Fully Automated Shape Analysis Based on Forest Automata

Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Forest automata (FA) have recently been proposed as a tool for shape analysis of complex heap structures. FA encode sets of tree decompositions of heap graphs in the form of tuples of tree automata. In order to allow for representing complex heap graphs, the notion of FA allowed one to provide user-defined FA (called boxes) that encode repetitive graph patterns of shape graphs to be used as alphabet symbols of other, higher-level FA. In this paper, we propose a novel technique of automatically learning the FA to be used as boxes that avoids the need of providing them manually. Further, we propose a significant improvement of the automata abstraction used in the analysis. The result is an efficient, fully-automated analysis that can handle even as complex data structures as skip lists, with the performance comparable to state-of-the-art fully-automated tools based on separation logic, which, however, specialise in dealing with linked lists only.

1 Introduction

Dealing with programs that use complex dynamic linked data structures belongs to the most challenging tasks in formal program analysis. The reason is a necessity of coping with infinite sets of reachable heap configurations that have a form of complex graphs. Representing and manipulating such sets in a sufficiently general, efficient, and automated way is a notoriously difficult problem.

In [6], a notion of *forest automata* (FA) has been proposed for representing sets of reachable configurations of programs with complex dynamic linked data structures. FA have a form of tuples of *tree automata* (TA) that encode sets of heap graphs decomposed into tuples of *tree components* whose leaves may refer back to the roots of the components. In order to allow for dealing with complex heap graphs, FA may be *hierarchically nested* by using them as alphabet symbols of other, higher-level FA. Alongside the notion of FA, a shape analysis applying FA in the framework of *abstract regular tree model checking* (ARTMC) [2] has been proposed in [6] and implemented in the Forester tool. ARTMC accelerates the computation of sets of reachable program configurations represented by FA by abstracting their component TA, which is done by collapsing some of their states. The analysis was experimentally shown to be capable of proving memory safety of quite rich classes of heap structures as well as to be quite efficient. However, it relied on the user to provide the needed nested FA—called *boxes*—to be used as alphabet symbols of the top-level FA.

In this paper, we propose a new shape analysis based on FA that avoids the need of manually providing the appropriate boxes. For that purpose, we propose a technique of automatically *learning* the FA to be used as boxes. The basic principle of the learning stems from the reason for which boxes were originally introduced into FA. In particular, FA must have a separate component TA for each node (called a *join*) of the represented graphs that has multiple incoming edges. If the number of joins is unbounded (as, e.g., in doubly linked lists, abbreviated as DLLs below), unboundedly many component TA are needed in flat FA. However, when some of the edges are hidden in a box (as, e.g., the prev and next links of DLLs in Fig. 1) and replaced by a single box-labelled edge, a finite number of component TA may suffice. Hence, the basic idea of our learning is to identify subgraphs of the FA-represented graphs that contain at least one join, and when they are enclosed—or, as we say later on, *folded*—into a box, the in-degree of the join decreases.

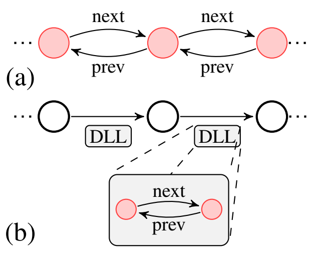

Fig. 1: (a) A DLL, (b) a hierarchical encoding of a DLL.

There are, of course, many ways to select the above mentioned subgraphs to be used as boxes. To choose among them, we propose several criteria that we found useful in a number of experiments. Most importantly, the boxes must be *reusable* in order to allow eliminating as many joins as possible. The general strategy here is to choose boxes that are *simple* and *small* since these are more likely to correspond to graph patterns that appear repeatedly in typical data structures. For instance, in the already mentioned case of DLLs, it is enough to use a box enclosing a single pair of next/prev links. On the other hand, as also discussed below, too simple boxes are sometimes not useful either.

Further, we propose a way how box learning can be efficiently integrated into the main analysis loop. In particular, we do not use the perhaps obvious approach of incrementally building a *database of boxes* whose instances would be sought in the generated FA. We found this approach inefficient due to the costly operation of finding instances of different boxes in FA-represented graphs. Instead, we always try to identify which subgraphs of the graphs represented by a given FA could be folded into a box, followed by looking into the so-far built database of boxes whether such a box has already been introduced or not. Moreover, this approach has the advantage that it allows one to use simple language inclusion checks for *approximate box folding*, replacing a set of subgraphs that appear in the graphs represented by a given FA by a larger set, which sometimes greatly accelerates the computation. Finally, to further improve the efficiency, we interleave the process of box learning with the *automata abstraction* into a single iterative process. In addition, we propose an FA-specific improvement of the basic automata abstraction which *accelerates the abstraction* of an FA using components of other FA. Intuitively, it lets the abstraction synthesize an invariant faster by allowing it to combine information coming from different branches of the symbolic computation.

We have prototyped the proposed techniques in Forester and evaluated it on a number of challenging case studies. The results show that the obtained approach is both quite general as well as efficient. We were, e.g., able to fully-automatically analyse programs with 2-level and 3-level skip lists, which, according to the best of our knowledge, no other fully-automated analyser can handle. On the other hand, our implementation achieves performance comparable and sometimes even better than that of Predator [4] (a winner of the heap manipulation division of SV-COMP'13) on list manipulating programs despite being able to handle much more general classes of heap graphs.

Related work. As discussed already above, we propose a new shape analysis based upon the notion of forest automata introduced in [6]. The new analysis is extended by a mechanism for automatically learning the needed nested FA, which is carefully integrated into the main analysis loop in order to maximize its efficiency. Moreover, we formalize the abstraction used in [6], which was not done in [6], and subsequently significantly refine it in order to improve both its generality as well as efficiency.

From the point of view of efficiency and degree of automation, the main alternative to our approach is the fully-automated use of separation logic with inductive list predicates as implemented in Space Invader [12] or SLAyer [1]. These approaches are, however, much less general than our approach since they are restricted to programs over certain classes of linked lists (and cannot handle even structures such as linked lists with data pointers pointing either inside the list nodes or optionally outside of them, which we can easily handle as discussed later on). A similar comparison applies to the Predator tool inspired by separation logic but using purely graph-based algorithms [4]. The work [9] on overlaid data structures mentions an extension of Space Invader to trees, but this extension is of a limited generality and requires some manual help.

In [5], an approach for synthesising inductive predicates in separation logic is proposed. This approach is shown to handle even tree-like structures with additional pointers. One of these structures, namely, the so-called mcf trees implementing trees whose nodes have an arbitrary number of successors linked in a DLL, is even more general than what can in principle be described by hierarchically nested FA (to describe mcf trees, recursively nested FA or FA based on hedge automata would be needed). On the other hand, the approach of [5] seems quite dependent on exploiting the fact that the encountered data structures are built in a "nice" way conforming to the structure of the predicate to be learnt (meaning, e.g., that lists are built by adding elements at the end only), which is close to providing an inductive definition of the data structure.

