Forest Automata for Verification of Heap Manipulation

FIT BUT Technical Report Series

Peter Habermehl, Lukáš Holík, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar

Technical Report No. FIT-TR-2011-001
Faculty of Information Technology, Brno University of Technology

Last modified: December 15, 2011

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.

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* that are nodes pointed to by a program variable or having several incoming edges. The tree components can refer to the boundaries of each other and hence they are "separated" much like heaps described by formulae joined by the separating conjunction in separation logic [14]. Sets of heaps with a bounded number of cut-points are then represented by the newly

proposed *forest automata* (FA) that are basically tuples of TA accepting trees whose leaves can refer back to the roots of any of the trees accepted by these automata. 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, as FA are not closed under union, we work with sets of forest automata, corresponding to, e.g., 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., [12, 15, 14, 4, 10, 13, 18, 17, 8, 11], 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, 17] 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, 17], 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., [13], 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[5]).

Further, compared to [4, 17], 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 [16]. In such cases, the abstraction used in [4, 17] 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], the newly proposed approach is a bit less general (we cannot, e.g., handle structures such as trees with linked leaves[6]), 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, 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 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

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

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

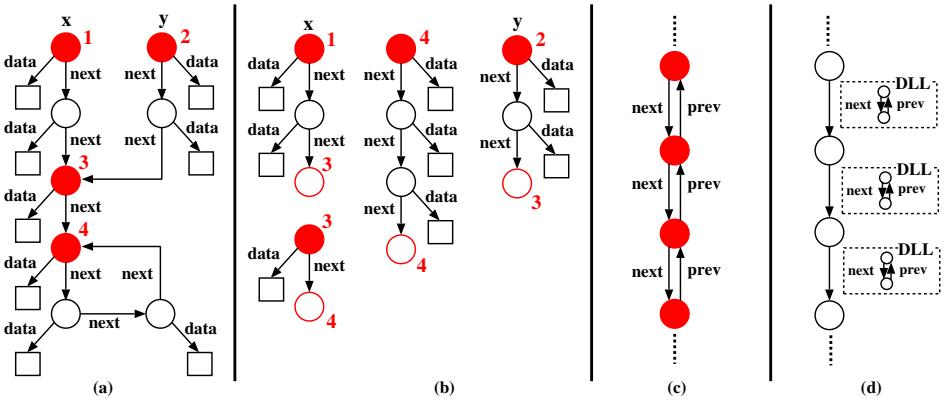

Fig. 1. (a) A heap graph with cut-points highlighted in red, (b) the canonical tree decomposition of the heap with x ordered before y, (c) a part of a DLL, (d) a hierarchical encoding of the DLL

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) which encode sets of heaps represented in a canonical way: if we select a tuple of trees from the languages of the given TA, construct a heap from them, and then canonically decompose it, we get the tuple of trees we started with. 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[7]. 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.[8] 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.[9]

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 $1 \leq i \leq n$. We call a set A *ranked* if there is a function $\# : A \to \mathbb{N}$. The value $\#(a)$ is called the *rank*

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

[8] 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.

[9] 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.

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[10]. 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 : $\forall i, j \in (I_F \cup O_F) : \ root(T_i) \preceq_p root(T_j) \iff i \leq j$.

[10] 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 Appendix A.1.

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 $t \xRightarrow{\pi} q$ 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 $t \xRightarrow{\pi} q$ 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* $\mathcal{L}_F(\mathcal{F}) = \{(T_1, \ldots, T_n, I, O) \mid (\forall 1 \leq i \leq n : ot(T_i) \in \mathcal{L}(\mathcal{A}_i)) \wedge (\forall 1 \leq i < j \leq n : T_i = (V_i, E_i) \wedge T_j = (V_j, E_j) \implies V_i \cap V_j = \emptyset)\}$. 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. In particular, we will restrict ourselves to working with non-empty SFA consisting of FA with non-empty languages only.

