Modelling Dynamic Memory Management in Constraint-Based Testing

Florence Charreteur	Bernard Botella	Arnaud Gotlieb
IRISA/Univ. de Rennes 1	CEA LIST	IRISA/INRIA
Campus de Beaulieu	Saclay	Campus de Beaulieu
35042 RENNES – France	91 191 Gif SurYvette – France	35042 RENNES – France
Florence.Charreteur@irisa.fr	Bernard.Botella@cea.fr	Arnaud.Gotlieb@irisa.fr

Abstract

Constraint-Based Testing (CBT) is the process of generating test cases against a testing objective by using constraint solving techniques. When programs contain dynamic memory allocation and loops, constraint reasoning becomes challenging as new variables and new constraints are created during the test data generation process. In this paper, we address this problem by proposing a new constraint model of C programs based on operators that model dynamic memory management. These operators apply deduction rules on abstract states of the memory allowing so to enhance the constraint reasoning process that permits to generate test data for these programs. We illustrate our approach on structural testing of a complex program that contains dynamic memory allocation/deallocation, structures and loops. An implementation is in progress and first experimental results obtained on this program show the highly deductive potential of the approach.

1. Introduction

A new trend in program testing is to combine static and dynamic analysis through the usage of the constraints technology. Constraint-Based Testing (CBT) was introduced fifteen years ago in the context of mutation testing [1] to model processes of test case generation using constraint solving techniques. Since then it has been continuously developed to cover several applications area including hardware verification [2,3], test data generation for structural testing [4,5,6,7] and functional testing [8,9], counter-example generation [10,11], or software verification [12,4,8]. Among the tools that implement the CBT approach, InKa [13,14], ATGen [15] and PathCrawler [5] are automated test data generators based on constraint propagation over finite domains. These tools extract a constraint program from the source code to be tested and then exploit constraint propagation and backtracking to find test data that cover a selected element (path, branch or statement) within the program. Moreover, these tools are able to detect parts of the program that cannot be executed (dead code) as they can sometimes detect unsatisfiability during constraint propagation.

Among the projects dedicated to CBT, the MUTT project [16] from Microsoft explores the combination of dynamic and static analysis for constraint-based automatic test data generation [6]. In France, CBT has been developed within the RNTL InKa and DANOCOPS projects, and the ACI V3F [17] which expressed the need for improving current constraint propagation capabilities of solvers exploited in CBT.

This paper addresses the problem of dynamic memory management in CBT and details our implementation based on specific constraint operators. We first illustrate this problem on a complex example extracted from the literature [18]. This program called Josephus is shown in figure 1. The first loop builds a circular simple-linked list of n nodes, while the second loop eliminates nodes at position m until only a single node remains in the list. Thus, this program contains dynamic memory allocation/deallocation within loops. This is considered as the main technical difficulty w.r.t. dynamic structures for CBT techniques.

An interesting and typical testing objective for the Josephus program is to find values for m and n such as the second loop (**while2**) is unrolled a given number of times as this constraints the length of the list. For CBT, such a problem involves complex analysis such as structures sharing and pointer aliasing. Moreover, dynamic allocation/deallocation in loops requires variables and constraints being created during the test data generation process. For example, consider the testing objective of generating a test datum on which

the loop **while2** is unrolled at least forty times[1]. Consequently, our system automatically deduces that the length of the list should be 41 while the remaining element at the end of the process is 31 when m=3.

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

int f(int n,int m){
1. int i; link t,x ;
2. t=(link)malloc(sizeof(struct node));
3. t->key = 1;
4. x = t;
5. i = 2;
6. while( i <= n ){          //while1
7.   t->next=
       (link)malloc(sizeof(struct node));
8.   t = t->next ;
9.   t->key = i;
10.  i++;}
11.t->next = x ;
12.while( t != t->next){  //while2
13.  i = 1;
14.  while( i <= m-1){   //while3
15.    t = t->next ;
16.    i++;}
17.  x = t->next ;
18.  t->next = (t->next)->next ;
19.  free(x);}
20.return t->key;}
```

Figure 1. Josephus program

This paper details the deduction rules that equip our operators that model dynamic memory management. We introduce each operator under the form of finite state machine that interacts with the constraint propagation solver. Our implementation translates a program under test written in a restricted subset of C into a constraint (logic) program. This program is then solved using classical finite domain constraint propagation and labeling.

Outline of the paper. Section 2 recalls the necessary background on Constraint (Logic) Programming. Section 3 describes our abstract memory model that tackles with dynamic allocated structures. Section 4 presents the deduction rules exploited in the constraint operators that model dynamic memory management. Section 5 gives preliminary results obtained on the Josephus program, while section 6 discusses related works. Finally, Section 7 presents our future work.

2. Background

2.1. Constraint reasoning

Our approach is based on Constraint Logic Programming over Finite Domains (CLP(FD)). Constraint Logic Programming replaces classical imperative assignment by unification and constraints that are logical relations between the variables of the problem. In CLP(FD), a finite domain is associated to each variable and a solution to the constraint system is a valuation of the variables within their domains that satisfy each constraint. Primitive constraints in CLP(FD) are built over variables, domains, arithmetical operators in {+,-,*,\,…} and relations {>,≥,=,≠,≤,<}. Non-primitive constraints include user-defined constraints and constraint operators that express high-level relation between other constraints. In this paper, we focus on the definition of constraint operators that tackle dynamic memory management.

Two interleaved processes intervene in the solving process of a constraint system.

Constraint propagation. Initially, the constraints are added to a main queue and fall into an ***evaluation*** state. Each constraint of the queue is considered one-by-one by the constraint propagation algorithm. The algorithm exploits each constraint to filter out the inconsistent values from the domains of the variables. When the domains of all variables of the constraint have been pruned, the constraint falls into the ***suspended*** state. When the domain of a variable is pruned, other constraints that involve this variable are reintroduced into the queue. In this case, these *suspended* constraints are woken up and return in the *evaluation* state. If the domain of at least one variable becomes empty, the constraint ***fails***: the constraint system is unsatisfiable. If the constraint succeeds, meaning that each of the tuples from the current domains are compatible with the constraint, then it falls into the ***entailed*** state. In this case, the constraint is removed from the queue of constraints as it is no more useful. When no more reduction is possible, the queue becomes empty and the constraint propagation ends. Figure 6 gives the model we use to represent constraint operators. It is based on this mechanism of constraint propagation. It will be described in the following.

Variable labeling. When the constraint propagation ends, enumeration on the possible values from the domains is usually required to get a solution. The labeling procedure tries to give a value to every variable one by one. When a value is chosen from the domain of a variable, the constraint propagation is re-run to prune the domains of other variables with the current hypothesis. If a contradiction appears during the resolution process, the procedure backtracks to other possible values. The process stops when a value is assigned to each variable.

[1] In the original ancient Josephus's decimation problem, there were 40 people killed, letting the emperor alive at position 31.

2.2. Notations and restrictions

In the following, parameters of operators that begin with a lower case letter denote identifiers whereas parameters that begin with a capital letter represent abstract variables of the model.

For the sake of simplicity, we confine ourselves to a small language over pointers whose grammar is given in figure 2. We suppose that dynamic allocation is only possible for structures as other cases are simpler. In this paper, we focus on dynamic memory management statements. We ask the reader to report to [13] to see how to deal with control structures such as if and while. Note that our approach addresses a greater subset of the C language that includes arrays, pointer arithmetic [19], and floating-point variables [20].

