Abstract Tree Regular Model Checking of Complex Dynamic Data Structures

with Ahmed Bouajjani, Adam Rogalewicz, Tomas Vojnar (Brno)

Peter Habermehl
LIAFA, University Paris 7

November 6th, 2006

Example program

```
// Doubly-Linked Lists
typedef struct {
    DLL *next, *prev;
} DLL;

DLL *DLL_reverse(DLL *x) {
    DLL *y,*z;
    z = NULL;
    y = x->next;
    while (y!=NULL) {
        x->next = z;
        x->prev = y;
        z = x; x = y;
        y = x->next
    }
    return x;
}
```

Properties

- The usual properties of use of memory (absence of intrinsic errors)

- Shape invariants

  - Using shape testers like

```
x = aDLLHead;
while (x != NULL && random())
   x = x->next;
if (x != NULL && x->next->prev != x)
   error();
```

  - Using formulæ of a logic like

$$l \xrightarrow{next^*} [\exists x.\ p \xrightarrow{next} x \ \wedge\ x \neq \bot \ \wedge\ \neg(x \xrightarrow{prev} p)]$$

which can be translated into shape testers

Verification approach

- Properties are translated to control line unreachability
- Verification using an automata-based framework
  - Encode memory configurations (shape graphs) as trees
  - Use finite-state tree automata to represent sets of configurations
  - Encode program statements as tree transducers (I/O automata)
  - Use Abstract Tree Regular Model Checking [BHRV '05]
    - Symbolic reachability analysis
    - Refinable abstractions on automata
- Implemented using Mona and applied to several case studies

Overview

- Properties considered
- Automata based verification approach
  - Encoding of sets of memory configurations
  - Encoding of program statements as transducers
- Experiments

Properties considered

- Basic consistency of pointer manipulations
  - absence of null and undefined pointer dereferences
  - no references to deleted nodes
- Shape invariance properties
  - like absence of sharing, acyclicity
  - if `x->next == y` in a DLL then also `y->prev == x`
- Absence of garbage

Specifying shape invariance properties

We describe negations of these properties

- Shape testers
- A logic of bad memory patterns
  - translated into shape testers

Shape testers

- Instrumentation code written in extended C
  - following pointers backwards
  - non-deterministic branching

- Added to the program

Checking consistency of the next and previous pointers

```
x = aDLLHead;
while (x != NULL && random())
   x = x->next;
if (x != NULL && x->next->prev != x)
   error();
```

A logic of bad memory patterns (LBMP)

- allows to describe bad shapes
- $\mathcal{V}$ finite set of program variables
- $\mathcal{S}$ finite set of selectors
- $\Phi ::= \exists w_1, ... w_n.\varphi$ with $\mathcal{W} = \{w_1, ..., w_n\}$ set of formulae variables
- $\varphi ::= \varphi \vee \varphi \mid \psi$, $\psi ::= \psi \wedge \psi \mid x\varrho y \mid x\varrho$
  $x, y \in \mathcal{V} \cup \mathcal{W}$ and ϱ is a reachability formula
- $\varrho ::= \stackrel{s}{\rightarrow} \mid \stackrel{s}{\leftarrow} \mid \varrho + \varrho \mid \varrho.\varrho \mid \varrho^* \mid [\sigma]$ where $s \in \mathcal{S}$ and σ a local neighbourhood formula
- LNF: $\exists u_1, ..., u_m.BC(x \stackrel{s}{\rightarrow} y, x = y)$ with $\mathcal{U} = \{u_1, ..., u_m\}$ a set of local formula variables, $s \in \mathcal{S}$, $x \in \mathcal{V} \cup \mathcal{W} \cup \mathcal{U} \cup \{p\}$, $y \in \mathcal{V} \cup \mathcal{W} \cup \mathcal{U} \cup \{p, \bot, \top\}$,
  p denotes the current position in the shape graph, $\bot$ is NULL and $\top$ undefined.