Note that any FA can be transformed (split) into an SCFA S. Each CFA of S represents hypergraphs having the same interconnection of the cut-points (see Appendix A.2 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 $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, I, O)$ 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 $\curlywedge_n^{I,O}(q_1, \ldots, q_n) \to q^{top}$ for each $(q_1, \ldots, q_n) \in F_1 \times \cdots \times F_n$. 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. To define this formally, we use an auxiliary operation *plug*. Let $e = (v, a, \bar{v}) \in E$ 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 $(o_1, \ldots, o_k)$ 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 $w = o_i, 1 \leq i \leq k$, $\rho(w) = \bar{v}.i$, 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 $E'' = (E \setminus \{e\}) \cup \{(v'', a', \bar{v}'') \mid \exists (v', a', \bar{v}') \in E' : \ \rho(v') = v'' \ \wedge \ \forall 1 \leq i \leq k : \ \rho(\bar{v}'.i) = \bar{v}''.i\}$. Now, the semantics of a hypergraph $H = (V, E, I, O)$ of level j is defined recursively as follows: Let $Plug(H) = \{plug(H, e, C) \mid e = (v, B, \bar{v}) \in E \ \wedge \ B \in \mathbb{B}_{j-1} \ \wedge \ C \in [\![B]\!]\}$. 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 $E = \{(v,a,\overline{v}) \mid (v,a,\overline{v}) \in E' \wedge a \in \Gamma\} \cup \{(v,\mathcal{L}_H(S),\overline{v}) \mid (v,S,\overline{v}) \in E' \wedge S \in X\}$.

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 $\mathcal{L}_H(S)$ (and $\mathcal{L}(S)$) 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 $\mathcal{L}(S)$ and $\mathcal{L}_H(S)$.

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.[11]

[11] 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.

Proper boxes and well-formed hypergraphs. Given a component C over Γ, we denote by $br(C)$ the set of indices i such that there is a path from the i-th output port of C to the input port of C—intuitively, $i \in br(C)$ means that the input port is *backward reachable* from the i-th output port. 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 $i \in br(C)$.[12] 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. This computation may easily be done by inspecting rules of the component TA of $\mathcal{F}$. Likewise, properness of nested SFA may be checked on the level of TA too—see Appendix A.3 for more details.

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

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.

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

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 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. More details on how $\preceq_H$ may be defined and used when dealing with sets of hypergraphs represented by hierarchical FA are a part of Appendix A.4.

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)* $\forall 1 \leq i \leq n : \mathcal{L}(\mathcal{A}^1_i) \subseteq \mathcal{L}(\mathcal{A}^2_i)$.

Language inclusion of sets of hierarchical CFA is handled in the same way as inclusion of ordinary SFA. Hierarchical FA that do not respect canonicity may be algorithmically split into several hierarchical CFA, similarly as ordinary CFA (see Appendix A.4 for more details).

Finally, note that despite we have not provided a way of precise inclusion checking for hierarchical SFA, it turns out that the described way of approximate inclusion checking is precise at least in some cases as discussed in Appendix A.5 (and the precision turns out to be sufficient for our case studies too).

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. Each allocated cell may have several next pointer selectors and contain data from some finite domain (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[13]. 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 by the following simple manipulation (we consider only the basic actions to which all the rest can be reduced). Nondestructive 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

[13] 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.

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 x->data = d_{new} replaces the edge (v_x, d_{old}) by the edge (v_x, 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 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.[14]

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 appearing on transitions adjacent with states accepting root-points. In the next phase, by splitting some of the component TA, we make every node referred to by x in the represented heaps an additional root-point every time such a node is currently not a root-point (cf. Appendix B for some more details). Third, we perform the actual update, which due to the previous step amounts to manipulation with TA transition rules adjacent with the states accepting root-points only (see Appendix B 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.

[14] 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.

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 the abstraction, we use a *backward symbolic execution* like 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 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 Appendix C.

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 size and testing language inclusion [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.

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

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	
SLL_{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 $2CDLLs_{Linux}$	0.11	T	
tree	0.11		3	tree+stack	0.10		
tree+parents	0.18			tree (DSW)	0.41		o.o.m.

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.

Acknowledgements 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.

References

1. P.A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, 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, 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, T. Vojnar. When Simulation Meets Antichains (On Checking Language Inclusion of NFAs). In *Proc. of TACAS'10*, *LNCS* 6015.
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, T. Vojnar. Abstract Regular Tree Model Checking. *ENTCS* 149(1), Elsevier, 2006.
7. A. Bouajjani, P. Habermehl, A. Rogalewicz, 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. B. Guo, N. Vachharajani, and D.I. August. Shape Analysis with Inductive Recursion Synthesis. In *Proc. of PLDI'07*, ACM Press, 2007.
11. P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. In *Proc. of POPL'11*, ACM Press, 2011.
12. A. Møller and M. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*, ACM Press, 2001.
13. 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.
14. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*, IEEE CS, 2002.
15. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
16. 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.
17. 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.
18. K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *Proc. of PLDI'08*, ACM Press, 2008.