```
<program>    ::= <statement>*

<statement>  ::=
     <int_assignment>
   | <pointer_assignment>
   | FREE(<pointer_var>)          %memory deallocation
   | IF cond THEN<statement>* ELSE<statement>*
   | WHILE cond <statement>*

<pointer_assignment>  ::=
     <pointer_var>  = NEW(type,id)  %memory allocation
   | <pointer_var>  = <pointer_val>
<pointer_var>::=
   symbol                         %pointer variable name
   | <pointer_val> - > f          %access to a field
<pointer_val>::=
   const                          %pointer constant
   | <pointer_var>

<int_assignment>  : :=
   <int_var>  = <int_val>
<int_var>     ::=
    symbol                        %integer variable name
   | <pointer_val> - > f          %access to a field
<int_val>     ::=
    const                         %integer constant
   | <int_var>
```

Figure 2. Grammar of the studied language

3. Description of an abstract memory

In our implementation, C operations over the memory are expressed with relations over two (abstract) memory states. The first memory state represents the memory before statement execution while the second memory state represents the memory after statement execution. A **memory state** is an abstract model of the physical memory, which contains information known *at a given step of the resolution process of the constraint system*. Figure 3 describes the model of an abstract memory state. It contains known

Description of sets :
MEM: set of abstract memories
$IDENT$: set of locations in abstract memories
VAR_i: set of abstract integer variables
VAR_p: set of abstract pointer variables
VAR_s : set of abstract structures
$TYPE_s$: set of structured types
$NAME_f$: set of names of fields
$TYPE$: set of types. $TYPE = \{integer, pointer\} \cup TYPE_s$

$\forall m \in MEM : m =< tab_i(m), tab_p(m), struct(m), closed(m) >$
with: $tab_i(m) \in \mathcal{P}(\{< ident, V_i > | ident \in IDENT, V_i \in VAR_i\})$
$tab_p(m) \in \mathcal{P}(\{< ident, V_p > | ident \in IDENT, V_p \in VAR_p\})$
$struct(m) \in \mathcal{P}(\{< type, V_s > | type \in TYPE_s, V_s \in VAR_s\})$
$closed(m) \in \{true, false\}$

$s_@ : VAR_s \rightarrow \mathcal{P}(IDENT)$ % set of program locations
$access_f : IDENT \times NAME_f \rightarrow IDENT$ % access to a field
$dom_i : VAR_i \rightarrow \mathcal{P}(\mathbb{N})$
$dom_p : VAR_p \rightarrow \mathcal{P}(IDENT \cup NULL) \cup \{all\}$
$ndom_p VAR_p \rightarrow \mathcal{P}(IDENT \cup NULL)$
$type_p : VAR_p \rightarrow TYPE$

Figure 3. Abstract memory

information about basic variables and dynamic allocations. MEM is the set of all the abstract memories. An abstract memory m contains four elements: struct(m), tab_i(m), tab_p(m), and closed(m) that are described below.

As an example, Figure 4 shows the abstract memory state m obtained before statement 11 of the C program of figure 1 after two iterations of the first loop.

tab_i(m)

ident	Var_i	dom_i
m	V_i1	Inf..sup
n	V_i2	3
i	V_i3	4
n(2).key	V_i4	1
n(7.1).key	V_i5	2
n(7.2).key	V_i6	3

tab_p(m)

ident	Var_p	dom_p	non_dom_p	$type_p$
t	V_p1	{n(7.2)}	empty	node
x	V_p2	{n(2)}	empty	node
n(2).next	V_p3	{n(7.1)}	empty	node
n(7.1).next	V_p4	{n(7.2)}	empty	node
n(7.2).next	V_p5	all	empty	node

struct(m)=<node, S_{node} >

Var	s_@
S_{node}	{ n(2), n(7.1),n(7.2)}

closed(m)=true

Figure 4. Abstract memory after the statement 11 of the Josephus program, at the end of the constraint propagation when the first loop is unrolled twice

3.1. Dynamic allocations: struct(m)

For an abstract memory m at a given step of the solving process, information about dynamic allocation of

structures is stored in struct(m). In a memory, to each structured type definition is associated a variable, and a function called s_@ gives the set of anonymous program locations associated to this variable. On the model, each abstract memory location is represented by the term ident. An abstract memory location can be anonymous in the case of dynamic allocation. In figure 4 where there are three dynamically allocated structures, {n(2), n(7.1), n(7.2)} is the set of the anonymous program locations. For example, n(7.2) denotes the anonymous program location obtained by the execution of the statement 7 in the second iteration of the loop. Function access_f associates to location loc of a structure and field name f, the complete name of the field location loc.f. For example, access_f(n(2),next) is n(2).next.

3.2. Basic types: $tab_i(m)$, $tab_p(m)$

Information about basic variables is memorized by pairs ident-Var; where ident is an abstract memory location, and Var is a variable of type integer or pointer. On the model, the sets of integer and pointer variables are noted respectively VAR_i and VAR_p. These pairs are stored in two data structures, called *tableaux*: $tab_i(m)$, $tab_p(m)$.

3.2.1. Integer variables. For abstract variables that represent integers, the function dom_i gives the set of possible values at a given step of the resolution. For example, the abstract memory location i in $tab_i(m)$, has value 4 in abstract memory of figure 4.

3.2.2. Pointer variables. For abstract variables that represent pointers, our model provides two functions dom_p and $ndom_p$. dom_p returns the set of possible abstract memory locations (symbolic names or anonymous program locations) for the pointer while $ndom_p$ returns the set of memory locations on which the pointer cannot points-to. For example, on a condition such as (p==&a||p==&b), we get dom_p(P)={@a,@b} where @a (resp. @b) denotes the address of a (resp. b) in the abstract memory model. On a condition such as (p!=&a), we get $ndom_p$(P)={@a}. If dom_p(P) contains a single value v, the element pointed by P is definitely known. For example, V_p2 in figure 4 is simplified to n(2) as x points to the first dynamically allocated structure in statement 2. If dom_p(P)=all, P can point to any location in the memory except the ones contained in $ndom_p$(P).

The function $type_p$ returns the type of the pointed element. In figure 4, all the pointers point to a node structure.

3.3. Closure of the memory: closed(m)

The last component of an abstract memory m is the predicate closed(m). It indicates whether all the elements of the *tableaux* and structures are defined or not in the current abstract memory. This predicate is used in the labeling process to force the input domain to contain only the data structures previously labeled. Without this predicate, labeling could invent new structures and values leading to a non-terminating process. Such status is of particular interest when one looks for the possible shapes of a dynamic structure during the labeling process as it strongly constrains the set of identifiers a pointer can point to.

4. Constraint operators on abstract memories

To find a test datum to reach a given testing objective, the program under test is translated into a constraint system on abstract memories. This section details the deduction rules applied within some of the constraint operators that tackle with dynamic structures. Figure 5 shows the translation of some basic statements of our language into these constraint operators. M_{in} and M_{out} are respectively the memories before and after the execution of a given statement. All the elements of the memory M_{in} and M_{out} but the input parameters of the constraint operators are identical (except the predicates closed(M_{in}) and closed(M_{out})). Asterisks in figure 5 denote the operators that will be detailed in the following. For other operators, we will only give an overview.

```
new(t,id)
(*)  new_s(Struct(M_in)(t), Struct(M_out)(t),id)
new_fields(M_in,M_out,t,id)

x=y->f
(*)  access_s(Struct(M_in)(typeY),Y,f,V_p)
(*)  load_elt(Tab_t(M_in),Vp,Val)
store_elt(Tab_i(M_in), Tab_i(M_out),X,Val)

y->f=x
(*)  access_s(Struct(M_in)(typeY),Y,f,V_p)
(*)  store_elt(Tab_i(M_in), Tab_i(M_out)t),Vp,X)

x=y
store_elt(Tab_p(M_in), Tab_p(M_out)t),X,Y)

