Modelling Dynamic Memory Management in Constraint-Based Testing

Florence Charreteur*, Bernard Botella**, Arnaud Gotlieb*

*IRISA, Rennes, France

**CEA, Saclay, France

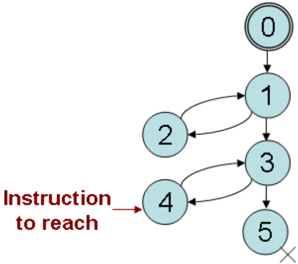
14th September 2007

Testing objective

- Context : automatic test case generation
- Aim : to find an input data to reach a given statement of the program
- Useful for some control-flow coverage criteria as *all-statements*, *all-branches*,...

Testing objective

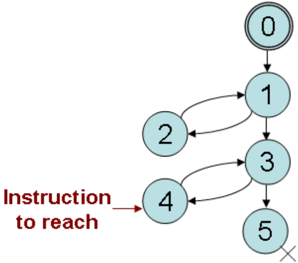
- Two approaches :
  - Path-oriented approach [Williams et al. ASE'04, Sen ESEC'05, Sai-ngern et al. IST'05]



[0 – 1 – 3 – 4]
[0 – 1 – 2 – 1 – 3 – 4]
[0 – 1 – 2 – 1 – 2 – 1 – 3 – 4]
...

Testing objective

- Two approaches :
  - Path-oriented approach [Williams et al. ASE'04, Sen ESEC'05, Sai-ngern et al. IST'05]
  - Goal-oriented approach [Korel 96,Gotlieb et al. ISSTA'98]



$4, 3, 1, 0$ mandatory
2 optional

Dynamic memory management

- Dynamic memory management in C
  - In C : memory can be dynamically managed
    ex : `malloc`, `free` statements
  - Values can then be stored in anonymous locations of the heap. We can only access them by their memory address (pointers).
- Problems for test case generation
  - Importance of the shape of the memory in addition to the values of the variables
  - Aliasing problem : possibly several pointers for the same memory location

Goal-oriented approach

Problems become more difficult in goal-oriented approach as several paths can be taken to reach a program point

- Depending on the followed path there can be :
  - different memory locations allocated,
  - different pointer values,
  - different aliasing relations
- The aliasing problem takes as much importance as it can be useful to perform deduction

Goal-oriented approach

Illustration :

- the value of the condition can depend on the followed path

```
...
if( C ){p = &i;}
i = 10;
*p = 0;
if( i < 5 )
...
```

- aliasing relations can be exploited to remove some paths and some inputs : backward reasoning

Contribution : modeling of constraint operators for dynamic memory management

Outline

1. Constraint-based testing
2. Dealing with dynamic memory management
3. Preliminary results
4. Further work

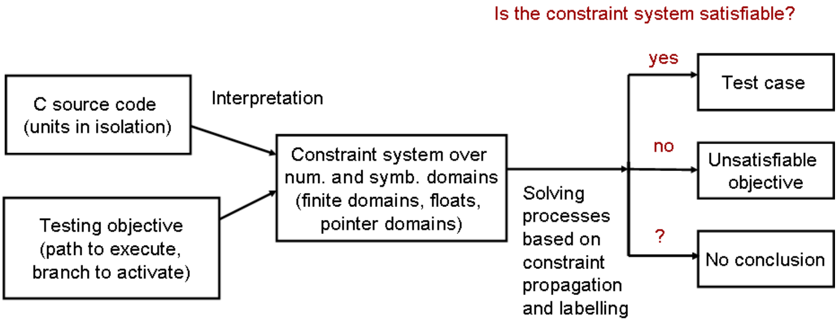
Outline

Principles

- CBT = process of generating test case against a testing objective by using constraint solving techniques
- Automatic test data generators based on constraint reasoning :
    - InKa [Gotlieb et al. ISSTA'98], PathCrawler [Williams et al. ASE'04], Cute [Sen ESEC'05]
    - Extraction of a constraint program from the source code to be tested

InKa [Gotlieb et al. ISSTA'98]

- Test case generation tool for C programs



- Extension to dynamic memory management

Outline

How do constraints lead to find an input state ?

