Abstract Regular Tree Model Checking of Complex Dynamic Data Structures

Ahmed Bouajjani[1], Peter Habermehl[1], Adam Rogalewicz[2], and Tomáš Vojnar[2]

[1] LIAFA, University of Paris 7, Case 7014, 2 place Jussieu, F-75251 Paris 5, France.
e-mail: {Ahmed.Bouajjani,Peter.Habermehl}@liafa.jussieu.fr
[2] FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic.
e-mail: {rogalew,vojnar}@fit.vutbr.cz

Abstract. We consider the verification of non-recursive C programs manipulating dynamic linked data structures with possibly *several* next pointer selectors and with finite domain non-pointer data. We aim at checking basic memory consistency properties (no null pointer assignments, etc.) and shape invariants whose violation can be expressed in an existential fragment of a first order logic over graphs. We formalise this fragment as a logic for specifying bad memory patterns whose formulae may be translated to testers written in C that can be attached to the program, thus reducing the verification problem considered to checking reachability of an error control line. We encode configurations of programs, which are essentially shape graphs, in an original way as extended tree automata and we represent program statements by tree transducers. Then, we use the abstract regular tree model checking framework for a fully automated verification. The method has been implemented and successfully applied on several case studies.

1 Introduction

Automated verification of programs manipulating dynamic linked data structures is currently a very live research area. This is partly due to the fact that programs manipulating pointers are often complex and tricky, and so methods for automatically analysing them are quite welcome, and also because automated verification of such programs is not easy. Programs manipulating dynamic linked data structures are typically infinite-state systems, their configurations have in general the form of unrestricted graphs (often referred to as the *shape graphs*), and the shape invariants of these graphs may be temporarily broken by the programs during destructive pointer updates.

In this paper, we propose a new fully-automated method for analysing various important properties of programs manipulating dynamic linked data structures. We consider non-recursive C programs (with variables over finite data domains) manipulating dynamic linked data structures with possibly *several* next pointer selectors. The properties we consider are basic consistency of pointer manipulations (no *null pointer assignments*, no use of *undefined pointers*, no references to *deleted elements*). Further undesirable behaviour of the verified programs (e.g., breaking of certain *shape invariants* such as an introduction of undesirable sharing, cycles, etc.) may be detected via *testers written in C* and attached to the verified procedures. Moreover, for a more declarative way of specifying undesirable behaviour of the considered programs, we introduce a special-purpose logic LBMP (*logic of bad memory patterns*) and we show that its formulae may be automatically translated into C testers. Then, verification of these properties reduces to reachability of a designated error location.

Our verification method is based on using the approach of *abstract regular tree model checking* (ARTMC) [9]. In regular tree model checking, configurations of the systems being examined are encoded as trees over a suitable ranked alphabet, sets of configurations are described by tree automata, and transitions of the systems are encoded as tree transducers. Subsequently, one computes the set of all configurations reachable from an initial set of configurations by repeatedly applying the tree transducers on the set of the so-far reached configurations (encoded as tree automata). In order to make the method terminate as often as possible and to fight the state explosion problem arising due to increasing sizes of the automata to be handled, various kinds of automatically refinable abstractions over automata are used in ARTMC.

In order to be able to apply ARTMC for verification of programs manipulating dynamic linked data structures, whose configurations (shape graphs) need not be tree-like, we propose an *original encoding of shape graphs based on tree automata*. We use trees to encode the *tree skeleton* of a shape graph. The edges of the shape graph that are not directly encoded in the tree skeleton are then represented by *routing expressions* over the tree skeleton—i.e., regular expressions over directions in a tree (as, e.g., left up, right down, etc.) and the kind of nodes that can be visited on the way. Both the tree skeletons and the routing expressions are automatically discovered by our method. The idea of using routing expressions is inspired by PALE [28] and graph types [24] although there, they have a bit different form (see below) and are defined manually.

Next, we show how all *pointer-manipulating statements* of the C programming language (without pointer arithmetics, recursion, and with finite-domain non-pointer data) may be *automatically translated to tree transducers* over the proposed tree-automata-based representation of sets of shape graphs.

We implemented our method in a prototype tool based on the Mona tree libraries [23]. We have tested it on a number of non-trivial procedures manipulating singly-linked lists (SLL), doubly-linked lists (DLL), trees (including the Deutsch-Schorr-Waite tree traversal), lists of lists, and also trees with linked leaves. To the best of our knowledge, verifying some properties on trees with linked leaves have so-far not been considered in any other fully automated tool. The experimental results obtained from our tool are quite encouraging (and, moreover, we believe that there is still a lot of room for further improvements as we have, e.g., not used the mechanism of Mona's guided tree automata, we have used general-purpose, not specialised abstractions as in [11], etc.).

Related Work. There have been and there are currently being investigated various approaches to verification of programs manipulating dynamic linked data structures that differ in the degree of automation, generality, and/or principles used. Out of these techniques, we mention TVLA based on 3-valued predicate logic with transitive closure [29, 26], PALE based on WS*k*S and tree automata [28], approaches based on predicate abstraction [4, 27], memory patterns [32, 15], graph grammars [25], separation logic [18], alias logic [14], or various (extended) automata [20, 17, 7]. Among these approaches, our method belongs to the most automated and at the same time most general ones.

The closest approach to what we propose here is the one of PALE that also uses tree automata (derived from WS*k*S formulae) as well as the idea of a tree skeleton and routing expressions. However, first, the encoding of PALE is different in that the routing expressions must deterministically choose their target, and also, for a given memory node, selector, and program line, the expression is fixed and cannot dynamically change during the run of the analysed program. Further, program statements are modelled as transformers on the level of WSkS formulae, not as transducers on the level of tree automata. Finally, the approach of PALE is not fully automatic as the user has to manually provide loop invariants and all needed routing expressions, which are automatically synthesised in our approach.

In [8], we proposed a method based on abstract regular *word* model checking for verifying programs with 1-selector dynamic data structures. The concept of regular word model checking was studied in a series of works—including, for instance, [22, 12, 1, 6, 11, 21, 31]. Several different works [30, 13, 2, 3, 9] have appeared on the subject of regular tree model checking as well. Our approach of abstract regular (tree) model checking provides efficiency and is the only one that has been so-far applied in the area of verifying programs with dynamic data structures.

Top-down tree automata on infinite trees are used for verification of pointer manipulating programs in [17]. Here, linked data structures are represented with unfolded loops as infinite trees. Unlike our general approach, the work identifies and concentrates on a decidable fragment of pointer manipulating programs and their properties. The allowed programs may be compiled into an automaton on pairs of trees, composed with the given input tree automaton, the undesirable output tree automaton, and emptiness of the product is then checked.

The logic LBMP we use is close to the existential (positive) fragment of the logic of reachable patterns (LRP) in linked data-structures [33] but there the purpose is to have a decidable logic for reasoning about post- and pre-conditions and closure under negation is important. In our work we only need to express negation of invariance properties, and our verification approach is model checking.

