Fourniture F1.6

Rapport :Expérimentation

Version 1.0

Historique

Date	Version	Description
18/07/2007	1.0	Version initiale

Auteurs

Auteur	Partenaire	
OURGHANLIAN Alain	EDF	Rédacteur principal
BOZGA Marius	Verimag	Contributeur
ROGALEWICZ Adam	Verimag	Contributeur
SANGNIER Arnaud	EDF – LSV	Contributeur

Table des matières

Expérimentation

1. Introduction

Dans le cadre du projet RNTL AVERILES plusieurs prototypes d'outils sont en cours d'élaboration pour d'une part générer des modèles formels à partir d'une analyse statique de code C d'un logiciel (voir lot 1 du projet), et d'autre part pour exploiter ces modèles afin de vérifier des propriétés portant sur la mémoire allouée dynamiquement par le logiciel à analyser (voir lot 2 du projet).

Les propriétés recherchées sont de deux types :

- L'absence de défauts intrinsèques dans la gestion de la mémoire. Par défauts intrinsèques, nous entendons notamment l'absence de fuite mémoire (une cellule allouée n'est plus accessible par le logiciel), l'accès à une zone précédemment désallouée ou à une zone mémoire invalide (défaut de segmentation).
- Et des propriétés plus fonctionnelles, comme la préservation de forme des structures de données construites dynamiquement. Par exemple un logiciel manipulant une liste simplement chaînée rend une liste simple simplement chaînée.

2. Présentation des cas d'études industrielle

Pour définir ses besoins en outils d'analyse de code et pour évaluer les prototypes d'outils développés par le projet, EDF a extrait deux cas d'étude issus d'applications industrielles.

2.1 Analyseur de trame réseau

L'analyseur de trame est utilisé pour observer les échanges de messages sur un réseau de type ethernet. Certains messages sont collectés et reconstitués à partir des trames observées, puis enfin stockés dans une base de données. D'autres applications vont alors se connecter à cette base pour exploiter ces informations qui ne font pas partie du cas d'étude.

L'architecture est composée de 6 threads coordonnées par des signaux, dont 4 principaux. Chacun des 4 threads réalise une étape du traitement de reconstitution des messages d'application à partir des trames réseaux observées. Ils utilisent l'allocation dynamique de mémoire pour stocker les informations collectées sur le réseau de communication. Bien que ce logiciel soit multi-tâches la gestion dynamique de la mémoire est intra-thread. Autrement dit, toute cellule allouée par un thread est rendue au système d'exploitation par ce même thread. L'allocation est faite uniquement sur des cellules atomiques, c'est à dire des cellules qui ne sont pas liées entre elles par des sélecteurs. Nous avons identifié deux grands types d'utilisation de la mémoire allouée dynamiquement.

La première utilisation, récurrente sur les 4 principaux thread, est illustrée par le code C suivant. Elle consiste, pour chaque thread, à récupérer dans une cellule allouée dynamiquement la trame en cours de construction élaborée par le thread de l'étape précédente. Cette trame est ensuite traitée par le thread. En fin d'étape de calcul la cellule allouée est détruite.

```
extern unsigned int dataSize;
extern unsigned char *bufLLC;

int Buf_Get(unsigned char *buffer, unsigned char **data, unsigned int taille_trame)
{
  int retVal = SPY_OK;
  volatile int erreur_buffer, buffer_vide;

  *data=NULL;
  if (!erreur_buffer){
    if (!buffer_vide){
      // allocation de la cellule pour y stocker la trame
      if ((*data=malloc((size_t)(taille_trame))) == NULL)
        return BUFFER_OVERFLOW;
      // Traitement initialisant la cellule allouée par memcpy
      return (SPY_OK);
    }
    else return(BUFFER_EMPTY);
```

```
  }
  else return(BUFFER_UNKNOWN);
}

int Recuperation_trame(void)
{
  unsigned char   *buffer;
  unsigned int    GO = TRUE;
  int             retVal;

  while (GO){
    retVal=Buf_Get(&bufLLC,&buffer,&dataSize );
    switch (retVal) {
    case BUFFER_UNKNOWN:
      GO=FALSE;
      break;

    case BUFFER_EMPTY:
      free(buffer);
      break;

    case SPY_OK :
      // Filtrage et ré-assemblage des trames réalisés dans un appel de fonction
      free(buffer);
      break;

    default:
      break;
    }
  }
}
```