free(x,t)
(*)  delete_s(Struc_t(M_in)(t), Struct(M_out)(t),X)
delete_fields(M_in,M_out,type,X)
```

Figure 5. Translation into operators

Statement **new(t,id)** generates two operators. The first one links the set of locations of type t between M_{in} and M_{out}. It needs the identifier id of the location to add. The second constraint operator reserves place for the fields of the new structure. The statement **x=y->f** generates

three operators. The first one gets a pointer Vp that points to the possible locations of y->f. Thanks to Vp, the second operator collects in Val the possible values for y->f. The third operator affects the domain of Val to X. Similarly, statement **y->f=x** generates two operators. The first one gets a pointer Vp that points to the possible locations in memory that can store y->f. The second one affects the value of x to y->f. Statement **x=y** is expressed by an operator that stores the abstract pointer variable that represents y in the location of x in the abstract memory. Statement **free(x,t)** generates two operators: the first one deletes the structure pointed by x in the memory while the second one deletes its fields.

We now turn on the description of these constraint operators: new_s for dynamic allocation, delete_s for deallocation, access_s to access structure field, store_element to store integer/pointer values in memory, and load_element to load values. The constraint operators for dynamic memory management have three roles: the propagation of the knowledge of the allocated locations in the memories, the filtering of the domains for pointers and integer values and the propagation of information about the closure of the memory.

4.1. Representation of a constraint operator

We propose to explain our constraint operators by using a simple model based on finite state machines. Figure 6 shows a generic state machine for constraint. Our representation gives the possible states of a constraint: it can be in evaluation, suspended, entailed or in a failure state. Deductions made thanks to the operators are described in the evaluation state. Labels on the arrows describe the events that permit to switch from one state to another. For each operator, we describe five possible transitions: 1) post event: the operator is posted in the propagation queue ; 2) waking-up event: the operator is woken up by some

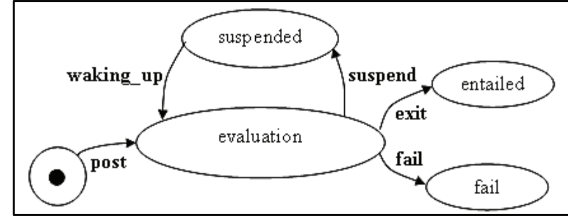


Figure 6. Representation of a constraint operator

additional information on the domain of its variables or on its status ; 3) suspend event: no more deduction rules can be exploited to prune the variation domains ; 4) exit event: the operator becomes entailed ; 5) fail event: some inconsistency has been detected which indicates failure of the current constraint system.