2 The Class of Programs and Properties Considered

2.1 The Considered Programs

We consider standard, non-recursive C programs manipulating dynamic linked data structures (with possibly *several* next pointer selectors). We do not consider pointer arithmetics. We suppose all non-pointer data to be abstracted to a finite domain by some of the existing techniques before our method is applied. In the paper, we concentrate on the following pointer manipulating program statements: `x=NULL`, `x=y`, `x = y->next`, `x->next = y`, `x = malloc()`, `free(x)`, and `if (x==y) goto L1; else goto L2;` for pointer variables `x` and `y` and program line labels `L1` and `L2`. We suppose some further, commonly used statements (such as `while` loops or nested dereferences) to be encoded by the listed statements. For brevity, we do not explicitly discuss manipulation of non-pointer finite-domain data, which is anyway straightforward. An example of a typical program that our method can handle is the reversion of doubly-linked lists (DLL) shown in Fig. 1, which we also use as our running example.

```
// 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;
}
```

Fig. 1. Reversing a DLL

2.2 The Considered Properties

First of all, the properties we intend to check include *basic consistency of pointer manipulations*, i.e. absence of null and undefined pointer dereferences and references to already deleted nodes. Further, we would like to check various *shape invariance properties* (such as absence of sharing, acyclicity, or, e.g., the fact that if `x->next == y` (and `y` is not null) in a DLL, then also `y->prev == x`, etc.). To define such properties we propose two approaches described below.

Shape Testers. First, we use the so-called shape testers written in the C language. They can be seen as instrumentation code trying to detect violations of the memory shape properties at selected control locations of the original program. We extend slightly the C language used by the possibility of following next pointers backwards and by non-deterministic branching. For our verification tool, the testers are just a part of the code being verified. An error is announced when a line denoted by an error label is reached. This way, we can check a whole range of properties (including acyclicity, absence of sharing and other shape invariants as the relation of next and previous pointers in DLLs—cf. Fig 2).

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

Fig. 2. Checking the consistency of the next and previous pointers

A Logic of Bad Memory Patterns. Second, in order to allow the undesired violations of the memory shape properties to be specified more easily, we propose a logic-based specification language—namely, a *logic of bad memory patterns* (LBMP)—that is a fragment of the existential first order logic on graphs with (regular) reachability predicates (and an implicit existential quantification over paths). When defining the logic, our primary concern is not to obtain a decidable logic but rather to obtain a logic whose formulae may be automatically translated to the above mentioned C testers allowing us to *efficiently* test whether some bad shapes may arise from the given program by testing reachability of a designated error control line of a tester.

Let $\mathcal{V}$ be a finite set of program variables and $\mathcal{S}$ a finite set of selectors. The *formulae of LBMP* have the form $\Phi ::= \exists w_1,...w_n.\varphi$ where $\mathcal{W} = \{w_1,...,w_n\}$, $\mathcal{V} \cap \mathcal{W} = \emptyset$, is a set of formulae variables, $\varphi ::= \varphi \vee \varphi \mid \psi$, $\psi ::= \psi \wedge \psi \mid x\rho y$, $x,y \in \mathcal{V} \cup \mathcal{W}$, and ρ is a reachability formula defined below. To simplify the formulae, we allow y in $x\rho y$ to be skipped if it is not referred to anywhere else. We suppose such a missing variable to be implicitly added and existentially quantified. Given a ψ formula, we define its associated graph to be the graph $G_\psi = (\mathcal{V} \cup \mathcal{W}, E)$ where $(x,y) \in E$ iff $x\rho y$ is a conjunct in ψ. To avoid guessing in the tester corresponding to a formula, we require G_ψ of every top level ψ formula to have all nodes reachable from elements of $\mathcal{V}$.

An LBMP *reachability formula* has the form $\rho ::= \xrightarrow{s} \mid \xleftarrow{s} \mid \rho + \rho \mid \rho.\rho \mid \rho^* \mid [\sigma]$ where $s \in \mathcal{S}$ and σ is a local neighbourhood formula. Finally, an LBMP *local neighbourhood* formula has the form $\exists u_1,...,u_m.BC(x \xrightarrow{s} y, x = y)$ where $\mathcal{U} = \{u_1,...,u_m\}$ is a set of local formula variables, $\mathcal{U} \cap (\mathcal{V} \cup \mathcal{W} \cup \{p\}) = \emptyset$, $p \notin \mathcal{V} \cup \mathcal{W}$, $s \in \mathcal{S}$, $x \in \mathcal{V} \cup \mathcal{W} \cup \mathcal{U} \cup \{p\}$, $y \in \mathcal{V} \cup \mathcal{W} \cup \mathcal{U} \cup \{p, \perp, \top\}$, and BC is the Boolean closure. Here, $\perp$

represents `NULL`, $\top$ an undefined value, and p is a special variable that always represents the *current position* in a shape graph. Moreover, to avoid guessing in the evaluation of the local neighbourhood formulae, we require that if σ is transformed into σ' in DNF, and we construct a graph based on the positive $\xrightarrow{s}$ literals for each disjunct of σ', each node of such a graph is reachable from p.

The semantics of LBMP formulae is relatively straightforward. Therefore we defer its description to the full version of the paper [10]. Instead, we illustrate the semantics of LBMP formulae on several examples expressing undesirable phenomena that we would like to avoid when manipulating *acyclic doubly-linked lists*. In their case, it is undesirable if one of the following happens after some operation (as, for instance, reversion) on a given list—we suppose the resulting list to be pointed via the program variable l:

1. the list does not end with null, which can be tested via $l \xrightarrow{n}^{*} [p = \top]$,
2. the predecessor of the first element is not null, which corresponds to $l[\neg(p \xrightarrow{b} \bot)]$,
3. the predecessor of the successor of a node n is not n, which can be detected via the formula $l \xrightarrow{n}^{*} [\exists x.\ p \xrightarrow{n} x \wedge x \neq \bot \wedge \neg(x \xrightarrow{p} p)]$, or
4. the list is cyclic, i.e. $\exists x.\ l \xrightarrow{n}^{*} [p = x] \xrightarrow{n} \xrightarrow{n}^{*} [p = x]$. (Note that this property is in fact implied by items 2 and 3.)

All the given formulae can be joint by disjunction into a single LBMP formula. Due to the space limitations, we do not provide more examples of LBMP formulae, but we note that for all the structures mentioned later in Section 5, we are able to specify all the commonly considered undesirable situations in LBMP (some more examples of LBMP formulae can then be found in the full version of the paper [10]).

