Forest Automata for Verification of Heap Manipulation[*]

Peter Habermehl[1], Lukáš Holík[2,4], Adam Rogalewicz[2],
Jiří Šimáček[2,3], and Tomáš Vojnar[2]

[1] LIAFA, Université Paris Diderot—Paris 7/CNRS, France
[2] FIT, Brno University of Technology, Czech Republic
[3] VERIMAG, UJF/CNRS/INPG, Gières, France
[4] Uppsala University, Sweden

Abstract. We consider verification of programs manipulating dynamic linked data structures such as various forms of singly and doubly-linked lists or trees. We consider important properties for this kind of systems like no null-pointer dereferences, absence of garbage, shape properties, etc. We develop a verification method based on a novel use of tree automata to represent heap configurations. A heap is split into several "separated" parts such that each of them can be represented by a tree automaton. The automata can refer to each other allowing the different parts of the heaps to mutually refer to their boundaries. Moreover, we allow for a hierarchical representation of heaps by allowing alphabets of the tree automata to contain other, nested tree automata. Program instructions can be easily encoded as operations on our representation structure. This allows verification of programs based on a symbolic state-space exploration together with refinable abstraction within the so-called abstract regular tree model checking. A motivation for the approach is to combine advantages of automata-based approaches (higher generality and flexibility of the abstraction) with some advantages of separation-logic-based approaches (efficiency). We have implemented our approach and tested it successfully on multiple non-trivial case studies.

1 Introduction

We address verification of sequential programs with complex *dynamic linked data structures* such as various forms of singly- and doubly-linked lists (SLL/DLL), possibly cyclic, shared, hierarchical, and/or having different additional (head, tail, data, and the like) pointers, as well as various forms of trees. We in particular consider C pointer manipulation, but our approach can easily be applied to any other similar language. We concentrate on *safety properties* of the considered programs which includes generic properties like absence of null dereferences, double free operations, dealing with dangling pointers, or memory leakage. Furthermore, to check various shape properties of the involved data structures one can use testers, i.e., parts of code which, in case some desired property is broken, lead the control flow to a designated error location.

[*] This work was supported by the Czech Science Foundation (projects P103/10/0306, P201/09/P531, and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the Czech-French Barrande project 021023, the BUT FIT project FIT-S-11-1, and the French ANR-09-SEGI project Veridyc .

G. Gopalakrishnan and S. Qadeer (Eds.): CAV 2011, LNCS 6806, pp. 424–440, 2011.

For the above purpose, we propose a novel approach of representing sets of heaps via *tree automata* (TA). In our representation, a heap is split in a canonical way into several *tree components* whose roots are the so-called *cut-points*. Cut-points are nodes pointed to by program variables or having several incoming edges. The tree components can refer to the roots of each other, and hence they are "separated" much like heaps described by formulae joined by the separating conjunction in separation logic [15]. Using this decomposition, sets of heaps with a bounded number of cut-points are then represented by the so called *forest automata* (FA) that are basically tuples of TA accepting tuples of trees whose leaves can refer back to the roots of the trees. Moreover, we allow alphabets of FA to contain *nested FA*, leading to a *hierarchical encoding of heaps*, allowing us to represent even sets of heaps with an unbounded number of cut-points (e.g., sets of DLL). Intuitively, a nested FA can describe a part of a heap with a bounded number of cut-points (e.g., a DLL segment), and by using such an automaton as an alphabet symbol an unbounded number of times, heaps with an unbounded number of cut-points are described. Finally, since FA are not closed under union, we work with sets of forest automata, which are an analogy of disjunctive separation logic formulae.

As a nice theoretical feature of our representation, we show that *inclusion* of sets of heaps represented by finite sets of non-nested FA (i.e., having a bounded number of cut-points) is decidable. This covers sets of complex structures like SLL with head/tail pointers. Moreover, we show how inclusion can be safely approximated for the case of nested FA. Further, C program statements manipulating pointers can be easily encoded as operations modifying FA. Consequently, the symbolic verification framework of *abstract regular tree model checking* [6,7], which comes with automatically refinable abstractions, can be applied.

The proposed approach brings the principle of *local heap manipulation* (i.e., dealing with separated parts of heaps) from separation logic into the world of automata. The motivation is to combine some advantages of using automata and separation logic. Automata provide higher generality and flexibility of the abstraction (see also below) and allow us to leverage the recent advances of efficient use of non-deterministic automata [2,3]. As further discussed below, the use of separation allows for a further increase in efficiency compared to a monolithic automata-based encoding proposed in [7].

We have implemented our approach in a prototype tool called *Forester* as a `gcc` plug-in. In our current implementation, if nested FA are used, they are provided manually (similar to the use of pre-defined inductive predicates common in works on separation logic). However, we show that Forester can already successfully handle multiple interesting case studies, proving the proposed approach to be very promising.

Related work. The area of verifying programs with dynamic linked data structures has been a subject of intense research for quite some time. Many different approaches based on logics, e.g., [13,16,15,4,10,14,19,18,8,12], automata [7,5,9], upward closed sets [1], and other formalisms have been proposed. These approaches differ in their generality, efficiency, and degree of automation. Due to space restrictions, we cannot discuss all of them here. Therefore, we concentrate on a comparison with the two closest lines of work, namely, the use of automata as described in [7] and the use of separation logic in the works [4,18] linked with the Space Invader tool. In fact, as is clear from the above, the approach we propose combines some features from these two lines of research.

Compared to [4,18], our approach is more general in that it allows one to deal with tree-like structures, too. We note that there are other works on separation logic, e.g., [14], that consider tree manipulation, but these are usually semi-automated only. An exception is [10] which automatically handles even tree structures, but its mechanism of synthesising inductive predicates seems quite dependent on the fact that the dynamic linked 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[1]).

Further, compared to [4,18], our approach comes with a more flexible abstraction. We are not building on just using some inductive predicates, but we combine a use of our nested FA with an automatically refinable abstraction on the TA that appear in our representation. Thus our analysis can more easily adjust to various cases arising in the programs being verified. An example is dealing with lists of lists where the sublists are of length 0 or 1, which is a quite practical situation [17]. In such cases, the abstraction used in [4,18] can fail, leading to an infinite computation (e.g., when, by chance, a list of regularly interleaved lists of length 0 or 1 appears) or generate false alarms (when modified to abstract even pointer links of length 1 to a list segment). For us, such a situation is easy to handle without any need to fine-tune the abstraction manually.

On the other hand, compared with the approach of [7], our newly proposed approach is a bit less general (we cannot, e.g., handle structures such as trees with linked leaves[2]), but on the other hand more scalable. The latter comes from the fact that the representation in [7] is monolithic, i.e., the whole heap is represented by one tree-like structure whereas our new representation is not monolithic anymore. Therefore, the different operations on the heap, e.g., corresponding to a symbolic execution of the verified program, influence only small parts of the encoding (unlike in [7], where the transducers used for this purpose are always operating on the entire automata). Also, the monolithic encoding of [7], based on a fixed tree skeleton over which additional pointer links were expressed using the so-called routing expressions, had problems with deletion of elements inside data structures and with detection of memory leakage (which was in theory possible, but it was so complex that it was never implemented).

2 From Heaps to Forests

In this section, we outline how sets of heaps can be represented by hierarchical forest automata. These automata are tuples of tree automata which accept trees that may refer to each other through the alphabet symbols. Furthermore their alphabet can contain strictly hierarchically nested forest automata. For the purpose of the explanation, *heaps* may be viewed as oriented graphs whose nodes correspond to allocated memory cells and edges to pointer links between these cells. The nodes may be labelled by non-pointer data stored in them (assumed to be from a finite data domain) and by program variables pointing to the nodes. Edges may be labelled by the corresponding selectors.