The work [10] proposes an approach which uses separation logic for generating numerical abstractions of heap manipulating programs allowing for checking both their safety as well as termination. The described experiments include even verification of programs with 2-level skip lists. However, the work still expects the user to manually provide an inductive definition of skip lists in advance. Likewise, the work [3] based on the so-called separating shape graphs reports on verification of programs with 2-level skip lists, but it also requires the user to come up with summary edges to be used for summarizing skip list segments, hence basically with an inductive definition of skip lists. Compared to [10,3], we did not have to provide any manual aid whatsoever to our technique when dealing with 2-level as well as 3-level skip lists in our experiments.

A concept of inferring graph grammar rules for the heap abstraction proposed in [8] has recently appeared in [11]. However, the proposed technique can so far only handle much less general structures than in our case.

2 Forest Automata

Given a word $\alpha = a_1 \dots a_n, n \geq 1$, we write α_i to denote its i-th symbol a_i. Given a total map $f : A \to B$, we use $dom(f)$ to denote its domain A and $img(f)$ to denote its image.

Graphs. A *ranked alphabet* is a finite set of symbols Σ associated with a mapping $\# : \Sigma \to \mathbb{N}_0$ that assigns ranks to symbols. A (directed, ordered, labelled) *graph* over Σ is a total map $g : V \to \Sigma \times V^*$ which assigns to every *node* $v \in V$ (1) a *label* from Σ, denoted as $\ell_g(v)$, and (2) a sequence of *successors* from V^*, denoted as $S_g(v)$, such that $\#\ell_g(v) = |S_g(v)|$. We drop the subscript g if no confusion may arise. Nodes v with $S(v) = \varepsilon$ are called *leaves*. For any $v \in V$ such that $g(v) = (a, v_1 \cdots v_n)$, we call the pair $v \mapsto (a, v_1 \cdots v_n)$ an *edge* of g. The *in-degree* of a node in V is the overall number of its occurrences in $g(v)$ across all $v \in V$. The nodes of a graph g with an in-degree larger than one are called *joins* of g.

A *path* from v to v' in g is a sequence $p = v_0, i_1, v_1, \ldots, i_n, v_n$ where $v_0 = v$, $v_n = v'$, and for each $j : 1 \leq j \leq n$, v_j is the i_j-th successor of v_{j-1}. The *length* of p is defined as $length(p) = n$. The *cost* of p is the sequence $i_1, \ldots, i_n$. We say that p is cheaper than another path p' iff the cost of p is lexicographically smaller than that of p'. A node u is *reachable* from a node v iff there is a path from v to u or $u = v$. A graph g is *accessible* from a node v iff all its nodes are reachable from v. The node v is then called the *root* of g. A *tree* is a graph t which is either empty, or it has exactly one root and each of its nodes is the i-th successor of at most one node v for some $i \in \mathbb{N}$.

Forests. Let $\Sigma \cap \mathbb{N} = \emptyset$. A Σ-labelled *forest* is a sequence of trees $t_1 \cdots t_n$ over $(\Sigma \cup \{1, \ldots, n\})$ where $\forall 1 \leq i \leq n : \#i = 0$. Leaves labelled by $i \in \mathbb{N}$ are called *root references*.

The forest $t_1 \cdots t_n$ represents the graph $\otimes t_1 \cdots t_n$ obtained by uniting the trees of $t_1 \cdots t_n$, assuming w.l.o.g. that their sets of nodes are disjoint, and interconnecting their roots with the corresponding root references. Formally, $\otimes t_1 \cdots t_n$ contains an edge $v \mapsto (a, v_1 \cdots v_m)$ iff there is an edge $v \mapsto (a, v'_1 \cdots v'_m)$ of some tree $t_i, 1 \leq i \leq n$, s.t. for all $1 \leq j \leq m$, $v_j = root(t_k)$ if v'_j is a root reference with $\ell(v'_j) = k$, and $v_j = v'_j$ otherwise.

Tree automata. A (finite, non-deterministic, top-down) *tree automaton* (TA) is a quadruple $A = (Q, \Sigma, \Delta, R)$ where Q is a finite set of *states*, $R \subseteq Q$ is a set of *root states*, Σ is a ranked alphabet, and Δ is a set of *transition rules*. Each transition rule is a triple of the form $(q, a, q_1 \ldots q_n)$ where $n \geq 0$, $q, q_1, \ldots, q_n \in Q$, $a \in \Sigma$, and $\#a = n$. In the special case where $n = 0$, we speak about the so-called *leaf rules*.

A *run* of A over a tree t over Σ is a mapping $\rho : dom(t) \to Q$ s.t. for each node $v \in dom(t)$ where $q = \rho(v)$, if $q_i = \rho(S(v)_i)$ for $1 \leq i \leq |S(v)|$, then Δ has a rule $q \to \ell(v)(q_1 \ldots q_{|S(v)|})$. We write $t \Longrightarrow_\rho q$ to denote that ρ is a run of A over t s.t. $\rho(root(t)) = q$. We use $t \Longrightarrow q$ to denote that $t \Longrightarrow_\rho q$ for some run ρ. The *language* of a state q is defined by $L(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of A is defined by $L(A) = \bigcup_{q \in R} L(q)$.

Graphs and forests with ports. We will further work with graphs with designated input and output points. An *io-graph* is a pair (g, ϕ), abbreviated as g_ϕ, where g is a graph and $\phi \in dom(g)^+$ a sequence of *ports* in which ϕ_1 is the *input port* and $\phi_2 \cdots \phi_{|\phi|}$ is a sequence of *output ports* such that the occurrence of ports in ϕ is unique. Ports and joins of g are called *cut-points* of g_ϕ. We use $cps(g_\phi)$ to denote all cut-points of g_ϕ. We say that g_ϕ is *accessible* if it is accessible from the input port ϕ_1.

An io-forest is a pair $f = (t_1 \cdots t_n, \pi)$ s.t. $n \geq 1$ and $\pi \in \{1, \ldots, n\}^+$ is a sequence of port indices, π_1 is the *input index*, and $\pi_2 \ldots \pi_{|\pi|}$ is a sequence of *output indices*, with no repetitions of indices in π. An io-forest encodes the io-graph $\otimes f$ where the ports of $\otimes t_1 \cdots t_n$ are roots of the trees defined by π, i.e., $\otimes f = (\otimes t_1 \cdots t_n, root(t_{\pi_1}) \cdots root(t_{\pi_n}))$.