Due to a lack of space, the procedure for translating LBMP formulae is described in the full version of the paper [10]. Intuitively, it is quite easy to see that the existentially quantified LBMP formulae with a stress on exploring paths through the examined linked data structures starting from program variables can be encoded in a slightly extended C code, put after the program being verified, and used in an efficient way for checking safety of the given program. We translate disjunctions to non-deterministic branching, conjunctions and series of reachability formulae to series of tests, iteration in the reachability expressions to non-deterministic while loops. The needed extension of C includes non-deterministic branching and the possibility of following next pointers backwards. Both of these features may easily be handled in our verification framework.

2.3 The Verification Problem

Our verification problem is model checking of the described undesirable existential properties against the given program. Above, we explain that for the specification of a violation of shape invariants, we use shape testers or LBMP whose formulae are translated into shape testers. For shape testers, we need to check unreachability of their designated error location. Moreover, we model all program statements such that if some basic memory consistency error (like a null pointer assignment) happens, the control is automatically transferred to a unique error control location. Thus, we are in general interested in *checking unreachability of certain error control locations* in a program.

3 Automata-based Verification Framework

In this section, we introduce the abstract tree regular model-checking framework based on tree automata and transducers that we use for solving our verification problem.

3.1 Tree Automata and Transducers

Terms and Trees. An *alphabet* Σ is a finite set of symbols. Σ is called *ranked* if there exists a *rank* function $\rho : \Sigma \to \mathbb{N}$. For each $k \in \mathbb{N}$, $\Sigma_k \subseteq \Sigma$ is the set of all symbols with rank k. Symbols of Σ_0 are called *constants*. Let χ be a denumerable set of symbols called *variables*. $T_\Sigma[\chi]$ denotes the set of *terms* over Σ and χ. The set $T_\Sigma[\emptyset]$ is denoted by T_Σ, and its elements are called *ground terms*. A term t from $T_\Sigma[\chi]$ is called *linear* if each variable occurs at most once in t.

A finite ordered *tree* t over a set of labels L is a mapping $t : \mathcal{P}os(t) \to L$ where $\mathcal{P}os(t) \subseteq \mathbb{N}^*$ is a finite, prefix-closed set of *positions* in the tree. A term $t \in T_\Sigma[\chi]$ can naturally also be viewed as a tree whose leaves are labelled by constants and variables, and each node with k sons is labelled by a symbol from Σ_k [16]. Therefore, below, we sometimes exchange terms and trees. We denote $\mathcal{N}lPos(t) = \{p \in \mathcal{P}os(t) \mid \neg\exists i \in \mathbb{N} : pi \in \mathcal{P}os(t)\}$ the set of *non-leaf* positions.

Tree Automata. A *bottom-up tree automaton* over a ranked alphabet Σ is a tuple $A = (Q, \Sigma, F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, and δ is a set of transitions of the following types: (i) $f(q_1, \ldots, q_n) \to_\delta q$, (ii) $a \to_\delta q$, and (iii) $q \to_\delta q'$ where $a \in \Sigma_0$, $f \in \Sigma_n$, and $q, q', q_1, \ldots, q_n \in Q$. Below, we call a bottom-up tree automaton simply a tree automaton.

Let t be a ground term. A run of a tree automaton A on t is defined as follows. First, leaves are labelled with states. If a leaf is a symbol $a \in \Sigma_0$ and there is a rule $a \to_\delta q \in \delta$, the leaf is labelled by q. An internal node $f \in \Sigma_k$ is labelled by q if there exists a rule $f(q_1, q_2, \ldots, q_k) \to_\delta q \in \delta$ and the first son of the node has the state label q_1, the second one q_2, ..., and the last one q_k. Rules of the type $q \to_\delta q'$ are called ε-*steps* and allow us to change a state label from q to q'. If the top symbol is labelled with a state from the set of final states F, the term t is accepted by the automaton A.

A set of ground terms accepted by a tree automaton A is called a *regular tree language* and is denoted by $L(A)$. Let $A = (Q, \Sigma, F, \delta)$ be a tree automaton and $q \in Q$ a state, then we define the *language of the state* q—$L(A, q)$—as the set of ground terms accepted by the tree automaton $A_q = (Q, \Sigma, \{q\}, \delta)$. The language $L^{\leq n}(A, q)$ is defined to be the set $\{t \in L(A, q) \mid height(t) \leq n\}$.

Tree Transducers. A *bottom-up tree transducer* is a tuple $\tau = (Q, \Sigma, \Sigma', F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ a set of final states, Σ an input ranked alphabet, Σ' an output ranked alphabet, and δ a set of transition rules of the following types: (i) $f(q_1(x_1), \ldots, q_n(x_n)) \to_\delta q(u), u \in T_{\Sigma'}[\{x_1, \ldots, x_n\}]$, (ii) $q(x) \to_\delta q'(u), u \in T_{\Sigma'}[\{x\}]$, and (iii) $a \to_\delta q(u), u \in T_{\Sigma'}$ where $a \in \Sigma_0$, $f \in \Sigma_n$, $x, x_1, \ldots, x_n \in \chi$, and $q, q', q_1, \ldots, q_n \in Q$. In the following, we call a bottom-up tree transducer simply a tree transducer. We always use tree transducers with $\Sigma = \Sigma'$.

A run of a tree transducer τ on a ground term t is similar to a run of a tree automaton on this term. First, rules of type *(iii)* are used. If a leaf is labelled by a symbol a and there is a rule $a \rightarrow_\delta q(u) \in \delta$, the leaf is replaced by the term u and labelled by the state q. If a node is labelled by a symbol f, there is a rule $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \rightarrow_\delta q(u) \in \delta$, the first subtree of the node has the state label q_1, the second one q_2, ..., and the last one q_n, then the symbol f and all subtrees of the given node are replaced according to the right-hand side of the rule with the variables $x_1, \ldots, x_n$ substituted by the corresponding left-hand-side subtrees. The state label q is assigned to the new tree. Rules of type *(ii)* are called ε-*steps*. They allow us to replace a q-state-labelled tree by the right hand side of the rule and assign the state label q' to this new tree with the variable x in the rule substituted by the original tree. A run of a transducer is successful if the root of a tree is processed and is labelled by a state from F.

A tree transducer is *linear* if all right-hand sides of its rules are linear (no variable occurs more than once). The class of linear bottom-up tree transducers is closed under composition. A tree transducer is called *structure-preserving* (or a *relabelling*) if it does not modify the structure of input trees and just changes the labels of their nodes. By abuse of notation, we identify a transducer τ with the relation $\{(t,t') \in T_\Sigma \times T_\Sigma \mid t \rightarrow_\delta^* q(t') \text{ for some } q \in F\}$. For a set $L \subseteq T_\Sigma$ and a relation $R \subseteq T_\Sigma \times T_\Sigma$, we denote $R(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w', w) \in R\}$ and $R^{-1}(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w, w') \in R\}$. If τ is a linear tree transducer and L is a regular tree language, then the sets $\tau(L)$ and $\tau^{-1}(L)$ are regular and effectively constructible [19, 16].

