Noname manuscript No.
(will be inserted by the editor)

Forest Automata for Verification of Heap Manipulation

Peter Habermehl · Lukáš Holík · Adam Rogalewicz · Jiří Šimáček · Tomáš Vojnar

Received: date / Accepted: date

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 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

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 EU/Czech IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070, the internal BUT project FIT-S-12-1, and the French ANR-09-SEGI project Veridyc.

P. Habermehl
Université Paris Diderot, Sorbonne Paris Cité, LIAFA, CNRS, France

L. Holík
FIT, Brno University of Technology, Czech Republic and
Uppsala University, Sweden

A. Rogalewicz
FIT, Brno University of Technology, Czech Republic

J. Šimáček
FIT, Brno University of Technology, Czech Republic and
VERIMAG, UJF/CNRS/INPG, Gières, France

T. Vojnar
FIT, Brno University of Technology, Czech Republic, E-mail: vojnar@fit.vutbr.cz

(efficiency). We have implemented our approach and tested it successfully on multiple non-trivial case studies.

Keywords Pointers · shape analysis · regular model checking · tree automata

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.

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 [16]. Using this decomposition, sets of heaps with a bounded number of cut-points are then represented by a new class of automata 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., [14,17,16,4,11,15,20,19,8,13], 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 [4,19] 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 [4,19], 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., [15], that consider tree manipulation, but these are usually semi-automated only. An exception is [11] 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 learned (meaning, e.g., that lists are built by adding elements at the end only[1]).

Further, compared to [4,19], 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 [18]. In such cases, the abstraction used in [4,19] 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.

Finally, compared with [7], our newly proposed approach is a bit less general. We cannot handle structures such as, e.g., trees with linked leaves. To handle these structures, we would have to introduce into our approach FA nested not just strictly hierarchically but in an arbitrary, possibly cyclic way, which is an interesting subject for future research. On the other hand, our new approach is more scalable than that of [7]. This is due to the fact that the heap representation in [7] is monolithic, i.e., the whole heap is represented by a single tree skeleton over which additional pointer links are expressed using the so-called routing expressions. The

[1] We did not find an available implementation of [11], and so we could not try it out ourselves.

new encoding is much more structured, and so the different operations on the heap, corresponding to a symbolic execution of the verified program, typically influence only small parts of the encoding and not all (or most) of it. The monolithic encoding of [7] has also problems with deletion of elements inside data structures since the routing expressions are built over a tree backbone that is assumed not to change (and hence deleted elements inside data structures are always kept, just marked as deleted). Moreover, the encoding of [7] has troubles with detection of memory leakage, which is in theory possible, but it is so complex that it has never been implemented.

Plan of the paper. In the next section, we provide an informal introduction to our proposal of hierarchical forest automata and their use for encoding sets of heaps. In Section 3, the notion of (non-hierarchical) forest automata is formalised, and we examine properties of forest automata from the point of view of inclusion checking. Subsequently, Section 4 generalises the notion of forest automata to hierarchical forest automata. In Section 5, we propose a verification procedure based on hierarchical forest automata. Section 6 provides a brief description of the Forester tool implementing the proposed approach as well as results obtained from experiments with Forester. Finally, Section 7 concludes the paper.

2 From Heaps to Forests

In this section, we outline in an informal way our proposal of hierarchical forest automata and the way how sets of heaps can be represented by them. 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 restrict ourselves to *garbage free heaps* in which all memory cells are reachable from pointer variables by following pointer links. However, this is not a restriction in practice since the emergence of garbage can be checked for each executed program statement. If some garbage arises, an error message can be issued and the symbolic computation stopped. Alternatively, the garbage can be removed and the computation continued.

It is easy to see that each heap graph can be *decomposed* into a set of *tree components* when the leaves of the tree components are allowed to reference back to the roots of these components. Moreover, given a total ordering on program variables and selectors, each heap graph may be decomposed into a tuple of tree components in a *canonical way* as illustrated in Figure 1(a) and (b). In particular, one can first identify the so-called *cut-points*, i.e., nodes that are either pointed to by a program variable or that have several incoming edges. Next, the cut-points can be canonically numbered using a depth-first traversal of the heap graph starting from nodes pointed to by program variables in the order derived from the order of the program variables and respecting the order of selectors. Subsequently, one can split the heap graph into tree components rooted at particular cut-points. These components should 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

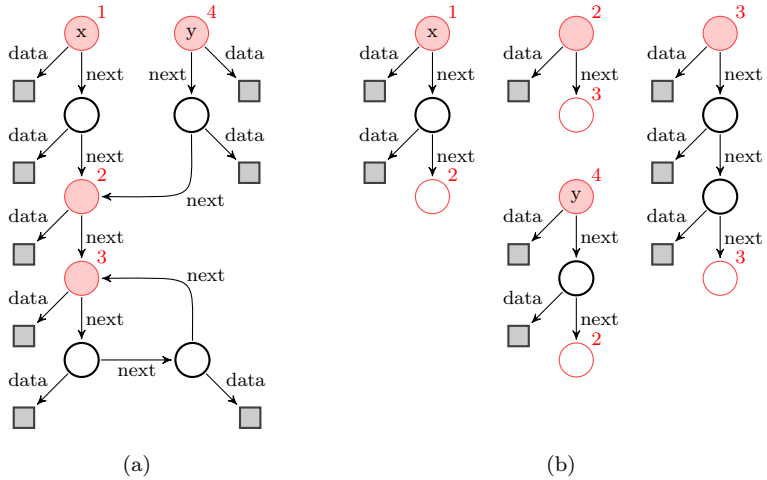

Fig. 1 (a) A heap graph with cut-points highlighted in red, (b) the canonical tree decomposition of the heap with `x` ordered before `y`.

its number. Finally, the tree components can then be canonically ordered according to the numbers of the cut-points representing their roots.

Our proposal of forest automata builds upon the described decomposition of heaps into tree components. In particular, a *forest automaton* (FA) is basically a tuple of tree automata (TA). Each of the tree automata accepts trees whose leaves may refer back to the roots of any of these trees. An FA then represents exactly the set of heaps that may be obtained by taking a single tree from the language of each of the component TA and by gluing the roots of the trees with the leaves referring to them.

Below, we will mostly concentrate on a subclass of FA that we call *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 testing inclusion component-wise on the languages of the TA constituting the given CFA.

Note, however, that FA are not closed under union. Even for 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). Thus, we represent unions of FA explicitly as *sets of FA* (SFA), which is similar to dealing with disjunctions of conjunctive separation logic formulae. However, as we will see, inclusion on the sets of heaps represented by SFA is still easily decidable.

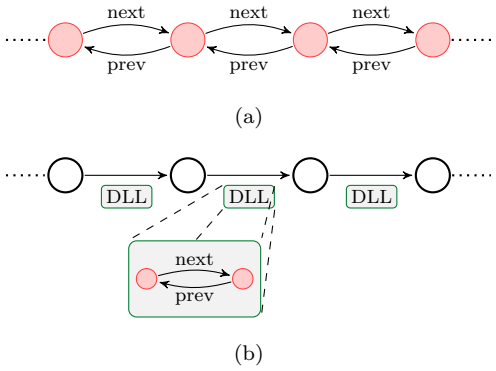

Fig. 2 (a) A part of a DLL, (b) a hierarchical encoding of the DLL.