A Forest Automata

A.1 Converting Unordered Trees to Ordered Trees

Let a Γ-labelled tree $T = (V,E)$ be given with $\preceq_\Gamma \subseteq \Gamma \times \Gamma$ being a total ordering on Γ. For a node $u \in V$, we let $a(u) = \{a \in \Gamma \mid \exists \overline{u} \in V^{\leq \#(\Gamma)} : \ (u,a,\overline{u}) \in E\}$, and for all $i \in \{1,\ldots,|a(u)|\}$, we let $ai(u,i) = a \in a(u) \iff |\{a' \in a(u) \mid a' \preceq_\Gamma a\}| = i$. Intuitively, $a(u)$ is the set of labels of edges leaving a node u, and $ai(u,i)$ gives the label of the i-th edge leaving u w.r.t. $\preceq_\Gamma$. We define Σ_Γ to be the alphabet 2^Γ ranked such that for $A \subseteq \Gamma$, $\#(A) = \Sigma_{a \in A} \#(a)$. We say that an ordered tree $t : \mathbb{N}^* \to \Sigma_\Gamma$ is an *ordered representation* of T iff there is a bijection $ot : V \to dom(t)$ such that $\forall u \in V \ \forall p \in \mathbb{N}^* : \ ot(u) = p \implies t(p) = a(u) \wedge \forall i \in \{1,\ldots,|a(u)|\} \ \forall \overline{u} \in V^{\leq \#(\Gamma)} : \ (u,ai(u,i),\overline{u}) \in E \iff \forall 1 \leq j \leq \#(ai(u,i)) : ot(\overline{u}.j) = p(\Sigma_{k=1}^{i-1} \#(ai(u,k)) + j - 1)$. Note that, in fact, there is a unique ordered representation of T which—referring to the appropriate bijection ot that is also unique—we may denote $ot(T)$ with a slight abuse of the notation.

A.2 Transforming FA into Canonicity Respecting FA

For transforming an FA into an SCFA, we first label the states of the component TA of the given FA by labels capturing in which possible orders root references appear in the leaves of the trees accepted at these states (and 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 roots of some of the trees accepted by some component TA need not be cut-points (and up to the ordering of the component TA). That is why, 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). For ordering the component TA, the labels of the accepting states can be conveniently used.

To be more precise, consider a forest automaton $\mathcal{F} = (\mathcal{A}_1,\ldots,\mathcal{A}_n,I,O)$, $n \geq 1$, and any of its component tree automata $\mathcal{A}_i = (Q_i,\Sigma,\Delta_i,F_i)$, $1 \leq i \leq n$. We can 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 $t \in \mathcal{L}(q)$ 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 the 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.[15] 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 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. This finally leads to a set of canonicity respecting FA.

Note that the above construction may sometimes unnecessarily split some component TA. This may happen, e.g., if the first component TA accepts trees referring once to the second component, and the second component TA accepts trees that refer either once or twice to the first component and then to the second component. However, this does not contradict with the fact that canonicity respecting forest automata representing the original set of hypergraphs are obtained using the above construction.

Also 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 statements (using labelling of states that is stored with the component TA and incrementally updated

[15] 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 only for a single labelling of accepting states in each of the copies.

when computing new component TA from the existing ones).[16] Therefore, transforming an FA obtained this way into the corresponding CFA involves only the elimination of redundant roots and the root reordering.

A.3 Checking Properness and Box-Connectedness

Properness of nested SFA may be checked relatively easily since we may take advantage of the fact that nested SFA of a proper SFA must be proper as well. 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. 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 the advantage of the fact that the nested SFA of level $j-1$ have been shown to be proper.

Let us have a canonicity respecting SFA S of some level such that its nested SFA are proper and we know the value of the function br for all of them.[17] We assume that S contains at least one FA, and that the languages of all $\mathcal{F} \in S$ are nonempty.

Moreover, to make the algorithms of checking properness and box-connectedness faster and simpler, we exploit the fact that the algorithm we describe in Section A.4 in fact produces automata respecting canonicity in a somewhat stronger sense than described in Section 4.3. We define the stronger notion of respecting canonicity below.

Given $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, I, O) \in S$, a *path* in a tree $t \in \mathcal{L}(\mathcal{A}_i)$ from v to w, $v, w \in dom(t)$, is a sequence $v = v_0, (a_1, k_1), v_1 \ldots, (a_m, k_m), v_m = w, 0 \leq m$, where for each $1 \leq i \leq m$, v_i is the k_i-th son of v_{i-1} and $t(v_i) = a_i$. The path is *backward passable* iff for each $1 \leq i \leq m$, the label a_i is backward passable at the k_i-th position, which means that there is a proper nested SFA $S_i \in a_i$ and $j \in br(S_i)$ such that $j + \sum_{\{b \in a_i \mid b \preceq_\Gamma S_i, b \neq S_i\}} \#(b) = k_i$.