Let $id \subseteq T_\Sigma \times T_\Sigma$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations $\tau^0 = id$, $\tau^{i+1} = \tau \circ \tau^i$ and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Below, we suppose $id \subseteq \tau$ meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

3.2 Abstract Regular Tree Model Checking

Let us recall the basic principles of abstract regular tree model checking (ARTMC) [9]. Let Σ be a ranked alphabet and $\mathbb{M}_\Sigma$ the set of all tree automata over Σ. Let $Init \in \mathbb{M}_\Sigma$ be a tree automaton describing a set of initial configurations, τ a tree transducer describing the behaviour of a system, and $Bad \in \mathbb{M}_\Sigma$ a tree automaton describing a set of bad configurations. The verification problem is to check whether

$$\tau^*(L(Init)) \cap L(Bad) = \emptyset \tag{1}$$

One of the methods how to check this is ARTMC [9]. Instead of computing the precise set of reachable configurations, it computes an overapproximation.

We define an abstraction function as a mapping $\alpha : \mathbb{M}_\Sigma \rightarrow \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. Given a tree transducer τ and an abstraction α, we define a mapping $\tau_\alpha : \mathbb{M}_\Sigma \rightarrow \mathbb{M}_\Sigma$ as $\forall M \in \mathbb{M}_\Sigma : \tau_\alpha(M) = \hat{\tau}(\alpha(M))$ where $\hat{\tau}(M)$ is the minimal deterministic automaton describing the language $\tau(L(M))$. An abstraction α is *finitary*, if the set $\mathbb{A}_\Sigma$ is finite.

For a given abstraction function α, we can compute iteratively the sequence of automata $(\tau_\alpha^i(Init))_{i\geq 0}$. If the abstraction α is finitary, then there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(Init) = \tau_\alpha^k(Init)$. The definition of the abstraction function α implies, that $L(\tau_\alpha^k(Init)) \supseteq \tau^*(L(Init))$.

If $L(\tau_\alpha^k(Init)) \cap L(Bad) = \emptyset$, then the verification problem (1) has a positive answer. If the intersection is non-empty, we must check whether it is a real counterexample, or a spurious one. The spurious counterexample may be caused by the used abstraction (the counterexample is not reachable from the set of initial configurations). Assume that $\tau_\alpha^k(Init) \cap L(Bad) \neq \emptyset$, which means that there is a symbolic path:

$$Init,\ \tau_\alpha(Init),\ \tau_\alpha^2(Init), \cdots \tau_\alpha^{n-1}(Init),\ \tau_\alpha^n(Init) \tag{2}$$

such that $L(\tau_\alpha^n(Init)) \cap L(Bad) \neq \emptyset$.

Let $X_n = L(\tau_\alpha^n(Init)) \cap L(Bad)$. Now, for each l, $0 \leq l < n$, we compute $X_l = L(\tau_\alpha^l(Init)) \cap \tau^{-1}(X_{l+1})$. Two possibilities may occur: (a) $X_0 \neq \emptyset$, which means that the verification problem (1) has a negative answer, and $X_0 \subseteq L(Init)$ is a set of dangerous initial configurations. (b) $\exists m, 0 \leq m < n, X_{m+1} \neq \emptyset \wedge X_m = \emptyset$ meaning that the abstraction function is too rough—we need to refine it and start the verification process again.

In [9], two general-purpose kinds of abstractions are proposed. Both are based on *automata state equivalences*. Tree automata states are split into several equivalence classes, and all states from one class are collapsed into one state. An abstraction becomes finitary if the number of equivalence classes is finite. The refinement is done by refining the equivalence classes. Both of the proposed abstractions allow for an automatic refinement to exclude the encountered spurious counterexample.

The first proposed abstraction is an *abstraction based on languages of trees of a finite height*. It defines two states equivalent if their languages up to the give height n are equivalent. There is just a finite number of languages of height n, therefore this abstraction is finitary. A refinement is done by an increase of the height n. The second proposed abstraction is an *abstraction based on predicate languages*. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $M = (Q, \Sigma, F, \delta)$ be a tree automaton. Then, two states $q_1, q_2 \in Q$ are equivalent if their languages $L(M, q_1)$ and $L(M, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$. Since there is just a finite number of subsets of $\mathcal{P}$, the abstraction is finitary. A refinement is done by adding new predicates, i.e. tree automata corresponding to the languages of all the states in the automaton of X_{m+1} from the analysis of spurious counterexample ($X_m = \emptyset$).

4 Tree Automata Encoding of Pointer Manipulating Programs

4.1 Encoding of Sets of Memory Configurations

Memory configurations of the considered programs with a finite set of pointer variables $\mathcal{V}$, a finite set of selectors $\mathcal{S} = \{1, ..., k\}$, and a finite domain $\mathcal{D}$ of data stored in dynamically allocated memory cells can be described as shape graphs of the following form. A *shape graph* is a tuple $SG = (N, S, V, D)$ where N is a finite set of memory nodes, $N \cap \{\perp, \top\} = \emptyset$ (we use $\perp$ to represent null, and $\top$ to represent an undefined pointer value), $N_{\perp,\top} = N \cup \{\perp, \top\}$, $S : N \times \mathcal{S} \rightarrow N_{\perp,\top}$ 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 data are stored in the particular memory nodes. We suppose

$\top \in \mathcal{D}$—the data value $\top$ is used to denote "zombies" of deleted nodes, which we keep and detect all erroneous attempts to access them.

To be able to describe the way we encode sets of shape graphs using tree automata, we first need a few auxiliary notions. First, to allow for dealing with more general shape graphs than tree-like, we do not simply identify the next pointers with the branches of the trees accepted by tree automata. Instead, we use the tree structure just as a backbone over which links between the allocated nodes are expressed using the so-called *routing expressions*, which are regular expressions over directions in a tree (like move up, move left down, etc.) and over the nodes that can be seen on the way. From nodes of the trees described by tree automata, we refer to the routing expressions via some symbolic names called *pointer descriptors* that we assign to them—we suppose dealing with a finite set of pointer descriptors $\mathcal{R}$. Moreover, we couple each pointer descriptor with a unique *marker* from a set $\mathcal{M}$ (and so $||\mathcal{R}|| = ||\mathcal{M}||$). The routing expressions may identify several target nodes for a single source memory node and a single selector. Markers associated with the target nodes can then be used to decrease the non-determinism of the description (only nodes marked with the right marker are considered as the target).

Let us now fix the sets $\mathcal{V}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{M}$. We use a *ranked alphabet* $\Sigma = \Sigma_2 \cup \Sigma_1 \cup \Sigma_0$ consisting of symbols of ranks $k = ||\mathcal{S}||$, 1, and 0. Symbols of rank k represent allocated memory nodes that may be pointed by some pointer variables, may be marked by some markers as targets of some next pointers, they contain some data and have k next pointers specified either as null, undefined, or via some next pointer descriptor. Thus, $\Sigma_2 = 2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D} \times (\mathcal{R} \cup \{\perp, \top\})^k$. Given an element $n \in \Sigma_2$, we use the notation $n.var$, $n.mark$, $n.data$, and $n.s$ (for $s \in \mathcal{S}$) to refer to the pointer variables, markers, data, and descriptors associated with n, respectively. Σ_1 is used for specifying nodes with undefined and null pointer variables, and so $\Sigma_1 = 2^{\mathcal{V}}$. Finally, in our trees, the leaves are all the same (with no special meaning), and so $\Sigma_0 = \{\bullet\}$.