Forest automata. A *forest automaton* (FA) over Σ is a pair $F = (A_1 \cdots A_n, \pi)$ where $n \geq 1$, $A_1 \cdots A_n$ is a sequence of tree automata over $\Sigma \cup \{1, \dots, n\}$, and $\pi \in \{1, \dots, n\}^+$ is a sequence of port indices as defined for io-forests. The *forest language* of F is the set of io-forests $L_f(F) = L(A_1) \times \cdots \times L(A_n) \times \{\pi\}$, and the *graph language* of F is the set of io-graphs $L(F) = \{\otimes f \mid f \in L_f(F)\}$.

Structured labels. We will further work with alphabets where symbols, called *structured labels*, have an inner structure. Let Γ be a ranked alphabet of *sub-labels*, ordered by a total ordering $\sqsubset_\Gamma$. We will work with graphs over the alphabet 2^Γ where for every symbol $A \subseteq \Gamma$, $\#A = \sum_{a \in A} \#a$. Let $e = v \mapsto (\{a_1, \dots, a_m\}, v_1 \cdots v_n)$ be an edge of a graph g where $n = \sum_{1 \leq i \leq m} \#a_i$ and $a_1 \sqsubset_\Gamma a_2 \sqsubset_\Gamma \cdots \sqsubset_\Gamma a_m$. The triple $e\langle i \rangle = v \to (a_i, v_k \cdots v_l)$, $1 \leq i \leq m$, from the sequence $e\langle 1 \rangle = v \to (a_1, v_1 \cdots v_{\#a_1}), \dots, e\langle m \rangle = v \to (a_m, v_{n-\#a_m+1} \cdots v_n)$ is called the i-th *sub-edge* of e (or the i-th sub-edge of v in g). We use $SE(g)$ to denote the set of all sub-edges of g. We say that a node v of a graph is *isolated* if it does not appear within any sub-edge, neither as an origin (i.e., $\ell(v) = \emptyset$) nor as a target. A graph g without isolated nodes is unambiguously determined by $SE(g)$ and vice versa (due to the total ordering $\sqsubset_\Gamma$ and since g has no isolated nodes). We further restrict ourselves to graphs with structured labels and without isolated nodes.

A counterpart of the notion of sub-edges in the context of rules of TA is the notion of rule-terms, defined as follows: Given a rule $\delta = (q, \{a_1, \dots, a_m\}, q_1 \cdots q_n)$ of a TA over structured labels of 2^Γ, *rule-terms* of δ are the terms $\delta\langle 1 \rangle = a_1(q_1 \cdots q_{\#a_1}), \dots, \delta\langle m \rangle = a_m(q_{n-\#a_m+1} \cdots q_n)$ where $\delta\langle i \rangle, 1 \leq i \leq m$, is called the *$i$-th rule-term of* δ.

Forest automata of a higher level. We let Γ_1 be the set of all forest automata over 2^Γ and call its elements forest automata over Γ *of level 1*. For $i > 1$, we define Γ_i as the set of all forest automata over ranked alphabets $2^{\Gamma \cup \Delta}$ where $\Delta \subseteq \Gamma_{i-1}$ is any nonempty finite set of FA of level $i-1$. We denote elements of Γ_i as forest automata over Γ *of level i*. The rank $\#F$ of an FA F in these alphabets is the number of its output port indices. When used in an FA F over $2^{\Gamma \cup \Delta}$, the forest automata from Δ are called *boxes* of F. We write Γ_* to denote $\cup_{i \geq 0} \Gamma_i$ and assume that Γ_* is ordered by some total ordering $\sqsubset_{\Gamma_*}$.

An FA F of a higher level over Γ accepts graphs where forest automata of lower levels appear as sub-labels. To define the semantics of F as a set of graphs over Γ, we need the following operation of *sub-edge replacement* where a sub-edge of a graph is substituted by another graph. Intuitively, the sub-edge is removed, and its origin and targets are identified with the input and output ports of the substituted graph, respectively.

Formally, let g be a graph with an edge $e \in g$ and its i-th sub-edge $e\langle i \rangle = v_1 \to (a, v_2 \cdots v_n), 1 \leq i \leq |S_g(v_1)|$. Let g'_ϕ be an io-graph with $|\phi| = n$. Assume w.l.o.g. that $dom(g) \cap dom(g') = \emptyset$. The sub-edge $e\langle i \rangle$ can be replaced by g' provided that $\forall 1 \leq j \leq n : \ell_g(v_j) \cap \ell_{g'}(\phi_j) = \emptyset$, which means that the node $v_j \in dom(g)$ and the corresponding port $\phi_j \in dom(g')$ do not have successors reachable over the same symbol. If the replacement can be done, the result, denoted $g[g'_\phi / e\langle i \rangle]$, is the graph g_n in the sequence $g_0, \dots, g_n$ of graphs defined as follows: $SE(g_0) = SE(g) \cup SE(g') \setminus \{e\langle i \rangle\}$, and for each $j : 1 \leq j \leq n$, the graph g_j arises from g_{j-1} by (1) deriving a graph h by replacing the origin of the sub-edges of the j-th port ϕ_j of g' by v_j, (2) redirecting edges leading to ϕ_j to v_j, i.e., replacing all occurrences of ϕ_j in $img(h)$ by v_j, and (3) removing ϕ_j.

If the symbol a above is an FA and $g'_\phi \in L(a)$, we say that $h = g[g'_\phi / e\langle i \rangle]$ is an *unfolding* of g, written $g \prec h$. Conversely, we say that g arises from h by *folding* g'_ϕ into

$e\langle i\rangle$. Let $\prec^*$ be the reflexive transitive closure of $\prec$. The Γ-*semantics* of g is then the set of graphs g' over Γ s.t. $g \prec^* g'$, denoted $[\![g]\!]_\Gamma$, or just $[\![g]\!]$ if no confusion may arise. For an FA F of a higher level over Γ, we let $[\![F]\!] = \bigcup_{g_\phi \in L(F)}([\![g]\!] \times \{\phi\})$.

Canonicity. We call an io-forest $f = (t_1 \cdots t_n, \pi)$ *minimal* iff the roots of the trees $t_1 \cdots t_n$ are the cut-points of $\otimes f$. A minimal forest representation of a graph is unique up to reordering of $t_1 \cdots t_n$. Let the *canonical ordering* of cut-points of $\otimes f$ be defined by the cost of the cheapest paths leading from the input port to them. We say that f is *canonical* iff it is minimal, $\otimes f$ is accessible, and the trees within $t_1 \cdots t_n$ are ordered by the canonical ordering of their roots (which are cut-points of $\otimes f$). A canonical forest is thus a unique representation of an accessible io-graph. We say that an FA *respects canonicity* iff all forests from its forest language are canonical. Respecting canonicity makes it possible to efficiently test FA language inclusion by testing TA language inclusion of the respective components of two FA. This method is precise for FA of level 1 and sound (not always complete) for FA of a higher level [6].