For each $1 \leq i \leq n$ and each $t \in \mathcal{L}(\mathcal{A}_i)$, we define the *reachability relation* $\rho_i^t \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$ on the roots of $\mathcal{F}$ that contains a pair (j,k) iff one of the following three conditions holds:

1. $i = j$ and there is a leaf v of t with $t(v) = k$, or
2. $i = k$, there is a leaf v of t with $t(v) = j$, and the path from the root of t to v is backward passable, or
3. there are nodes u, v, w of t such that both v and w are leaves of the subtree rooted by u, $t(v) = j$, $t(w) = k$, and the path from u to v is backward passable.

[16] Moreover, note that the need to split automata appears only when some additional component TA are introduced in the second step of the symbolic execution of pointer manipulating statements (cf. Section 5 or Appendix B).

[17] Notice that even though respecting canonicity assumes properness of nested SFA and we require here proper SFA to be respect canonicity, this is not a circular dependency. For an SFA to respect canonicity, we only require its nested SFA to be proper. So, for an SFA of level j to respect canonicity, we require properness of SFA of level $j-1$ only. Respecting canonicity in an SFA of level 0 does not depend on the notion of properness since SFA of level 0 have no nested SFA. Properness on level j then depends on respecting canonicity on level j.

We say that $\mathcal{F}$ is an *FA with uniform reachability* iff for each $1 \leq i \leq n$, ρ_i^t is the same for all $t \in \mathcal{L}(\mathcal{A}_i)$. If it is the case, then we denote the reachability relation as ρ_i. We further say that an SFA S *strongly respects canonicity* iff it respects canonicity as defined in Section 4.3 and all its elements are FA with uniform reachability. In Section A.4, we show how to transform FA into SFA that strongly respect canonicity and we also show how to compute the relation ρ_i.

If $\mathcal{F}$ is an FA with uniform reachability, we define the *global reachability relation* $\rho = (\bigcup_{1 \leq i \leq n} \rho_i)^*$ on roots of $\mathcal{F}$.[18] Notice that $(i,j) \in \rho$ iff for all $H \in [\![\mathcal{L}_H(\mathcal{F})]\!]$ there are two nodes u and v that correspond to the i-th and j-th root of $\mathcal{F}$, respectively, and such that there is a path from u to v in H.

Properness of an SFA representing some box and computing br on it is then done as follows. First, a singleton SFA $\{\mathcal{F}\}$ with ι being the only input port of $\mathcal{F}$ is proper iff (1) for all $1 \leq i \leq n$, $(\iota, i) \in \rho$ and (2) for all $o, o' \in O$, $(o, o') \in \rho \implies (o', \iota) \in \rho$. If $\{\mathcal{F}\}$ is proper, then $br(\{\mathcal{F}\})$ equals the set $\{o \in O \mid (o, \iota) \in \rho\}$. Finally, assuming that an SFA S strongly respects canonicity, S is proper iff all its elements agree on the values of I and O, and all the singleton SFA $\{\mathcal{F}\}$, $\mathcal{F} \in S$, are proper and agree on the value of $br(\mathcal{F})$. This value then equals $br(S)$.

Box-connectedness of an SFA S that strongly respects canonicity and that has proper nested SFA for which we know the values of br can be checked similarly as properness, i.e., using the relation ρ. Particularly, S is box-connected if and only if for all $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, I, O) \in S$ and for all $1 \leq k \leq n$ there is some $\iota \in I$ such that $(\iota, k) \in \rho$.

A.4 Transforming Hierarchical FA into Canonicity Respecting Hierarchical FA

The labelling considered in Section A.2 when transforming (non-hierarchical) FA into sets of canonicity respecting FA does not cover the cases when the nodes which are the roots of some tree components are reachable from nodes pointed by program variables only when considering backward reachability through boxes. We solve this problem by extending the labelling from Section A.2 as described below. Consider a hierarchical forest automaton $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, I, O)$, $n \geq 1$, with a set X of nested SFA that are proper and its component tree automaton $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$, $1 \leq i \leq n$. We label each $q \in Q_i$ by a set of extended labels (w, Y, Z_1, Z_2), $w \in \{1, \ldots, n\}^*$, $Y \subseteq \{1, \ldots, n\}$, $Z_1 \subseteq \{1, \ldots, n\} \times X \times \mathbb{N}$, and $Z_2 \subseteq \{1, \ldots, n\} \times X \times \mathbb{N} \times \{1, \ldots, n\}^*$ for which there is a tree $t \in \mathcal{L}(q)$ such that

