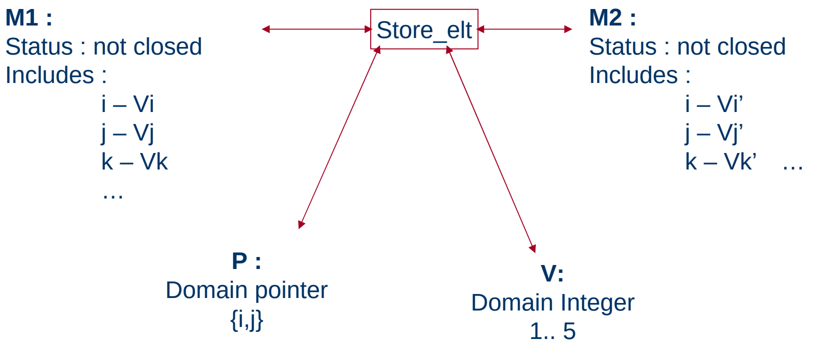
Constraints on memories

- `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)`



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

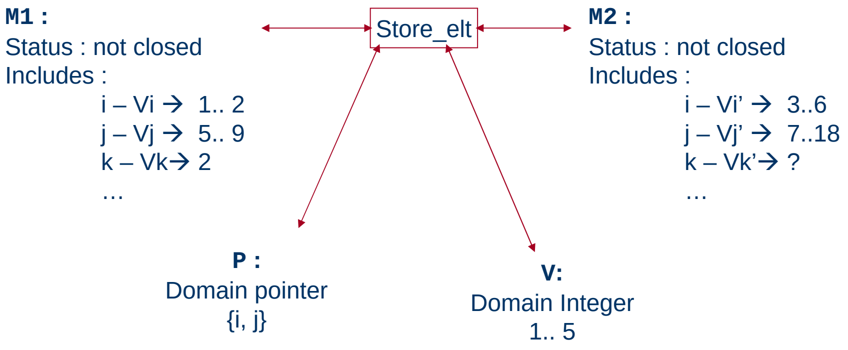
Store_elt

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’→ ?
…

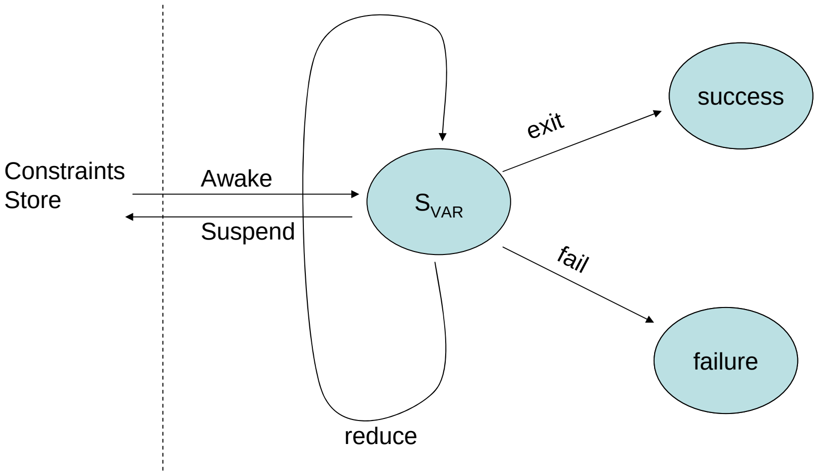
`P :`
Domain pointer
{i, j}

V:
Domain Integer
1.. 5

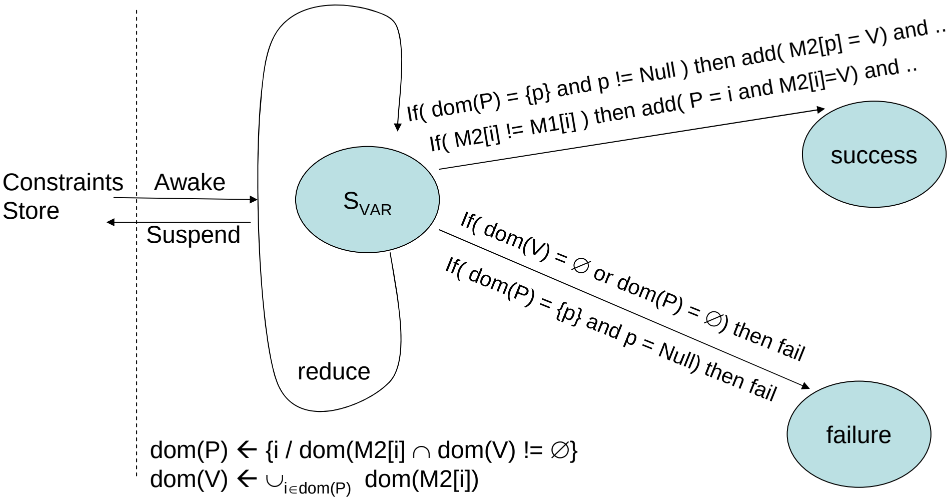
Automatic deductions after the constraint propagation step :

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

Metamodel for the definition of a new constraint



`store_elt(P,V,M1,M2)`



If(dom(P) = {p} and p != Null) then add(M2[p] = V) and ..
If(M2[i] != M1[i]) then add(P = i and M2[i]=V) and ..