- The aim is to find an input memory state to cover an objective
  Memory state = list of allocated memory locations + values for data
- Abstract memory state
  - A set of memory states is associated with each program point
  - It is an over approximation of the memory states compatible with the objective at a point of the program
  - It is possibly infinite
  - It is represented by an abstract memory state
- Role of constraints
  - In a first time, pruning the domain of memory states,
  - In a second time, finding a particular input memory state to reach the objective.

Abstract memory states

Abstract memory state

integers:

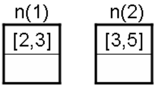
location	possible values
n(1).key	[2,3]
n(2).key	[3,5]

pointers:

location	possible values
n(1).next	{n(2),null}
n(2).next	all

structures:

type	locations
node	{n(1),n(2)}



struct node {int key; node* next}

Abstract memory states

Abstract memory state

integers:

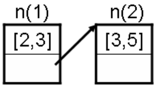
location	possible values
n(1).key	[2,3]
n(2).key	[3,5]

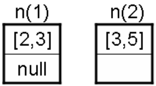
pointers:

location	possible values
n(1).next	{n(2),null}
n(2).next	all

structures:

type	locations
node	{n(1),n(2)}





struct node {int key; node* next}

Abstract memory states

Abstract memory state

integers:

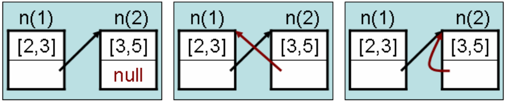
location	possible values
n(1).key	[2,3]
n(2).key	[3,5]

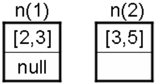
pointers:

location	possible values
n(1).next	{n(2),null}
n(2).next	all

structures:

type	locations
node	{n(1),n(2)}





struct node {int key; node* next}

Abstract memory states

Abstract memory state

integers:

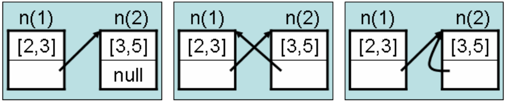
location	possible values
n(1).key	[2,3]
n(2).key	[3,5]

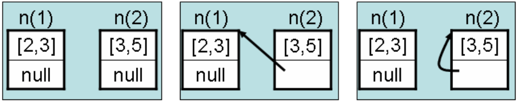
pointers:

location	possible values
n(1).next	{n(2),null}
n(2).next	all

structures:

type	locations
node	{n(1),n(2)}





struct node {int key; node* next}

Abstract memory states

Abstract memory state

integers:

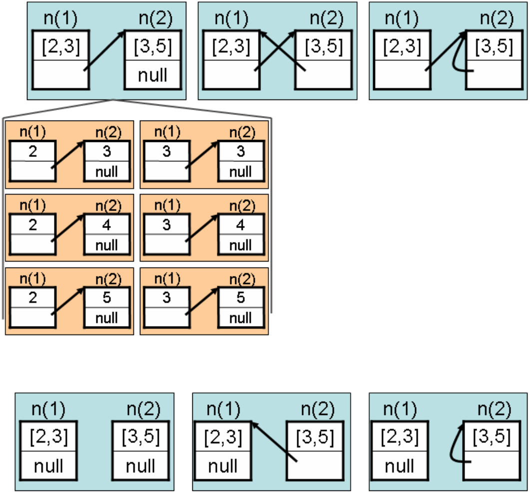
location	possible values
n(1).key	[2,3]
n(2).key	[3,5]

pointers:

location	possible values
n(1).next	{n(2),null}
n(2).next	all

structures:

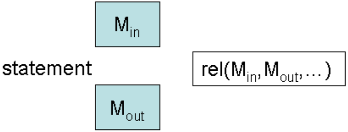
type	locations
node	{n(1),n(2)}



struct node {int key; node* next}

Operators to describe the program

- Statements are represented in the constraints model by constraint operators that link the memory before the statement (M_{in}) with the memory after the statement (M_{out}).
- The operators take as parameters M_{in}, M_{out} as well as other parameters peculiar to the statement.