In practice, we keep automata in the so called *state uniform* form, which simplifies maintaining of the canonicity respecting form [6] (and it is also useful when abstracting and "folding", as discussed in the following). It is defined as follows. Given a node v of a tree t in an io-forest, we define its *span* as the pair (α, V) where $\alpha \in \mathbb{N}^*$ is the sequence of labels of root references reachable from the root of t ordered according to the prices of the cheapest paths to them, and $V \subseteq \mathbb{N}$ is the set of labels of references which occur more than once in t. The state uniform form then requires that all nodes of forests from $L(F)$ that are labelled by the same state q in some accepting run of F have the same span, which we denote by $span(q)$.

3 FA-based Shape Analysis

We now provide a high-level overview of the main loop of our shape analysis. The analysis automatically discovers memory safety errors (such as invalid dereferences of `null` or undefined pointers, double frees, or memory leaks) and provides an FA-represented over-approximation of the sets of heap configurations reachable at each program line. We consider sequential non-recursive C programs manipulating the heap. Each heap cell may have several *pointer selectors* and *data selectors* from some finite data domain (below, *PSel* denotes the set of pointer selectors, *DSel* denotes the set of data selectors, and $\mathbb{D}$ denotes the data domain).

Heap representation. A single heap configuration is encoded as an io-graph $g_{\mathtt{sf}}$ over the ranked alphabet of structured labels 2^Γ with sub-labels from the ranked alphabet $\Gamma = PSel \cup (DSel \times \mathbb{D})$ with the ranking function that assigns each pointer selector 1 and each data selector 0. In this graph, an allocated memory cell is represented by a node v, and its internal structure of selectors is given by a label $\ell_g(v) \in 2^\Gamma$. Values of data selectors are stored directly in the structured label of a node as sub-labels from $DSel \times \mathbb{D}$, so, e.g., a singly linked list cell with the data value 42 and the successor node x_{next} may be represented by a node x such that $\ell_g(x) = \{\mathtt{next}(x_{next}), (\mathtt{data}, 42)(\varepsilon)\}$. Selectors with undefined values are represented such that the corresponding sub-labels are not in $\ell_g(x)$. The null value is modelled as the special node `null` such that $\ell_g(\mathtt{null}) = \emptyset$. The input port $\texttt{sf}$ represents a special node that contains the *stack frame* of the analysed function, i.e. a structure where selectors correspond to variables of the function.

In order to represent (infinite) *sets* of heap configurations, we use state uniform FA of a higher level to represent sets of canonical io-forests representing the heap configurations. The FA used as boxes are learnt during the analysis using the learning algorithm presented in Sec. 4.

Symbolic Execution. The verification procedure performs standard abstract interpretation with the abstract domain consisting of sets of state uniform FA (a single FA does not suffice as FA are not closed under union) representing sets of heap configurations at particular program locations. The computation starts from the initial heap configuration given by an FA for the io-graph $g_{\texttt{sf}}$ where g comprises two nodes: `null` and `sf` where $\ell_g(\texttt{sf}) = \emptyset$. The computation then executes abstract transformers corresponding to program statements until the sets of FA held at program locations stabilise. We note that abstract transformers corresponding to pointer manipulating statements are exact. Executing the abstract transformer $\tau_{\texttt{op}}$ over a set of FA $\mathcal{S}$ is performed separately for every $F \in \mathcal{S}$. Some of boxes are first *unfolded* to uncover the accessed part of the heaps, then the update is performed. The detailed description of these steps can be found in [7].

At junctions of program paths, the analysis computes unions of sets of FA. At loop points, the union is followed by widening. The widening is performed by applying box *folding* and *abstraction* repeatedly in a loop on each FA from $\mathcal{S}$ until the result stabilises. An elaboration of these two operations, described in detail in Sec. 4 and 5 respectively, belongs to the main contribution of the presented paper.

4 Learning of Boxes

Sets of graphs with an unbounded number of joins can only be described by FA with the help of boxes. In particular, boxes allow one to replace (multiple) incoming sub-edges of a join by a single sub-edge, and hence lower the in-degree of the join. Decreasing the in-degree to 1 turns the join into an ordinary node. When a box is then used in a cycle of an FA, it effectively generates an unbounded number of joins.

The boxes are introduced by the operation of *folding* of an FA F which transforms F into an FA F' and a box B used in F' such that $[\![F]\!] = [\![F']\!]$. However, the graphs in $L(F')$ may contain less joins since some of them are hidden in the box B, which encodes a set of subgraphs containing a join and appearing repeatedly in the graphs of $L(F)$. Before we explain folding, we give a characterisation of subgraphs of graphs of $L(F)$ which we want to fold into a box B. Our choice of the subgraphs to be folded is a compromise between two high-level requirements. On the one hand, the folded subgraphs should contain incoming edges of joins and be as simple as possible in order to be reusable. On the other hand, the subgraphs should not be too small in order not to have to be subsequently folded within other boxes (in the worst case, leading to generation of unboundedly nested boxes). Ideally, the hierarchical structuring of boxes should respect the natural hierarchical structuring of the data structures being handled since if this is not the case, unboundedly many boxes may again be needed.

4.1 Knots of Graphs

A graph h is a *subgraph* of a graph g iff $SE(h) \subseteq SE(g)$. The *border* of h in g is the subset of the set $dom(h)$ of nodes of h that are incident with sub-edges in $SE(g) \setminus SE(h)$. A *trace* from a node u to a node v in a graph g is a set of sub-edges $t = \{e_0, \ldots, e_n\} \subseteq SE(g)$ such that $n \geq 1$, e_0 is an outgoing sub-edge of u, e_n is an incoming sub-edge of v, the origin of e_i is one of the targets of e_{i-1} for all $1 \leq i \leq n$, and no two sub-edges have the same origin. We call the origins of $e_1, \ldots, e_n$ the *inner nodes* of the trace. A trace from u to v is *straight* iff none of its inner nodes is a cut-point. A *cycle* is a trace from a node v to v. A *confluence* of g_ϕ is either a cycle of g_ϕ or it is the union of two disjoint traces starting at a node u, called the *base*, and ending in the node v, called the *tip* (for a cycle, the base and the tip coincide).