```
   a= new(t,new(1)) ;
M0
   b= new(t,new(2)) ;
M1
   c= new(t,new(3)) ;
   a-> f =1;
   b-> f =2;
   c->f =3;
   if(cond1){
      p=a;
   }else{
      if(cond2){
          p=b;
      }else{p=c}}
   if(cond3){
M2
      free(p);
M3
   }else{
M4
      p->t=i;
M5
      j=p->t;}
   if(p!=a && b->t=6 && j>2){
```

Figure 7. Program foo

Figure 7 shows a part of a C program that illustrates the interest of the deduction rules that we are going to present. In the following, we will refer to this program implicitly by showing only the abstract state of the

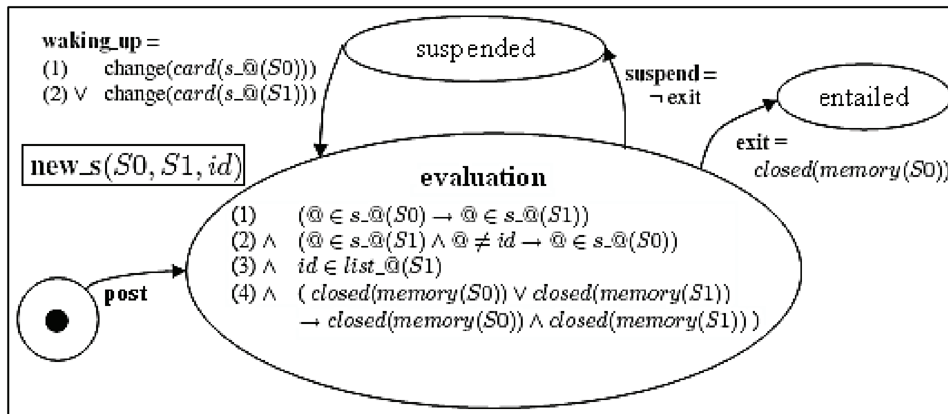


Figure 8. new_s operator

memories M0,..,M5.

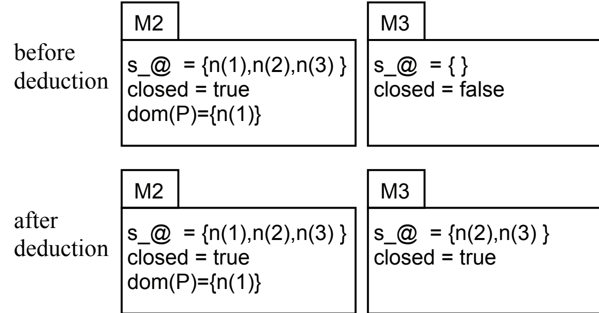
4.2. The new_s operator

The operator new_s(S0,S1,id) is added whenever a structure of type t is dynamically allocated. Figure 8 shows the model of this operator that establishes a relation between S0, S1 and id. In the model, we suppose that $S0=struct(M_{in})(t)$, $S1=struct(M_{out})(t)$, where struct(M)(t) denotes the variable associated with t in struct(M), and id is the identifier of the anonymous dynamic location. The operator is awoken when a location is added to the set of locations of S0 or S1.

If the operator is awoken, some new deductions can be performed. S0 and S1 should contain the same identifiers, except for id that is only in S1. Firstly, S1 should contain all the locations that are present in S0 (proposition 1) as well as the location id (proposition 3). Secondly, all the locations that are in S1, except id, should be in S0 (proposition 2). Moreover, if one abstract memory is closed, the second one should also be closed (proposition 4). Indeed, as new_s adds only one new location, the closure of M_{in} (resp. M_{out}) implies the closure of M_{out} (resp. M_{in}). Moreover, the operator succeeds as soon as M_{in} is closed (cf exit arrow), while it suspends otherwise.

4.3. The delete_s operator

The operator delete_s(S0,S1,X) is added whenever a structure of type t, pointed by X, is removed from the abstract memory. Figure 9 illustrates the delete_s operator, which maintains a relation between three parameters: $S0=struct(M_{in})(t)$, $S1=struct(M_{out})(t)$, and X. The operator delete_s is woken up either when a location is added to the set of locations S0 or S1, or when the variation domain of pointer X is modified (for example, learning that X points to only one location). Such a modification is noted change(X) in our model. The relation maintains the fact that X should be non-null (proposition 1). As an illustration of the deduction rules, suppose the information on abstract memories linked by a delete_s operator shown below is available (memories before deduction).



Here, P points definitely to n(1) which is the deleted location. So, it should not appear in M3 (proposition 2.1) and other locations are not touched (proposition 2.2). Moreover, as M2 is closed and the location to delete is known, M3 is also closed (proposition 4) and the operator succeeds (exit arrow). We obtain the memories after deduction shown above.
If there is no precise information on the pointed location or the available memory, then the operator is suspended.

4.4. The access_s operator

The operator access_s(S,X,f,Vp) is added when we access to a numeric field of a structure of type t. Figure 10 illustrates this operator that maintains a relation between four parameters: S=struct(M)(t), X the pointer to the possible locations of the structure, f the field name and Vp the pointer to the possible locations of the field.

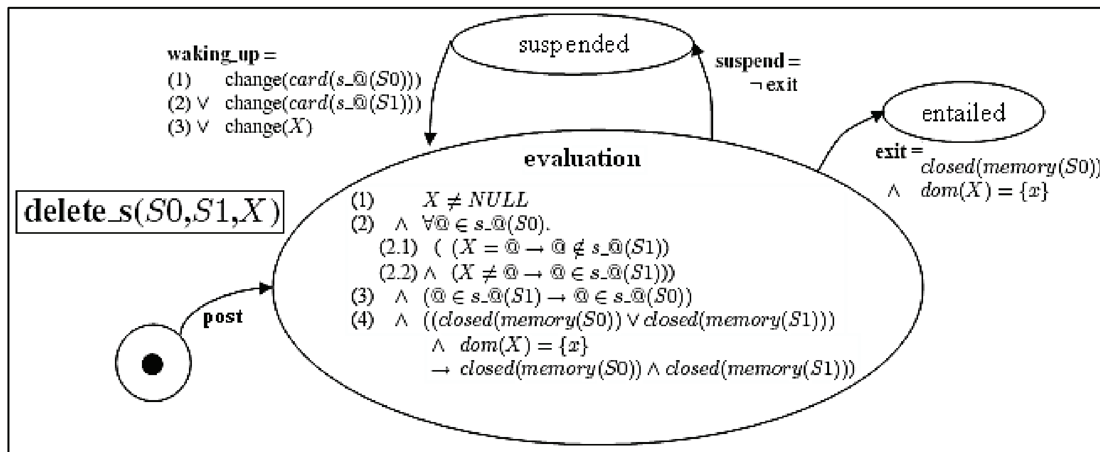


Figure 9. delete_s operator