In what follows, we are representing sets of *garbage free* heaps only, i.e., all memory cells are reachable from pointer variables by following pointer links. However,

[1] We did not find an available implementation of [10], and so we could not try it out ourselves.

[2] Unless a generalisation to FA nested not just strictly hierarchically, but in an arbitrary, possibly cyclic way is considered, which is an interesting subject for future research.

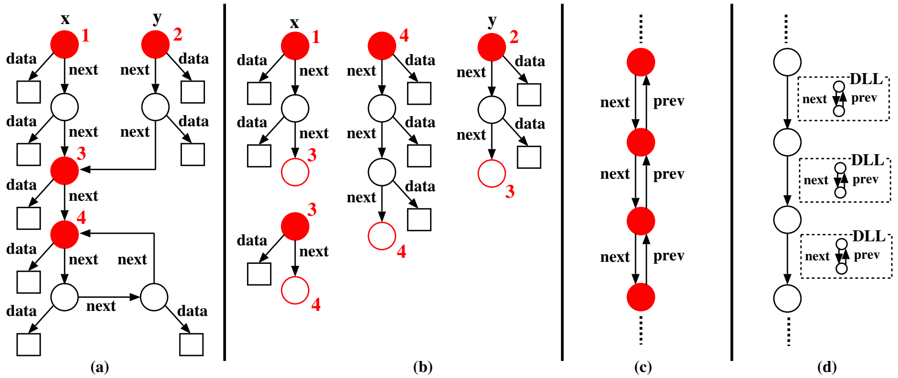

Fig. 1. (a) A heap graph with cut-points highlighted in red, (b) the canonical tree decomposition of the heap with x ordered before y, (c) a part of a DLL, (d) a hierarchical encoding of the DLL

practically this is not a restriction since the emergence of garbage can be checked for each program statement to be fired and if garbage arises, an error message can be issued and the computation stopped or the garbage removed and the computation continued.

Now, note that each heap graph may be *canonically decomposed* into a tuple of trees as follows. We first identify the *cut-points*, i.e. nodes that are either pointed to by a program variable or that have several incoming edges. Then, we totally order program variables and selectors. Next, cut-points are canonically numbered using a depth-first traversal of the heap graph starting from nodes pointed to by program variables, taking them in accordance with their order, and exploring the graph according to the order of selectors. Finally, we split the heap graph into tree components rooted at particular cut-points. These components contain all the nodes reachable from their root while not passing through any cut-point, plus a copy of each reachable cut-point, labelled by its number. The tree components are then canonically ordered according to the numbers of their root cut-points. For an illustration of the decomposition, see Figure 1 (a) and (b).

Now, tuples of tree automata (TA), called *forest automata* (FA), accepting tuples of trees whose leaves may refer to the root of any tree out of a given tuple, may be viewed as representing a set of heaps as follows. We simply take a tree from the language of each of the TA and obtain a heap by gluing the tree roots corresponding to cut-points with the leaves referring to them.

Further, we consider in particular *canonicity respecting forest automata* (CFA). CFA encode sets of heaps decomposed in a canonical way, i.e., such that if we take any tuple of trees accepted by the given CFA, construct a heap from them, and then canonically decompose it, we get the tuple of trees we started with. This means that in the chosen tuple there is no tree with a root that does not correspond to a cut-point and that the trees are ordered according to the depth-first traversal as described above. The canonicity respecting form allows us to test inclusion on the sets of heaps represented by CFA by component-wise testing inclusion on the languages of the TA constituting the given CFA.

Note, however, that FA are not closed under union. Clearly, even if we consider FA having the same number of components, uniting the TA component-wise may yield an FA overapproximating the union of the sets of heaps represented by the original FA (cf.

Section 3). Hence, we will have to represent unions of FA explicitly as *sets of FA* (SFA), which is similar to dealing with disjunctions of separation logic formulae. However, as we shall see, inclusion on the sets of heaps represented by SFA is still easily decidable.

The described encoding allows one to represent sets of heaps with a bounded number of cut-points. However, to handle many common dynamic data structures one needs to represent sets of heaps with an *unbounded number of cut-points*. Indeed, in doubly-linked lists (DLLs) for instance, every node is a cut-point. We solve this problem by representing heaps in a *hierarchical way*. In particular, we collect sets of repeated subgraphs (called *components*) in the so-called *boxes*. Every occurrence of such components can then be replaced by a single hyperedge labelled by the appropriate box[3]. In this way, a set of heap graphs with an unbounded number of cut-points can be transformed into a set of *hierarchical heap hypergraphs* with a bounded number of cut-points at each level of the hierarchy. Figures 1 (c) and (d) illustrate how this approach can reduce DLLs into singly-linked lists (with a DLL segment used as a kind of meta-selector). Sets of heap hypergraphs corresponding either to the top level of the representation or to boxes of different levels can then be decomposed into (hyper)tree components and represented using FA whose alphabet can contain nested FA.[4] Intuitively, FA that appear in the alphabet of some superior FA play a role similar (but not equal) to that of inductive predicates in separation logic.[5]

The question of deciding inclusion on sets of heaps represented by hierarchical FA remains open. However, we propose a *canonical decomposition of hierarchical hypergraphs* allowing inclusion to be decided for sets of heap hypergraphs represented by FA in the case when the nested FA labelling hyperedges are taken as atomic alphabet symbols. Note that this decomposition is by far not the same as for non-hierarchical heap graphs due to a need to deal with nodes that are not reachable on the top level, but are reachable through edges hidden in some boxes. This result allows one to safely approximate inclusion checking on hierarchically represented heaps, which appears to work quite well in practice.

3 Hypergraphs and Their Representation

We now formalise the notion of hypergraphs and forest automata.

3.1 Hypergraphs

Given a set A and $n \in \mathbb{N}$, let A^n denote the n^{th}-Cartesian power of A and let $A^{\leq n} = \bigcup_{0 \leq i \leq n} A^i$. For an n-tuple $\overline{a} = (a_1, \ldots, a_n) \in A^n$, $n \geq 1$, we let $\overline{a}.i = a_i$ for any $1 \leq i \leq n$.

[3] We may obtain hyperedges here since we allow components to have a single designated input node, but possibly several output nodes.

[4] Since graphs are a special case of hypergraphs, in the following, we will work with hypergraphs only. Moreover, to simplify the definitions, we will work with hyperedge-labelled hypergraphs only. Node labels mentioned above will be put at specially introduced nullary hyperedges leaving from the nodes whose label is to be represented.

[5] For instance, we use a nested FA encoding a DLL segment of length 1, not of length 1 or more as in separation logic: the repetition of the segment is encoded in the structure of the top-level FA.

We call a set A *ranked* if there is a function $\# : A \to \mathbb{N}$. The value $\#(a)$ is called the *rank* of $a \in A$. We call $\#(A) = max(\{\#(a) \mid a \in A\})$ the maximum rank of an element in the given set. For any $n \geq 0$, we denote by A_n the set of all elements of rank n from A.

Given a finite ranked set Γ called a hyperedge alphabet, a Γ-labelled oriented *hypergraph* with designated input and output ports—denoted simply as a hypergraph if no confusion may arise—is a tuple $G = (V, E, I, O)$ where V is a finite set of vertices, $E \subseteq V \times \Gamma \times V^{\leq \#(\Gamma)}$ is a set of hyperedges such that $\forall (v, a, \overline{v}) \in E : \ \overline{v} \in V^{\#(a)}$, and $I, O \subseteq V$ are sets of input and output ports, respectively[6]. We assume that there is a total ordering $\preceq_p \subseteq P \times P$ on the set $P = I \cup O$ of all ports of G. The sets I, O of input/output ports may be empty in which case we may drop them from the hypergraph. For symbols $a \in \Gamma$ with $\#(a) = 0$, we write $(v, a) \in E$ to denote that $(v, a, ()) \in E$.