We can now specify the *tree memory backbones* we use to encode memory configurations as the trees that belong to the language of the tree automaton with the following rules[3]: (1) $\bullet \to q_i$, (2) $\Sigma_2(q_i/q_m, \ldots, q_i/q_m) \to q_m$, (3) $\Sigma_1(q_m/q_i) \to q_n$, and (4) $\Sigma_1(q_n) \to q_u$. Intuitively, q_i, q_m, q_n, and q_u are automata states, where q_i accepts the leaves, q_m accepts the memory nodes, q_n accepts the node encoding null variables, and q_u, which is the accepting state, accepts the node with undefined variables. Note that there is always a single node with undefined variables, a single node with null variables, and then a sub-tree with the memory allocated nodes. Thus, every memory tree t can be written as $t = undef(null(t'))$ for $undef, null \in \Sigma_1$. We say a memory tree $t = undef(null(t'))$ is *well-formed* if the pointer variables are assigned unique meanings, i.e. $undef \cap null = \emptyset \wedge \forall p \in \mathcal{N}lPos(t') : t'(p).var \cap (null \cup undef) = \emptyset \wedge \forall p_1 \neq p_2 \in \mathcal{N}lPos(t') : t'(p_1).var \cap t'(p_2).var = \emptyset$ where $\mathcal{N}lPos$ are non-leaf positions—cf. Section 3.1.

We let $\mathcal{S}^{-1} = \{s^{-1} \mid s \in \mathcal{S}\}$ be a set of "inverted selectors" allowing one to follow the links in a shape graph in a reverse order. A *routing expression* may then be formally defined as a regular expression on pairs $s.p \in (\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma_2$. Intuitively, each pair used as

[3] If we put a set into the place of the input symbol in a transition rule, we mean we can use any element of the set. Moreover, if we use q_1/q_2 instead of a single state, one can take either q_1 or q_2, and if there is a k-tuple of states, one considers all possible combinations of states.

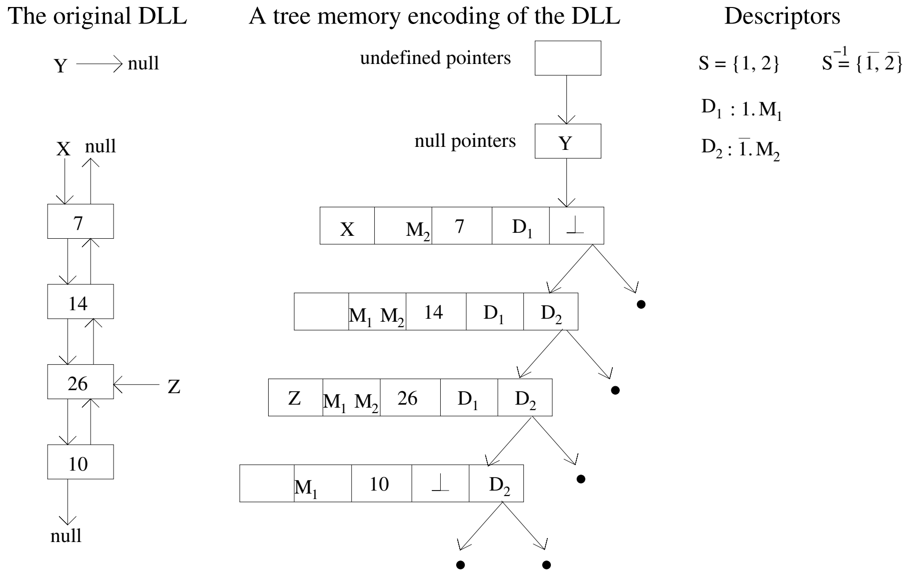

Fig. 3. An example of a tree memory encoding—a doubly linked list (DLL)

a basic building block of a routing expression describes one step over the tree memory backbone: we follow a certain branch up or down and then we should see a certain node (most often, we will use the node components of routing expressions to check whether a certain marker is set in a particular node).

A *tree memory encoding* is a tuple (t,μ) where t is a tree memory backbone and μ a mapping from the set of pointer descriptors $\mathcal{R}$ to routing expressions over the set of selectors $\mathcal{S}$ and the memory node alphabet Σ_2 of t. An example of a tree memory encoding for a *doubly-linked list* (DLL) is shown in Fig. 3.

Let (t,μ), $t = undef(null(t'))$, be a tree memory encoding with a set of selectors $\mathcal{S}$ and a memory node alphabet Σ_2. We call $\pi = p_1 s_1 ... p_l s_l p_{l+1} \in \Sigma_2.((\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma_2)^l$ a *path in t* of length $l \geq 1$ iff $p_1 \in \mathcal{P}os(t')$ and $\forall i \in \{1,...,l\} : (s_i \in \mathcal{S} \wedge p_i.s_i = p_{i+1} \wedge p_{i+1} \in \mathcal{P}os(t')) \vee (s_i \in \mathcal{S}^{-1} \wedge p_{i+1}.s_i = p_i)$. For $p, p' \in \mathcal{N}lPos(t')$ and a selector $s \in \mathcal{S}$, we write $p \xrightarrow{s} p'$ iff (1) $t'(p).s \in \mathcal{R}$, (2) there is a path $p_1 s_1 ... p_l s_l p_{l+1}$ in t for some $l \geq 0$ such that $p = p_1$, $p_{l+1} = p'$, and (3) $s_1 t'(p_2) ... t'(p_l) s_l t'(p_{l+1}) \in \mu(t'(p).s)$.

The *set of shape graphs represented by a tree memory encoding* (t,μ) with $t = undef(null(t'))$ is denoted by $[\![(t,\mu)]\!]$ and given as all the shape graphs $SG = (N,S,V,D)$ for which there is a bijection $\beta : \mathcal{P}os(t') \to N$ such that:

1. $\forall p, p' \in \mathcal{N}lPos(t')\ \forall s \in \mathcal{S} : (t'(p).s \notin \{\bot,\top\} \wedge p \xrightarrow{s} p') \Leftrightarrow S(\beta(p),s) = \beta(p')$.
   (The links between memory nodes are respected.)
2. $\forall p \in \mathcal{N}lPos(t')\ \forall s \in \mathcal{S}\ \forall x \in \{\bot,\top\} : t'(p).s = x \Leftrightarrow S(\beta(p),s) = x$.
   (Null and undefined successors are respected.)
3. $\forall v \in \mathcal{V}\ \forall p \in \mathcal{P}os(t') : v \in t'(p).var \Leftrightarrow V(v) = \beta(p)$.
   (Assignment of memory nodes to variables is respected.)