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, for instance, in doubly-linked lists (DLLs), 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*) containing cut-points in the so-called *boxes*. Every occurrence of such components can then be replaced by a single edge labelled by the appropriate box. To specify how a subgraph enclosed within a box is connected to the rest of the graph, the subgraph is equipped with the so-called input and output ports. The source vertex of a box then matches the input port of the subgraph, and the target vertex of the edge matches the output port.[2] In this way, a set of heap graphs with an unbounded number of cut-points can be transformed into a set of *hierarchical heap graphs* with a bounded number of cut-points at each level of the hierarchy. Figures 2(a) and (b) illustrate how this approach can basically reduce DLLs into singly-linked lists (with a DLL segment used as a kind of meta-selector).

In general, we allow a box to have more than one output port. Boxes with multiple output ports, however, reduce heap graphs not to graphs but *hypergraphs* with *hyperedges* having a single source node, but multiple target nodes. This situation is illustrated on a simple example shown in Figure 3. The tree with linked brothers from Figure 3(a) is turned into a hypergraph with binary hyperedges shown in Figure 3(c) using the box B from Figure 3(b). The subgraph encoded by the box B can be connected to its surroundings via its input port i and *two* output ports $o1$, $o2$. Therefore, the hypergraph from Figure 3(c) encodes it by a hyperedge with one source and *two* target nodes.

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 *hierarchical FA* whose alphabet can contain

[2] Later on, the term input port will be used to refer to the nodes pointed to by program variables too since these nodes play a similar role as the inputs of components.

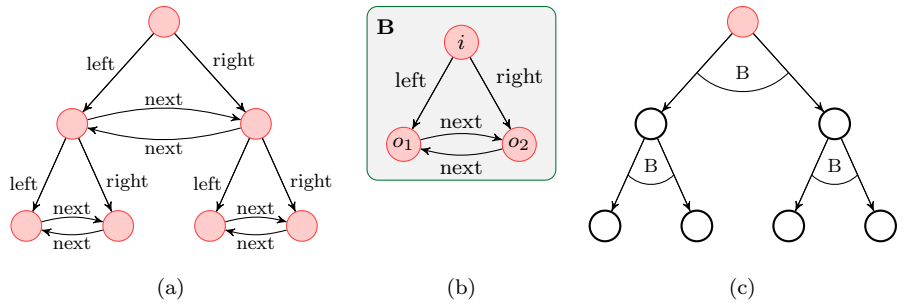

Fig. 3 (a) A tree with linked brother nodes, (b) a pattern that repeats in the structure and that is linked in such a way that all nodes in the structure are cut-points, (c) the tree with linked brother nodes represented using hyperedges labelled by the box B.

nested FA.[3] Intuitively, FA appearing in the alphabet of some superior FA play a role similar to that of inductive predicates in separation logic.[4] We restrict ourselves to automata that form a finite and strict hierarchy (i.e., there is no circular use of the automata in their alphabets).

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 provided that 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 us 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

A *ranked alphabet* is a finite set Γ of symbols associated with a map $\# : \Gamma \to \mathbb{N}$. The value $\#(a)$ is called the *rank* of $a \in \Gamma$. We use $\#(\Gamma)$ to denote the maximum rank of a symbol in Γ. A ranked alphabet Γ is a *hypergraph alphabet* if it is associated with

[3] 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.

[4] For instance, we use a nested FA to encode a DLL segment of length 1. In separation logic, the corresponding induction predicate would represent segments of length 1 or more. In our approach, the repetition of the segment is encoded in the structure of the top-level FA.

a total ordering $\preceq_\Gamma$ on its symbols. For the rest of the section, we fix a hypergraph alphabet Γ.

An (oriented, Γ-labelled) *hypergraph* (with designated input/output ports) is a tuple $G = (V, E, P)$ where:

- V is a finite set of *vertices*.
- E is a finite set of *hyperedges* such that every hyperedge $e \in E$ is of the form $(v, a, (v_1, \ldots, v_n))$ where $v \in V$ is the *source* of e, $a \in \Gamma$, $n = \#(a)$, and $v_1, \ldots, v_n \in V$ are *targets* of e and *a-successors* of v.
- P is the so-called *port specification* that consists of a set of *input ports* $I_P \subseteq V$, a set of *output ports* $O_P \subseteq V$, and a total ordering $\preceq_P$ on $I_P \cup O_P$.

We use $\overline{v}$ to denote a sequence $v_1, \ldots, v_n$ and $\overline{v}.i$ to denote its i^{th} vertex v_i. For symbols $a \in \Gamma$ with $\#(a) = 0$, we write $(v, a) \in E$ to denote that $(v, a, ()) \in E$. Such hyperedges may simulate labels assigned to vertices.

A *path* in a hypergraph $G = (V, E, P)$ 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}. G is called *deterministic* iff $\forall (v, a, \overline{v}), (v, a', \overline{v}') \in E$: $a = a' \implies \overline{v} = \overline{v}'$. G is called *well-connected* iff each node $v \in V$ is reachable through some path from some input port of G.

As we have already mentioned in Section 2, 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(a) shows a hypergraph with two input ports corresponding to the variables `x` and `y`. The hyperedges are labelled by selectors `data` and `next`. All the hyperedges are of arity 1. A simple example of a hypergraph with hyperedges of arity 2 is given in Figure 3(c).

3.2 A Forest Representation of Hypergraphs

We will now define the forest representation of hypergraphs. For that, we will first define a notion of a tree as a basic building block of forests. We will define trees much like hypergraphs but with a restricted shape and without input/output ports. The reason for the latter is that the ports of forests will be defined on the level of the forests themselves, not on the level of the trees that they are composed of.

Formally, an (unordered, oriented, Γ-labelled) *tree* $T = (V, E)$ consists of a set of vertices and hyperedges defined as in the case of hypergraphs with the following additional requirements: (1) V contains a single node with no incoming hyperedge (called the *root* of T and denoted $root(T)$). (2) All other nodes of T are reachable from $root(T)$ via some path. (3) Each node has at most one incoming hyperedge. (4) Each node appears at most once among the target nodes of its incoming hyperedge (if it has one). Given a tree, we call its nodes with no successors *leaves*.

Let us assume that $\Gamma \cap \mathbb{N} = \emptyset$. An (ordered, Γ-labelled) *forest* (with designated input/output ports) is a tuple $F = (T_1, \ldots, T_n, R)$ such that:

- For every $i \in \{1, \ldots, n\}$, $T_i = (V_i, E_i)$ is a tree that is labelled by the alphabet $(\Gamma \cup \{1, \ldots, n\})$.
- R is a (forest) port specification consisting of a set of *input ports* $I_R \subseteq \{1, \ldots, n\}$, a set of *output ports* $O_R \subseteq \{1, \ldots, n\}$, and a total ordering $\preceq_R$ of $I_R \cup O_R$.

– For all $i, j \in \{1, \ldots, n\}$, (1) if $i \neq j$, then $V_i \cap V_j = \emptyset$, (2) $\#(i) = 0$, and (3) a vertex v with $(v, i) \in E_j$ is not a source of any other edge (it is a leaf). We call such vertices *root references* and denote by $rr(T_i)$ the set of all root references in T_i, i.e., $rr(T_i) = \{v \in V_i \mid (v, k) \in E_i, k \in \{1, \ldots, n\}\}$.