If(dom(V) = ∅ or dom(P) = ∅) then fail
If(dom(P) = {p} and p = Null) then fail

dom(P) ← {i / dom(M2[i] ∩ dom(V) != ∅}
dom(V) ← $\cup_{i \in dom(P)}$ dom(M2[i])
dom(M1[i]) ← dom(M2[i]) ∩ dom(M1[i]) if(i ∉ dom(P))
dom(M2[i]) ← dom(M1[i]) ∩ dom(M2[i]) if(i ∉ dom(P))
dom(M2[i]) ← dom(M1[i] ∪ dom(V)) otherwise

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)

- **affect**(A ,R)
- **affect**(**const**(ATOM) ,R)
- **affect**(**in**(Binf,Bsup) ,R)
- **affect**('**-**' ,A,R)
- **affect**(**conv(float,double)** **,**A,R)
- **affect**(**conv(double,float)** ,A,R)
- **affect**(**conv(double,long)** **,**A,R)
- **affect**(**conv(long,double)** **,**A,R)
- **affect**(**conv(float,long)** ,A,R)
- **affect**(**conv(long,float)** ,A,R)

- **affect('+'** ,A,B,R)
- **affect('-'** ,A,B,R)
- **affect('*'** ,A,B,R)
- **affect('/'** ,A,B,R)
- **affect('=<'**,A,B,R)
- **affect('<'** ,A,B,R)
- **affect('>='**,A,B,R)
- **affect('>'** ,A,B,R)
- **affect('=='**,A,B,R)
- **affect('!='** ,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)

- **affect(**X ,Y)
- **affect(in**(DOM), X)
- **affect(const('0')**,X)
- **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) X: pointer, Y:integer, Z:pointer
- **affect('-' ,** X,Y,Z) X: pointer, Y:integer, Z: pointer or X pointer, Y pointer, Z: integer

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

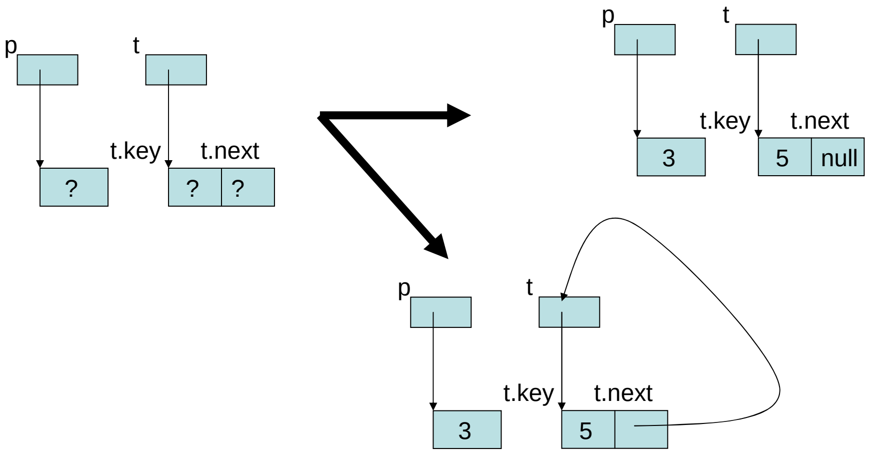
Tries first to avoid creating anything else in the memory and then instantiates all the tableaux

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

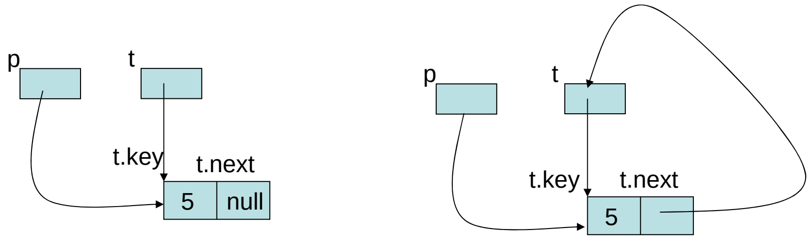
- Creates a single object for each pointer and instantiates all the tableaux



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*