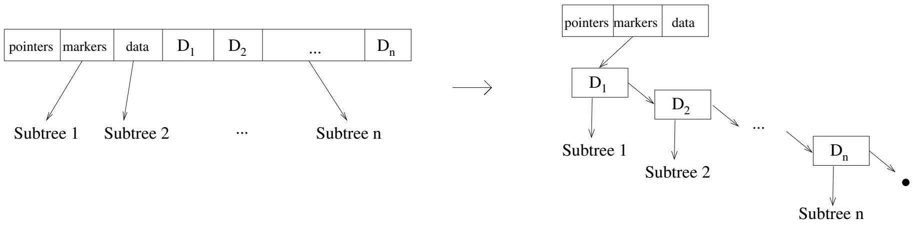

Fig. 4. Splitting memory nodes in Mona into data and next pointer nodes

4. $\forall v \in \mathcal{V} : (v \in null \Leftrightarrow V(v) = \bot) \wedge (v \in undef \Leftrightarrow V(v) = \top)$.
   (Assignment of null and undefinedness of variables are respected.)
5. $\forall p \in \mathcal{N}lPos(t')\ \forall d \in \mathcal{D} : t'(p).data = d \Leftrightarrow D(\beta(p)) = d$.
   (Data stored in memory nodes is respected.)

A *tree automata memory encoding* is a tuple (A,μ) where A is a tree automaton accepting a regular set of tree memory backbones and μ is a mapping as above. Naturally, A represents the set of shape graphs defined by $[\![(A,\mu)]\!] = \bigcup_{t \in L(A)} [\![(t,\mu)]\!]$.

Remarks. We use ARTMC as our verification method. It syntactically manipulates tree automata A whose languages can be interpreted as shape graphs using our encoding. Notice, that (A,μ) and $[\![(A,\mu)]\!]$ are two different notions, since the encoding is not canonical as a given shape graph can be possibly obtained by several different tree memory encodings. In Section 4.3, we argue that program statements can, nevertheless, be encoded faithfully as tree transducers. Another important property of the encoding is that given a tree automata memory encoding (A,μ), the set $[\![(A,\mu)]\!]$ can be empty although $L(A)$ is not empty (since the routing expressions can be incompatible with the tree automaton). Of course, if $L(A)$ is empty, then $[\![(A,\mu)]\!]$ is also empty. Therefore, checking emptiness of $[\![(A,\mu)]\!]$ (which is important for applying the ARTMC framework, see Section 4.4) can be done in a sound way by checking emptiness of $L(A)$.

4.2 Tree Memory Configurations in Mona

In our implementation, we use the tree automata library from the Mona project [23]. As the library supports binary trees only, and we need n-ary ones, we *split each memory node* labelled with $\Sigma_2 = 2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D} \times (\mathcal{R} \cup \{\bot, \top\})^k$ in the above definition of a tree memory encoding into a data node labelled with $2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D}$ and a series of k next pointer nodes, each labelled with $\mathcal{R} \cup \{\bot, \top\}$—cf. Fig. 4.

As for the set of *pointer descriptors* $\mathcal{R}$, we currently fix it by introducing a unique pointer descriptor for each destructive update `x->`s `= y` or `x->`s `= new` that appears in the program. This is because they are the statements that establish new links among the allocated memory nodes. In addition, we might have some further descriptors if they are a part of the specification of the input configurations (see section 4.4).

Further, in our Mona-based framework, we encode *routing expressions* using tree transducers. A transducer representing a routing expression r simply copies the input tree memory backbone on which it is applied up to: (1) looking for a data node n_1 that is labelled with a special token $\blacklozenge \notin \mathcal{V} \cup \mathcal{M} \cup \mathcal{D}$ and (2) moving $\blacklozenge$ to a data node n_2 that is the target of the next pointer described by r and that is also marked with the appropriate marker. As described in the next section, we can then implement program statements that follow the next pointers (e.g., `x = y->`s) by putting the token $\blacklozenge$ to a node pointed to by `x`, applying the transducer implementing the appropriate routing expression, and making `y` point to the node to which $\blacklozenge$ was moved. Due to applying abstraction, the target may not always be unique—in such a case, the transducer implementing the routing expression simply returns a set of trees in which $\blacklozenge$ is put to some target data node such that all possibilities where it can get via the given routing expression are covered.

Note that the use of tree transducers for encoding routing expressions allows us in theory to express more than using just regular expressions. In particular, we can refer to the tree context of the nodes via which the given route is going. In our current implementation, we, however, do not use this fact.

4.3 Encoding Program Statements as Tree Transducers

We encode every of the considered pointer-manipulating statements as a tree transducer. In the transducer, we expect the tree memory encoding to be extended by a new root symbol which corresponds to the *current program line* or to an error indication when an error is found during the analysis. We now briefly describe how the transducers corresponding to the program statements work. Each transducer is constructed in such a way, that it simulates the effect of a program statement on a set of shape graphs represented by a tree automata memory encoding: if a shape SG represented by a tree memory encoding is transformed by the program statement to a shape graph SG', then the transducer transforms the tree memory encoding such that it represents SG'. This makes sure, that although the encoding is non canonical (see end of section 4.1), we simulate faithfully a program statement.

Non-destructive Updates and Tests. The simplest is the case of the `x = NULL` assignment. The transducer implementing it just goes through the input tree and copies it to the output with the exception that (1) it removes `x` from the labelling of the node in which it currently is and adds it to the labelling of the *null* node and (2) changes the current line appropriately. The transducer implementing an assignment `x = y` is similar, it just puts `x` not to the *null* node, but to the node which is currently labelled by `y`.

The transducers for the tests of the form `if (x == null) goto l1; else goto l2;` are very similar to the above—they just do not change the node in which `x` is, but only change the current program line to either `l1` or `l2` according to whether or not `x` is in the *null* node. If `x` is in *undef*, an error indication is used instead of `l1` or `l2`. The transducers for `if (x == y) goto l1; else goto l2;` are similar—they just test whether or not `x` and `y` appear in the same node (both different from *undef*).

The transducer for an `x = y->`s statement is a union of several complementary actions. If `y` is in *null* or *undef*, an error is indicated. If `y` is in a regular data node and its s-th next pointer node contains either $\perp$ or $\top$, the transducer removes `x` from the node it is currently in and puts it into the *null* or *undef* node, respectively. If `y` is in a regular data node n and its s-th next pointer node contains some pointer descriptor $r \in \mathcal{R}$, the ♦ token is put to n. Then, the routing expression transducer associated with r is applied. Finally, `x` is removed from its current node and put into the node to which ♦ was moved by the applied routing expression transducer.