La seconde utilisation de l'allocation dynamique de mémoire permet de stocker les trames en cours de construction. En effet l'information à reconstituer peut être répartie sur plusieurs trames réseau. Pour cette utilisation le code C est plus complexe, car il utilise de l'arithmétique de pointeur (pour « décortiquer » les différents champs des trames) et de l'arithmétique entière car ces cellules sont stockées dans des tableaux (par exemple liste des liaisons observées par l'analyseur, historique des connexions). Pour être analysable par les outils ce deuxième cas d'utilisation nécessitera une réécriture du code.

Les propriétés recherchées sont :

- Le logiciel ne libère une cellule précédemment désallouée (le double free() en C a un comportement non défini par la norme C ANSI).
- Absence de fuite mémoire.
- Pas d'accès à une cellule désallouée.

Ce premier cas d'étude n'a pas encore fait l'objet d'évaluation par le projet AVERILES. La définition du C simplifié (voir lot 1) ainsi que la mise à disposition prochaine d'un générateur de modèle à partir du code source C simplifié permettra de mener une évaluation des premiers prototypes.

2.2 Gestion de données de configuration de tranches

Ce second cas d'étude est issu d'une application de gestion des données de configuration d'une installation de production d'énergie. Les données de configuration caractérisent tous les matériels présents sur l'installation (capteurs, moteurs, pompes, …) et qui doivent être recensés dans le système de supervision de la centrale. Au cours du temps, des opérations de maintenance ou des évolutions de matériel font que des matériels sont changés ou modifiés. Il s'agit alors de mettre à jour les données de configuration.

Le logiciel de gestion de ces données stocke sous forme de listes doublement chaînées la configuration d'une installation. Le cas d'étude est issu de la bibliothèque de fonctions utilisée pour gérer ces listes chaînées : primitives d'ajout, de suppression d'un élément de la liste et primitive de destruction de la liste.

Les propriétés recherchées sont :

- L'absence de fuite mémoire.
- Pas d'accès à une cellule désallouée.
- La conservation de la forme de la liste par la fonction d'ajout ou de suppression d'un élément de la liste, et l'obtention d'une liste vide pour la destruction.

Une première évaluation de ce cas d'étude a été réalisée avec l'outil *artmc* (§ 3.1).

3. Expérimentations des outils

3.1 L'outil ARTMC (Abstract Regular Tree Model Checking) de VERIMAG

3.1.1 Présentation des cas d'études

Nous avons effectué des expérimentations avec des listes simplement chaînées (SLL), des listes doublement chaînées (DLL), des arbres, des listes de listes, et des listes de tâches. Pour tous ces exemples nous avons vérifié l'absence d'erreur de segmentation générée par l'accès aux pointeurs NULL, non-définis ou des-alloués. De plus, dans certains cas, nous avons aussi vérifié quelques propriétés sur la forme de la mémoire (comme l'absence de partage, l'acyclicité des listes, la préservation des éléments en entrée, etc.).

Listes simplement chaînées

Nous avons expérimenté deux études de cas sur les listes simplement chaînées (SLL) – la création de listes et le renversement. Dans le premier cas, la configuration initiale était vide. Dans le deuxième cas, nous avons utilisé un modèle de « toutes les listes simplement chaînées ». Dans les deux cas, nous avons vérifié la propriété suivante sur la forme : *la liste est acyclique*. De plus, dans le deuxième cas, nous avons vérifié que la procédure implémente réellement le renversement.