Examples for doubly linked lists

- $l \stackrel{next^*}{\longrightarrow} [p = \top]$

Examples for doubly linked lists

- $l \overset{next^*}{\longrightarrow} [p = \top]$
  The list ends with undefined

Examples for doubly linked lists

- $l \stackrel{next^*}{\longrightarrow} [p = \top]$
  The list ends with undefined

- $l[\neg(p \stackrel{prev}{\longrightarrow} \bot)]$

Examples for doubly linked lists

- $l \stackrel{next^*}{\longrightarrow} [p = \top]$
  The list ends with undefined

- $l[\neg(p \stackrel{prev}{\longrightarrow} \perp)]$
  the predecessor of the first element is not null

Examples for doubly linked lists

- $l \stackrel{next^*}{\longrightarrow} [p = \top]$
  The list ends with undefined

- $l[\neg(p \stackrel{prev}{\longrightarrow} \bot)]$
  the predecessor of the first element is not null

- $l \stackrel{next^*}{\longrightarrow} [\exists x.\ p \stackrel{next}{\longrightarrow} x \ \wedge \ x \neq \bot \ \wedge \ \neg(x \stackrel{prev}{\longrightarrow} p)]$

Examples for doubly linked lists

- $l \stackrel{next^*}{\longrightarrow} [p = \top]$
  The list ends with undefined

- $l[\neg(p \stackrel{prev}{\longrightarrow} \bot)]$
  the predecessor of the first element is not null

- $l \stackrel{next^*}{\longrightarrow} [\exists x.\ p \stackrel{next}{\longrightarrow} x \ \wedge \ x \neq \bot \ \wedge \ \neg(x \stackrel{prev}{\longrightarrow} p)]$
  the predecessor of the successor of a node is not the node itself

Examples for doubly linked lists

- $l \overset{next^*}{\longrightarrow} [p = \top]$
  The list ends with undefined

- $l[\neg(p \overset{prev}{\longrightarrow} \bot)]$
  the predecessor of the first element is not null

- $l \overset{next^*}{\longrightarrow} [\exists x.\ p \overset{next}{\longrightarrow} x \ \wedge \ x \neq \bot \ \wedge \ \neg(x \overset{prev}{\longrightarrow} p)]$
  the predecessor of the successor of a node is not the node itself

- $\exists x.\ l \overset{next^*}{\longrightarrow} [p = x] \overset{next}{\longrightarrow}\overset{next^*}{\longrightarrow} [p = x]$

Examples for doubly linked lists

- $l \overset{next^*}{\longrightarrow} [p = \top]$
  The list ends with undefined

- $l[\neg(p \overset{prev}{\longrightarrow} \bot)]$
  the predecessor of the first element is not null

- $l \overset{next^*}{\longrightarrow} [\exists x.\ p \overset{next}{\longrightarrow} x \ \wedge\ x \neq \bot \ \wedge\ \neg(x \overset{prev}{\longrightarrow} p)]$
  the predecessor of the successor of a node is not the node itself

- $\exists x.\ l \overset{next^*}{\longrightarrow} [p = x] \overset{next}{\longrightarrow}\overset{next^*}{\longrightarrow} [p = x]$
  the list is cyclic

Translation from LBMP to shape testers

- Suppose that all variables refered to in formula are reachable from $\mathcal{V}$
- One starts by exploring the memory configurations starting from the variables
- go to special location if formula holds
- Disjunction : non-deterministic branching
- Conjunction : series of tests
- Reachability formulae $\varrho ::= \xrightarrow{s} | \xleftarrow{s} | \; \varrho + \varrho \mid \varrho . \varrho \mid \varrho^* \mid [\sigma]$
  - $\xrightarrow{s} | \xleftarrow{s}$: following the appropriate selectors (forward or backward)
  - $\varrho . \varrho$: sequencing
  - $\varrho + \varrho$ and ϱ^* : non deterministic branching
- $[\sigma]$ can also be tested easily

The verification problem

- If a basic consistency error is encountered, the program goes to some designated error location.
- Negations of shape invariance properties are expressed as formulæ of LBMP.
- They are translated into shape testers.
- If an error location is reached, the shape invariance property is broken.