Given a hyperedge $e = (v, a, (v_1, \ldots, v_n)) \in E$ of a hypergraph $G = (V, E, I, O)$, v is the *source* of e and $v_1, \ldots, v_n$ are *a-successors* of v in G. An (oriented) *path* in G is a sequence $\langle v_0, a_1, v_1, \ldots, a_n, v_n \rangle$, $n \geq 0$, where for all $1 \leq i \leq n$, v_i is an a_i-successor of v_{i-1} in G. G is called *deterministic* iff $\forall (v, a, \overline{v}), (v, a', \overline{v}') \in E$: $a = a' \implies \overline{v} = \overline{v}'$. A hypergraph G is *well-connected* if each node $v \in V$ is reachable through some path from some input port of G. Figure 1 (a) shows a (hyper)graph with two input ports corresponding to the two variables. Edges are labelled by selectors `data` and `next`.

3.2 A Forest Representation of Hypergraphs

A Γ-labelled hypergraph $T = (V, E)$ without input and output ports is an unordered, oriented Γ-labelled *tree* (denoted simply as a tree below) iff (1) it has a single node with no incoming hyperedge (called the *root* of T, denoted $root(T)$), (2) all other nodes of T are reachable from $root(T)$ via some path, and (3) each node has at most one incoming hyperedge. Nodes with no successors are called *leaves*.

Given a finite ranked hyperedge alphabet Γ such that $\Gamma \cap \mathbb{N} = \emptyset$, we call a tuple $F = (T_1, \ldots, T_n, I, O)$, $n \geq 1$, an ordered Γ-labelled *forest* with designated input and output ports (or just a forest) iff (1) for every $i \in \{1, \ldots, n\}$, $T_i = (V_i, E_i)$ is a $\Gamma \cup \{1, \ldots, n\}$-labelled tree where $\forall i \in \{1, \ldots, n\}$, $\#(i) = 0$ and a vertex v with $(v, i) \in E$ is not a source of any other edge (hence it is a leaf), (2) $\forall 1 \leq i_1 < i_2 \leq n : \ V_{i_1} \cap V_{i_2} = \emptyset$, and (3) $I, O \subseteq \{1, \ldots, n\}$ denote the input and output ports, respectively.

We call the sources of edges labelled by $\{1, \ldots, n\}$ *root references* and denote by $rr(T_i)$ the set of all root references in T_i, i.e., $rr(T_i) = \{v \in V_i \mid (v, k) \in E_i, k \in \{1, \ldots, n\}\}$ for each $i \in \{1, \ldots, n\}$. A forest $F = (T_1, \ldots, T_n, I_F, O_F)$, $n \geq 1$, *represents* the hypergraph $\otimes F$ that is obtained by first uniting the trees $T_1, \ldots, T_n$ and then removing every root reference $v \in V_i$, $1 \leq i \leq n$, and redirecting the hyperedges leading to v to the root of T_k where $(v, k) \in E_i$. Formally, $\otimes F = (V, E, I, O)$ where:

- $V = \bigcup_{i=1}^{n} V_i \setminus rr(T_i)$, $E = \bigcup_{i=1}^{n} \{(v, a, \overline{v}') \mid a \in \Gamma \wedge \exists (v, a, \overline{v}) \in E_i \ \forall 1 \leq j \leq \#(a) :$ if $\exists (\overline{v}.j, k) \in E_i$ with $k \in \{1, \ldots, n\}$, then $\overline{v}'.j = root(T_k)$, else $\overline{v}'.j = \overline{v}.j\}$,
- $I = \{root(T_i) \mid i \in I_F\}$, $O = \{root(T_i) \mid i \in O_F\}$,
- the ordering of the set of ports $P = I \cup O$ is defined by : $\forall i, j \in (I_F \cup O_F) : \ root(T_i) \preceq_p root(T_j) \iff i \leq j$.

[6] Intuitively, in hypergraphs representing heaps, input ports correspond to nodes pointed to by program variables or to input nodes of components, and output ports correspond to output nodes of components.

Figure 1 (b) shows a forest decomposition of the graph of Figure 1 (a). It is decomposed into four trees which have designated roots which are referred to in the trees. The decomposition respects the ordering of the two ports corresponding to the variables.

3.3 Minimal and Canonical Forests

We call a forest $F = (T_1,\ldots,T_n,I_F,O_F)$ representing the well-connected hypergraph $G = (V,E,I,O) = \otimes F$ *minimal* iff the roots of the trees $T_1,\ldots,T_n$ correspond to the *cut-points* of G which are those nodes that are either ports or that have more than one incoming hyperedge in G. A minimal forest representation of a hypergraph is unique up to permutations of $T_1,\ldots,T_n$. In order to get a canonical forest representation of a well-connected *deterministic* hypergraph $G = (V,E,I,O)$, we need to canonically order the trees in its minimal forest representation. We do this as follows: First, we assume the set of hyperedge labels Γ to be totally ordered via some ordering $\preceq_\Gamma$. Then, a depth-first traversal (DFT) on G is performed starting with the DFT stack containing the set $I \cup O$ in the given order $\preceq_p$, the smallest node being on top of the stack. We now call a forest representation $F = (T_1,\ldots,T_n,I_F,O_F)$ of G *canonical* iff it is minimal and the trees $T_1,\ldots,T_n$ appear in F in the following order: First, the trees whose roots correspond to ports appear in the order given by $\preceq_p$, and then the rest of the trees appears in the same order in which their roots are visited in the described DFT of G. A canonical representation is obtained this way since we consider G to be deterministic. Clearly the forest of Figure 1 (b) is a canonical representation of the graph of Figure 1 (a).

3.4 Forest Automata

We now define forest automata as tuples of tree automata encoding sets of forests and hence sets of hypergraphs. To be able to use classical tree automata, we will need to work with trees that are ordered, node-labelled, with the node labels being ranked.

Ordered Trees. Let ε denote the empty sequence. An *ordered tree* t over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ satisfying the following conditions: (1) $dom(t)$ is a finite, prefix-closed subset of $\mathbb{N}^*$, and (2) for each $p \in dom(t)$, if $\#(t(p)) = n \geq 0$, then $\{i \mid pi \in dom(t)\} = \{1,\ldots,n\}$. Each sequence $p \in dom(t)$ is called a *node* of t. For a node p, the i^{th} *child* of p is the node pi, and the i^{th} *subtree* of p is the tree t' such that $t'(p') = t(pip')$ for all $p' \in \mathbb{N}^*$. A *leaf* of t is a node p with no children, i.e., there is no $i \in \mathbb{N}$ with $pi \in dom(t)$. Let $\mathbb{T}(\Sigma)$ be the set of all ordered trees over Σ.

For an $\preceq_\Gamma$-ordered hyperedge alphabet Γ, it is easy to convert Γ-labelled trees into node-labelled ordered trees and back (up to isomorphism). We label a node of an ordered tree by the set of labels of the hyperedges leading from the corresponding node in the original tree, and we order the successors of the node w.r.t. the hyperedge labels through which they are reachable (while always keeping tuples of nodes reachable via the same hyperedge together). The rank of the new node label is then given by the sum of the original hyperedge labels embedded into it. Below, we use the notion Σ_Γ to denote the ranked node alphabet obtained from Γ as described above (w.r.t. a total ordering $\preceq_\Gamma$ that we will from now on assume to be always associated with Γ) and $ot(T)$ to denote the ordered tree obtained from a Γ-labelled tree T. For a formal description, see [11].