Listes doublement chaînées

Nous avons considéré 4 études de cas manipulant des listes doublement chaînées (DLL) – insertion d'un élément (dans une position arbitraire), extraction d'un élément (dans une position arbitraire), renversement d'un DLL et tri par insertion d'une DLL. Pour tous ces cas d'étude, nous avons considéré comme entrée un modèle couvrant tous les types de listes doublement chaînées possibles. A l'exception du tri par insertion, nous avons vérifié les propriétés suivantes : (1*) la liste est acyclique* et (2) *la liste est correctement chaînée* (x->next->back = x). De plus, dans le cas du renversement nous avons aussi vérifié que la procédure l'implémente correctement.

Arbres

Nous avons vérifié 4 cas d'étude opérant sur des structures d'arbre. Le premier est l'insertion dans un arbre. Il démarre avec un arbre vide, et réalise un nombre arbitraire d'insertion dans les feuilles. Pour cet exemple, nous avons vérifié que : (1) *il n'y a pas de partage* et (2*) il n'y a pas de cycle* dans la structure construite. Le deuxième cas d'étude est le parcours des arbres en profondeur d'abord. Comme configuration initiale, nous avons pris un modèle de tous les arbres binaires avec des pointeurs vers les nœuds parents, et où chaque nœud a une donnée booléenne *«marked = false»*. A la fin, nous avons vérifié que tous les nœuds ont été visités : *pour tous les nœuds, nous avons « marked = true »*. Ensuite, nous avons considéré le parcours Deutsh-Schorr-Waite. Nous avons pris comme entrée un modèle de tous les arbres binaires. Pour cet exemple, nous n'avons pas vérifié de propriétés sur la structure. Finalement, le dernier cas d'étude réalise le chaînage des feuilles d'un arbre dans une liste simplement chaînée. Comme entrée, nous avons pris un modèle de tous les arbres binaires possibles, avec des pointeurs vers le nœud parent. A la fin, nous avons vérifié la propriété suivante sur la forme : *quelque soit deux feuilles, arbitrairement choisies dans l'arbre initial, elles sont correctement liées* après l'exécution de la procédure.