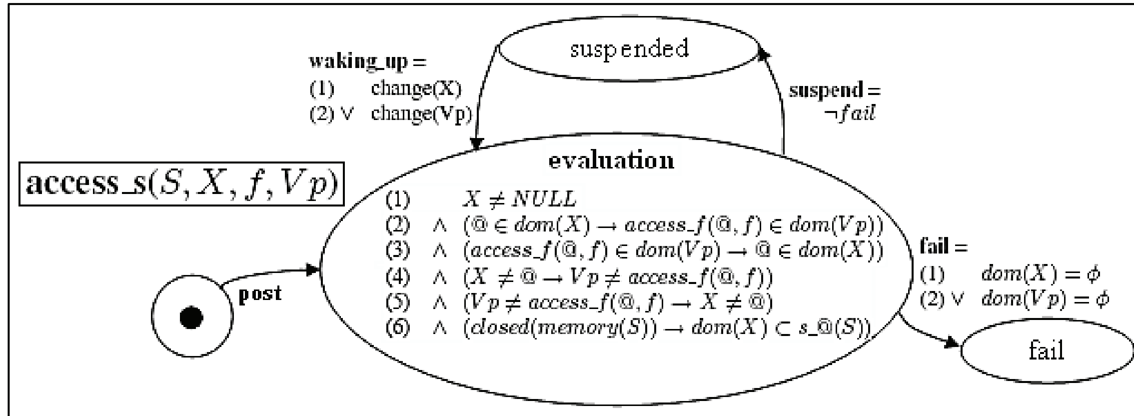
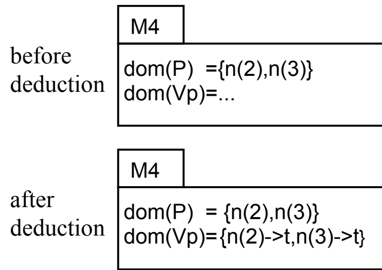
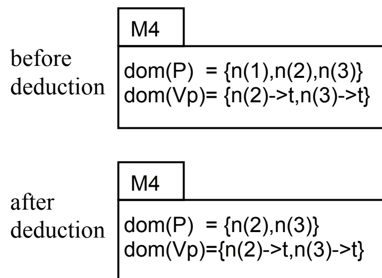


Figure 10. access_s operator

The operator access_s is awoken when we get more information about the values pointed by X or by Vp. X should not be null (proposition 1). Statement `p->t=i;` in figure 7 leads to add two constraint operators including access_s(struct(M4)(t),P,t,Vp). The following draw illustrates the deductions made by this operator.



For each location of the domain of P, the relation maintains that Vp can point to the location associated with the field (proposition 2). In the following example, Vp can only point to n(2)->t or n(3)->t so P can only point to n(2) or n(3) (proposition 5).



Proposition 6 gives complementary information: if the memory M is closed, all the possible locations pointed by X belong to the set of addresses of S. Indeed all the possible locations for a structure of type t are known and X points to such a structure.

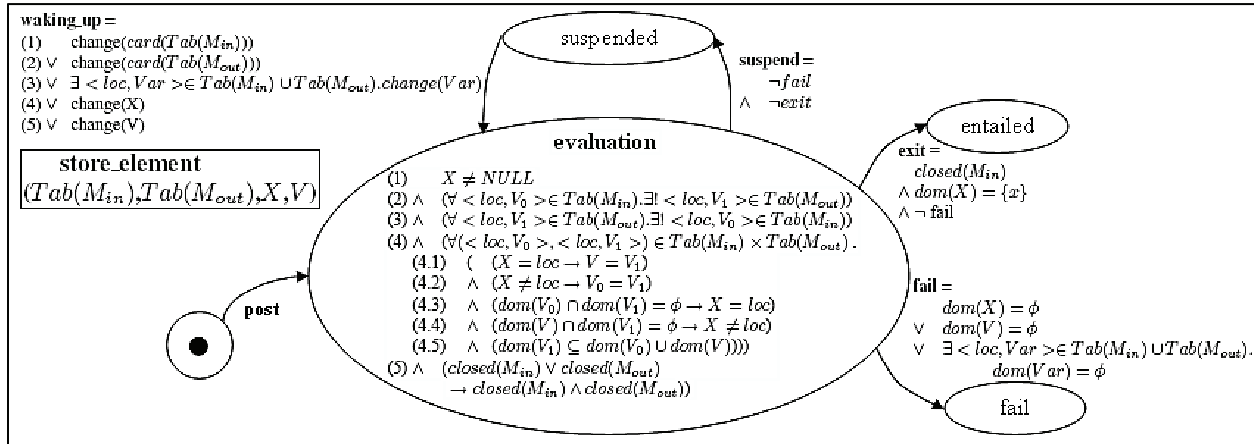
During the resolution process, if the domain of a variable becomes empty, the constraint resolution process fails. Indeed, it means that there is no assignment for all the variables of the system such that all the constraints are satisfied. For the operator access_s, the only domains that can become empty are the domain of X and the domain of Vp (in this case X or Vp cannot point to any location anymore).

4.5. The store_element operator

The operator store_element(Tab(M_{in}),Tab(M_{out}),X,V) is added when the statement stores a value in memory at a given address. Figure 11 illustrates this operator that maintains a relation between X, Tab(M_{in}), Tab(M_{out}) and V. X is a pointer to a location that stores an integer or a pointer value. The store_element operator is awoken when 1) a pair ident-Var is added in one of the *tableaux* Tab(M_{in}) and Tab(M_{out}) (conditions 1 and 2) ; 2) when the domain of a variable in Tab(M_{in}) or Tab(M_{out}) changes (condition 3); 3) when the information about the pointer X changes; 4) when the domain of the value V to store changes.

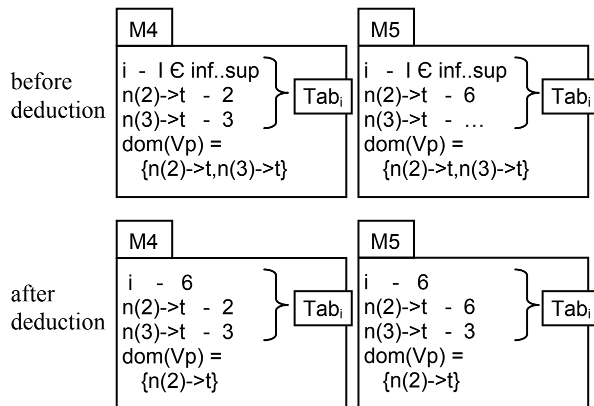
X should be non-null (proposition 1). Consider again statement `p->t=i;` in figure 7 and let Vp be a variable that points to the possible locations of p->t. The storage of the value of I in Vp is performed within the relation maintained by the store_element operator.

In the figure below, in the memories before deduction, as domains of n(2)->t before and after the storing statement are distinct, the value of n(2)-> t is changed and the value of I is stored in n(2)->t. As a consequence, Vp points to n(2)->t (proposition 4.3). dom(n(2)-> t) in M5 and dom(I) should be intersected in order to find the values of I and n(2)->t in M5 (proposition 4.1). The domain of n(3) -> t remains the same in both memories (proposition 4.2). We obtain the following memories:



$$
\begin{array}{ll}
(1) & X \neq NULL \\
(2)\wedge & (\forall <loc, V_0> \in Tab(M_{in}).\exists! <loc, V_1> \in Tab(M_{out})) \\
(3)\wedge & (\forall <loc, V_1> \in Tab(M_{out}).\exists! <loc, V_0> \in Tab(M_{in})) \\
(4)\wedge & (\forall (<loc, V_0>, <loc, V_1>) \in Tab(M_{in}) \times Tab(M_{out}) . \\
(4.1) & (\ (X = loc \rightarrow V = V_1) \\
(4.2) & \wedge\ (X \neq loc \rightarrow V_0 = V_1) \\
(4.3) & \wedge\ (dom(V_0) \cap dom(V_1) = \phi \rightarrow X = loc) \\
(4.4) & \wedge\ (dom(V) \cap dom(V_1) = \phi \rightarrow X \neq loc) \\
(4.5) & \wedge\ (dom(V_1) \subseteq dom(V_0) \cup dom(V)))) \\
(5)\wedge & (closed(M_{in}) \vee closed(M_{out}) \\