Verification amounts to checking for control location unreachability

- Our approach: use abstract regular tree model-checking

Regular Tree Model-Checking

[KMMPS ’97, BT ’02, AJMO ’02, ALOR ’05]

- Natural generalisation of Regular model-checking
- Configurations : trees (terms)
- Sets of configurations : finite tree automata (bottom-up)
- Operations: finite tree transducers (noted τ)
- Basic verification problem : Computing the transitive closure of a finite tree transducer

The verification problem

- Check: $\tau^*(Init) \cap Bad = \emptyset$
- Compute τ^* or
- For a given tree automaton A, compute $\tau^*(A)$

Abstract Regular (Tree) Model Checking

- Compute $(\alpha \circ \tau)^*(Init)$ instead of $\tau^*(Init)$

$$\tau^*(Init) \subseteq (\alpha \circ \tau)^*(Init)$$

- If $(\alpha \circ \tau)^*(Init) \cap Bad = \emptyset$ then answer YES

- else if $(\alpha \circ \tau)^*(Init) \cap Bad$ contains a real counterexample,
  then answer NO
  else refine the abstraction and start again

Automata state collapsing as abstractions

- We define an equivalence relation $\equiv$ on automata states
- We define an abstraction function $\alpha(A) = A/\equiv$
- We propose several equivalence relations to define abstractions
  - States are equivalent if they accept the same trees up to some fixed height
  - States are equivalent if their languages have non-empty intersections with the same predicate tree automata.
  - States are equivalent if they are neighbours
- Refinement: choose finer equivalence relation

Tree automata encoding of pointer manipulating programs

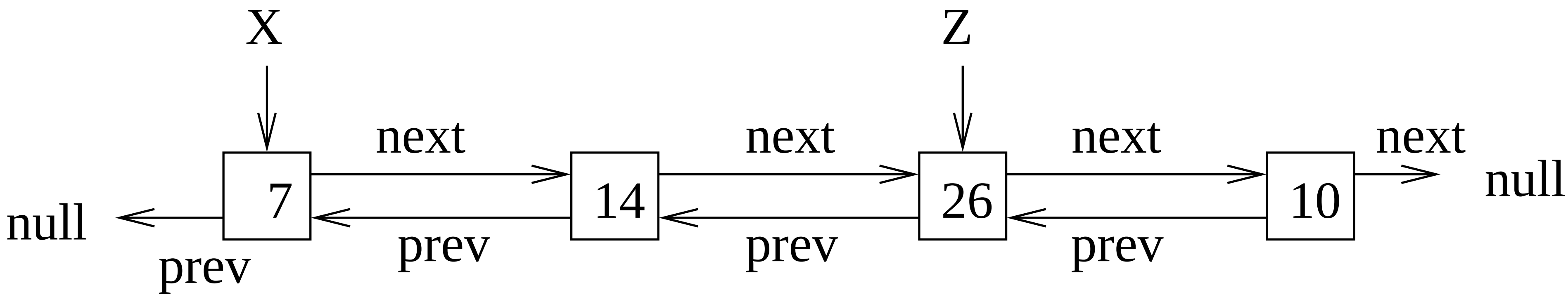
- Encoding of sets of memory configurations
- Encoding of program statements as transducers

Encoding of shape graphs as trees

- $\mathcal{S} = \{1, \ldots, k\}$ finite set of selectors, $\mathcal{V}$ finite set of pointer variables

- A shape graph is a tuple $SG = (N, S, V, D)$ where
  - N is a finite set of memory nodes,
  - $N_{\perp,\top} = N \cup \{\perp, \top\}$
  - $S : N \times \mathcal{S} \rightarrow N_{\perp,\top}$ is a successor function
  - $V : \mathcal{V} \rightarrow N_{\perp,\top}$ is a mapping that defines where the pointer variables are currently pointing to, and
  - $D : N \rightarrow \mathcal{D}$ defines what (finite) data is stored in the particular memory nodes.