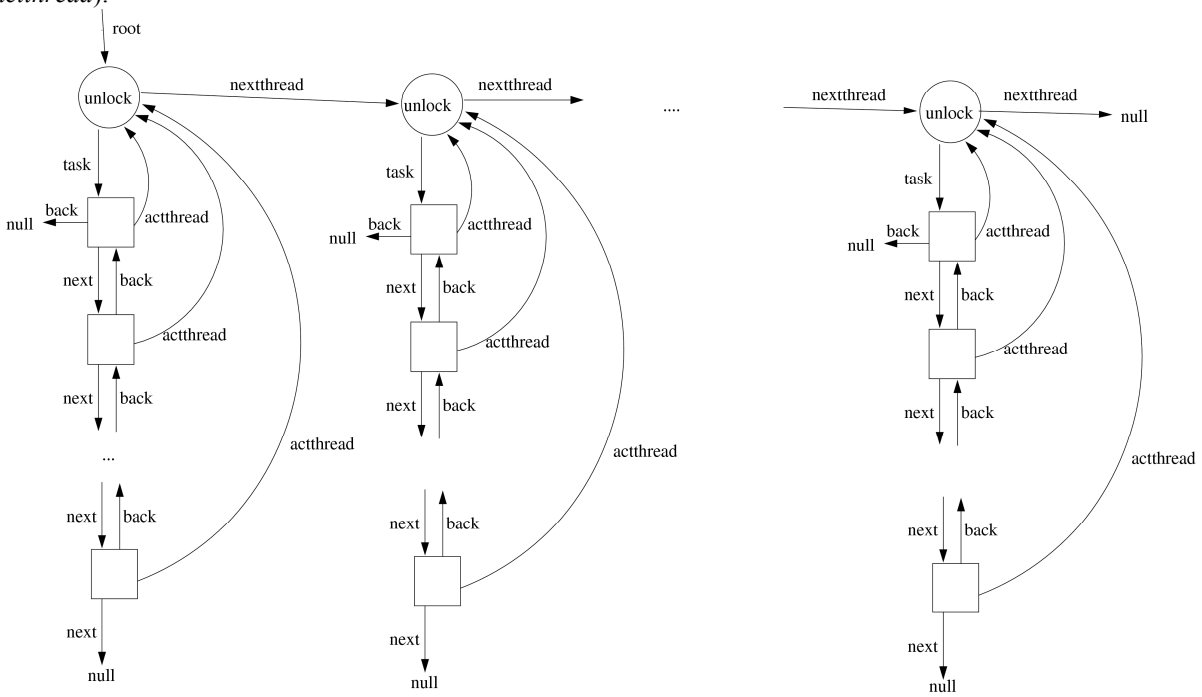
Listes de listes

Dans ce cas d'étude, nous avons considéré une liste simplement chaînée, dont les éléments sont à leur tour, des listes simplement chaînées. Nous prenons comme configuration initiale, une liste vide et nous effectuons un nombre arbitraire d'insertions. A la fin, nous avons vérifie la propriété suivante sur la forme obtenue : il n'y a pas de partage entre deux listes différentes.

Liste de tâches

Cet étude de cas considère une structure de données rencontrée habituellement dans les noyaux des systèmes d'exploitation [Berdine'07]. La structure est illustrée dans la figure suivante. Elle est composée d'une liste simplement chaînée de threads (nœuds ronds), et pour chaque thread, une liste doublement chaînée des tâches

(nœuds carrés). Tous les threads ont un pointeur vers le thread suivant (*nextthread*) et un autre vers la première tâche. Le premier thread est pointé par une variable du programme (*root*). De plus chaque thread possède un lock pour assurer l'exclusion mutuelle en cas de manipulation concurrente de la liste des tâches associées. Finalement, chaque tâche a des pointeurs vers la tâche suivante (*next*), précédente (*prev*) et vers le thread la contenant (*actthread*).



Nous avons utilisé cette structure pour deux études de cas, respectivement, l'extraction d'un thread de la liste et l'insertion d'un nouveau thread dans cette liste. Dans les deux cas, nous avons considéré un modèle représentant toutes les listes de tâches possibles. Dans les deux cas nous avons également vérifié la propriété suivante sur la forme : (1) *il n'y a pas de partage entre les listes des tâches attachées a deux threads différents*. De plus, dans le cas de l'insertion, nous avons vérifié aussi que : (2) *il n'y a pas de boucles dans les listes des tâches*, (3) *tous les threads sont unlocked*, et (4) *tous les pointeurs des tâches vers le thread parent (actthread) sont positionnés correctement*.

Second cas d'étude EDF

Nous mené une évaluation sur deux des fonctions du second cas d'étude EDF (voir § 2.2). Il s'agit des fonctions d'insertion d'un élément à la fin d'une liste doublement chaînée et de la suppression d'un élément particulier désigné par un pointeur transmis comme argument à la fonction. Sur ces deux fonctions nous avons vérifié la propriété de forme suivante : *La fonction retourne une liste correctement chaînée (x->next->back = x).*

3.1.2 Bilan des analyses

Le tableau suivant résume les résultats obtenus par notre outil *artmc* sur les exemples décrits ci-dessus. Nous précisons les meilleurs résultats obtenus et avec quelle abstraction parmi les deux schémas générales et l'abstraction *one-simple-neighbour* (prouvé très utile en pratique). La mention *restricted* annote les cas où l'abstraction a été appliquée seulement à certains points (les boucles). Les résultats ont été obtenus sur un 64bit Opteron 2,8 GHz. La colonne $|Q|$ donne de l'information sur la taille (nombre d'états) du plus grand automate construit, et *Nref* donne de l'information sur le nombre de pas de raffinement.