A forest $F = (T_1, \ldots, T_n, R)$ *represents* the hypergraph $\otimes F$ obtained by uniting the trees $T_1, \ldots, T_n$ and interconnecting their roots with the corresponding root references. In particular, for every root reference $v \in V_i$, $i \in \{1, \ldots, n\}$, hyperedges leading to v are redirected to the root of T_j where $(v, j) \in E_i$, and v is removed. The sets I_R and O_R then contain indices of the trees whose roots are to be input/output ports of $\otimes F$, respectively. Finally, their ordering $\preceq_P$ is defined by the $\preceq_R$-ordering of the indices of the trees whose roots they are. Formally, $\otimes F = (V, E, P)$ 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 $\overline{v}'.j = \overline{v}.j\}$,
– $I_P = \{root(T_i) \mid i \in I_R\}$, $O_P = \{root(T_i) \mid i \in O_R\}$,
– $\forall u, v \in I_P \cup O_P$ such that $u = root(T_i)$ and $v = root(T_j)$: $u \preceq_P v \iff i \preceq_R j$.

3.3 Minimal and Canonical Forests

We now define the canonical form of a forest which will be important later for deciding language inclusion on forest automata, acceptors of sets of hypergraphs.

We call a forest $F = (T_1, \ldots, T_n, R)$ representing the well-connected hypergraph $\otimes F$ *minimal* iff the roots of the trees $T_1, \ldots, T_n$ correspond to the *cut-points* of $\otimes F$, i.e., those nodes that are either ports, have more than one incoming hyperedge in $\otimes F$, or appear more than once as a target of some hyperedge. A minimal forest representation of a hypergraph is unique up to permutations of $T_1, \ldots, T_n$.

In order to get a truly unique canonical forest representation of a well-connected *deterministic* hypergraph $G = (V, E, P)$, it remains to canonically order the trees in its minimal forest representation. To do this, we use the total ordering $\preceq_P$ on ports P and the total ordering $\preceq_\Gamma$ on hyperedge labels Γ of G. We then order the trees according to the order in which their roots are visited in a depth-first traversal (DFT) of G. If all nodes are not reachable from a single port, a series of DFTs is used. The DFTs are started from the input ports in I_P in the order given by $\preceq_P$. During the DFTs, a priority is given to the hyperedges that are smaller in $\preceq_\Gamma$. A canonical representation is obtained this way since we consider G to be deterministic.

Figure 1(b) shows a forest decomposition of the heap graph of Figure 1(a). The nodes pointed to by variables are input ports of the heap graph. Assuming that the ports are ordered such that the port pointed by `x` precedes the one pointed by `y`, then the forest of Figure 1(b) is a canonical representation of the heap graph of Figure 1(a).

3.4 Tree Automata

Next, we will work towards defining forest automata as tuples of tree automata encoding sets of forests and hence sets of hypergraphs. We start by classical definitions of tree automata and their languages.

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.

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 $f(q_1, \ldots, q_n) \to q$ 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*.

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 $1 \leq i \leq n$, $t(p)(q_1, \ldots, q_n) \to q$. We write $t \xrightarrow{\pi} q$ to denote that π is a run of $\mathcal{A}$ over t such that $\pi(\epsilon) = q$. We use $t \Longrightarrow q$ to denote that $t \xrightarrow{\pi} q$ for some run π. The *language* of a state q is defined by $L(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of $\mathcal{A}$ is defined by $L(\mathcal{A}) = \bigcup_{q \in F} L(q)$.

3.5 Forest Automata

We will now define forest automata as tuples of tree automata extended by a port specification. Tree automata accept trees that are ordered and node-labelled. Therefore, in order to be able to use forest automata to encode sets of forests, we must define a conversion between ordered, node-labelled trees and our unordered, edge-labelled trees.

We convert a deterministic Γ-labelled unordered tree T into a node-labelled ordered tree $ot(T)$ by (1) transferring the information about labels of edges of a node into the symbol associated with the node and by (2) ordering the successors of the node. More concretely, we label each node of the ordered tree $ot(T)$ by the set of labels of the hyperedges leading from the corresponding node in the original tree T. Successors of the node in $ot(T)$ correspond to the successors of the original node in T, and are ordered w.r.t. the order $\preceq_\Gamma$ of hyperedge labels through which the corresponding successors are reachable in T (while always keeping tuples of nodes reachable via the same hyperedge together, ordered in the same way as they were ordered within the hyperedge). The rank of the new node label is given by the sum of ranks of the original hyperedge labels embedded into it. Below, we use Σ_Γ to denote the ranked node alphabet obtained from Γ as described above.

The Notion of Forest Automata. A *forest automaton* over Γ (with designated input/output ports) is a tuple $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ where:

- For all $1 \leq i \leq n$, $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$ is a TA with $\Sigma = \Sigma_\Gamma \cup \{1, \ldots, n\}$ and $\#(i) = 0$.

– R is defined as for forests, i.e., it consists of input and output ports $I_R, O_R \subseteq \{1, \ldots, n\}$ and a total ordering $\preceq_R$ on $I_R \cup O_R$.

The *forest language* of $\mathcal{F}$ is the set of forests $L_F(\mathcal{F}) = \{(T_1, \ldots, T_n, R) \mid \forall 1 \leq i \leq n : \ ot(T_i) \in L(\mathcal{A}_i)\}$, i.e., the forest language is obtained by taking the Cartesian product of the tree languages, unordering the trees that appear in its elements, and extending them by the port specification. The forest language of $\mathcal{F}$ in turn defines the *hypergraph language* of $\mathcal{F}$ which is the set of hypergraphs $L(\mathcal{F}) = \{\otimes F \mid F \in L_F(\mathcal{F})\}$.

An FA $\mathcal{F}$ *respects canonicity* iff for each forest $F \in L_F(\mathcal{F})$, the hypergraph $\otimes F$ is well-connected, and F is its canonical representation. 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 proposition.

Proposition 1 *Let $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ and $\mathcal{F}' = (\mathcal{A}'_1, \ldots, \mathcal{A}'_m, R')$ be two CFA. Then, $L(\mathcal{F}) \subseteq L(\mathcal{F}')$ iff $n = m$, $R = R'$, and $\forall 1 \leq i \leq n : \ L(\mathcal{A}_i) \subseteq L(\mathcal{A}'_i)$.*

3.6 Transforming FA into Canonicity Respecting FA

In order to facilitate inclusion checking, each FA can be algorithmically transformed (split) into a finite set of CFA such that the union of their languages equals the original language. We describe the transformation in a more detailed way below.

First, we label the states of the component TA of the given FA by special labels. For each state, these labels capture all possible orders in which root references appear in the leaves of the trees accepted at this state when the left-most (i.e., the first) appearance of each root-reference is considered only. Moreover, the labels capture which of the references appear multiple times. Intuitively, following the first appearances of the root references in the leaves of tree components is enough to see how a depth first traversal through the represented hypergraph orders the roots of the tree components. The knowledge of multiple references to the same root from a single tree is then useful for checking which nodes should really be the roots.