Example of a shape graph



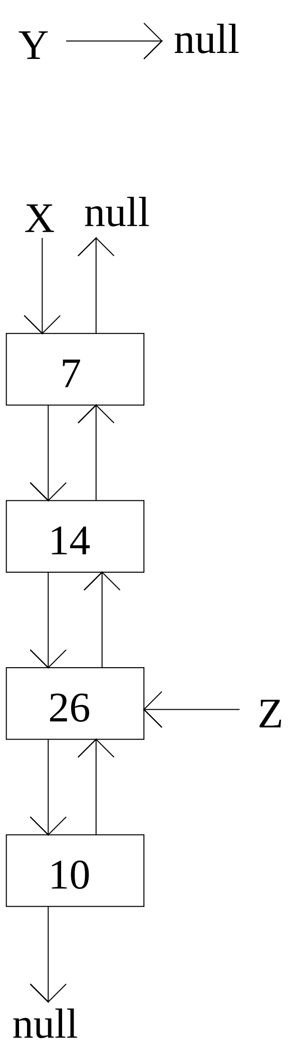
Encoding of shape graphs as trees

in the spirit of graph types [KS ’93] and PALE [MS ’02] but different

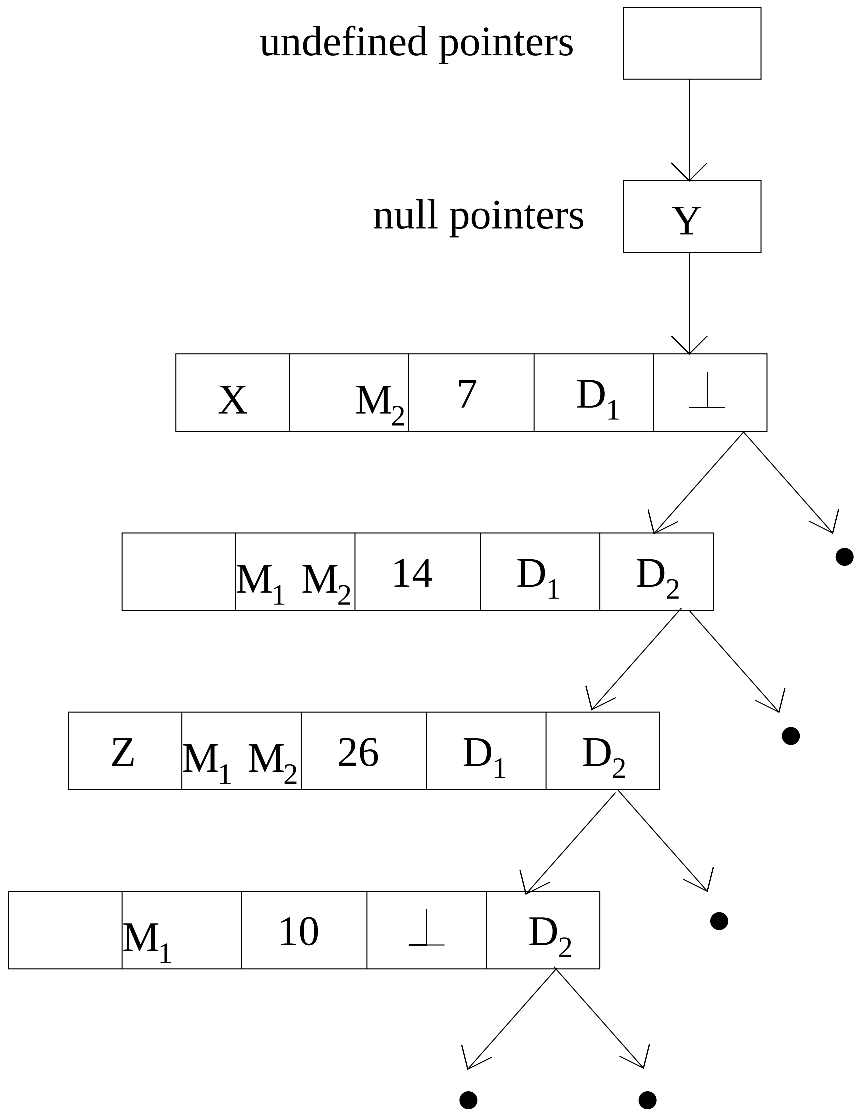
- Use trees as backbones
- describe links between nodes of the trees using pointer descriptors (with routing expressions expressing paths in the tree)