Operators to describe the program

- allocation of a new structure of type t :

  new_s(M_{in}, M_{out}, t, id)

  new_fields(M_{in}, M_{out}, t, id)
- storage in the memory :

  store_elt(M_{in}, M_{out}, P, V)
- deallocation :

  delete_s(M_{in}, M_{out}, P)

  delete_fields(M_{in}, M_{out}, P)
- access to the memory :

  load_elt(M_{in}, Y, V)

  access_s(M_{in}, t, Y, f, P)

Operators to describe the program

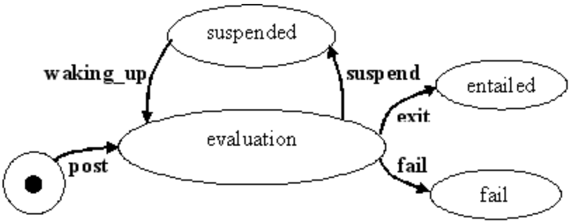
- Translation of the statements into constraint operators

$free(p)$
load_elt(M_{in}, p, P)
delete_s(M_{in}, M_{out}, P)
delete_fields(M_{in}, M_{out}, P)

$y- > f = x$
load_elt(M_{in}, y, Y)
access_s(M_{in}, Y, f, P)
load_elt(M_{in}, x, X)
store_elt(M_{in}, M_{out}, P, X)

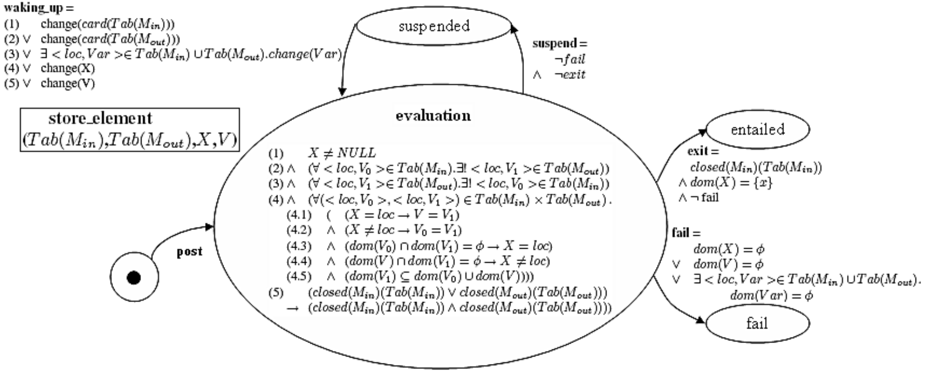
Description of operators

- Modelling the operators : finite state machine

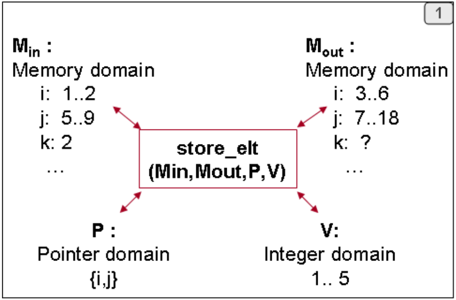


Description of operators

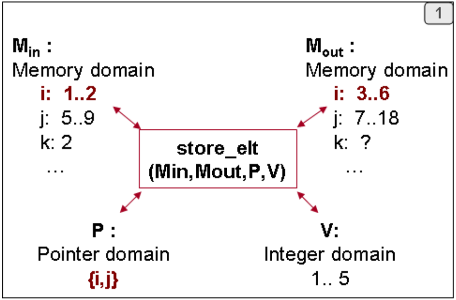
- store_elt(M_{in}, M_{out}, X, V)



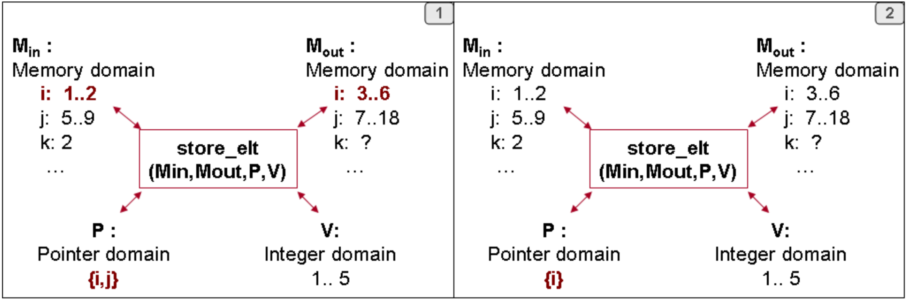
store_elt operator : Illustration of deductions performed