The computed labels are subsequently used to possibly split the given FA into several FA such that the accepting states of the component TA of each of the obtained FA are labelled in a unique way. This guarantees that the obtained FA are canonicity respecting up to the fact that the roots of some of the trees accepted by component TA need not be cut-points (and up to the ordering of the component TA). Thus, subsequently, some of the TA may get merged. Finally, we order the remaining component TA in a way consistent with the DFT ordering on the cut-points of the represented hypergraphs (which after the splitting is the same for all the hypergraphs represented by each obtained FA). To order the component TA, the labels of the accepting states can be conveniently used.

More precisely, consider a forest automaton $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$, $n \geq 1$, and any of its component tree automata $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$, $1 \leq i \leq n$. We label each state $q \in Q_i$ by a set of labels (w, Y), $w \in \{1, \ldots, n\}^*$, $Y \subseteq \{1, \ldots, n\}$, for which there is a tree $t \in L(q)$ such that

– w is the string that records the order in which root references appear for the first time in the leaves of t (i.e., w is the concatenation of the labels of the

leaves labelled by root references, restricted to the first occurrence of each root reference), and
- Y is the set of root references that appear more than once in the leaves of t.

Such labelling can be obtained by first labelling states w.r.t. the leaf rules and then propagating the so-far obtained labels bottom-up. If the final states of $\mathcal{A}_i$ get labelled by several different labels, we make a copy of the automaton for each of these labels, and in each of them, we preserve only the transitions that allow trees with the appropriate label of the root to be accepted.[5] This way, all the component automata can be processed and then new forest automata can be created by considering all possible combinations of the transformed TA.

Clearly, each of the FA created above represents a set of hypergraphs that have the same number of cut-points (corresponding either to ports, nodes referenced at least twice from a single component tree, or referenced from several component trees) that get ordered in the same way in the depth first traversal of the hypergraphs. However, it may be the case that some roots need not correspond to cut-points. This is easy to detect by looking for a root reference that does not appear in the set part of any label of some final state and that does not appear in the labels of two different component tree automata. A useless root can then be eliminated by adding transition rules of the appropriate component tree automaton $\mathcal{A}_i$ to those of the tree automaton $\mathcal{A}_j$ that refers to that root and by gluing final states of $\mathcal{A}_i$ with the states of $\mathcal{A}_j$ accepting the root reference i.

It remains to order the component TA within each of the obtained FA in a way consistent with the DFT ordering of the cut-points of the represented hypergraphs (which is now the same for all the hypergraphs represented by a single FA due to the performed splitting). To order the component TA of any of the obtained FA, one can use the w-part of the labels of its accepting states. One can then perform a DFT on the component TA, considering the TA as atomic objects. One starts with the TA that accept trees whose roots represent ports and processes them wrt. the ordering of ports. When processing a TA $\mathcal{A}$, one considers as its successors the TA that correspond to the root references that appear in the w-part of the labels of the accepting states of $\mathcal{A}$. Moreover, the successor TA are processed in the order in which they are referenced from the labels. When the DFT is over, the component TA may get reordered according to the order in which they were visited.

Subsequently, the port specification R and root references in leaves must be updated to reflect the reordering. If the original sets I_R or O_R contain a port i, and the i^{th} tree was moved to the j^{th} position, then i must be substituted by j in I_R, O_R, and $\preceq_R$ as well as in all root references. This finally leads to a set of canonicity respecting FA.

Note that, in practice, it is not necessary to tightly follow the above described process. Instead, one can arrange the symbolic execution of statements in such a way that when starting with a CFA, one obtains an FA which already meets some requirements for CFA. Most notably, the splitting of component TA—if needed—can be efficiently done already during the symbolic execution of the particular

[5] More technically, given a labelled TA, one can first make a separate copy of each state for each of its labels, connect the states by transitions such that the obtained singleton labelling is respected, then make a copy of the TA for each label of accepting states, and keep the accepting status for a single labelling of accepting states in each of the copies only.

statements. Therefore, transforming an FA obtained this way into the corresponding CFA involves the elimination of redundant roots and the root reordering only.

3.7 Sets of Forest Automata

The class of languages of FA (and even CFA) is not closed under union since a forest language of a FA corresponds to the Cartesian product of the languages of all its components, and not every union of Cartesian products may be expressed as a single Cartesian product. For instance, consider two CFA $\mathcal{F} = (\mathcal{A}, \mathcal{B}, R)$ and $\mathcal{F}' = (\mathcal{A}', \mathcal{B}', R)$ such that $L_F(\mathcal{F}) = \{(a, b, R)\}$ and $L_F(\mathcal{F}') = \{(c, d, R)\}$ where a, b, c, d are distinct trees. The forest language of the FA $(\mathcal{A} \cup \mathcal{A}', \mathcal{B} \cup \mathcal{B}', R)$ is $\{(x, y, R) \mid (x, y) \in \{a, c\}\times\{b, d\}\})$, and there is no FA with the hypergraph language equal to $L(\mathcal{F}) \cup L(\mathcal{F}')$.

Due to the above, we cannot transform a set of CFA obtained by canonising a given FA into a single CFA. Likewise, when we obtain several CFA when symbolically executing several program paths leading to the same program location, we cannot merge them into a single CFA without risking a loss of information. Consequently, we will explicitly work with *finite sets of (canonicity-respecting) forest automata*, S(C)FA for short, where the language $L(\mathcal{S})$ of a finite set $\mathcal{S}$ of FA is defined as the union of the languages of its elements. This, however, means that we need to be able to decide language inclusion on SFA.

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

We will transform every FA $\mathcal{F}$ in $\mathcal{S}$ and $\mathcal{S}'$ into a TA $\mathcal{A}^{\mathcal{F}}$ which accepts the language of trees where:

- The root of each of these trees is labelled by a special fresh symbol (parameterised by n and the port specification of $\mathcal{F}$).
- The root has n children, one for each tree automaton of $\mathcal{F}$.
- For each $1 \leq i \leq n$, the i^{th} child of the root is the root of a tree accepted by the i^{th} tree automaton of $\mathcal{F}$.

Trees accepted by $\mathcal{A}^{\mathcal{F}}$ are therefore unique encodings of hypergraphs in $L(\mathcal{F})$. We will then test the inclusion $L(\mathcal{S}) \subseteq L(\mathcal{S}')$ by testing the tree automata language inclusion between the union of TA obtained from $\mathcal{S}$ and the union of TA obtained from $\mathcal{S}'$.

Formally, let $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ be an FA where $\mathcal{A}_i = (\Sigma, Q_i, \Delta_i, F_i)$ for each $1 \leq i \leq n$. Without a loss of generality, assume that $Q_i \cap Q_j = \emptyset$ for each $1 \leq i < j \leq n$. We define the TA $\mathcal{A}^{\mathcal{F}} = (\Sigma \cup \{\lambda_n^R\}, Q, \Delta, \{q^{top}\})$ where:

- $\lambda_n^R \notin \Sigma$ is a fresh symbol with $\#(\lambda_n^R) = n$,
- $q^{top} \notin \bigcup_{i=1}^{n} Q_i$ is a fresh accepting state,
- $Q = \bigcup_{i=1}^{n} Q_i \cup \{q^{top}\}$, and
- $\Delta = \bigcup_{i=1}^{n} \Delta_i \cup \Delta^{top}$ where Δ^{top} contains the rule $\lambda_n^R(q_1, \ldots, q_n) \to q^{top}$ for each $(q_1, \ldots, q_n) \in F_1 \times \cdots \times F_n$.