Exemple	***Test of shape properties***	***Time***	***Abstraction method***	***|Q|***	***Nref***
Creation of SLLs	Yes	1s	predicates, restricted	25	0

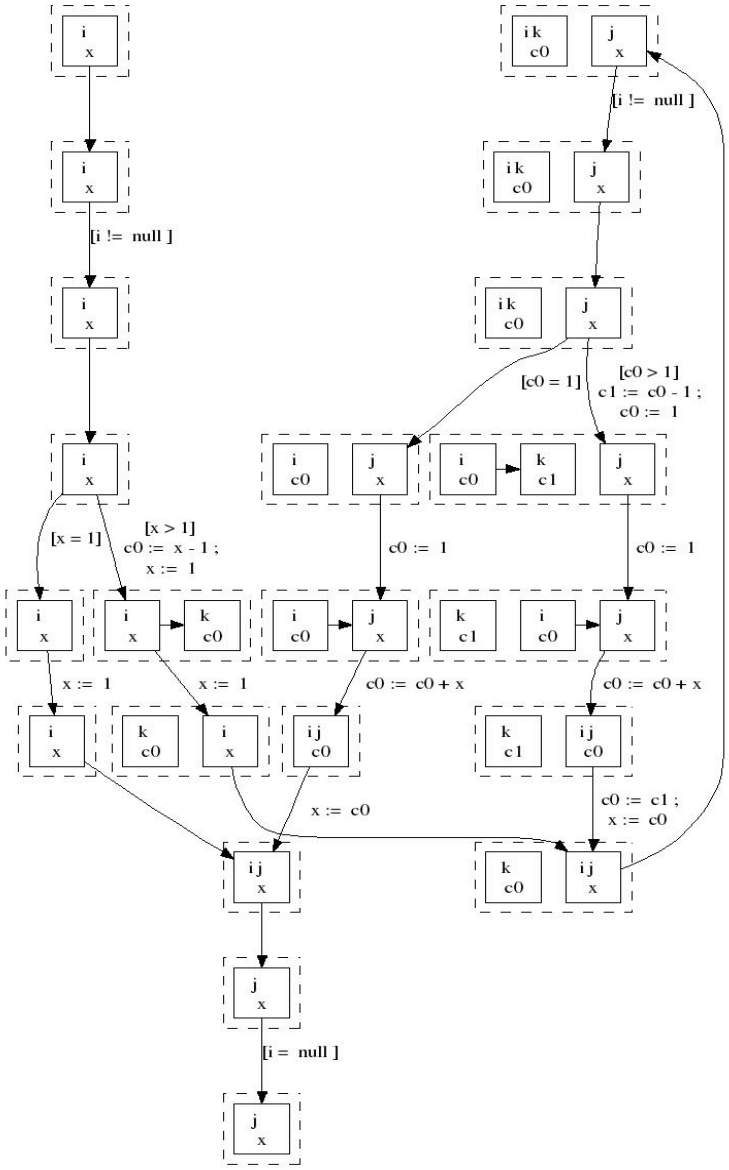
Exemple	*Test of shape properties*	*Time*	*Abstraction method*	*|Q|*	*Nref*
Reversion of SLLs	Yes	5s	Predicates	52	0
Deletion from DLLs	Yes	6s	finite height	100	0
Insertion into DLLs	Yes	10s	neighbour, restricted	106	0
Reversion of DLLs	Yes	7s	Predicates	54	0
Insertsort of DLLs	No	2s	predicates	51	0
Inserting into trees	Yes	23s	predicates, restricted	65	0
Depth-first search	Yes	11s	predicates	67	1
Linking leaves in trees	Yes	40s	predicates	75	2
Inserting into a list of lists	Yes	5s	predicates, restricted	55	0
Deutsch-Schorr-Waite tree traversal	No	47s	predicates	126	0
Insertion into task-lists	Yes	11m 25s	finite height, restricted	277	0
Deletion in task-Lists	Yes	1m 41s	predicates, restricted	420	0
2nd EDF case study DLL-insertion	Yes	3s	predicates, restricted	50	0
2nd EDF case study DLL-remove	Yes	11s	finite height	132	0

3.2 L'outil L2CA (List To Counter Automata) de VERIMAG

3.2.1 Présentation des cas d'études

Nous avons appliqué l'outil *l2ca* sur des programmes de manipulation (insertion, destruction, renversement) des listes, et sur des programmes (abstraits) de tri. Au cas par cas, nous avons considéré une ou plusieurs tas en entrée de ces programmes.