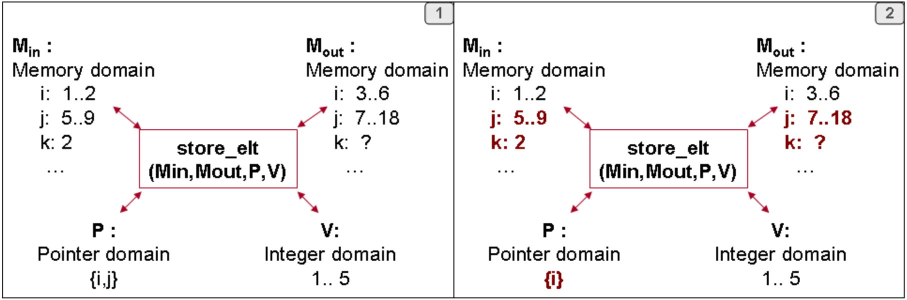
store_elt operator : Illustration of deductions performed



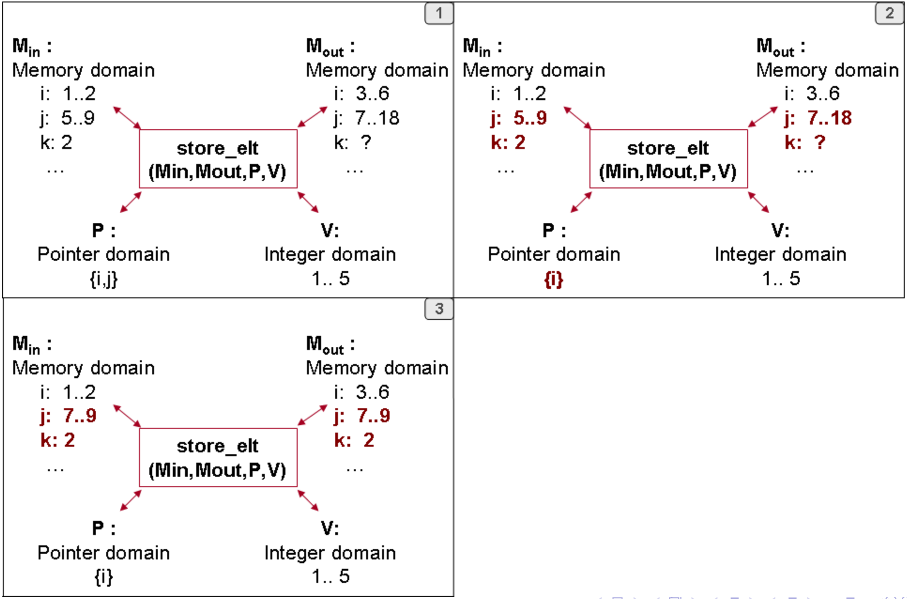
store_elt operator : Illustration of deductions performed



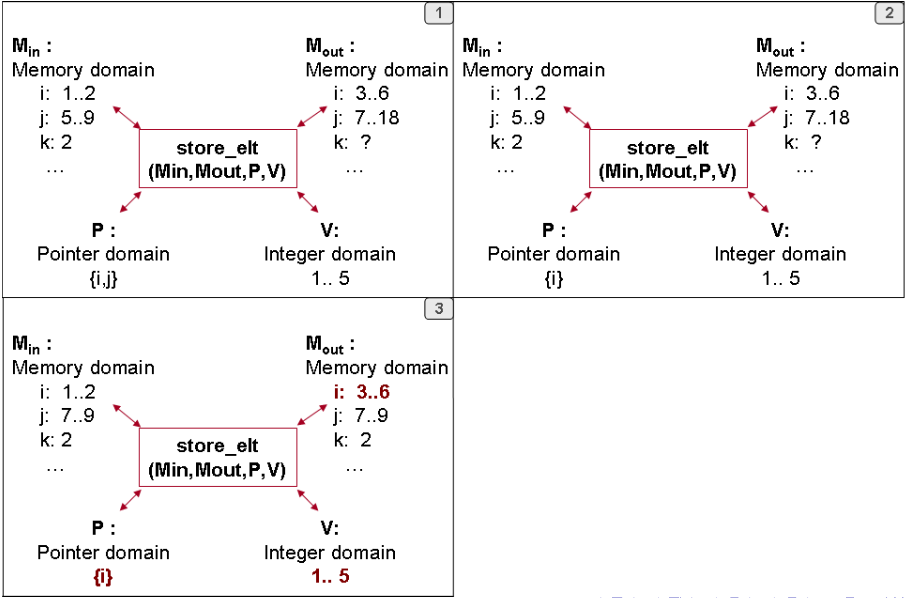
store_elt operator : Illustration of deductions performed