Tree Automata. A (finite, non-deterministic, bottom-up) *tree automaton* (abbreviated as TA in the following) is a quadruple $\mathcal{A} = (Q, \Sigma, \Delta, F)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is a ranked alphabet, and Δ is a set of transition rules. Each transition rule is a triple of the form $((q_1, \ldots, q_n), f, q)$ where $n \geq 0$, $q_1, \ldots, q_n, q \in Q$, $f \in \Sigma$, and $\#(f) = n$. We use $(q_1, \ldots, q_n) \xrightarrow{f} q$ to denote that $((q_1, \ldots, q_n), f, q) \in \Delta$. In the special case where $n = 0$, we speak about the so called *leaf rules*, which we sometimes abbreviate as $\xrightarrow{f} q$.

A *run* of $\mathcal{A}$ over a tree $t \in \mathbb{T}(\Sigma)$ is a mapping $\pi : dom(t) \to Q$ such that, for each node $p \in dom(t)$ where $q = \pi(p)$, if $q_i = \pi(pi)$ for $1 \leq i \leq n$, then Δ has a rule $(q_1, \ldots, q_n) \xrightarrow{t(p)} q$. We write $t \overset{\pi}{\Longrightarrow} q$ to denote that π is a run of $\mathcal{A}$ over t such that $\pi(\varepsilon) = q$. We use $t \Longrightarrow q$ to denote that $t \overset{\pi}{\Longrightarrow} q$ for some run π. The *language* of a state q is defined by $\mathcal{L}(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of $\mathcal{A}$ is defined by $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in F} \mathcal{L}(q)$.

Forest Automata. Let Γ be a ranked hyperedge alphabet ordered by $\preceq_\Gamma$. We call an n-tuple $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, I, O)$, $n \geq 1$, a *forest automaton* with designated input/output ports (called also FA) over Γ iff for all $1 \leq i \leq n$, $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$ is a TA with $\Sigma = \Sigma_{\Gamma \cup \{1,\ldots,n\}}$ where $\forall 1 \leq i \leq n : \#(i) = 0$. The sets $I, O \subseteq \{1, \ldots, n\}$ are sets of input/output ports, respectively. $\mathcal{F}$ defines the *forest language* $\mathcal{L}_F(\mathcal{F}) = \{(T_1, \ldots, T_n, I, O) \mid (\forall 1 \leq i \leq n : ot(T_i) \in \mathcal{L}(\mathcal{A}_i)) \wedge (\forall 1 \leq i < j \leq n : T_i = (V_i, E_i) \wedge T_j = (V_j, E_j) \Longrightarrow V_i \cap V_j = \emptyset)\}$. The *hypergraph language* of $\mathcal{F}$ is then the set $\mathcal{L}(\mathcal{F}) = \{\otimes F \mid F \in \mathcal{L}_F(\mathcal{F})\}$. An FA $\mathcal{F}$ *respects canonicity* iff each forest $F \in \mathcal{L}_F(\mathcal{F})$ is a canonical representation of some well-connected hypergraph, namely, the hypergraph $G = \otimes F$. We abbreviate canonicity respecting FA as CFA. It is easy to see that comparing sets of hypergraphs represented by CFA can be done *component-wise* as described in the below lemma.

Lemma 1. *Let $\mathcal{F}_1 = (\mathcal{A}^1_1, \ldots, \mathcal{A}^1_{n_1}, I_1, O_1)$ and $\mathcal{F}_2 = (\mathcal{A}^2_1, \ldots, \mathcal{A}^2_{n_2}, I_2, O_2)$ be two CFA. Then, $\mathcal{L}(\mathcal{F}_1) \subseteq \mathcal{L}(\mathcal{F}_2)$ iff (1) $n_1 = n_2$, (2) $I_1 = I_2$, (3) $O_1 = O_2$, and (4) $\forall 1 \leq i \leq n : \mathcal{L}(\mathcal{A}^1_i) \subseteq \mathcal{L}(\mathcal{A}^2_i)$.*

Sets of Forest Automata. The class of languages of forest automata is not closed under union. The reason is that a forest language of an FA is the Cartesian product of the languages of all its components and that not every union of Cartesian products may be expressed as a single Cartesian product. For instance, consider two CFA $\mathcal{F} = (\mathcal{A}, \mathcal{B}, I, O)$ and $\mathcal{F}' = (\mathcal{A}', \mathcal{B}', I, O)$ such that $\mathcal{L}_F(\mathcal{F}) = \{(a, b, I, O)\}$ and $\mathcal{L}_F(\mathcal{F}') = \{(c, d, I, O)\}$ where a, b, c, d are distinct trees. The forest language of the FA $(\mathcal{A} \cup \mathcal{A}', \mathcal{B} \cup \mathcal{B}', I, O)$ is $\{(x, y, I, O) \mid (x, y) \in \{a, c\} \times \{b, d\}\})$ and thus there is no CFA with the hypergraph language equal to $\mathcal{L}(\mathcal{F}) \cup \mathcal{L}(\mathcal{F}')$. Therefore, we will work with *finite sets of (canonicity-respecting) forest automata*, S(C)FA for short, where the language $\mathcal{L}(S)$ of a finite set S of FA is defined as the union of the languages of its elements.

Note that any FA can be transformed (split) into an SCFA whose CFA represent hypergraphs having a different interconnection of the cut-points (see [11] for details).

Testing Inclusion on SFA. The problem of checking inclusion on SFA, this is, checking whether $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ where S, S' are SFA, can be reduced to a problem of checking inclusion on tree automata. We may w.l.o.g. assume that S and S' are SCFA.

For an FA $\mathcal{F} = (\mathcal{A}_1, \dots, \mathcal{A}_n, I, O)$ where $\mathcal{A}_i = (\Sigma, Q_i, \Delta_i, F_i)$ for each $1 \leq i \leq n$, we define the TA $\mathcal{A}^{\mathcal{F}} = (\Sigma \cup \{\curlywedge_n^{I,O}\}, Q, \Delta, \{q^{top}\})$ where $\curlywedge_n^{I,O} \notin \Sigma$ is a symbol with $\#(\curlywedge_n^{I,O}) = n$, $q^{top} \notin \bigcup_{i=1}^{n} Q_i$, $Q = \bigcup_{i=1}^{n} Q_i \cup \{q^{top}\}$, and $\Delta = \bigcup_{i=1}^{n} \Delta_i \cup \Delta^{top}$. The set Δ^{top} contains the rule $\curlywedge_n^{I,O}(q_1, \dots, q_n) \rightarrow q^{top}$ for each $(q_1, \dots, q_n) \in F_1 \times \cdots \times F_n$. Intuitively, $\mathcal{A}^{\mathcal{F}}$ accepts the trees where n-tuples of ordered trees representing hypergraphs from $\mathcal{L}(\mathcal{A})$ are topped by a designated root node labelled by $\curlywedge_n^{I,O}$. It is now easy to see that the following lemma holds (in the lemma, "$\cup$" stands for the usual tree automata union).