Destructive Updates. The destructive pointer update `x->`s `= y` is implemented as follows. If `x` is in *null* or *undef*, an error is indicated. If `x` is defined and if `y` is in *null* or *undef*, the transducer puts $\perp$ or $\top$ into the s-th next pointer node below `x`, respectively. Otherwise, the transducer puts the pointer descriptor r associated with the particular `x->`s `= y` statement being fired into the s-th next pointer node below `x`, and it marks the node in which `y` is by the marker coupled with r. Then, the routing expression transducer associated with r is updated such that it includes the path from the node of `x` to the node of `y`.

One could think of various strategies how to *extract the path* going from the node of `x` to the node of node `y`. Currently, we use a simple strategy, which is, however, successful in many practical examples as our experiments show: We extract the shortest path between `x` and `y` on the tree memory backbone, which consists of going a certain number of steps upwards to the closest common parent of `x` and `y` and then going a certain number of steps downwards. (The upward or the downward phase may also be skipped when going just down or up, respectively.) When extracting this path, we project away all information about nodes we see on the way and about nodes not directly lying on the path. Only the directions (left/right up/down) and the number of steps are preserved.

Note that we, in fact, perform the operation of routing expression extraction on a tree automaton, and we extract all possible paths between where `x` and `y` may currently be. The result is transformed into a transducer τ_{xy} that moves the token ♦ from the position of `x` to the position of `y`, and τ_{xy} is then united with the current routing expression transducer associated with the given pointer descriptor r. The extraction of the routing paths is done partly by rewriting the input tree automaton via a special transducer τ_π that in one step identifies all the shortest paths between all `x` and `y` positions and projects away the non-necessary information about the nodes on the way. The transducer τ_π is simple: it just checks that we are going one branch up from `x` and one branch down to `y` while meeting in a single node. The transition relation of the resulting transducer is then post-processed by changing the context of the path to an arbitrary one which cannot be done by transducing in Mona where structure preserving transducers may only be used.

Dynamic Allocation and Deallocation. The `x = malloc()` statement is implemented by rewriting the right-most • leaf node to a new data node pointed to by `x`. Below the node, the procedure also creates all the k next pointer nodes whose contents is set to $\top$.

In order to exploit the regularity that is always present in algorithms allocating new data structures, which typically add new elements at the end/leaves of the structure, we also explicitly support an `x.`s `= malloc()` statement. We even try to pre-process programs and compact all successive pairs of statements of the form `x = malloc();`

`y->`*s* `= x` (provided `x` is not used any further) to `y->`*s*`= malloc()`. Such a statement is then implemented by adding the new element directly under the node pointed to by *y* (provided it is a leaf) and joining it by a simple routing expression of the form "one level down via a certain branch". This typically allows us to work with much simpler and more precise routing expressions.

Finally, a `free(x)` statement is implemented by a transducer that moves all variables that are currently in the node pointed to by `x` to the *undef* node (if `x` is in *null* or *undef*, an error is indicated). Then, the node is marked by a special marker as a deleted node, but it stays in our tree memory encoding with all its current markers set. In addition to all the other tests mentioned above as done within the transducer implementing an `x = y->`*s* assignment, it is also tested whether the target is not deleted—if so, an error is indicated.

4.4 Verification of Programs with Pointers using ARTMC

Input structures. We consider two possibilities how to encode the input structures. First, we can directly use the tree automata memory encoding—e.g., a tree automata memory encoding (with two pointer descriptors *next* and *prev* and the corresponding routing expressions) describing all possible doubly-linked lists pointed to by some program variable. Such an encoding can be provided manually or derived automatically from a description of the concerned linked data structure provided, e.g., as a graph type [24]. The main advantage is that the verification process starts with an exact encoding of the set of all possible instances of the considered data structure.

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

Fig. 5. Generating DLLs

Another possible approach is to start with the unique "empty" shape graph where all variables are undefined. We can encode such a shape graph using a tree automata encoding where all variables are in *undef*, *null* is empty, there are no other nodes, and all the routing expressions are empty. The set of structures on which the examined procedure should be verified is then supposed to be generated by a *constructor written in C* by the user (as, e.g., in Fig. 5). This constructor is then put before the verified procedure and the whole program is given to the model checker. The advantage is that no further notation is necessary. The disadvantage is that we have more code that is subject to the verification and the set of automatically obtained input structures need not be encoded in the optimal way leading to a slow-down of the verification.

Applying ARTMC. In Section 3.2, we have given an overview of ARTMC. We supposed that one transducer τ is used to describe the behaviour of the whole system. In the application described in this paper, we use a variant of this approach by considering each program statement as one transducer. Then, we compute an overapproximation of the reachable configurations for each program line by starting from an initial set of shape graphs represented by a tree automata memory encoding and iterating the abstract fixpoint computation described in Section 3.2 through the program structure. The fixpoint computation stops if the abstraction α is finitary. In such a case, the number of the abstracted tree automata encoding sets of the memory backbones that can arise in the program being checked is finite. Moreover, the number of the arising routing expressions is also finite as they are extracted from the bounded number of the tree automata describing the encountered sets of memory backbones.[4]

During the computation, we check whether a designated error location in the program is reached or whether a fixpoint is attained. In the latter case, the property is satisfied (the error control location is not reachable). In the former case, we compute backwards to check if the counterexample is spurious as explained in Section 3.2. However, as said in Section 4.1, the check for emptiness is not exact and therefore we might conclude that we have obtained a real counterexample although this is not the case. Such a case does not happen in any of our experiments and could be detected by replaying the path from the initial configurations.

5 Implementation and Experimental Results

An ARTMC Tool for Tree Automata Memory Encodings. We have implemented the above proposed method in a prototype tool based on the Mona tree automata libraries [23]. We use a depth-first strategy when iterating the transducers corresponding to the particular program lines.

We have also refined the basic finite-height and predicate abstractions proposed in [9]. In particular, we do not allow collapsing of data nodes with next pointer nodes, collapsing of next pointer nodes corresponding to different selectors, and we prevent the abstraction of allowing a certain pointer variable to point to several memory nodes at the same time.

We have also proposed one new abstraction schema called the *neighbour abstraction*. Under this schema, only the tree automata states are collapsed that (1) accept equal data memory nodes with equal next pointer nodes associated with them and (2) that directly follow each other (are neighbours). This strategy is very simple, yet it proved useful in some practical cases.

Finally, we allow the abstraction to be applied either at all program lines or only at the loop closing points. In some cases, the latter approach is more advantageous due to some critical destructive pointer updates are done without being interleaved with abstraction. This way, we may avoid having to remove lots of spurious counterexamples that may otherwise arise when the abstraction is applied while some important shape invariant is temporarily broken.