It is now easy to see that the following proposition holds (in the proposition, "$\cup$" stands for the usual tree automata union).

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

4 Hierarchical Hypergraphs

As discussed informally in Section 2, simple forest automata cannot express sets of data structures with unbounded numbers of cut-points like, e.g., the set of all doubly-linked lists or the set of all trees with linked brothers (Figures 2 and 3). To capture such data structures, we will enrich the expressive power of forest automata by allowing them to be hierarchically nested. For the rest of the section, we fix a hypergraph alphabet Γ.

4.1 Hierarchical Hypergraphs, Components, and Boxes

We first introduce hypergraphs with hyperedges labelled by the so-called boxes which are sets of hypergraphs (defined up to isomorphism[6]). A hypergraph G with hyperedges labelled by boxes encodes a set of hypergraphs. The hypergraphs encoded by G can be obtained by replacing every hyperedge of G labelled by a box by some hypergraph from the box. The hypergraphs within the boxes may themselves have hyperedges labelled by boxes, which gives rise to a hierarchical structure (which we require to be of a finite depth).

Let Υ be a hypergraph alphabet. First, we define an Υ-labelled *component* as an Υ-labelled hypergraph $C = (V, E, P)$ which satisfies the requirement that $|I_P| = 1$ and $I_P \cap O_P = \emptyset$. Then, an Υ-labelled *box* is a non-empty set B of Υ-labelled components such that all of them have the same number of output ports. This number is called the *rank of the box B* and denoted by $\#(B)$. Let $\mathbb{B}[\Upsilon]$ be the ranked alphabet containing all Υ-labelled boxes such that $\mathbb{B}[\Upsilon] \cap \Upsilon = \emptyset$. The operator $\mathbb{B}$ gives rise to a hierarchy of alphabets $\Gamma_0, \Gamma_1, \ldots$ where:

- $\Gamma_0 = \Gamma$ is the set of *plain symbols*,
- for $i \geq 0$, $\Gamma_{i+1} = \Gamma_i \cup \mathbb{B}[\Gamma_i]$ is the set of *symbols of level $i+1$*.

A Γ_i-labelled hypergraph H is then called a *Γ-labelled (hierarchical) hypergraph of level i*, and we refer to the Γ_{i-1}-labelled boxes appearing on edges of H as to *nested boxes of H*. A Γ-labelled hypergraph is sometimes called a *plain* Γ-labelled hypergraph.

Semantics of hierarchical hypergraphs. A Γ-labelled hierarchical hypergraph H encodes a set $[\![H]\!]$ of plain hypergraphs, called the *semantics* of H. For a set S of hierarchical hypergraphs, we use $[\![S]\!]$ to denote the union of semantics of its elements.

If H is plain, then $[\![H]\!]$ contains just H itself. If H is of level $j > 0$, then hypergraphs from $[\![H]\!]$ are obtained in such a way that hyperedges labelled by

[6] Dealing with hypergraphs and later also automata defined up to isomorphism avoids a need to deal with classes instead of sets. We will not repeat this fact later on.

boxes $B \in \Gamma_j$ are substituted in all possible ways by plain components from $[\![B]\!]$. The substitution is similar to an ordinary hyperedge replacement used in graph grammars. When an edge e is substituted by a component C, the input port of C is identified with the source node of e, and the output ports of C are identified with the target nodes of e. The correspondence of the output ports of C and the target nodes of e is defined using the order of the target nodes in e and the ordering of ports of C. The edge e is finally removed from H.

Formally, given a Γ-labelled hierarchical hypergraph $H = (V, E, P)$, a hyperedge $e = (v, a, \overline{v}) \in E$, and a component $C = (V', E', P')$ where $\#(a) = |O_{P'}| = k$, the substitution of e by C in H results in the hypergraph $H[C/e]$ defined as follows. Let $o_1 \preceq_P \ldots \preceq_P o_k$ be the ports of O_P ordered by $\preceq_P$. W.l.o.g., assume $V \cap V' = \emptyset$. C will be connected to H by identifying its ports with their matching vertices of e. We define for every vertex $w \in V'$ its matching vertex $\mathit{match}(w)$ such that (1) if $w \in I_{P'}$, $\mathit{match}(w) = v$ (the input port of C matches the source of e), (2) if $w = o_i, 1 \leq i \leq k$, $\mathit{match}(w) = \overline{v}.i$ (the output ports of C match the corresponding targets of e), and (3) $\mathit{match}(w) = w$ otherwise (an inner node of C is not matched with any node of H). Then $H[C/e] = (V'', E'', P)$ where $V'' = V \cup (V' \setminus (I_{P'} \cup O_{P'}))$ and $E'' = (E \setminus \{e\}) \cup \{(v'', a', \overline{v}'') \mid \exists (v', a', \overline{v}') \in E' : \ \mathit{match}(v') = v'' \ \wedge \ \forall 1 \leq i \leq k : \ \mathit{match}(\overline{v}'.i) = \overline{v}''.i\}$.

We can now give an inductive definition of $[\![H]\!]$. Let $e_1 = (v_1, B_1, \overline{v}_1)$, $\ldots$, $e_n = (v_n, B_n, \overline{v}_n)$ be all edges of H labelled by Γ-labelled boxes. Then, $G \in [\![H]\!]$ iff it is obtained from H by successively substituting every e_i by a component $C_i \in [\![B_i]\!]$, i.e.,

$$[\![H]\!] = \{H[C_1/e_1] \ldots [C_n/e_n] \mid C_1 \in [\![B_1]\!], \ldots, C_n \in [\![B_n]\!]\}.$$

Figure 2(b) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)graph of Figure 2(a). Similarly, Figure 3(c) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)-graph of Figure 3(a).

4.2 Hierarchical Forest Automata

We now define hierarchical forest automata that represent sets of hierarchical hypergraphs. The hierarchical FA are FA whose alphabet can contain symbols which encode boxes appearing on edges of hierarchical hypergraphs. The boxes are themselves represented using hierarchical FA.

To define an alphabet of hierarchical FA, we will take an approach similar to the one used for the definition of hierarchical hypergraphs. First, we define an operator $\mathbb{A}$ which for a hypergraph alphabet Υ returns the ranked alphabet containing the set of all SFA $\mathcal{S}$ over (a finite subset of) Υ such that $L(\mathcal{S})$ is an Υ-labelled box and such that $\mathbb{A}[\Upsilon] \cap \Upsilon = \emptyset$. The rank of $\mathcal{S}$ in the alphabet $\mathbb{A}[\Upsilon]$ is the rank of the box $L(\mathcal{S})$. The operator $\mathbb{A}$ gives rise to a hierarchy of alphabets $\mathbf{\Gamma}_0, \mathbf{\Gamma}_1, \ldots$ where:

- $\mathbf{\Gamma}_0 = \Gamma$ is the set of *plain symbols*,
- for $i \geq 0$, $\mathbf{\Gamma}_{i+1} = \mathbf{\Gamma}_i \cup \mathbb{A}[\mathbf{\Gamma}_i]$ is the set of *symbols of level* $i + 1$.

A hierarchical FA $\mathcal{F}$ over $\mathbf{\Gamma}_i$ is then called a Γ-labelled *(hierarchical) FA of level* i, and we refer to the hierarchical SFA over $\mathbf{\Gamma}_{i-1}$ appearing within alphabet symbols of $\mathcal{F}$ as to *nested SFA of* $\mathcal{F}$.