Lemma 2. *For two SCFA S and S', $\mathcal{L}(S) \subseteq \mathcal{L}(S') \iff \mathcal{L}(\bigcup_{\mathcal{F} \in S} \mathcal{A}^{\mathcal{F}}) \subseteq \mathcal{L}(\bigcup_{\mathcal{F}' \in S'} \mathcal{A}^{\mathcal{F}'})$.*

4 Hierarchical Hypergraphs

We inductively define hierarchical hypergraphs as hypergraphs with hyperedges possibly labelled by hierarchical hypergraphs of a lower level. Let Γ be a ranked alphabet.

4.1 Hierarchical Hypergraphs, Components, and Boxes

A Γ-labelled (hierarchical) *hypergraph of level* 0 is any Γ-labelled hypergraph. For $j \in \mathbb{N}$, a *hypergraph of level* $j+1$ is defined as a hypergraph over the alphabet $\Gamma \cup \mathbb{B}_j$.

To define the set $\mathbb{B}_j$, we first define a Γ-labelled *component of level* j as a hypergraph $C = (V, E, I, O)$ of level j which satisfies the requirement that $|I| = 1$ and $I \cap O = \emptyset$.

Then, $\mathbb{B}_j$ is the set of Γ-labelled *boxes* of level j where each box $B \in \mathbb{B}_j$ is a set of Γ-labelled components of level j which all have the same number of output ports. We call this number the *rank* of B, require that $\Gamma \cap \mathbb{B}_j = \emptyset$ and call boxes over Γ that appear as labels of hyperedges of a hierarchical hypergraph H over Γ *nested boxes* of H.

Semantics of hierarchical hypergraphs and boxes. We are going to define the semantics of a hierarchical hypergraph H as a set of hypergraphs $[\![H]\!]$. If H is of level 0, then $[\![H]\!] = \{H\}$. The semantics of a box B, denoted $[\![B]\!]$, is the union of semantics of its elements (i.e., it is a set of components of level 0). In the semantics of a hypergraph $H = (V, E, I, O)$ of level $j > 0$, each hyperedge labelled by a box $B \in \mathbb{B}_{j-1}$ is substituted in all possible ways by components from the semantics of B (as in ordinary hyperedge replacement used in graph grammars). To define this formally, we use an auxiliary operation *plug*. Let $e = (v, a, \overline{v}) \in E$ be a hyperedge with $\#(a) = k$ and let $C = (V', E', I', O')$ be a component of level $j-1$ to be plugged into H instead of e. Let $(o_1, \dots, o_k)$ be the set O' ordered according to $\preceq_P$. W.l.o.g., assume $V \cap V' = \emptyset$. For any $w \in V'$, we define an auxiliary port matching function $\rho(w)$ such that (1) if $w \in I'$, $\rho(w) = v$, (2) if $w = o_i, 1 \leq i \leq k$, $\rho(w) = \overline{v}.i$, and (3) $\rho(w) = w$ otherwise. We define $plug(H, e, C) = (V'', E'', I, O)$ by setting $V'' = V \cup (V' \setminus (I' \cup O'))$ and $E'' = (E \setminus \{e\}) \cup \{(v'', a', \overline{v}'') \mid \exists (v', a', \overline{v}') \in E' : \rho(v') = v'' \wedge \forall 1 \leq i \leq k : \rho(\overline{v}'.i) = \overline{v}''.i\}$. Now, the semantics of a hypergraph $H = (V, E, I, O)$ of level j is defined recursively as follows: Let $Plug(H) = \{plug(H, e, C) \mid e = (v, B, \overline{v}) \in E \wedge B \in \mathbb{B}_{j-1} \wedge C \in [\![B]\!]\}$. If $Plug(H) = \emptyset$, $[\![H]\!] = \{H\}$, otherwise $[\![H]\!] = \bigcup_{H' \in Plug(H)} [\![H']\!]$. Figure 1 (d) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)graph of Figure 1 (c) obtained using *Plug*. The only box used represents a DLL segment.

4.2 Hierarchical Forest Automata

To represent sets of deterministic hierarchical hypergraphs, we propose to use (hierarchical) FA whose alphabet contains SFA representing the needed nested boxes. For a hierarchical FA $\mathcal{F}$, we will denote by $\mathcal{L}_H(\mathcal{F})$ the set of hierarchical hypergraphs represented by it. Likewise, for a hierarchical SFA S, we let $\mathcal{L}_H(S) = \bigcup_{\mathcal{F} \in S} \mathcal{L}_H(\mathcal{F})$.

Let Γ be a *finite* ranked alphabet. Formally, an FA $\mathcal{F}$ over Γ of level 0 is an ordinary FA over Γ, and we let $\mathcal{L}_H(\mathcal{F}) = \mathcal{L}(\mathcal{F})$. For $j \in \mathbb{N}$, $\mathcal{F}$ is an FA over Γ of level $j+1$ iff $\mathcal{F}$ is an ordinary FA over an alphabet $\Gamma \cup X$ where X is a finite set of SFA of level j (called *nested SFA* of $\mathcal{F}$) such that for every $S \in X$, $\mathcal{L}_H(S)$ is a box over Γ of level j. The rank $\#(S)$ of S equals the rank of the box $\mathcal{L}_H(S)$.

For FA of level $j+1$, $\mathcal{L}_H(\mathcal{F})$ is defined as the set of hierarchical hypergraphs that arise from the hypergraphs in $\mathcal{L}(\mathcal{F})$ by replacing SFA on their edges by the boxes they represent. Formally, $\mathcal{L}_H(\mathcal{F})$ is the set of hypergraphs of level $j+1$ such that $(V,E,I,O) \in \mathcal{L}_H(\mathcal{F})$ iff there is a hypergraph $(V,E',I,O) \in \mathcal{L}(\mathcal{F})$ where $E = \{(v,a,\bar{v}) \mid (v,a,\bar{v}) \in E' \wedge a \in \Gamma\} \cup \{(v,\mathcal{L}_H(S),\bar{v}) \mid (v,S,\bar{v}) \in E' \wedge S \in X\}$.

Notice that a hierarchical SFA of any level has finitely many nested SFA of a lower level only, and the number of levels if finite. Therefore, a hierarchical SFA is a finitely representable object. Notice also that even though the maximum number of cut-points of hypergraphs from $\mathcal{L}_H(S)$ is fixed (SFA always accept hypergraphs with a fixed maximum number of cut-points), the number of cut-points of hypergraphs in $[\![\mathcal{L}_H(S)]\!]$ may be unbounded. The reason is that hypergraphs from $\mathcal{L}_H(S)$ may contain an unbounded number of hyperedges labelled by boxes B such that hypergraphs from $[\![B]\!]$ contain cut-points too. These cut-points then appear in hypergraphs from $[\![\mathcal{L}_H(S)]\!]$, but they are not visible at the level of hypergraphs from $\mathcal{L}_H(S)$.

Hierarchical SFA are therefore finite representations of sets of hypergraphs with possibly unbounded numbers of cut-points.