 & \rightarrow closed(M_{in}) \wedge closed(M_{out}))
\end{array}
$$

Figure 11. store_element operator



Other deductions associated with the operator include the following rules:

- Any couple loc-Var existing in one of the two memories should also appear in the other memory (propositions 2 and 3);
- For all the pairs (<loc-V_0>,<loc-V_1>) in Tab(M_{in}) x Tab(M_{out}):
  - If dom(V)∩dom(V_1)≠Ø, the variable V cannot be stored at the location loc, so X ≠ loc (proposition 4.4)
  - In other cases, we can deduce that dom(V_1) is included in dom(V_0) ∪ dom(V) (proposition 4.5)

The solving process fails if the domain of X, or the domain of an abstract variable stored in M_{in} or M_{out}, or the domain of V becomes empty.

After constraint propagation, store_element succeeds if M_{in} is closed and the value pointed by X is known. Indeed, in this case all the information that permits to deduce the contents of Tab(M_{in}) and Tab(M_{out}) is available.

4.6. The load_element operator

The operator load_element(Tab(M),X,V) is added when the program loads an integer or pointer value from the memory at a given address. Loading a value does not modify the memory so it constraints only a single memory. Figure 12 illustrates the operator that maintains a relation between X, V and M, where pointer X points to a variable V in the corresponding tableau of M.

Proposition 2 expresses that for all the pairs <loc-V_{ar}> in Tab(M):

waking_up =
(1) change(card(Tab(M)))
(2) ∨ ∃ <loc, Var> ∈ Tab(M). change(Var)
(3) ∨ change(X)
(4) ∨ change(V)

load_element (Tab(M),X,V)

suspended — suspend = ¬fail ∧ ¬exit

evaluation

$$
\begin{array}{ll}
(1) & X \neq NULL \\
(2)\wedge & \forall (<loc, Var>) \in Tab(M) . \\
(2.1) & (\ (X = loc \rightarrow V = Var) \\
(2.2) & \wedge\ (dom(V) \cap dom(Var) = \phi \rightarrow x \neq loc))
\end{array}
$$

entailed — exit = ∃ <loc, Var> ∈ Tab(M). (X = loc ∧ V = Var) ∧ ¬ fail

fail — fail = dom(X) = ϕ ∨ dom(V) = ϕ

post

Figure 12. load_element operator

- If X points to loc, the variable V is loaded from the location loc and $V=V_{ar}$. The domain of V and V_{ar} is then $dom(V) \cap dom(V_{ar})$.
- If $dom(V) \cap dom(V_{ar}) \neq \emptyset$, the variable V cannot be loaded from the loc location and then $X \neq loc$.

The solving process fails if the domain of X or V becomes empty, while it succeeds if there is a pair $<loc,V_{ar}>$ in Tab(M) such that X=loc and $V=V_{ar}$.

5. Experimental results on the Josephus program

We implemented the operators that are described above. Our system is able to take a program written in a restricted syntax of the C language and generates automatically test data w.r.t. some testing objectives. The system is developed in C and Prolog and follows the principles of the previous implementation InKa [13,14]. In our first experiment, we generated test data for the Josephus program in figure 1. We considered several testing objectives. Among them, we generated a test suite that covers all the branches of the program in less than 120sec of CPU time. The results were computed on an Intel Pentium, 2.16 GHz machine running Windows XP with 2.0Go of RAM. The test suite contains {(0,0), (0,2) ,(2,2)} as values for m and n. We also dealt with more complex requests as the one described in introduction of this paper. For the objective of unrolling k times the loop **while2**, we obtain the results shown in table 1. Although being too preliminary to conclude on our approach, we claim that these results are promising because they confirm the high deductive potential of our approach.

Table 1. Testing objective: 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