Experimental Results. We have performed several experiments with singly-linked lists (SLL), doubly-linked lists (DLL), trees, lists of lists, and trees with linked leaves. All three mentioned types of automata abstraction—the finite height abstraction (with the initial height being one), predicate abstraction (with no initial predicates), and neighbour abstraction—proved useful (gave the best achieved result) in different examples. All examples were automatically verified for null/undefined/deleted pointer

[4] The non-canonicity of our encoding does not prevent the computation from stopping. It may just take longer since several encodings for the same graph could be added.

exceptions. Additionally, some further shape properties (such as absence of sharing, acyclicity, preservation of input elements, etc.) were verified in some case studies too. All these properties were specified in the LBMP logic from Sect. 2.2 and translated to C testers. We give a detailed overview of the performed experiments in the full version of the paper [10].

Table 1 contains verification times for the experiments mentioned above (the "+ test" in the name of an experiment means that some shape invariants were checked). We give the best result obtained using the three mentioned abstraction schemas and say for which abstraction schema the result was obtained. The note "restricted" accompanying the abstraction method means that the abstraction was applied at the loop points only. The experiments were performed on a 64bit Xeon 3,2 GHz with 3 GB of memory. The column $|Q|$ gives information about the size of the biggest encountered automaton, and N_{ref} gives the number of refinements.

Despite the prototype nature of the tool, which can still be optimised in multiple ways (some of them are mentioned in the conclusions), the results are quite competitive. For example, for one of the most complex examples—the Deutsch-Schorr-Waite tree traversal, TVLA took 3 minutes on the same machine with manually provided instrumentation predicates and predicate transformers. The verification time for the trees with linked leaves is relatively high, but we are not aware of any other fully automated tool with which experiments with this structure have been performed.

Table 1. Results of experimenting with the prototype implementation of the presented method

Example	Time	Abstraction method	$\|Q\|$	N_{ref}
SLL-creation + test	0.5s	predicates, restricted	22	0
SLL-reverse + test	6s	predicates	45	1
DLL-delete + test	8s	finite height	100	0
DLL-insert + test	11s	neighbour, restricted	94	0
DLL-reverse + test	13s	predicates	48	1
DLL-insertsort	3s	predicates	38	0
Inserting into trees + test	12s	predicates, restricted	91	0
Linking leaves in trees + test	11min 15s	predicates	217	10
Inserting into a list of lists + test	27s	predicates, restricted	125	1
Deutsch-Schorr-Waite tree traversal	3min 14s	predicates	168	0

6 Conclusion

We have proposed a new, fully automated method for verification of programs manipulating complex dynamic linked data structures. The method is based on the framework of ARTMC. In order to able to use ARTMC, we proposed a new representation of sets of shape graphs based on tree automata and a representation of the standard C pointer manipulating statements as tree transducers (with some extensions). In particular, we considered verification of the basic memory consistency properties (no null pointer assignments, etc.) and of shape invariants whose corruption may be described in an existential fragment of a first-order logic on graphs. We formalised this fragment as a special-purpose logic called LBMP whose formulae may be translated to C-based testers that may be attached to the verified programs, thus transforming the verification problem to be considered to the control line reachability. We have implemented the technique in a prototype tool and obtained some promising experimental results.

In the future, we would like to optimise the performance of our Mona-based prototype tool, e.g., by exploiting the concept of *guided tree automata* that are suggested as very helpful in many situations by the authors of Mona [5] and that we have not used yet. Further, it is interesting to try come up with some *special purpose automata abstractions* for the considered domain—so-far we have used mostly general purpose tree automata abstractions, and we have an experience from [8] that special purpose abstraction may bring very significant speed-ups (in [8], it was sometimes two orders of magnitude or even more). Further research directions then include, for instance, checking of other kinds of properties (as, e.g., absence of garbage, which we know to be possible—cf. the full version of the paper [10]—but which we have not yet implemented), experimenting with combinations of our technique with techniques of non-pointer data abstraction, or termination checking.

Acknowledgement. This work was supported in part by the French Ministry of Research (ACI project Securité Informatique) and by the Czech Grant Agency within projects 102/05/H050, 102/04/0780, and 102/03/D211.

References

1. P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Regular Model Checking Made Simple and Efficient. In *Proc. of CONCUR'02*, volume 2421 of *LNCS*. Springer, 2002.
2. P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.
3. P.A. Abdulla, A. Legay, J. d'Orso, and A.Rezine. Simulation-Based Iteration of Tree Transducers. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
4. I. Balaban, A. Pnueli, and L. Zuck. Shape Analysis by Predicate Abstraction. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.
5. M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for Guided Tree Automata. In *Proc. of WIA'96*, volume 1260 of *LNCS*. Springer, 1997.
6. B. Boigelot, A. Legay, and P. Wolper. Iterating Transducers in the Large. In *Proc. of CAV'03*, volume 2725 of *LNCS*. Springer, 2003.
7. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. Technical Report TR-2006-3, Verimag, UJF/CNRS/INPG, Grenoble, 2006.
8. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
9. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS*, 149:37–48, 2006. A preliminary version was presented at Infinity'05.
10. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures, 2006. Full version available on URL: `http://www.fit.vutbr.cz/~vojnar/pubs.php`.

11. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*. Springer, 2004.
12. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
13. A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.
14. M. Bozga, R. Iosif, and Y. Lakhnech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003.
15. M. Češka, P. Erlebach, and T. Vojnar. Pattern-Based Verification of Programs with Extended Linear Linked Data Structures. *ENTCS*, 145:113–130, 2006. A preliminary version was presented at AVOCS'05.
16. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005. URL: `http://www.grappa.univ-lille3.fr/tata`.
17. J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
18. D. Distefano, P.W. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
19. J. Engelfriet. Bottom-up and Top-down Tree Transformations—A Comparison. *Mathematical System Theory*, 9:198–231, 1975.
20. P. Habermehl, R. Iosif, and T. Vojnar. Automata-Based Verification of Programs with Tree Updates. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
21. P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. *ENTCS*, 138:21–36, 2005. A preliminary version was presented at Infinity'04.
22. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.
23. N. Klarlund and A. Møller. MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark.
24. N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*. ACM Press, 1993.
25. O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proc. of ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.
26. A. Loginov, T. Reps, and M. Sagiv. Abstraction Refinement via Inductive Learning. In *Proc. of CAV'05*, volume 3576 of *LNCS*. Springer, 2005.
27. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.
28. A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001. Also in SIGPLAN Notices 36(5), 2001.
29. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
30. E. Shahar and A. Pnueli. Acceleration in Verification of Parameterized Tree Networks. Technical Report MCS02-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2002.
31. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using Language Inference to Verify Omega-regular Properties. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.
32. T. Yavuz-Kahveci and T. Bultan. Automated Verification of Concurrent Linked Lists with Counters. In *Proc. of SAS'02*, volume 2477 of *LNCS*. Springer, 2002.
33. G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A Logic of Reachable Patterns in Linked Data-Structures. In *Proc. of FOSSACS'06*, volume 3921 of *LNCS*. Springer, 2006.