4.3 Inclusion and Well-Connectedness on Hierarchical SFA

In this section, we aim at checking well-connectedness and inclusion of sets of hypergraphs represented by hierarchical FA. Since considering the full class of hierarchical hypergraphs would unnecessarily complicate our task, we introduce restrictions of hierarchical automata that rule out some rather artificial scenarios and that allow us to handle the automata hierarchically (i.e., using some pre-computed information for nested FA rather than having to unfold the entire hierarchy all the time). In particular, we enforce that for a hierarchical hypergraph H, well-connectedness of hypergraphs in $[\![H]\!]$ is equivalent to the so-called box-connectedness of H introduced below, and, further, determinism of graphs from $[\![H]\!]$ is equivalent to determinism of H.[7]

Proper boxes and well-formed hypergraphs. Given a component C of level 0 over Γ, we define its *backward reachability* set $br(C)$ as the set of indices i for which there is

[7] Notice that for a general hierarchical hypergraph H, well-connectedness of H is nor implied neither implies well-connectedness of hypergraphs from $[\![H]\!]$. This holds also for determinism. The reason is that a component C in a nested box of H may interconnect its ports in an arbitrary way. It may contain paths from output ports to both input and output ports, but it may be missing paths from the input port to some of the output ports.

a path from the i-th output port of C *back* to the input port of C. Given a box B over Γ, we inductively define B to be *proper* iff all its nested boxes are proper, $br(C_1) = br(C_2)$ for any $C_1, C_2 \in [\![B]\!]$ (we use $br(B)$ to denote $br(C)$ for $C \in [\![B]\!]$), and the following holds for all components $C \in [\![B]\!]$: (1) C is well-connected. (2) If there is a path from the i-th to the j-th output port of C, $i \neq j$, then $i \in br(C)$.[8] A hierarchical hypergraph H is called *well-formed* if all its nested boxes are proper. In that case, the conditions above imply that either all or no graphs from $[\![H]\!]$ are well-connected and that well-connectedness of graphs in $[\![H]\!]$ may be judged based only on the knowledge of $br(B)$ for each nested box B of H, without a need to reason about the semantics of B (in particular, Condition 2 guarantees that we do not have to take into account paths that interconnect output ports of B). This is formalised below.

Box-connectedness. Let $H = (V, E, I, O)$ be a well-formed hierarchical hypergraph over Γ with a set X of nested boxes. We define the *backward reachability graph* of H as the hypergraph $H^{br} = (V, E \cup E^{br}, I, O)$ over $\Gamma \cup X \cup X^{br}$ where $X^{br} = \{(B,i) \mid B \in X \wedge i \in br(B)\}$ and $E^{br} = \{(v_i, (B,i), (v)) \mid B \in X \wedge (v, B, (v_1, \ldots, v_n)) \in E \wedge i \in br(B)\}$. Then we say that H is *box-connected* iff H^{br} is well-connected. The below lemma clearly holds.

Lemma 3. *If H is a well-formed hierarchical hypergraph, then the hypergraphs from $[\![H]\!]$ are well-connected iff H is box-connected. Moreover, if hypergraphs from $[\![H]\!]$ are deterministic, then both H and H^{br} are deterministic hypergraphs.*

We straightforwardly extend the above notions to hypergraphs with hyperedges labelled by hierarchical SFA, treating these SFA-labels as if they were the boxes they represent. Particularly, we call a hierarchical SFA S proper iff it represents a proper box, we let $br(S) = br([\![\mathcal{L}_H(S)]\!])$, and for a hypergraph H over $\Gamma \cup Y$ where Y is a set of proper SFA, its backward reachability hypergraph H^{br} is defined based on br in the same way as backward reachability hypergraph of a hierarchical hypergraph above (just instead of boxes, we deal with their SFA representations). We also say that H is box-connected iff H^{br} is well-connected.

Given an FA $\mathcal{F}$ over Γ with proper nested SFA, we can check well-connectedness of graphs from $[\![\mathcal{L}_H(\mathcal{F})]\!]$ as follows: (1) for each nested SFA S of $\mathcal{F}$, we compute (and cache for further use) the value $br(S)$, and (2) using this value, we check box-connectedness of graphs in $\mathcal{L}(\mathcal{F})$ without a need of reasoning about the inner structure of the nested SFA. In [11], we describe how this computation may be done by inspecting rules of the component TA of $\mathcal{F}$. Properness of nested SFA may be checked on the level of TA too as also described [11].

Checking inclusion on hierarchical automata over Γ with nested boxes from X, i.e., given two hierarchical FA $\mathcal{F}$ and $\mathcal{F}'$, checking whether $[\![\mathcal{L}_H(\mathcal{F})]\!] \subseteq [\![\mathcal{L}_H(\mathcal{F}')]\!]$, is a hard problem, even under the assumption that nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ are proper. We have not even answered the question of its decidability yet. In this paper, we choose a pragmatic approach and give only a semialgorithm that is efficient and works well in practical cases. The idea is simple. Since the implications $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}') \implies \mathcal{L}_H(\mathcal{F}) \subseteq \mathcal{L}_H(\mathcal{F}') \implies [\![\mathcal{L}_H(\mathcal{F})]\!] \subseteq [\![\mathcal{L}_H(\mathcal{F}')]\!]$ obviously hold, we may safely approximate the

[8] Notice that this definition is correct since boxes of level 0 have no nested boxes, and the recursion stops at them.

solution of the inclusion problem by deciding whether $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ (i.e., we abstract away the semantics of nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ and treat them as ordinary labels).

From now on, assume that our hierarchical FA represent only deterministic well-connected hypergraphs, i.e., that $[\![\mathcal{L}_H(\mathcal{F})]\!]$ and $[\![\mathcal{L}_H(\mathcal{F}')]\!]$ contain only well-connected deterministic hypergraphs. Note that this assumption is in particular fulfilled for hierarchical FA representing garbage-free heaps.

We cannot directly use the results on inclusion checking of Section 3.4, based on a canonical forest representation and canonicity respecting FA, since they rely on well-connectedness of hypergraphs from $\mathcal{L}(\mathcal{F})$ and $\mathcal{L}(\mathcal{F}')$, which is now *not* necessarily the case. However, by Lemma 3, every graph H from $\mathcal{L}(\mathcal{F})$ or $\mathcal{L}(\mathcal{F}')$ is box-connected and both H and H^{br} are deterministic. As we show below, these properties are still sufficient to define a canonical forest representation of H, which in turn yields a canonicity respecting form of hierarchical FA.

Canonicity respecting hierarchical FA. Let Y be a set of proper SFA over Γ. We aim at a canonical forest representation $F = (T_1, \ldots, T_n, I, O)$ of a $\Gamma \cup Y$-labelled hypergraph $H = \oplus F$ which is box-connected and such that both H and H^{br} are deterministic. By extending the approach used in Section 3.4, this will be achieved via an unambiguous definition of the *root-points* of H, i.e., the nodes of H that correspond to the roots of the trees $T_1, \ldots, T_n$, and their ordering.

The root-points of H are defined as follows. First, every cut-point (port or a node with more than one incoming edge) is a *root-point of Type 1*. Then, every node with no incoming edge is a *root-point of Type 2*. Root-points of Type 2 are entry points of parts of H that are not reachable from root-points of Type 1 (they are only backward reachable). However, not every such part of H has a unique entry point which is a root-point of Type 2. Instead, there might be a simple loop such that there are no edges leading into the loop from outside. To cover a part of H that is reachable from such a loop, we have to choose exactly one node of the loop to be a root-point. To choose one of them unambiguously, we define a total ordering $\preceq_H$ on nodes of H and choose the smallest node wrt. this ordering to be a *root-point of Type 3*. After unambiguously determining all root-points of H, we may order them according to $\preceq_H$ and we are done.

A suitable total ordering $\preceq_H$ on V can be defined taking an advantage of the fact that H^{br} is well-connected and deterministic. Therefore, it is obviously possible to define $\preceq_H$ as the order in which the nodes are visited by a deterministic depth-first traversal that starts at input ports. The details on how this may be algorithmically done on the structure of forest automata may be found in [11].

We say that a hierarchical FA $\mathcal{F}$ over Γ with proper nested SFA and such that hypergraphs from $[\![\mathcal{L}_H(\mathcal{F})]\!]$ are deterministic and well-connected *respects canonicity* iff each forest $F \in \mathcal{L}_F(\mathcal{F})$ is a canonical representation of the hypergraph $\otimes F$. We abbreviate canonicity respecting hierarchical FA as hierarchical CFA. Analogically as for ordinary CFA, respecting canonicity allows us to compare languages of hierarchical CFA component-wise as described in the below lemma.