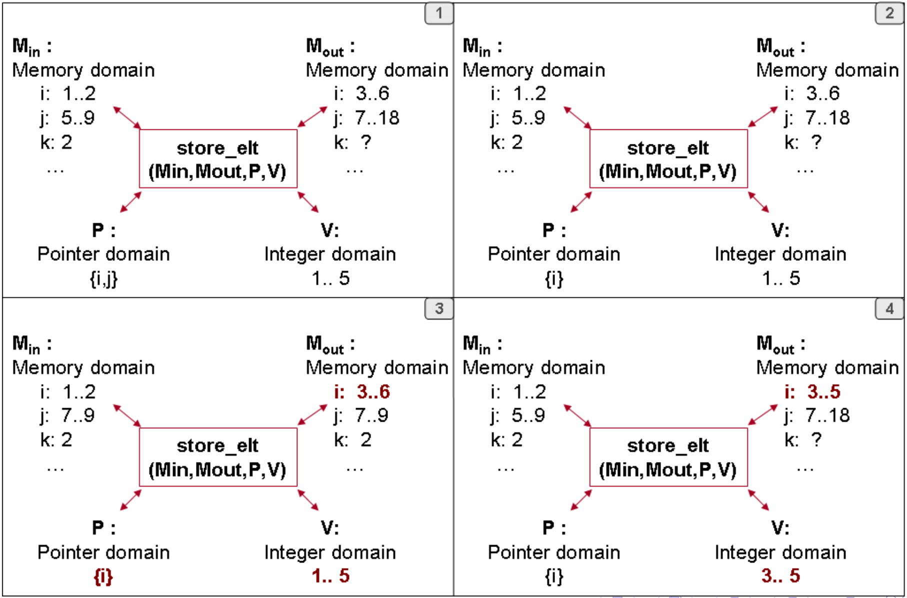
store_elt operator : Illustration of deductions performed



store_elt operator : Illustration of deductions performed



store_elt operator : Illustration of deductions performed



Outline

- A program with linked structures : Josephus

```
typedef struct node *link;
struct node { int key ; link next;};

int f(int n,int m){
  int i; link t,x ;
  t=(link)malloc(sizeof(struct node));
  t->key = 1;
  x = t;
  i = 2;
  while( i <= n ){        //while1
    t->next=
       (link)malloc(sizeof(struct node));
    t = t->next ;
    t->key = i;
    i++;
  }
  t->next = x ;
  while( t != t->next){ //while 2
    i = 1;
    while( i <= m-1){  //while 3
      t = t->next ;
      i++;
    }
    x = t->next ;
    t->next = (t->next)->next ;
    free(x);
  }
  return t->key;
}
```

- **while1** : construction of a circular simple-linked list with n nodes
- **while2** : elimination of the node at position m until only a single node remains

- Branch-coverage of the Josephus program

CPU time required to generate test data
120 sec for three test data

- Branch-coverage of the Josephus program

CPU time required to generate test data
120 sec for three test data

- Reaching k iterations of `while2` in the Josephus program

k	Test data	CPU time (in sec) required to generate test data
5	m=0,n=6	0.4
10	m=0,n=11	1.4
15	m=0,n=16	6.8
20	m=0,n=21	13.2

Outline

Further work

- Extension of the model to deal with C statements closer to the memory
  - pointer cast
  - union
- Integration of a language of preconditions on the memory shape
- Performing a formal proof of the operators

Modelling Dynamic Memory Management in Constraint-Based Testing

Florence Charreteur*, Bernard Botella**, Arnaud Gotlieb*

*IRISA, Rennes, France

**CEA, Saclay, France

14th September 2007