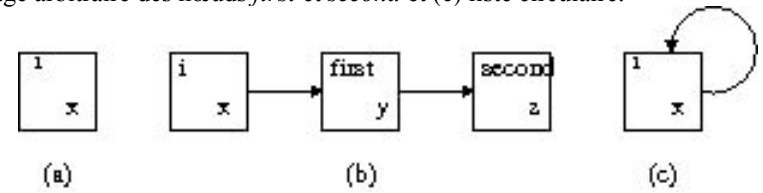
La figure suivante illustre le type de résultat produit par *l2ca*. Il s'agit d'un automate a compteurs, qui peut être écrit sous différents formats et analysé avec des outils appropriés (*fast*, *artmc*, *aspic*, ...).



Concernant les propriétés à vérifier, pour l'instant, nous nous sommes intéressés seulement aux erreurs (intrinsèques) de segmentation. La présence de telles erreurs dans le programme s'expriment par l'atteignabilité d'un état prédéfini dans l'automate à compteurs que nous construisons. Nous avons utilisé l'outil *aspic* (http://www-verimag.imag.fr/~gonnord/aspic/aspic.html) pour vérifier cette propriété.

3.2.1.1 Manipulation des listes

Nous avons analysé des programmes simples implémentant des opérations élémentaires sur des listes, circulaires et/ou non-circulaires. Pour la création d'une liste, le tas en entrée est vide, pour les autres opérations, les tas considérés en entrée sont illustrés dans la figure suivante, respectivement (a) liste non circulaire simple, (b) liste non-circulaire avec marquage arbitraire des nœuds *first* et *second* et (c) liste circulaire.



Création d'une liste

```
pointer i, j;
int k;

i = null;
k = 0;
while ( !(k == 10)) do
       j = new;  j.next = i;  i = j;
       k = k+1;
od
```

Destruction d'une liste

```
pointer i;
int k;

k = 0;
while ( !(i == null) ) do
       i = i.next;
       k = k+1;
od
```

Renversement d'une liste

```
pointer i,j,k;

while !(i == null) do
       k = i.next;  i.next = j;  j = i;  i = k;
od
```

3.2.1.2 Algorithmes de tri

La deuxième partie de nos expérimentations porte sur des programmes implémentant les algorithmes de tri les plus connus - par fusion, par sélection, par insertion et à bulles - opérant sur des données stockées dans les cellules d'une liste simplement chaînée. Nous sommes partis des programmes opérant *sur place* c'est-à-dire, pendant le tri, les données ne change pas de cellule, seulement les sélecteurs sont modifiés pour obtenir finalement une liste correctement triée. Cependant, nous avons considéré des versions *abstraites* de ces programmes où nous avons ignoré les données et remplacé le résultat de leur comparaison par du non-déterminisme.

Les tas considérés en entrée par les programmes de tri sont illustrés dans la figure suivante, respectivement (a) pour le tri par fusion et (b) pour les autres cas.

(a) (b)

Tri par fusion

```
pointer i, j, k, t;

k = null; t = null;
while ( (!(i == null)) && (!(j==null)) ) do
       if ( void ) then
              t = i;  i = i.next;
       else
              t = j;  j = j.next;
       fi
       t.next = k;  k = t;
```

```
od
if ( i == null ) then
       i = j;
fi
while (! (i == null) ) do
       t = i;  i = i.next;  t.next = k;  k = t;
od
```

Tri par sélection

```
pointer l, L, t;
pointer min, pmin, i, pi;

L = null;
if (! (l==null) ) then
       L = l;  t = l;  l = l.next;  t.next = null;
fi
while (! (l == null) ) do
       min = l;  pmin = null;  i = l.next;  pi = l;
       while (! (i == null) ) do
              if ( void ) then
                     min = i;  pmin = pi;
              fi
              pi = i;  i = i.next;
       od
       if ( !(pmin == null)) then
              pmin.next = min.next;
       else
              l = l.next;
       fi
       t.next = min;  t = t.next;  t.next = null;
od
```