Lemma 4. *Let $\mathcal{F}_1 = (\mathcal{A}^1_1, \ldots, \mathcal{A}^1_{n_1}, I_1, O_1)$ and $\mathcal{F}_2 = (\mathcal{A}^2_1, \ldots, \mathcal{A}^2_{n_2}, I_2, O_2)$ be two hierarchical CFA. Then, $\mathcal{L}(\mathcal{F}_1) \subseteq \mathcal{L}(\mathcal{F}_2)$ iff (1) $n_1 = n_2$, (2) $I_1 = I_2$, (3) $O_1 = O_2$, and (4) $\forall 1 \leq i \leq n : \mathcal{L}(\mathcal{A}^1_i) \subseteq \mathcal{L}(\mathcal{A}^2_i)$.*

Lemma 4 allows us to safely approximate inclusion of the sets of hypergraphs encoded by hierarchical FA (i.e., to safely approximate the test $[\![\mathcal{L}_H(\mathcal{F})]\!] \subseteq [\![\mathcal{L}_H(\mathcal{F}')]\!]$ for hierarchical FA $\mathcal{F}$, $\mathcal{F}'$). This turns out to be sufficient for all our case studies (cf. Section 6). Moreover, the described inclusion checking is precise at least in some cases as discussed in [11]. A generalization of the result to sets of hierarchical CFA can be obtained as for ordinary SFA. Hierarchical FA that do not respect canonicity may be algorithmically split into several hierarchical CFA, similarly as ordinary CFA (see [11]).

5 The Verification Procedure Based on Forest Automata

We now briefly describe our verification procedure. As already said, we consider sequential, non-recursive C programs manipulating dynamic linked data structures via program statements given below[9]. Each allocated cell may have several next pointer selectors and contain data from some finite domain[10] (below, *Sel* denotes the set of all selectors and *Data* denotes the data domain). The cells may be pointed by program variables (whose set is denoted as *Var* below).

Heap Representation. As discussed in Section 2, we encode a single heap configuration as a deterministic $(Sel \cup Data \cup Var)$-labelled hypergraph with the ranking function being such that $\#(x) = 1 \Leftrightarrow x \in Sel$ and $\#(x) = 0 \Leftrightarrow x \in Data \cup Var$, in which nodes represent allocated memory cells, unary hyperedges (labelled by symbols from *Sel*) represent selectors, and the nullary hyperedges (labelled by symbols from $Data \cup Var$) represent data values and program variables[11]. Input ports of the hypergraphs are nodes pointed to by program variables. Null and undefined values are modelled as two special nodes `null` and `undef`. We represent sets of heap configurations as hierarchical $(Sel \cup Data \cup Var)$-labelled SCFA.

Symbolic Execution. The symbolic computation of reachable heap configurations is done over a control flow graph (CFG) obtained from the source program. A control flow action a applied to a hypergraph H (i.e., to a single configuration) returns a hypergraph $a(H)$ that is obtained from H as follows. Nondestructive actions $\mathtt{x} = \mathtt{y}$, $\mathtt{x} = \mathtt{y\text{->}s}$, or $\mathtt{x} = \mathtt{null}$ remove the x-label from its current position and label with it the node pointed by y, the s-successor of that node, or the `null` node, respectively. The destructive action $\mathtt{x\text{->}s} = \mathtt{y}$ replaces the edge $(v_{\mathtt{x}}, \mathtt{s}, v)$ by the edge $(v_{\mathtt{x}}, \mathtt{s}, v_{\mathtt{y}})$ where $v_{\mathtt{x}}$ and $v_{\mathtt{y}}$ are the nodes pointed to by x and y, respectively. Further, `malloc(x)` moves the x-label to a newly created node, `free(x)` removes the node pointed to by x (and links x and all aliased variables with `undef`), and $\mathtt{x\text{->}data} = \mathtt{d}_{new}$ replaces the edge $(v_{\mathtt{x}}, \mathtt{d}_{old})$ by the edge $(v_{\mathtt{x}}, \mathtt{d}_{new})$. Evaluating a guard g applied on H amounts to a simple test of equality of nodes or equality of data fields of nodes. Dereferences of `null` and `undef`

[9] Most C statements for pointer manipulation can be translated to these statements, including most type casts and restricted pointer arithmetic.

[10] No abstraction for such data is considered.

[11] Below, to simplify the informal description, we say that a node is labelled by a variable instead of saying that the variable labels a nullary hyperedge leaving from that node.

are of course detected (as an attempt to follow a non-existing hyperedge) and an error is announced. Emergence of garbage is detected iff $a(H)$ is not well-connected.[12]

We, however, compute not on single hypergraphs representing particular heaps but on sets of them represented by hierarchical SCFA. For now, we assume the nested SCFA used to be provided by the user. For a given control flow action (or guard) x and a hierarchical SCFA S, we need to symbolically compute an SCFA $x(S)$ s.t. $[\![\mathcal{L}_H(x(S))]\!]$ equals $\{x(H) \mid H \in [\![\mathcal{L}_H(S)]\!]\}$ if x is an action and $\{H \in [\![\mathcal{L}_H(S)]\!] \mid x(H)\}$ if x is a guard.

Derivation of the SCFA $x(S)$ from S involves several steps. The first phase is materialisation, where we unfold nested SFA representing boxes that hide data values or pointers referred to by x. We note that we are unfolding only SFA in the closest neighbourhood of the involved pointer variables; thus, on the level of TA, we touch only nested SFA adjacent to root-points. In the next phase, we introduce additional root-points for every node referred to by x to the forest representation. Third, we perform the actual update, which due to the previous step amounts to manipulation with root-points only (see [11] for details). Last, we repeatedly fold (apply) boxes and normalise (transform the obtained SFA into a canonicity respecting form) until no further box can be applied, so that we end up with an SCFA. We note that like unfolding, folding is also done only in the closest neighbourhood of root-points.

Unfolding is, loosely speaking, done by replacing a TA rule labelled by a nested SFA by the nested SFA itself (plus the proper binding of states of the top-level SFA to ports of the nested SFA). Folding is currently based on detecting isomorphism of a part of the top-level SFA and a nested SFA. The part of the top-level SFA is then replaced by a single rule labelled by the nested SFA. We note that this may be further improved by using language inclusion instead of isomorphism of automata.

The Fixpoint Computation. The verification procedure performs a classical (forward) control-flow fixpoint computation over the CFG, where flow values are hierarchical SCFA that represent sets of possible heap configurations at particular program locations. We start from the input location with the SCFA representing an empty heap with all variables undefined. The join operator is the union of SCFA. With every edge from a source location l labelled by x (an action or a guard), we associate the flow transfer function f_x. Function f_x takes the flow value (SCFA) S at l as its input and (1) computes the SCFA $x(S)$, (2) applies abstraction to $x(S)$, and returns the result.

Abstraction may be done by applying the general techniques described in [6] to the individual TA inside FA. Particularly, the abstraction collapses states with similar languages (based on their languages up-to certain tree depth or using predicate languages).