Given an io-graph g_ϕ, the *signature* of a sub-graph h of g is the minimum subset $sig(h)$ of $cps(g_\phi)$ that (1) contains $cps(g_\phi) \cap dom(h)$ and (2) all nodes of h, except the nodes of $sig(h)$ themselves, are reachable by straight traces from $sig(h)$. Intuitively, $sig(h)$ contains all cut-points of h plus the closest cut-points to h which lie outside of h but which are needed so that all nodes of h are reachable from the signature. Consider the example of the graph g_u in Fig. 2 in which cut-points are represented by •. The signature of g_u is the set $\{u, v\}$. The signature of the highlighted subgraph h is also equal to $\{u, v\}$. Given a set $U \subseteq cps(g_\phi)$, a *confluence of* U is a confluence of g_ϕ with the signature within U. Intuitively, the confluence of a set of cut-points U is a confluence whose cut-points belong to U plus in case the base is not a cut-point, then the closest cut-point from which the base is reachable is also from U. Finally, the *closure* of U is the smallest subgraph h of g_ϕ that (1) contains all confluences of U and (2) for every inner node v of a straight trace of h, it contains all straight traces from v to leaves of g. The closure of the signature $\{u, v\}$ of the graph g_u in Fig. 2 is the highlighted subgraph h. Intuitively, Point 1 includes into the closure all nodes and sub-edges that appear on straight traces between nodes of U apart from those that do not lie on any confluence (such as node u in Fig. 2). Note that nodes x and y in Fig. 2, which are leaves of g_u, are not in the closure as they are not reachable from an inner node of any straight trace of h. The *closure of a subgraph* h of g_ϕ is the closure of its signature, and h is *closed* iff it equals its closure.

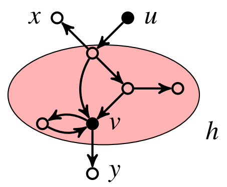

Fig. 2: Closure.

Knots. For the rest of Sec. 4.1, let us fix an io-graph $g_\phi \in L(F)$. We now introduce the notion of a knot which summarises the desired properties of a subgraph k of g that is to be folded into a box. A *knot* k of g_ϕ is a subgraph of g such that: (1) k is a confluence, (2) k is the union of two knots with intersecting sets of sub-edges, or (3) k is the closure of a knot. A *decomposition* of a knot k is a set of knots such that the union of their sub-edges equals $SE(k)$. The *complexity of a decomposition* of k is the maximum of sizes of signatures of its elements. We define the *complexity of a knot* as the minimum of the complexities of its decompositions. A knot k of complexity n is an *optimal knot of complexity* n if it is maximal among knots of complexity n and if it has a root. The root must be reachable from the input port of g_ϕ by a trace that does not intersect with sub-edges of the optimal knot. Notice that the requirement of maximality implies that optimal knots are closed.

The following lemma, proven in [7], implies that optimal knots are uniquely identified by their signatures, which is crucial for the folding algorithm presented later.

Lemma 1. *The signature of an optimal knot of g_ϕ equals the signature of its closure.*

Next, we explain what is the motivation behind the notion of an optimal knot:

Confluences. As mentioned above, in order to allow one to eliminate a join, a knot must contain some join v together with at least one incoming sub-edge in case the knot is based on a loop and at least two sub-edges otherwise. Since g_ϕ is accessible (meaning that there do not exist any traces that cannot be extended to start from the same node), the edge must belong to some confluence c of g_ϕ. If the folding operation does not fold the entire c, then a new join is created on the border of the introduced box: one of its incoming sub-edges is labelled by the box that replaces the folded knot, another one is the last edge of one of the traces of c. Confluences are therefore the smallest subgraphs that can be folded in a meaningful way.

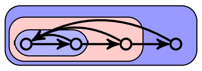

Fig. 3: A list with head pointers.

Uniting knots. If two different confluences c and c' share an edge, then after folding c, the resulting edge shares with c' two nodes (at least one being a target node), and thus c' contains a join of g_ϕ. To eliminate this join too, both confluences must be folded together. A similar reasoning may be repeated with knots in general. Usefulness of this rule may be illustrated by an example of the set of lists with head pointers. Without uniting, every list would generate a hierarchy of knots of the same depth as the length of the list, as illustrated in Fig. 3. This is clearly impractical since the entire set could not be represented using finitely many boxes. Rule 2 unites all knots into one that contains the entire list, and the set of all such knots can then be represented by a single FA (containing a loop accepting the inner nodes of the lists).

Complexity of knots. The notion of complexity is introduced to limit the effect of Rule 2 of the definition of a knot, which unites knots that share a sub-edge, and to hopefully make it follow the natural hierarchical structuring of data structures. Consider, for instance, the case of singly-linked lists (SLLs) of cyclic doubly-linked lists (DLLs). In this case, it is natural to first fold the particular segments of the DLLs (denoted as DLSs below), i.e., to introduce a box for a single pair of next and prev pointers. This way, one effectively obtains SLLs of cyclic SLLs. Subsequently, one can fold the cyclic SLLs into a higher-level box. However, uniting all knots with a common sub-edge would create knots that contain entire cyclic DLLs (requiring unboundedly many joins inside the box). The reason is that in addition to the confluences corresponding to DLSs, there are confluences which traverse the entire cyclic DLLs and that share sub-edges with all DLSs (this is in particular the case of the two circular sequences consisting solely of next and prev pointers respectively). To avoid the undesirable folding, we exploit the notion of complexity and fold graphs in successive rounds. In each round we fold all optimal knots with the smallest complexity (as described in Sec. 4.2), which should correspond to the currently most nested, not yet folded, sub-structures. In the previous example, the algorithm starts by folding DLSs of complexity 2, because the complexity of the confluences in cyclic DLLs is given by the number of the DLSs they traverse.

Closure of knots. The closure is introduced for practical reasons. It allows one to identify optimal knots by their signatures, which is then used to simplify automata constructions that implement folding on the level of FA (cf. Sec. 4.2).

Root of an optimal knot. The requirement for an optimal knot k to have a root is to guarantee that if an io-graph h_ψ containing a box B representing k is accessible, then the io-graph $h_\psi[k/B]$ emerging by substituting k for a sub-edge labelled with B is accessible, and vice versa. It is also a necessary condition for the existence of a canonical forest representation of the knot itself (since one needs to order the cut-points w.r.t. the prices of the paths leading to them from the input port of the knot).

4.2 Folding in the Abstraction Loop

In this section, we describe the operation of folding together with the main abstraction loop of which folding is an integral part. The pseudo-code of the main abstraction loop is shown in Alg. 1. The algorithm modifies a set of FA until it reaches a fixpoint. Folding on line 5 is a sub-procedure of the algorithm which looks for substructures of FA that accept optimal knots, and replaces these substructures by boxes that represent the corresponding optimal knots. The operation of folding is itself composed of four consecutive steps: *Identifying indices*, *Splitting*, *Constructing boxes*, and *Applying boxes*. For space reasons, we give only an overview of the steps of the main abstraction loop and folding. Details may be found in [7].

```
1 Unfold solitaire boxes
2 repeat
3     Normalise
4     Abstract
5     Fold
6 until fixpoint
```

Alg. 1: Abstraction Loop

Unfolding of solitaire boxes. Folding is in practice applied on FA that accept partially folded graphs (only some of the optimal knots are folded). This may lead the algorithm to hierarchically fold data structures that are not hierarchical, causing the symbolic execution not to terminate. For example, consider a program that creates a DLL of an arbitrary length. Whenever a new DLS is attached, the folding algorithm would enclose it into a box together with the tail which was folded previously. This would lead to creation of a hierarchical structure of an unbounded depth (see Fig. 4), which would cause the symbolic execution to never reach a fixpoint. Intuitively, this is a situation when a repetition of subgraphs may be expressed by an automaton loop that iterates a box, but it is instead misinterpreted as a recursive nesting of graphs. This situation may happen when a newly created box contains another box that cannot be iterated since it does not appear on a loop (e.g, in Fig. 4 there is always one occurrence of a box encoding a shorter DLL fragment inside a higher-level box). This issue is addressed in the presented algorithm by first unfolding all occurrences of boxes that are not iterated by automata loops before folding is started.