Let $\mathcal{F}$ be a hierarchical FA. We now define an operator $\sharp$ that translates any $\mathbf{\Gamma}_i$-labelled hypergraph $G = (V, E, P) \in L(\mathcal{F})$ to a Γ-labelled hierarchical hypergraph H of level i (i.e., it translates G by transforming the SFA that appear on its edges to the boxes they represent). Formally, $G^\sharp$ is defined inductively as the Γ-labelled hierarchical hypergraph $H = (V, E', P)$ of level i that is obtained from the hypergraph G by replacing every edge $(v, \mathcal{S}, \overline{v}) \in E$, labelled by a Γ-labelled hierarchical SFA $\mathcal{S}$, by the edge $(v, L(\mathcal{S})^\sharp, \overline{v})$, labelled by the box $L(\mathcal{S})^\sharp$ where $L(\mathcal{S})^\sharp$ denotes the set (box) $\{X^\sharp \mid X \in L(\mathcal{S})\}$. Then, we define the semantics of a hierarchical FA $\mathcal{F}$ over Γ as the set of Γ-labelled (plain) hypergraphs $[\![\mathcal{F}]\!] = [\![L(\mathcal{F})^\sharp]\!]$.

Notice that a hierarchical SFA of any level has finitely many nested SFA of a lower level only. Therefore, a hierarchical SFA is a finitely representable object. Notice also that even though the maximum number of cut-points of hypergraphs from $L(\mathcal{S})^\sharp$ is fixed (SFA always accept hypergraphs with a fixed maximum number of cut-points), the number of cut-points of hypergraphs in $[\![\mathcal{S}]\!]$ may be unbounded. The reason is that hypergraphs from $L(\mathcal{S})^\sharp$ 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 $[\![\mathcal{S}]\!]$, but they are not visible at the level of hypergraphs from $L(\mathcal{S})^\sharp$.

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 enforce a restricted form of hierarchical automata that rules out some rather artificial scenarios and that allows 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, the restricted form guarantees that:

1. For a hierarchical hypergraph H, well-connectedness of hypergraphs in $[\![H]\!]$ is equivalent to the so-called box-connectedness of H. Box-connectedness is a property introduced below that can be easily checked and that basically considers paths from input ports to output ports and vice versa, in the latter case through hyperedges hidden inside nested boxes.
2. Determinism of hypergraphs from $[\![H]\!]$ implies determinism of H.

The two above properties simplify checking inclusion and well-connectedness considerably since for a general hierarchical hypergraph H, well-connectedness of H is neither implied nor it 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 (including paths from an output port to another output port not passing the input port), but it may be missing paths from the input port to some of the output ports.

Using the above restriction, we will show below a safe approximation of inclusion checking on hierarchical SFA, and we will also show that this approximation

is precise in some cases. Despite the introduced restriction, the description is quite technical, and it may be skipped on the first reading. Indeed, it turns out that in practice, an even more aggressive approximation of inclusion checking in which nested boxes are taken as atomic symbols is often sufficient.

Properness and Box-connectedness. Given a Γ-labelled component C of level 0, we define its *backward reachability set* $br(C)$ as the set of indices i for which there is 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 $C_1, C_2 \in [\![B]\!]$, 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 $i \in br(C)$.[7]

For a proper box B, we use $br(B)$ to denote $br(C)$ for $C \in [\![B]\!]$. A hierarchical hypergraph H is called *well-formed* iff all its nested boxes are proper. In that case, the conditions above imply that either all or no hypergraphs from $[\![H]\!]$ are well-connected and that well-connectedness of hypergraphs 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, Point 2 in the above definition of proper boxes guarantees that we do not have to take into account paths that interconnect output ports of B). This is formalised below.

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

Proposition 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 $\mathcal{S}$ proper iff it represents a proper box $[\![\mathcal{S}]\!]$, we let $br(\mathcal{S}) = br([\![\mathcal{S}]\!])$, and for a $\Gamma \cup Y$-labelled hypergraph G where Y is a set of proper SFA, its backward reachability hypergraph G^{br} is defined based on br in the same way as the backward reachability hypergraph of a hierarchical hypergraph above (just instead of boxes, we deal with their SFA representations). We also say that G is box-connected iff G^{br} is well-connected.

Checking Properness and Well-connectedness. We now outline algorithms for checking properness of nested SFA and well-connectedness of SFA.

Properness of nested SFA can be checked relatively easily since we can take advantage of the fact that nested SFA of a proper SFA must be proper as well.

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

We start with nested SFA of level 0 which contain no nested SFA, we check their properness and compute the values of the backward reachability function br for them. To do this we can label TA states similarly to Section 3.6. A unique label of each root in the SFA representing the box guarantees that the br function will be equal for all hypergraphs hidden in the box. Then, we iteratively increase the level j and for each j, we check properness of the nested SFA of level j and compute the values of the function br. For this, we use the values of br that we have computed for the nested SFA of level $j-1$, and we can also take advantage of the fact that the nested SFA of level $j-1$ have been shown to be proper. We can again use the labels attached to all tree automata states. The difference from level 0 is that we have to extend the labels in order to capture also the backward reachability of the edges labelled by nested SFA.

Now, given an FA $\mathcal{F}$ over Γ with proper nested SFA, we can check well-connectedness of hypergraphs from $[\![\mathcal{F}]\!]$ as follows: (1) for each nested SFA $\mathcal{S}$ of $\mathcal{F}$, we compute like above (and cache for further use) the value $br(\mathcal{S})$, and (2) using this value, we check box-connectedness of hypergraphs in $L(\mathcal{F})$ without a need of reasoning about the inner structure of the nested SFA [12].

The Problem of Checking Inclusion on Hierarchical FA. 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 $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$, is a hard problem, even under the assumption that nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ are proper. Its decidability is not known. In this paper, we choose a pragmatic approach and give only a semi-algorithm that is efficient and works well in practical cases. The idea is simple. Since the implications $L(\mathcal{F}) \subseteq L(\mathcal{F}') \implies L(\mathcal{F})^\sharp \subseteq L(\mathcal{F}')^\sharp \implies [\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ obviously hold, we may safely approximate the solution of the inclusion problem by deciding whether $L(\mathcal{F}) \subseteq L(\mathcal{F}')$ (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 $[\![\mathcal{F}]\!]$ and $[\![\mathcal{F}']\!]$ 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 3.5, based on a canonical forest representation and canonicity respecting FA, since they rely on well-connectedness of hypergraphs from $L(\mathcal{F})$ and $L(\mathcal{F}')$, which is now *not* necessarily the case. The reason is that hypergraphs represented by a not well-connected hierarchical hypergraph H can themselves still be well-connected via backward links hidden in boxes. However, by Proposition 3, every hypergraph G from $L(\mathcal{F})$ or $L(\mathcal{F}')$ is box-connected, and both G and G^{br} are deterministic. As we show below, these properties are still sufficient to define a canonical forest representation of G, 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 $F = (T_1, \ldots, T_n, R)$ of a $\Gamma \cup Y$-labelled hypergraph $G = \otimes F$ which is box-connected and such that both G and G^{br} are deterministic. By extending the approach used in Section 3.5, this will be achieved via an unambiguous definition of the *root-points* of G, i.e., the nodes of G that correspond to the roots of the trees $T_1, \ldots, T_n$, and their ordering.