Tri par insertion

```
pointer l, L;
pointer i, pi, pn;

L = null;
if (! (l == null) ) then
       L = l;  l = l.next;  L.next = null;
fi
while (! (l == null)) do
       i = l;  l = l.next;
       if ( void ) then
              i.next = L;  L = i;
       else
              pi = L;
              while (void) do
                     pn = pi.next;
                     if (! (pn == null) ) then
                            pi = pi.next;
                     fi
              od
       i.next = pi.next;  pi.next = i;
       fi
od
```

Tri à bulles

```
pointer l, i, pi, j, t;
int sorted;

if (! (l==null)) then
       t = l.next;
       if (! (t == null)) then
              sorted = 0;
              while (sorted == 0) do
                     sorted = 1;
                     pi = null;  i = l;  j = l.next;
                     while (! (j == null)) do
                            if (void) then
                                   t = j.next;  i.next = t;
                                   j.next = i;  i = j;  j = j.next;
                                   if (pi == null) then
                                          l = i;
                                   else
                                          pi.next = i;
                                   fi
                                   sorted = 0;
                            fi
                            pi = i;  i = j;  j = j.next;
                     od
              od
       fi
fi
```

3.2.2 Bilan des analyses

Le tableau suivant résume les expérimentations faites avec *l2ca*. Pour chaque exemple considéré, nous précisons la taille de l'automate à compteurs construit (*Q* nombre d'états, *T* nombre de transitions, *X* nombre de compteurs), le temps de génération par *l2ca* (*temps*, en secondes) puis, l'existence ou non des erreurs de segmentation.

Exemple	***Q***	***T***	***X***	***temps***	***SegF***
List creation	18	18	4	0.148	No
List destruction	11	11	4	0.148	No
Noncircular list reversal	21	22	4	0.156	No
Noncircular list reversal	86	93	7	0.204	No
Circular List reversal	68	73	5	0.196	No
Merge sort	109	119	6	0.188	No
Selection sort	3564	4109	9	3.260	No
Insertion sort	1337	1633	8	0.612	No
Bubble sort	505	577	8	0.308	No

3.3 L'outil TOPICS (Translation Of Program Into Counter System) du LSV – EDF

3.3.1 Présentation générale

L'outil TOPICS est en cours de développement. Toutefois, nous avons dans un premier temps implémenté un prototype en OCAML de façon à tester la traduction de programmes manipulant des listes simplement chaînées vers des automates à compteurs présentée dans [1]. Nous avons réalisé des expérimentations sur quelques exemples de programmes. Ces programmes ont été abstraits pour ne considérer que la structure des listes simplement chaînées sans considérer les données contenues dans les cellules des listes. Ainsi lorsque celles-ci interviennent dans un test, nous transformons ce test en un branchement indéterministe. Par exemple si la structure de donnée considérée est la suivante (dans la syntaxe du C) alors tous les appels au champ data sont éliminées.

```
typedef struct _list{
  int data ;
  struct _list * next ;
} * liste ;
```

L'outil produit des automates à compteurs au format de l'outil FAST [2]. La vérification du programme s'effectue donc en deux phases :

1. Traduction du programme vers un automate à compteurs
2. Vérification de l'automate à compteurs en utilisant FAST

Les propriétés vérifiées sur ces programmes sont l'absence de fuite mémoire (memory leak) et l'absence de violation mémoire (segmentation fault).

Nous avons également la possibilité de donner la forme du tas mémoire (i.e. de la liste simplement chaînée) donnée en entrée des programmes. Pour les exemples présentés ci-dessous, nous avons à chaque fois considéré le cas où la liste donnée était circulaire ou non.

3.3.2 Programmes vérifiés

Nous donnons ici, dans la syntaxe du langage C, les programmes manipulant des listes simplement chaînées sur lesquels nous avons mené nos expérimentations.