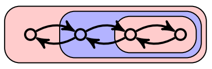
Fig. 4: DLL.

Normalising. We define the *index* of a cut-point $u \in cps(g_\phi)$ as its position in the canonical ordering of cut-points of g_ϕ, and the *index* of a closed subgraph h of g_ϕ as the set of indices of the cut-points in $sig(h)$. The folding algorithm expects the input FA F to satisfy the property that all io-graphs of $L(F)$ have the same indices of closed knots. The reason is that folding starts by identifying the index of an optimal knot of an arbitrary io-graph from $L(F)$, and then it creates a box which accepts all closed subgraphs of the io-graphs from g_ϕ with the same index. We need a guarantee that *all* these subgraphs are indeed optimal knots. This guarantee can be achieved if the io-graphs from $L(F)$ have equivalent interconnections of cut-points, as defined below.

We define the relation $\sim_{g_\phi} \subseteq \mathbb{N} \times \mathbb{N}$ between indices of closed knots of g_ϕ such that $N \sim_{g_\phi} N'$ iff there is a closed knot k of g_ϕ with the index N and a closed knot k' with the index N' such that k and k' have intersecting sets of sub-edges. We say that two io-graphs g_ϕ and h_ψ are *interconnection equivalent* iff $\sim_{g_\phi} = \sim_{h_\psi}$.

Lemma 2. *Interconnection equivalent io-graphs have the same indices of optimal knots.*

Interconnection equivalence of all io-graphs in the language of an FA F is achieved by transforming F to the *interconnection respecting form*. This form requires that the language of every TA of the FA consists of interconnection equivalent trees (when viewing root references and roots as cut-points with corresponding indices). The transformation is described in [7]. The normalisation step also includes a transformation into the state uniform and canonicity respecting form.

Abstraction. We use abstraction described in Sec. 5 that preserves the canonicity respecting form of TA as well as their state uniformity. It may break interconnection uniformity, in which case it is followed by another round of normalisation. Abstraction is included into each round of folding for the reason that it leads to learning more general boxes. For instance, an FA encoding a cyclic list of one particular length is first abstracted into an FA encoding a set of cyclic lists of all lengths, and the entire set is then folded into a single box.

Identifying indices. For every FA F entering this sub-procedure, we pick an arbitrary io-graph $g_\phi \in L(F)$, find all its optimal knots of the smallest possible complexity n, and extract their indices. By Lemma 2 and since F is normalised, indices of the optimal knots are the same for all io-graphs in $L(F)$. For every found index, the following steps fold all optimal knots with that index at once. Optimal knots of complexity n do not share sub-edges, the order in which they are folded is therefore not important.

Splitting. For an FA $F = (A_1 \cdots A_n, \pi)$ and an index I of an optimal knot found in the previous step, splitting transforms F into a (set of) new FA with the same language. The nodes of the borders of I-indexed optimal knots of io-graphs from $L(F)$ become roots of trees of io-forests accepted by the new FA. Let $s \in I$ be a position in F such that the s-indexed cut-points of io-graphs from $L(F)$ reach all the other I-indexed cut-points. The index s exists since an optimal knot has a root. Due to the definition of the closure, the border contains all I-indexed cut-points, with the possible exception of s. The s-th cut-point may be replaced in the border of the I-indexed optimal knot by the base e of the I-indexed confluence that is the first one reached from the s-th cut-point by a straight path. We call e the *entry*. The entry e is a root of the optimal knot, and the s-th cut-point is the only I-indexed cut-point that might be outside the knot. If e is indeed different from the s-th cut-point, then the s-th tree of forests accepted by F must be split into two trees in the new FA: The subtree rooted at the entry is replaced by a reference to a new tree. The new tree then equals the subtree of the original s-th tree rooted at the entry.

The construction is carried out as follows. We find all states and all of their rules that accept entry nodes. We denote such states and rules as entry states and rules. For every entry state q, we create a new FA F_q^0 which is a copy of F but with the s-th TA A_s split to a new s-th TA A'_s and a new $(n+1)$-th TA A_{n+1}. The TA A'_s is obtained from A_s by changing the entry rules of q to accept just a reference to the new $(n+1)$-th root and by removing entry rules of all other entry states (the entry states are processed separately in

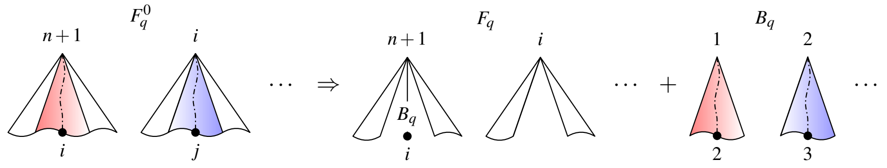

Fig. 5: Creation of F_q and B_q from F_q^0. The subtrees that contain references $i, j \in J$ are taken into B_q, and replaced by the B_q-labelled sub-edge in F_q.

order to preserve possibly different contexts of entry nodes accepted at different states). The new TA A_{n+1} is a copy of A_s but with the only accepting state being q. Note that the construction is justified since due to state uniformity, each node that is accepted by an entry rule and that does not appear below a node that is also accepted by an entry rule is an entry node. In the result, the set $J = (I \setminus \{s\}) \cup \{n+1\}$ contains the positions of the trees of forests of F_q^0 rooted at the nodes of the borders of I-indexed optimal knots.

Constructing boxes. For every F_q^0 and J being the result of splitting F according to an index I, a box B_q is constructed from F_q^0. We transform TA of F_q^0 indexed by the elements of J. The resulting TA will accept the original trees up to that the roots are stripped from the children that cannot reach a reference to J. To turn these TA into an FA accepting optimal knots with the index I, it remains to order the obtained TA and define port indices, which is described in detail in [7]. Roughly, the input index of the box will be the position j to which we place the modified $(n+1)$-th TA of F_q^0 (the one that accepts trees rooted at the entry). The output indices are the positions of the TA with indices $J \setminus \{j\}$ in F_q^0 which accept trees rooted at cut-points of the border of the optimal knots.