Example

The original DLL



A tree memory encoding of the DLL



Descriptors

$S = \{1, 2\}$ $S^{-1} = \{\bar{1}, \bar{2}\}$

$D_1 : 1.M_1$

$D_2 : \bar{1}.M_2$

Encoding

- Let $\mathcal{S}^{-1}$ be the set of inverted selectors
- We fix a number of pointer descriptors
  - which have a unique marker (indicating where the pointer can point to)
  - with a routing expression describing paths in the tress backbone
- Each routing expression is a regular expression on the alphabet of pairs $s.n \in (\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma$ where Σ is the alphabet for nodes (data, markers, etc.)
- A tree memory encoding is a tuple (t, μ) where t is a tree memory backbone and μ a mapping from pointer descriptors to routing expressions.

Encoding

- $[\![(t,\mu)]\!]$ is the set of shape graphs represented by t.
  - The nodes of the graph are internal nodes of the tree.
  - Links are obtained by following routing expressions

- A tree automata memory encoding is a tuple $[\![(A,\mu)]\!]$ with a tree automaton A

- A tree automata memory encoding represents the set of shape graphs $[\![(A,\mu)]\!] = \bigcup_{t\in L(A)} [\![(t,\mu)]\!]$.

- Remarks
  - The encoding is not canonical
  - (A,μ) and $[\![(A,\mu)]\!]$ are two different notions
  - Given (A,μ), $[\![(A,\mu)]\!]$ can be empty although A is not empty.

Encoding in Mona

- Use binary trees
- Routing expressions are “implemented” as tree transducers

Encoding of program statements as transducers

- Each pointer manipulation statement is encoded as a tree transducer
- We add also the current program line (or error location) to the configuration
- Transducers are constructed such that they simulate the effect of program statements on the corresponding shape graphs

Non-destructive updates and tests

- `x = null`
- `x = y`
- `if (x == null) then goto l1 else goto l2;`
- `x = y->s`
  - if `y->s` undef or null update `x` accordingly
  - else mark the `y` node with ⧫
  - apply corresponding routing expression transducer and move ⧫
  - remove `x` and put it into node marked by ⧫
  - can be non-deterministic if several targets are possible

Destructive updates

`x->s = y`

- To each statement like this a pointer descriptor is associated
- Add the particular pointer descriptor below `x`
- Add the marker at `y`
- Update the routing expression by adding the path from `x` to `y`
  - take shortest path from `x` to `y`
  - All possible paths will be added

Dynamic allocation and reallocation

- `x = malloc()`
  - transform a leaf node
  - and add corresponding nodes for selectors
- `x.s = malloc()`
  - use the leaf node below `x`
  - and add simple routing expression
- `free(x)`

Verification of programs with pointers using ARTMC

- Input structures
  - start with a tree automata memory encoding (for example DLLs)
  - start with empty shape graph and use a constructor written in C

```
aDLLHead = malloc();
aDLLHead->prev = null;
x = aDLLHead;
while (random()) {
  x->next = malloc();
  x->next->prev = x;
  x = x->next;
}
x->next = null;
```

- Applying ARTMC : Check for emptiness is not exact

Experimental results

Example	Time	Abs. method	$\|Q\|$	N_{ref}
SLL-creation + test	0.5s	predicates	22	0
SLL-reverse + test	6s	predicates	45	1
DLL-delete + test	8s	finite height	100	0
DLL-insert + test	11s	neighbour, predicates	94	0
DLL-reverse + test	13s	predicates	48	1
DLL-insertsort	3s	predicates	38	0
Inserting into trees + test	12s	predicates	91	0
Linking leaves in trees + test	11m15s	predicates	217	10
Inserting into list of lists + test	27s	predicates	125	1
Deutsch-Schorr-W. tree traversal	3m14s	predicates	168	0

SLL: Singly-linked list, DLL: Doubly-linked list

Conclusion and further work

- new, automatic method for verification of programs with complex dynamic data structures
- Optimising the prototype implementation
- Checking absence of garbage
- Show termination