- w and Y are as in the non-hierarchical case.
- $(r, S, i) \in Z_1$ iff there is a backward passable[19] path from a leaf labelled by the root reference r to the root of the tree t, and this leaf is an S_i-successor of some node in the unordered tree t' where $ot(t') = t$.
- $(r, S, i, w') \in Z_2$ records the fact that in the tree t, there is a subtree t' with the root labelled by (w', Y', Z_1', Z_2') such that $(r, S, i) \in Z_1'$. This labelling is used to resolve cases where there is a backward passable path from a root reference into some intermediate node in the tree, but not to the root of the tree t.

[18] Here, the * stands for the reflexive and transitive closure.

[19] See A.3 for the definition of a backward passable path.

Such labelling can be obtained in a similar way as in the case of non-hierarchical automata—i.e., by first labelling states w.r.t. the leaf rules and then propagating the so far obtained labels bottom-up. Elements of the Z_1 sets are not propagated when a transition rule reads an edge without backward reachability at the concerned position. 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 accepting trees with the appropriate label of the root (in a similar way as in Appendix A.2).

The extended labels guarantee that each FA $\mathcal{F}$ obtained above is an FA with *uniform reachability* (see Appendix A.3). The relation ρ_i can be derived directly from the label of the final states of $\mathcal{A}_i$. Particularly, if the label is (w, Y, Z_1, Z_2), then $(j,k) \in \rho_i$ iff:

1. $i = j$ and k appears in w, or
2. $i = k$ and $(j,S,l) \in Z_1$ for some S and l, or
3. $(j,S,l,w') \in Z_2$ for some S and l such that k appears in w'.

Clearly, each of the FA created above represents a set of hierarchical hypergraphs that have the same number of roots. However, as in the case of non-hierarchical hypergraphs, some roots need not correspond to root-points. This problem is solved in the same way as in Appendix A.2.

In order to transform each obtained FA $\mathcal{F}$ into the (strongly) canonicity respecting form, its component TA are subsequently ordered according to the depth-first traversal on the so-called *root reference reachability graph*. In this graph, nodes correspond to the roots of the forest representation of the hypergraphs encoded by $\mathcal{F}$, and edges represent the reachability relation $\bigcup_{1 \leq i \leq n} \rho_i$. The edges of the root reference reachability graph are labelled by natural numbers using the extended labels as described below. Successors of nodes in the root reference reachability graph are then explored according to these numbers in the depth-first traversal on the graph.

Let us denote by $x_{\mathcal{A}}$ the node of the root reference reachability graph corresponding to a component TA $\mathcal{A}$ (i.e., to the roots of the trees accepted by $\mathcal{A}$). For each component TA $\mathcal{A}$, assuming that its final states are labelled by (w, Y, Z_1, Z_2), we label the edges leading from $x_{\mathcal{A}}$ to the nodes corresponding to TA referenced from w by natural numbers assigned in the order given by w. Then, for each component TA $\mathcal{A}$ of $\mathcal{F}$ whose labelling of final states contains a Z_1 triple (r,S,i) where r references a TA $\mathcal{A}'$, we label the edge leading from $x_{\mathcal{A}'}$ to $x_{\mathcal{A}}$ by a number assigned to the pair (S,i) in the lexicographic ordering on all such pairs that appear in the Z_1 triples of the labels of the component TA of $\mathcal{F}$. (We use numbers greater than those used in the previous phase of numbering). Finally, for each label (r,S,i,w') of the final states of some component TA $\mathcal{A}$, the TA $\mathcal{A}'$ referenced by r, and each TA $\mathcal{A}''$ referenced from w', we label the edge leading from $x_{\mathcal{A}'}$ to $x_{\mathcal{A}''}$ by a number obtained from the lexicographic ordering of the triples (S,i,r') where r' ranges over references that appear in the w' parts of the Z_2 labels. (We again use numbers greater than those used in the previous phases of numbering.) If multiple numbers are assigned to a single edge, the smallest is chosen.