Applying boxes. This is the last step of folding. For every F_q^0, J, and B_q which are the result of splitting F according to an index I, we construct an FA F_q that accepts graphs of F where knots enclosed in B_q are substituted by a sub-edge with the label B_q. It is created from F_q^0 by (1) leaving out the parts of root rules of its TA that were taken into B_q, and (2) adding the rule-term $B_q(r_1, \ldots, r_m)$ to the rule-terms of root rules of the $(n+1)$-th component of F_q^0 (these are rules used to accept the roots of the optimal knots enclosed in B_q). The states $r_1, \ldots, r_m$ are fresh states that accept root references to the appropriate elements of J (to connect the borders of knots of B_q correctly to the graphs of F_q—the details may be found in [7]). The FA F_q now accepts graphs where optimal knots of graphs of $L(F)$ with the signature I are hidden inside B_q. Creation of B_q and of its counterpart F_q from F_q^0 is illustrated in Fig. 5 where $i, j, \ldots \in J$.

During the analysis, the discovered boxes must be stored in a database and tested for equivalence with the newly discovered ones since the alphabets of FA would otherwise grow with every operation of folding *ad infinitum*. That is, every discovered box is given a unique name, and whenever a semantically equivalent box is folded, the newly created edge-term is labelled by that name. This step offers an opportunity for introducing another form of acceleration of the symbolic computation. Namely, when a box B is found by the procedure described above, and another box B' with a name N s.t. $[\![B']\!] \subset [\![B]\!]$ is already in the database, we associate the name N with B instead of with B' and restart the analysis (i.e., start the analysis from the scratch, remembering just the updated database of boxes). If, on the other hand, $[\![B]\!] \subseteq [\![B']\!]$, the folding is performed using the name N of B', thus overapproximating the semantics of the folded FA. As presented in Sec. 6, this variant of the procedure, called *folding by inclusion*, performs in some difficult cases significantly better than the former variant, called *folding by equivalence*.

5 Abstraction

The abstraction we use in our analysis is based on the general techniques described in the framework of abstract regular (tree) model checking [2]. We, in particular, build on the *finite height abstraction* of TA. It is parameterised by a height $k \in \mathbb{N}$, and it collapses TA states q, q' iff they accept trees with the same sets of prefixes of the height at most k (the prefix of height k of a tree is a subgraph of the tree which contains all paths from the root of length at most k). This defines an equivalence on states denoted by $\approx_k$. The equivalence $\approx_k$ is further refined to deal with various features special for FA. Namely, it has to work over tuples of TA and cope with the interconnection of the TA via root references, with the hierarchical structuring, and with the fact that we use a *set* of FA instead of a single FA to represent the abstract context at a particular program location.

Refinements of $\approx_k$. First, in order to maintain the same basic shape of the heap after abstraction (such that no cut-point would, e.g., suddenly appear or disappear), we refine $\approx_k$ by requiring that equivalent states must have the same spans (as defined in Sec. 2). When applied on $\approx_1$, which corresponds to equivalence of data types, this refinement provided enough precision for most of the case studies presented later on, with the exception of the most difficult ones, namely programs with skip lists [13]. To verify these programs, we needed to further refine the abstraction to distinguish automata states whenever trees from their languages encode tree components containing a different number of unique paths to some root reference, but some of these paths are hidden inside boxes. In particular, two states q, q' can be equivalent only if for every io-graph g_ϕ from the graph language of the FA, for every two nodes $u, v \in dom(g_\phi)$ accepted by q and q', respectively, in an accepting run of the corresponding TA, the following holds: For every $w \in cps(g_\phi)$, both u and v have the same number of outgoing sub-edges (selectors) in $[\![g_\phi]\!]$ which start a trace in $[\![g_\phi]\!]$ leading to w. According to our experiments, this refinement does not cost almost any performance, and hence we use it by default.

Abstraction for Sets of FA. Our analysis works with sets of FA. We observed that abstracting individual FA from a set of FA in isolation is sometimes slow since in each of the FA, the abstraction widens some selector paths only, and it takes a while until an FA in which all possible selector paths are widened is obtained. For instance, when analysing a program that creates binary trees, before reaching a fixpoint, the symbolic analysis generates many FA, each of them accepting a subset of binary trees with some of the branches restricted to a bounded length (e.g., trees with no right branches, trees with a single right branch of length 1, length 2, etc.). In such cases, it helps when the abstraction has an opportunity to combine information from several FA. For instance, consider an FA that encodes binary trees degenerated to an arbitrarily long left branch, and another FA that encodes trees degenerated to right branches only. Abstracting these FA in isolation has no effect. However, if the abstraction is allowed to collapse states from both of these FA, it can generate an FA accepting all possible branches.

Unfortunately, the natural solution to achieve the above, which is to unite FA before abstraction, cannot be used since FA are not closed under union (uniting TA component-wise overapproximates the union). However, it is possible to enrich the automata structure of an FA F by TA states and rules of another one without changing the language of F, and in this way allow the abstraction to combine the information from both FA. In particular, before abstracting an FA $F = (A_1 \cdots A_n, \pi)$ from a set S of FA, we pre-process it as follows. (1) We pick automata $F' = (A'_1 \cdots A'_n, \pi) \in S$ which are compatible with F in that they have the same number of TA, the same port references, and for each $1 \leq i \leq n$, the root states of A'_i have the same spans as the root states of A_i. (2) For all such F' and each $1 \leq i \leq n$, we add rules and states of A'_i to A_i, but we keep the original set of root states of A_i. Since we assume that the sets of state of TAs of different FA are disjoint, the language of A_i stays the same, but its structure is enriched, which helps the abstraction to perform a coarser widening.

6 Experimental Results

We have implemented the above proposed techniques in the Forester tool and tested their generality and efficiency on a number of case studies. In the experiments, we compare two configurations of Forester, and we also compare the results of Forester with those of Predator [4], which uses a graph-based memory representation inspired by separation logic with higher-order list predicates. We do not provide a comparison with Space Invader [12] and SLAyer [1], based also on separation logic with higher-order list predicates, since in our experiments they were outperformed by Predator.

In the experiments, we considered programs with various types of lists (singly and doubly linked, cyclic, nested, with skip pointers), trees, and their combinations. In the case of skip lists, we had to slightly modify the algorithms since their original versions use an ordering on the data stored in the nodes of the lists (which we currently do not support) in order to guarantee that the search window delimited on some level of skip pointers is not left on any lower level of the skip pointers. In our modification, we added an additional explicit end-of-window pointer. We checked the programs for memory safety only, i.e., we did not check data-dependent properties.