6. Related Works

Williams et al. [5], Sai-ngern et al. [21] and Sen [7] address the problem of generating test data for C functions with dynamic structures by using symbolic execution and constraint solving techniques. In their approaches, constraints on input values, representing preconditions, force pointing relationships in generated tests. But in the absence of such preconditioning constraints, separation of input pointers values is assumed. Consequently it is not possible to produce aliasing relations between inputs that haven't been required by the user. In our approach, we propose dynamic memory management operators able to deal with pointer aliasing problems that are due to the source code, and also to produce aliasing relationships on inputs if they are necessary to fulfil the test objective.

PathCrawler [5] and CUTE [7] are two test data generators which try to cover all the feasible paths of C programs. Both systems try to solve path conditions in order to find the next test data that will follow a path that improves the current coverage of the program. These tools are path-oriented, meaning that they require a path (or path-prefix) to be selected first. When the coverage criteria is weaker (all branches for example), considering all paths that reach a given branch is usually unreasonable as the number of control flow paths can be exponential on the number of decisions of the program, and the number of decisions depends on the bounds of loops that are often only limited by the size of integers. Our approach is goal-oriented, and is able, taking into account the goal objective, to quickly detect and abandon paths incompatible with it. Moreover, thanks to the constraint reasoning on operators, our implementation allows more complex testing objectives, such as reaching a selected point at certain iterations of a while loop, to be expressed. This is particularly interesting for programs that build dynamic data structures as such requests help verify their shapes during testing. However, one disadvantage of our approach is that it requires the constraint solver to be adapted and modified which prevents the usage of some commercial solvers that are sometimes more efficient.

7. Conclusion and perspectives

In this paper, we have presented new constraint operators on which is based a constraint-based method for generating automatically structural test data for programs with dynamic structures. These operators handle dynamic memory allocation, deallocation, loading and update. We have implemented these operators within a test data generator for programs written in a restricted subset of C programs. We also have generated test data for the complex program Josephus that creates and modifies dynamically allocated structures. This paper has focussed on intra-procedural test data generation and our future work is dedicated to extend this approach to inter-procedural test data generation and to complex constructions of the C language. Dealing with function calls and dynamic memory allocation requires paying attention to how constraint systems are built as the number of

constraints can grow exponentially with the number of function calls. This is our main challenging problem in order for the approach to scale up to large-sized programs

8. References

[1] R.A. DeMillo and J.A Offut, *"Constraint-Based Automatic Test Data Generation"*, TSE, 1991, pp. 900-910.

[2] D. Lewin, L. Fournier, M. Levinger and E. Roytman and G. Shurek, *"Constraint Satisfaction for Test Program Generation"*, IEEE International Phoenix Conference on Communication and Computers, 1995.