The just described ordering of the component TA of a given FA based on the root reference reachability graph orders the component TA in a way consistent with the order $\preceq_H$ that is induced on the root-points of the represented hypergraphs by the further described deterministic depth-first traversal on the corresponding backward reachability

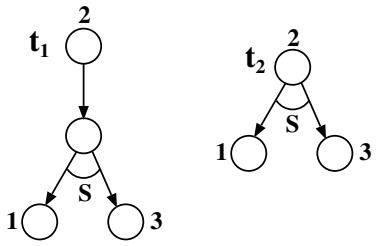

Fig. 2. Two tree components problematic for handling the order of roots

hypergraphs H^{br}. In particular, the corresponding DFT on the backward reachability hypergraphs starts from the input ports and it is driven by the fixed ordering on the input ports and the labels of hyperedges. The ordering of the inverted hyperedges (labelled by symbols from X^{br}) is inherited from the ordering of the hyperedges on which they are based and from the number of the output port used. Moreover, the DFT normally explores first original hyperedges and only then the inverted hyperedges. However, the inverted hyperedges are prioritised whenever the traversal comes to a node x using an inverted hyperedge and x is not a root of some tree. In such a case, the DFT continues the search first by inverted hyperedges and only then by the regular hyperedges.

The above described handling of the inverted hyperedges forces the DFT on backward reachability hypergraphs to reach a root of a tree component before alternating the direction of the DFT inside the tree component. We explain the reason behind this on an example. Suppose that we have a TA accepting tree components t_1 and t_2 rooted at the root-point 2 as depicted in Figure 2. Suppose that all the edges in these trees are backward passable and that the accepting state of this automaton is labelled by the following label $(\text{“13”}, \{1,3\}, \{(1,S,1),(3,S,2)\}, \{(1,S,1,\text{“13”}),(3,S,2,\text{“13”})\})$. Further, assume that the root-point r_1[20] is the only input port of the represented hypergraphs. Hence, the DFT on the described trees will start from the reference to r_1 (represented by the node labelled by 1 in Figure 2). In the backward reachability hypergraph based on t_1, a DFT without the described priority of inverted hyperedges would go from r_1 to the internal node of t_1 using the inverted hyperedge $(S,1)$ and then it would continue back to r_1, backtrack, and then go to r_3 via the different output ports of S. So, the induced ordering of the root-points would be $1 \preceq_H 3 \preceq_H 2$. On the other hand, in the backward reachability hypergraph based on t_2, such a DFT would go from r_1 to r_2 and then to r_3, giving a different induced ordering of the root-points, namely $1 \preceq_H 2 \preceq_H 3$. The described DFT forces the search in the backward reachability hypergraph based on t_1 to continue from the internal node of t_1 by an inverted hyperedge to r_2 and only after that to continue to r_3. So, the induced ordering is the same in both the cases. Note that the same ordering is obtained by the DFT on the root reference graph where the edge from r_1 to r_2 has a bigger priority than the edge from r_1 to r_3.

[20] Let us denote the i-th root point as r_i.

The FA obtained after splitting based on the extended labels described above, removing the redundant roots, and re-arranging the particular TA according to the root reference reachability graph are still not necessarily canonicity respecting. Respecting of canonicity is not guaranteed in cases when there exist *root-points of Type 3* in the represented hypergraphs—i.e., when there exist loops without any incoming edge in these hypergraphs. This situation can easily be detected by looking for a component TA accepting a tree representation of such loops. In particular, this amounts to looking for a component TA accepting trees such that (1) their roots are not ports in the forest representation, (2) they are not referred from the trees accepted by any other TA—this can easily be checked by inspecting the label w of the final states of the other TA, and (3) they contain a single leaf with a root reference to itself—this can be checked by inspecting the labels w and Y of the accepting states of the concerned TA. The canonisation procedure then rearranges the concerned TA such that the root of each accepted tree is the smallest node according to the above described depth-first traversal on H^{br} (i.e., the node that should be the canonically chosen root-point of Type 3). Intuitively, this amounts to a rotation of the concerned backward reachable loops represented by tree automata rules so that the nodes that are identified as the root-points become the roots of the tree representation.