Table 1 gives running times in seconds (the average of 10 executions) of the tools on our case studies. “Basic” stands for Forester with the abstraction applied on individual FA only and “SFA” stands for Forester with the abstraction for sets of FA. The value T means that the running time of the tool exceeded 30 minutes, and the value Err means that the tool reported a spurious error. The names of the examples in the table contain the name of the data structure manipulated in the program, which is “SLL” for singly linked lists, “DLL” for doubly linked lists (the “C” prefix denotes cyclic lists), “tree” for binary trees, “tree+parents” for trees with parent pointers. Nested variants of SLL (DLL) are named as “SLL (DLL) of” and the type of the nested structure. In particular, “SLL of 0/1 SLLs” stands for SLL of a nested SLL of length 0 or 1, and “SLL of 2CDLLs” stands for SLL whose each node is a root of two CDLLs. The “+head” flag stands for a list where each element points to the head of the list and the subscript “Linux” denotes the implementation of lists used in the Linux kernel, which uses type casts and a restricted pointer arithmetic. The “DLL+subdata” stands for a kind of a DLL with data

Table 1: Results of the experiments

Example	basic	SFA	boxes	Predator	Example	basic	SFA	boxes	Predator
SLL (delete)	0.03	0.04		0.04	DLL (reverse)	0.04	0.06	1 / 1	0.03
SLL (bubblesort)	0.04	0.04		0.03	DLL (insert)	0.06	0.07	1 / 1	0.05
SLL (mergesort)	0.08	0.15		0.10	DLL (insertsort1)	0.35	0.40	1 / 1	0.11
SLL (insertsort)	0.05	0.05		0.04	DLL (insertsort2)	0.11	0.12	1 / 1	0.05
SLL (reverse)	0.03	0.03		0.03	DLL of CDLLs	5.67	1.25	8 / 7	0.22
SLL+head	0.05	0.05		0.03	DLL+subdata	0.06	0.09	- / 2	T
SLL of 0/1 SLLs	0.03	0.03		0.11	CDLL	0.03	0.03	1 / 1	0.03
SLL$_{\text{Linux}}$	0.03	0.03		0.03	tree	0.14	0.14		Err
SLL of CSLLs	2.07	0.73	3 / 4	0.12	tree+parents	0.18	0.21	2 / 2	T
SLL of 2CDLLs$_{\text{Linux}}$	0.16	0.17	13 / 5	0.25	tree+stack	0.09	0.08		Err
skip list$_2$	0.66	0.42	- / 3	T	tree (DSW)	1.74	0.40		Err
skip list$_3$	T	9.14	- / 7	T	tree of CSLLs	0.32	0.42	- / 4	Err

pointers pointing either inside the list nodes or optionally outside of them. For a "skip list", the subscript denotes the number of skip pointers. In the example "tree+stack", a randomly constructed tree is deleted using a stack, and "DSW" stands for the Deutsch-Schorr-Waite tree traversal (the Lindstrom variant). All experiments start with a random creation and end with a disposal of the specified structure while the indicated procedure (if any) is performed in between. The experiments were run on a machine with the Intel i7-2600 (3.40 GHz) CPU and 16 GiB of RAM.

The table further contains the column "boxes" where the value "X/Y" means that X manually created boxes were provided to the analysis that did not use learning while Y boxes were learnt when the box learning procedure was enabled. The value "-" of X means that we did not run the given example with manually constructed boxes since their construction was too tedious. If user-defined boxes are given to Forester in advance, the speedup is in most cases negligible, with the exception of "DLL of CDLLs" and "SLL of CSLLs", where it is up to 7 times. In a majority of cases, the learnt boxes were the same as the ones created manually. However, in some cases, such as "SLL of 2CDLLs$_{\text{Linux}}$", the learning algorithm found a smaller set of more elaborate boxes than those provided manually.

In the experiments, we use folding by inclusion as defined in Sec. 4.2. For simpler cases, the performance matched the performance of folding by equivalence, but for the more difficult examples it was considerably faster (such as for "skip list$_2$" when the time decreased from 3.82 s to 0.66 s), and only when it was used the analysis of "skip list$_3$" succeeded. Further, the implementation folds optimal knots of the complexity ≤ 2 which is enough for the considered examples. Finally, note that the performance of Forester in the considered experiments is indeed comparable with that of Predator even though Forester can handle much more general data structures.

7 Conclusion

We have proposed a new shape analysis using forest automata which—unlike the previously known approach based on FA—is fully automated. For that purpose, we have proposed a technique of automatically learning FA called boxes to be used as alphabet symbols in higher-level FA when describing sets of complex heap graphs. We have also proposed a way how to efficiently integrate the learning with the main analysis algorithm. Finally, we have proposed a significant improvement—both in terms of generality as well as efficiency—of the abstraction used in the framework. An implementation of the approach in the Forester tool allowed us to fully-automatically handle programs over quite complex heap structures, including 2-level and 3-level skip lists, which—to the best of our knowledge—no other fully-automated verification tool can handle. At the same time, the efficiency of the analysis is comparable with other state-of-the-art analysers even though they handle less general classes of heap structures.

For the future, there are many possible ways how the presented approach can be further extended. First, one can think of using recursive boxes or forest automata using hedge automata as their components in order to handle even more complex data structures (such as mcf trees). Another interesting direction is that of integrating FA-based heap analysis with some analyses for dealing with infinite non-pointer data domains (e.g., integers) or parallelism.

Acknowledgement. This work was supported by the Czech Science Foundation (projects P103/10/0306, 13-37876P), the Czech Ministry of Education, Youth, and Sports (project MSM 0021630528), the BUT FIT project FIT-S-12-1, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

References

1. J. Berdine, B. Cook, and S. Ishtiaq. Memory Safety for Systems-level Code. In *Proc. of CAV'11*, *LNCS* 6806, Springer, 2011.
2. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular (Tree) Model Checking. *STTT* 14(2), Springer, 2012.
3. B.-Y.E. Chang, X. Rival, and G.C. Necula. Shape Analysis with Structural Invariant Checkers. In *Proc. of SAS'07*, *LNCS* 4634, Springer, 2007.
4. K. Dudka, P. Peringer, and T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In *Proc. of CAV'11*, *LNCS* 6806, 2011.
5. B. Guo, N. Vachharajani, and D.I. August. Shape Analysis with Inductive Recursion Synthesis. In Proc. of PLDI'07, ACM Press, 2007.
6. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV'11*, LNCS 6806, Springer, 2011.
7. L. Holík, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. Fully Automated Shape Analysis Based on Forest Automata. Tech. rep. FIT-TR-2013-01, FIT BUT, 2013.
8. J. Heinen, T. Noll, and S. Rieger. Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. ENTCS 266, Elsevier, 2010.
9. O. Lee, H. Yang, and R. Petersen. Program Analysis for Overlaid Data Structures. In *Proc. of CAV'11*, LNCS 6806, Springer, 2011.
10. S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic Numeric Abstractions for Heap-manipulating programs. In *Proc. of POPL'10*, ACM Press, 2010.
11. A.D. Weinert. Inferring Heap Abstraction Grammars. BSc thesis, RWTH Aachen, 2012.
12. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, *LNCS* 5123, Springer, 2008.
13. W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. Commun. ACM, 33(6): 668–676, ACM, 1990.