[3] E. Bin, R. Emek, G. Shurek and A. Ziv, *"Using a constraint satisfaction formulation and solution techniques for random test program generation"*, IBM Systems Journal 41 (3), 2002, pp. 386-402.

[4] A. Gotlieb, B. Botella and M. Rueher, *"Automatic Test Data Generation Using Constraint Solving Techniques"*, ISSTA98, 23 (2), Clearwater Beach, FL, USA, 1998, pp. 53-63.

[5] N. Williams, B. Marre, P. Mouy and M. Roger, *"PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis"*, In Proc. Dependable Computing, EDCC 2005: 5th European Dependable Computing Conference, Eds. Mario Dal Cin, Mohamed Kaâniche, András Pataricza, LNCS Vol. 3463/2005, Springer-Verlag GmbH, Budapest, Hungary, 2005, pp. 281-292.

[6] N. Tillmann and W. Schulte, *"Parameterized unit tests"*, ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, Lisbon, Portugal, 2005, pp. 253-262.

[7] K. Sen, D. Marinov and G. Agha, *"CUTE: a concolic unit testing engine for C"*, ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, Lisbon, Portugal, 2005, pp. 263-272.

[8] B. Marre and A. Arnould, *"Test Sequences Generation from LUSTRE Descriptions: GATEL"*, Proc. of the 15th IEEE Conference on Automated Software Engineering (ASE'00), IEEE CS Press, Grenoble, France, 2000, pp. 229-237.

[9] B. Seljimi and I. Parissis, "*Using CLP to Automatically Generate Test Sequences for Synchronous Programs with Numeric Inputs and Outputs"*, ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering, IEEE Computer Society, vol. 0, Washington, DC, USA, 2006, pp. 105-116.

[10] D. Jackson and M. Vaziri, *"Finding bugs with a constraint solver"*, ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, ACM Press, Portland, Oregon, United States, 2000, pp. 14-25.

[11] C. Pasareanu, M. Dwyer and W. Visser, *"Finding Feasible Abstract Counter-Examples",* International Journal on Software Tools for Technology Transfer (STTT), 5 (1), 2003, pp. 34-48.

[12] H. Collavizza, and M. Rueher, "*Exploration of the Capabilities of Constraint Programming for Software Verification"*, Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), Vienna, Austria, 2006, pp. 182-196.

[13] A. Gotlieb, B. Botella and M. Rueher, "*A CLP Framework for Computing Structural Test Data*", Proceedings of Computational Logic (CL'2000), LNAI 1891, 2000, pp. 399-413.

[14] A. Gotlieb, B. Botella and M. Watel, *"InKa: Ten years after the first ideas"*, 19th International Conference on Software and Systems Engineering and their Applications (ICSSEA'06), Paris, France, 2006

[15] C. Meudec, "*ATGen: automatic test data generation using constraint logic programming and symbolic execution"*, Software Testing, Verification and Reliability (STVR), 11 (2), 2001, pp. 81-96.

[16] http://research.microsoft.com/projects/mutt/

[17] http://www.polytech.unice.fr/~rueher/CEP/en/

[18] R.Sedgewick, *"algorithms in C"*, Addison-Westley publishing company, 1988.

[19] A. Gotlieb, T. Denmat and B. Botella, *"Goal-oriented test data generation for programs with pointer variables"*, 29th IEEE Annual International Computer Software and Applications Conference (COMPSAC'05), Edinburh, Scotland, 2005, pp. 449-454.

[20] B.Botella, A. Gotlieb and C. Michel, *"Symbolic execution of floating-point computations"*, The Software Testing, Verification and Reliability journal, 16 (2), John Wiley and Sons Ltd., Chichester, UK, 2006, pp. 97-121.

[21] S. Sai-ngern, C. Lursinap and P. Sophatsathit, *"An address mapping approach for test data generation of dynamic linked structures",* Information and Software Technology 47 (3), 2005, pp. 199-214.