More formally, let $\mathcal{A}_r$ be the TA that we need to rotate, r being the number of $\mathcal{A}_r$ in the concerned FA, and let $(w^{\mathcal{A}_r}, Y^{\mathcal{A}_r}, Z_1^{\mathcal{A}_r}, Z_2^{\mathcal{A}_r})$ be the label associated with the accepting states of $\mathcal{A}_r$. The states of $\mathcal{A}_r$ that accept the nodes that should become the new roots are identified as follows. Let k be the number of the root-point from where the DFT on the represented hypergraphs comes (without passing through any other root-points) to the nodes corresponding to the roots of the tree components represented by $\mathcal{A}_r$. Identifying k is easy since $x_{\mathcal{A}_k}$ is the predecessor of $x_{\mathcal{A}_r}$ in the DFT performed on the root reference reachability graph. The edge from $x_{\mathcal{A}_k}$ to $x_{\mathcal{A}_r}$ exists in the root reference reachability graph due to existence of a backward passable path from the root of each tree represented by $\mathcal{A}_r$ to a root reference to k. Existence of this path is captured by the label $Z_1^{\mathcal{A}_r}$, concretely by an element $(x,S,i) \in Z_1^{\mathcal{A}_r}$.[21]

Now, the TA $\mathcal{A}_r = (Q_r, \Sigma, \Delta_r, F_r)$ can be rotated as described in the following. Let $R = (\ldots, q_{i_1}, \ldots, q_{i_2}, \ldots) \xrightarrow{a} q \in \Delta$ be a rule such that the w label of q_{i_1} contains r, and the Z_1 label of q_{i_2} contains (k,S,i).[22] Such a rule appears exactly once in each run of $\mathcal{A}$ since a reference to r may appear only once at the leaf level (otherwise we would not be dealing with a root-point of Type 3), and also S_i can link with a single leaf only (otherwise we would obtain a non-deterministic hypergraph). If there are more rules like R in $\mathcal{A}_r$, then we may split $\mathcal{A}_r$ to several automata containing a single rule of the described kind, and process each of the automata separately (which we assume to be the case in the following). When we transform $\mathcal{A}_r$ to $\mathcal{A}'_r$ that accepts trees in which the concerned loops are represented in the appropriately rotated way, the rule R is redirected to a newly introduced accepting state, the rules that used to originally lead to accepting states are redirected to states originally reading a reference to r, and

[21] If there are more backward passable paths from the roots of the trees represented by $\mathcal{A}_r$ to a root reference to k, then each such path has a different record in $Z_1^{\mathcal{A}_r}$. In such a case, we choose the one with the smallest (S,i).

[22] Note that q_{i_1} and q_{i_2} can appear swapped on the left-hand side of the rule too.

reading a reference to r while going to q is allowed. Formally, for some $q_{fin} \notin Q_r$, $\mathcal{A}'_r = (Q_r \cup \{q_{fin}\}, \Sigma, \Delta'_r, \{q_{fin}\})$ where $\Delta' = (\Delta_r \setminus (\{R\} \cup \{\xrightarrow{r} q\})) \cup \Delta''$. The set of the newly added rules is defined as $\Delta'' = \{(\ldots, q_{i_1}, \ldots, q_{i_2}, \ldots) \xrightarrow{a} q_{fin}\} \cup \{(q_1, \ldots, q_k) \xrightarrow{b} q_r \mid (q_1, \ldots, q_k) \xrightarrow{b} q_f \in \Delta_r, q_f \in F_r, \xrightarrow{r} q_r \in \Delta_r\} \cup \{\xrightarrow{r} q\}$.

As a consequence of the rotation, the final state of the rotated TA may have a different label (w, Y, Z_1, Z_2) than the original one.[23] This may cause that the FA with the new TA inside has a different root reference reachability graph and hence different ordering of the roots. Therefore after each TA rotation, we recompute the root reachability graph and reorder the forest. Note that the order of the roots that are originally ordered before the root of the rotated TA is not affected. Therefore, even if there are more root-points of Type 3, the rotations on each of them will be done at most once, and hence the canonisation procedure terminates.

As mentioned in Appendix A.3, the described canonisation procedure yields FA that strongly respect canonicity which allows us to check properness and box-connectedness by computing the reachability relation on the roots of the FA.

A.5 Cases When Inclusion on Hierarchical FA is Precise

In many practical cases, approximating the inclusion $[\![\mathcal{L}_H(\mathcal{F})]\!] \subseteq [\![\mathcal{L}_H(\mathcal{F}')]\!]$ by deciding $\mathcal{L}(\mathcal{F}) \subseteq \mathcal{L}(\mathcal{F}')$ actually is precise. Two general conditions that guarantee this are the following:

1. Distinct nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ represent distinct boxes (i.e., there are no two nested SFA that represent the same box but are not identical and hence represent two different labels in the alphabet of the component tree automata of $\mathcal{F}$) and
2. $\forall H \in \mathcal{L}_H(\mathcal{F}) \forall H' \in \mathcal{L}_H(\mathcal{F}') : H \neq H' \implies [\![H]\!] \cap [\![H']\!] = \emptyset$.