Reverse : Renverse l'ordre des éléments de la liste

```
List reverse(List x) {
       List y,t;
        y=NULL;
       while(x!=NULL){
              t=y;
              y=x;
              x=x->next;
              y->next=t;
              t=NULL; }
       return y; }
```

Delete : Efface un élément de la liste

```
List delete(List x, int delval) {
       List elem,prev,temp;
       elem=x; prev=NULL;
       temp=NULL;
       while (elem!=NULL) {
              if (elem->data == delval) {
                     if (prev==NULL) {
                            x=elem->next; }
                     else {
                            temp=elem->next;
                            prev->next=temp; }
                     elem->next=NULL;
                     free(elem);
                     return x; }
              prev=elem;
              elem=elem->next; }
       }
```

DeleteAll : Efface tous les éléments de la liste

```
void deleteAll(List x) {
       List elem;
       while(x!=NULL) {
              elem=x;
              x=x->next;
```

```
                elem->next=NULL;
                free(elem); }
}
```

Insert : Insère un élément dans une liste ordonnée

```
List insert(List x, int value) {
       List e,p,t;
       e=x;
       p=NULL;
       t=NULL;
       if (x==NULL) {
              x=malloc(sizeof(struct _list));
              x->next=e; }
       else {
              if (value >= x->val) {
                     x=malloc(sizeof(struct _list));
                     x->next=e; }
              else {
                     while (e!=NULL && value < e->val) {
                            p=e; e=e->next; }
                     t=malloc(sizeof(struct _list));
                     t->next=e;
                     p->next=t; } }
       return x; }
```

Merge : Regroupe dans une seule liste ordonnée deux listes ordonnées

```
List merge(list p, List q) {
       List h,t;
       h=NULL;
       t=NULL;
       if (p==NULL) { h=q; }
       else {
              if (q==NULL) { h=p; }
              else {
                     if (p->data < q ->data) {
                            h=p;
                            p=p->next; }
                      else { h=q;
                            q=q->next; }
                     t=h;
                     while (p!=NULL && q!=NULL) {
                            if (p->val < q->val) {
                                   t->next=p;
                                   p=p->next; }
                            else {
                                   t->next=q;
                                   q=q->next; }
                            t=t->next; }
                     if (p !=NULL) {
                            t->next=p; }
                     else {
                            if (q!=NULL) {
                                   t->next=q; } } } }
       return h; }
```

3.3.3 Bilan des analyses

Le tableau ci-dessus donne les résultats obtenus suite aux expérimentations. La colonne Q donne le nombre d'états de contrôle de l'automate à compteurs obtenus, la colonne T donne son nombre de transitions et la colonne X le nombre de compteurs utilisés. Chacun de ces exemples a été montré comme étant correct. Nous avons de plus

modifié volontairement certains de ces programmes afin qu'ils réalisent une erreur (de type fuite mémoire ou violation mémoire) et constater que les erreurs étaient effectivement détectées.

Exemple	***Q***	***T***	***X***	***MemLeak***	***SegFault***
Reverse (liste classique)	26	27	3	Non	Non
Reverse (liste ciculaire)	48	50	4	Non	Non
Delete (liste classique)	50	50	4	Non	Non
Delete (liste circulaire)	64	67	7	Non	Non
DeleteAll (liste classique)	9	9	2	Non	Non
DeleteAll (liste circulaire)	15	16	2	Non	Non
Insert (liste classique)	34	39	4	Non	Non
Insert (liste circulaire)	36	34	4	Non	Non
Merge (liste classique)	100	139	5	Non	Non

4. Références

[Berdine'07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang, Shape analysis for composite data structures, In Proc. of CAV'07.

[1] S. Bardin, A. Finkel, É. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. In R. Bharadwaj, editor, AVIS, Vienna, Austria, Apr. 2006.

[2] S. Bardin, J. Leroux and G. Point. FAST Extended Release. In Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06), Seattle, Washington, USA, August 2006, LNCS 4144, pages 63-66