The root-points of G 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 G that are not reachable from root-points of Type 1 (they are only backward reachable). However, not every such part of G 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 G 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 $\preceq_G$ on nodes of G and choose the smallest node wrt. this ordering to be a *root-point of Type 3*. After unambiguously determining all root-points of G, we may order them according to $\preceq_G$, and we are done.

A suitable total ordering $\preceq_G$ on V can be defined taking advantage of the fact that G^{br} is well-connected and deterministic. Therefore, it is obviously possible to define $\preceq_G$ 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 [12].

We say that a hierarchical FA $\mathcal{F}$ over Γ with proper nested SFA and such that hypergraphs from $[\![\mathcal{F}]\!]$ are deterministic and well-connected *respects canonicity* iff each forest $F \in L_F(\mathcal{F})$ 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 proposition.

Proposition 4 *Let* $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, R)$ *and* $\mathcal{F}' = (\mathcal{A}'_1, \ldots, \mathcal{A}'_m, R')$ *be hierarchical CFA. Then,* $L(\mathcal{F}) \subseteq L(\mathcal{F}')$ *iff* $n = m$, $R = R'$, *and* $\forall 1 \leq i \leq n : \ L(\mathcal{A}_i) \subseteq L(\mathcal{A}'_i)$.

Proposition 4 allows us to safely approximate inclusion of the sets of hypergraphs encoded by hierarchical FA (i.e., to safely approximate the test $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ 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 below. A generalisation 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 [12].

Precise Inclusion on Hierarchical FA. In many practical cases, approximating the inclusion $[\![\mathcal{F}]\!] \subseteq [\![\mathcal{F}']\!]$ by deciding $L(\mathcal{F}) \subseteq L(\mathcal{F}')$ is actually precise. A condition that guarantees this is the following:

Condition 1. $\forall H \in L(\mathcal{F})^\sharp \ \forall H' \in L(\mathcal{F}')^\sharp : H \neq H' \implies [\![H]\!] \cap [\![H']\!] = \emptyset$. Intuitively, this means that one cannot have two distinct hierarchical hypergraphs representing the same plain hypergraph.

Clearly, Condition 1 holds if the following two more concrete conditions hold:

Condition 2. Nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ represent a set of boxes X that *do not overlap*.

Condition 3. Every $H \in L(\mathcal{F})^\sharp \cup L(\mathcal{F}')^\sharp$ is *maximally boxed* by boxes from X.

The notions of maximally boxed hypergraphs and non-overlapping boxes are defined as follows. A hierarchical hypergraph H is *maximally boxed* by boxes from a set X iff all its nested boxes are from X, and no part of H can be "hidden" in a box from X, this is, there is no hypergraph G and no component $C \in B, B \in X$ such that $G[C/e] = H$ for some edge e of G. Boxes from a set of boxes X over Γ *do not overlap* iff for every hypergraph G over Γ, there is only one hierarchical hypergraph H over Γ which is maximally boxed by boxes from X and such that $G \in [\![H]\!]$.

We note that the boxes represented by the nested SFA that appear in the case studies presented in this paper satisfy Conditions 2 and 3, and so Condition 1 is satisfied too. Hence, inclusion tests performed within our case studies are precise.

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 `x = y`, `x = y->s`, `x = null`, `x->s = y`, `malloc(x)`, and `free(x)` together with pointer and data equality tests and common control flow statements as discussed in more details below[8]. Each allocated cell may have several next pointer selectors and contain data from some finite domain[9]. We use *Sel* to denote the set of all selectors and *Data* to denote 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 the hypergraph, the 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[10]. 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 G (i.e., to a single configuration) returns a hypergraph $a(G)$ that is obtained from G as follows. Non-destructive actions `x = y`, `x = y->s`, or `x = 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 `x->s = 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`),

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

[9] No abstraction for such data is considered.

[10] 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.

and `x->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 G amounts to a simple test of equality of nodes or equality of data fields of nodes. Dereferences of `null` and `undef` are of course detected (as an attempt to follow a non-existing hyperedge) and an error is announced. Emergence of garbage is detected iff $a(G)$ is not well-connected.[11]

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 $\mathcal{S}$, we need to symbolically compute an SCFA $x(\mathcal{S})$ s.t. $[\![x(\mathcal{S})]\!]$ equals $\{x(G) \mid G \in [\![\mathcal{S}]\!]\}$ if x is an action and $\{G \in [\![\mathcal{S}]\!] \mid x(G)\}$ if x is a guard.

Derivation of the SCFA $x(\mathcal{S})$ from $\mathcal{S}$ 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 [12]. 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 the operation of 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 appropriate 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. Note that this may be further improved by using language inclusion instead of isomorphism of automata.

A simplified example of a symbolic execution is provided in Figure 4. In the left part of the figure, we provide concrete heaps (the dashed edges represent sequences of one or more edges linked into a linked-list), and in the right part, we provide their forest automata representation (for a better readability, top-down tree automata are used). The initial configuration is depicted in Fig. 4(a), and Figure (b), (c), and (d) represent the sets of heaps obtained after successively applying the statements `x` $=$ `y->next`, `x->next` $=$ `z`, and `z` $=$ `x`.

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. The function f_x takes the

[11] 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.