To detect spurious counterexamples and to refine abstraction, we use a *backward run* similarly as in [6]. This is possible since the steps of the symbolic execution may be reversed, and it is also possible to compute almost precise intersections of hierarchical SFA. More precisely, given SCFA S_1 and S_2, we can compute an SCFA S such that $[\![\mathcal{L}_H(S)]\!] \subseteq [\![\mathcal{L}_H(S_1)]\!] \cap [\![\mathcal{L}_H(S_2)]\!]$. This underapproximation is safe since it can lead

[12] Further, we note that we also handle a restricted pointer arithmetic. This is basically done by indexing elements of *Sel* by integers to express that the target of a pointer is an address of a memory cell plus or minus a certain offset. The formalism described in the paper may be easily adapted to support this feature.

Table 1. Experimental results

Example	Forester	Invader	ARTMC	Example	Forester	Invader	ARTMC
SLL (delete)	0.04	0.1	0.5	SLL (reverse)	0.04	0.03	
SLL (bubblesort)	0.12	Err		SLL (insertsort)	0.09	0.1	
SLL (mergesort)	0.12	Err		SLL of CSLLs	0.11	T	
SLL+head	0.04	0.06		SLL of 0/1 SLLs	0.13	T	
$\text{SLL}_{\text{Linux}}$	0.05	T		DLL (insert)	0.07	0.08	0.4
DLL (reverse)	0.05	0.09	1.4	DLL (insertsort1)	0.35	0.18	1.4
DLL (insertsort2)	0.16	Err		CDLL	0.04	0.09	
DLL of CDLLs	0.32	T		SLL of $\text{2CDLLs}_{\text{Linux}}$	0.11	T	
tree	0.11		3	tree+stack	0.10		
tree+parents	0.18			tree (DSW)	0.41		o.o.m.

neither to false positives nor to false negatives (it could only cause the computation not to terminate). Moreover, for the SCFA that appear in the case studies in this paper, the intersection we compute actually is precise. More details can be found in [11].

6 Implementation and Experimental Results

We have implemented the proposed approach in a prototype tool called *Forester*, having the form of a gcc plug-in. The core of the tool is our own library of TA that uses the recent technology for handling nondeterministic automata (particularly, methods for reducing the size of TA and for testing language inclusion on them [2,3]). The fixpoint computation is accelerated by the so-called finite height abstraction that is based on collapsing states of TA that have the same languages up to certain depth [6].

Although our implementation is an early prototype, the results are encouraging with regard to the generality of structures the tool can handle, precision of the generated invariants as well as the running times. We tested the tool on sample programs with various types of lists (singly, doubly linked, cyclic, nested), trees, and their combinations. Basic memory safety properties—in particular, absence of null and undefined pointer dereferences, double free operations, and absence of garbage—were checked.

We have compared performance of our tool with the tool Space Invader [4] based on separation logic and also with the tool ARTMC [7] based on abstract regular tree model checking. The comparison with Space Invader was done against examples with lists only since Invader does not handle trees. A higher flexibility of our automata abstraction manifests itself on several examples where Invader does not terminate. This is particularly well visible at the test case with a list of sublists of lengths 0 or 1 (discussed already in the introduction). Our technique handles this example smoothly (without any need to add some special inductive predicates that could decrease the performance or generate false alarms). The ARTMC tool can, in principle, handle more general structures than we can currently handle (such as trees with linked leaves). However, the used representation of heap-configurations is much heavier which causes ARTMC not to scale that well. (Since it is difficult to encode the input for ARTMC, we have tried only some interesting cases.)

Table 1 summarises running times (in seconds) of the three tools on our case studies. The value T means that the running time exceeded 30 minutes, o.o.m. means that the tool ran out of memory, and the value Err stands for a failure of symbolic execution. The names of experiments in the table contain the name of the data structure handled by the program, which ranges over "SLL" for singly-linked lists, "DLL" for doubly linked lists (the prefix "C" means cyclic), "tree" for binary trees, "tree+parents" for trees with parent pointers. Nested variants of SLL are named as "SLL of" and the type of the nested list. In particular, "SLL of 0/1 SLLs" stands for SLL of nested SLL of length 0 or 1. "SLL+head" stands for a list where each element points to the head of the list, "SLL of 2CDLLs" stands for SLL whose each node is a source of two CDLLs. The flag "Linux" denotes the implementation of lists used in the Linux kernel that uses a restricted pointer arithmetic which we can also handle. All experiments start with a random creation and end with a disposal of the specified structure. An indicated procedure (if any) is performed in between the creation and disposal phase. In the experiment "tree+stack", a randomly created tree is disposed using a stack in a top-down manner such that we always dispose a root of a subtree and save its subtrees into the stack. "DSW" stands for the Deutsch-Schorr-Waite tree traversal (the Lindstrom variant). We have run our tests on a machine with Intel T9600 (2.8GHz) CPU and 4GiB of RAM.

7 Conclusion

We have proposed hierarchically nested forest automata as a new means of encoding sets of heap configurations when verifying programs with dynamic linked data structures. The proposal brings the principle of separation from separation logic into automata, allowing us to combine some advantages of automata (generality, less rigid abstraction) with a better scalability stemming from local heap manipulation. We have shown some interesting properties of our representation from the point of view of inclusion checking. We have implemented the approach and tested it on multiple non-trivial cases studies, demonstrating the approach to be really promising.

In the future, we would like to first improve the implementation of our tool Forester, including support for predicate language abstraction within abstract regular tree model checking [6] as well as implementation of automatic learning of nested FA. From a more theoretical perspective, it is interesting to show whether inclusion checking is or is not decidable for the full class of nested FA. Another interesting direction is then a possibility of allowing truly recursive nesting of FA, which would allow us to handle very general structures such as trees with linked leaves.

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic Abstraction for Programs with Dynamic Memory Heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008)
2. Abdulla, P.A., Bouajjani, A., Holík, L., Kaati, L., Vojnar, T.: Computing Simulations over Tree Automata. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 93–108. Springer, Heidelberg (2008)

3. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When Simulation Meets Antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
4. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
5. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with Lists Are Counter Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
6. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking. In: ENTCS, vol. 149(1), Elsevier, Amsterdam (2006)
7. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
8. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-abduction. In: Proc. of POPL 2009. ACM Press, New York (2009)
9. Deshmukh, J.V., Emerson, E.A., Gupta, P.: Automatic Verification of Parameterized Data Structures. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 27–41. Springer, Heidelberg (2006)
10. Guo, B., Vachharajani, N., August, D.I.: Shape Analysis with Inductive Recursion Synthesis. In: Proc. of PLDI 2007. ACM Press, New York (2007)
11. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest Automata for Verification of Heap Manipulation. Technical Report FIT-TR-2011-01, FIT BUT, Czech Republic (2011), `http://www.fit.vutbr.cz/~isimacek/pub/FIT-TR-2011-01.pdf`
12. Madhusudan, P., Parlato, G., Qiu, X.: Decidable Logics Combining Heap Structures and Data. In: Proc. of POPL 2011. ACM Press, New York (2011)
13. Møller, A., Schwartzbach, M.: The Pointer Assertion Logic Engine. In: Proc. of PLDI 2001. ACM Press, New York (2001)
14. Nguyen, H.H., David, C., Qin, S.C., Chin, W.-N.: Automated Verification of Shape and Size Properties Via Separation Logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
15. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: Proc. of LICS 2002. IEEE Computer Society Press, Los Alamitos (2002)
16. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic. TOPLAS 24(3) (2002)
17. Yang, H., Lee, O., Calcagno, C., Distefano, D., O'Hearn, P.W.: On Scalable Shape Analysis. Technical report RR-07-10, Queen Mary, University of London (2007)
18. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
19. Zee, K., Kuncak, V., Rinard, M.: Full Functional Verification of Linked Data Structures. In: Proc. of PLDI 2008. ACM Press, New York (2008)