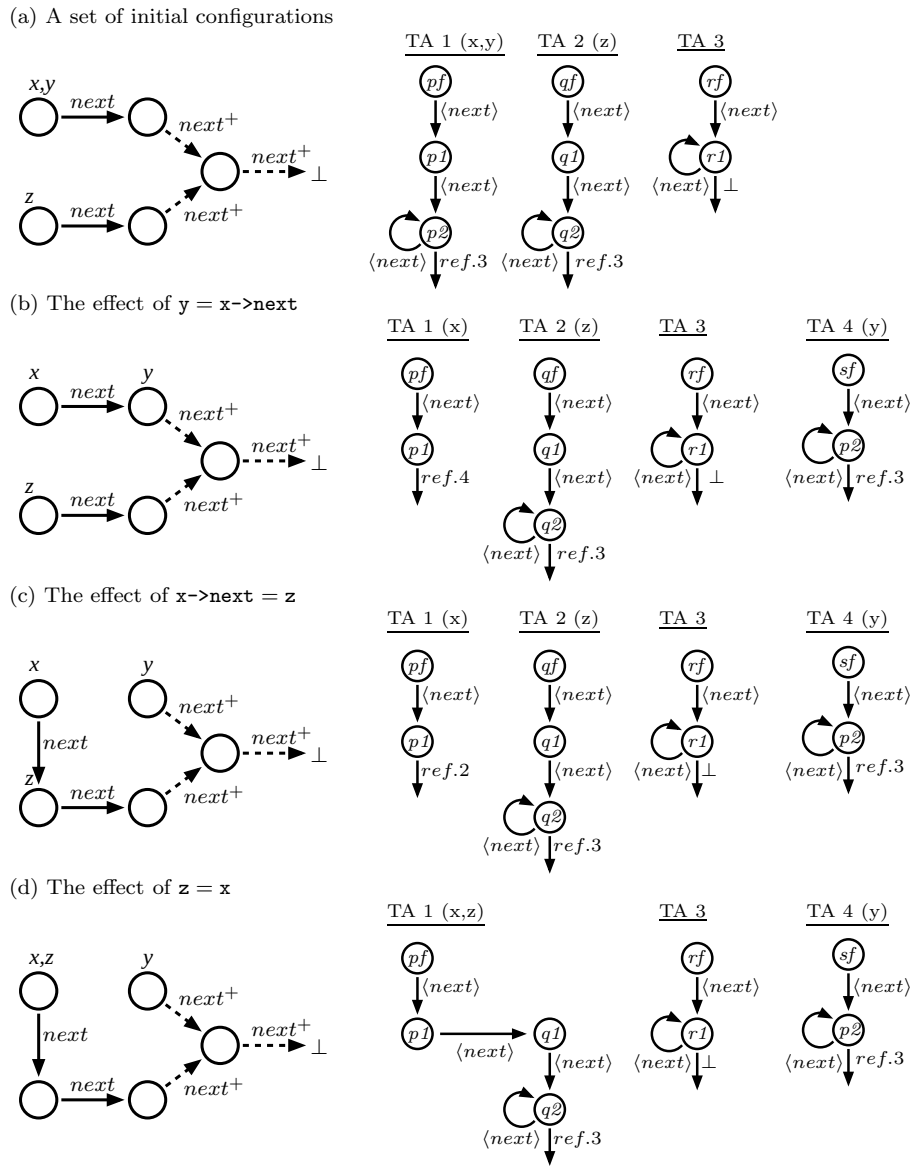

Fig. 4 A concrete (on the left) and symbolic execution (on the right) of statements `y = x->next`, `x->next = z`, and `z = x`. For the sake of simplicity, the presented FA are not strictly in their canonical form.

flow value (SCFA) $\mathcal{S}$ at l as its input and (1) computes the SCFA $x(\mathcal{S})$, (2) applies *abstraction* to $x(\mathcal{S})$, and returns the result.

The abstraction may be implemented by applying the general techniques described in the framework of abstract regular tree model checking [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, one can 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 safely approximate intersections of hierarchical SFA. More precisely, given SCFA $\mathcal{S}_1$ and $\mathcal{S}_2$, one can compute an SCFA $\mathcal{S}$ such that $[\![\mathcal{S}]\!] \subseteq [\![\mathcal{S}_1]\!] \cap [\![\mathcal{S}_2]\!]$. This under-approximation is safe since it can lead neither to false positives nor to false negatives (it can only cause the computation not to terminate). Moreover, for the SCFA that appear in the case studies in this paper, the intersection we compute is actually precise. More details can be found in [12].

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 a prototype, the results are very 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-linked, 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 the performance of our tool with that of Space Invader [4], the first fully automated tool based on separation logic, Predator [10], a new fully automated tool based in principle on separation logic (although it represents sets of heaps using graphs), and also with the ARTMC tool [7] based on abstract regular tree model checking[12]. The comparison with Space Invader and Predator was done on examples with lists only since Invader and Predator do not handle trees. The higher flexibility of our automata abstraction shows up, for example, in the test case with a list of sublists of lengths 0 or 1 (discussed already in the introduction) for which Space Invader does not terminate. Our technique handles this example smoothly (without any need to add special inductive predicates that could decrease the performance or generate false alarms). Predator can also handle this test case, but to achieve that, the algorithms implemented in it must have been manually extended to use a new kind of list segment of length 0 or 1, together with an appropriate modification of the implementation of Predator's join and abstraction operations[13]. On the other hand, 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.

Table 1 summarises running times (in seconds) of the four tools on our case studies. The value T means that the running time exceeded 30 minutes, o.o.m.

[12] Since it is quite difficult to encode the input for ARTMC, we have tried it on some interesting cases only.

[13] The operations were carefully tuned not to easily generate false alarms, but the risk of generating them has anyway been increased.

Table 1 Experimental results

Example	Forester	Invader	Predator	ARTMC
SLL (delete)	0.01	0.10	0.01	0.50
SLL (reverse)	< 0.01	0.03	< 0.01	
SLL (bubblesort)	0.02	Err	0.02	
SLL (insertsort)	0.02	0.10	0.01	
SLL (mergesort)	0.07	Err	0.13	
SLL of CSLLs	0.07	T	0.12	
SLL+head	0.01	0.06	0.01	
SLL of 0/1 SLLs	0.02	T	0.03	
SLL_{Linux}	< 0.01	T	< 0.01	
DLL (insert)	0.02	0.08	0.03	0.40
DLL (reverse)	0.01	0.09	0.01	1.40
DLL (insertsort1)	0.20	0.18	0.15	1.40
DLL (insertsort2)	0.06	Err	0.03	
CDLL	< 0.01	0.09	< 0.01	
DLL of CDLLs	0.18	T	0.13	
SLL of $2CDDLs_{Linux}$	0.03	T	0.19	
tree	0.06			3.00
tree+stack	0.02			
tree+parents	0.10			
tree (DSW)	0.16			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. In particular, "SLL" stands 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 implementation of lists used in the Linux kernel with restricted pointer arithmetic [10] which we can also handle. All experiments start with a random creation and end with a disposal of the specified structure. If some further operation is performed in between the creation phase and the disposal phase, it is indicated in brackets. 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 and tested the approach on multiple non-trivial cases studies, demonstrating the approach to be promising.

In the future, we plan to improve the implementation of our tool Forester, including a support for predicate language abstraction within abstract regular tree model checking [6]. We also plan to implement the 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. P.A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic Abstraction for Programs with Dynamic Memory Heaps. In *Proc. of CAV'08*, *LNCS* 5123, Springer, 2008.
2. P.A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. Computing Simulations over Tree Automata: Efficient Techniques for Reducing TA. In *Proc. of TACAS'08*, *LNCS* 4963, 2008.
3. P.A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When Simulation Meets Antichains (On Checking Language Inclusion of NFAs). In *Proc. of TACAS'10*, *LNCS* 6015, Springer, 2010.
4. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. CAV'07*, *LNCS* 4590, Springer, 2007.
5. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, *LNCS* 4144, Springer, 2006.
6. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS* 149(1), Elsevier, 2006.
7. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, *LNCS* 4134, Springer, 2006.
8. C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-abduction. In *Proc. of POPL'09*, ACM Press, 2009.
9. J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, *LNCS* 3920, Springer, 2006.
10. 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, Springer, 2011.
11. B. Guo, N. Vachharajani, and D.I. August. Shape Analysis with Inductive Recursion Synthesis. In *Proc. of PLDI'07*, ACM Press, 2007.
12. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. 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`
13. P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. In *Proc. of POPL'11*, ACM Press, 2011.
14. A. Møller and M. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*, ACM Press, 2001.
15. H. H. Nguyen, C. David, S. Qin, and W. N. Chin Automated Verification of Shape and Size Properties via Separation Logic. In *Proc. of VMCAI'07*, *LNCS* 4349, Springer, 2007.
16. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*, IEEE CS, 2002.
17. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
18. H. Yang, O. Lee, C. Calcagno, D. Distefano, and P.W. O'Hearn. On Scalable Shape Analysis. Technical report RR-07-10, Queen Mary, University of London, 2007.
19. 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.
20. K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *Proc. of PLDI'08*, ACM Press, 2008.