It can be seen that (2) holds if the following two conditions hold:

a. nested SFA of $\mathcal{F}$ and $\mathcal{F}'$ represent a set of boxes X that *do not overlap* and
b. every $H \in \mathcal{L}_H(\mathcal{F}) \cup \mathcal{L}_H(\mathcal{F}')$ is *maximally boxed* by boxes from X.

The notions of maximally boxes hypergraphs and non-overlapping boxes are defined as follows. A 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 $plug(G, e, C) = H$ 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 nested SFA that appear in the case studies presented in this paper satisfy Condition (1). Moreover, the boxes represented by them satisfy Points (a) and (b), therefore Condition (2) is satisfied too. Hence, inclusion tests performed within our case studies are precise.

[23] The order of root references in the string w can be different, and in each $(k, S, i, w') \in Z_2$, w' can be ordered differently as well. Y and Z_1 stay unchanged.

B Symbolic Execution

In Section 5, we have outlined several phases of symbolic execution of a program statement over an FA. The first phase may involve unfolding of some of the nested SFA hiding data values or pointers to be manipulated. Next, certain statements can require additional root nodes (and hence component TA) to be added, which is therefore done in the second phase. The third phase implements the actual effect of the statements in the form of a syntactical manipulation with the rules and states of the FA. Finally, the modified FA is transformed into the canonicity respecting form (including possible folding of nested SFA). The main idea of the first and fourth point is discussed in Section 5. Below, we briefly discuss the second and the third point.

Introduction of Additional Roots. In certain cases, one cannot execute the effect of a program statement directly on the FA at hand. Consider for example an FA $\mathcal{F}$ and the statement `y := x->s`. Here, for any hypergraph represented by $\mathcal{F}$, `x` points to a cut-point that corresponds to the root of a tree accepted by some component TA of $\mathcal{F}$. We want `y` to point to the node which is reachable from the node pointed to by `x` via the selector `s`. After executing the statement, `y` will point to a cut-point. However, it may be the case that the node `x->s` (i.e., the node that is the successor of the node pointed to by `x` via the selector `s`) is currently not a cut-point and it is accepted at an ordinary automaton state (not an accepting state as in the case of a root). Therefore, the TA accepting trees whose roots are pointed to by `x` has to be split into a new pair of TA such that the first automaton accepts trees that have a reference to the second automaton as the `s`-successors of their roots nodes, and the second automaton describes the part of the heap starting at the `x->s` nodes in the trees accepted by the original automaton.[24]

For the sake of simplicity, we now assume that the TA to be split contains a single transition leading to an accepting state only and that the accepting state does not appear inside any left hand side of a rule of the TA. A general TA can be easily transformed into a set of several TA obeying this restriction. Let $\mathcal{A} = (Q, \Sigma, \Delta, \{q_f\})$ be the TA that we want to split, let $(\ldots, q_s, \ldots) \xrightarrow{a} q_f \in \Delta$ be the only transition that leads to q_f, and let $q_s \in Q$ be the state at which the nodes accessible via the selector `s` from the roots of the trees accepted by $\mathcal{A}$ are accepted. We replace $\mathcal{A}$ by TA $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\mathcal{A}_1$ references $\mathcal{A}_2$ via `s`. Formally, $\mathcal{A}_1 = (Q, \Sigma, \Delta', \{q_f\})$, $\Delta' = \Delta \setminus \{(\ldots, q_s, \ldots) \xrightarrow{a} q_f\} \cup \{(\ldots, q_r, \ldots) \xrightarrow{a} q_f, \xrightarrow{r} q_r\}$ where r is a root reference to the newly created TA $\mathcal{A}_2$, and $\mathcal{A}_2 = (Q, \Sigma, \Delta, \{q_s\})$. Since this transformation my cause that many states of $\mathcal{A}_1$ and $\mathcal{A}_2$ become useless the automata are subsequently reduced (by removing

[24] Sometimes, it may happen that in some trees accepted by the given TA, a split is needed whereas in the others not. This can happen when in the trees accepted by the TA there is a tree where `x->s` is a root reference and another one where `x->s` is an intermediate node (accepted at an ordinary state). As an example, there may be sequences of `s`-selectors below the node pointed by `x` of length one or more. For length one, one does not need to introduce a new root since some root is already reached via `s`, which is, however, not the case for the other lengths. In such a scenario, the TA has to be first split into two TA accepting trees that do or do not need a split (which can be done by inspecting the rules in the immediate neighbourhood of the accepting states), and then the split is done only in the latter case.

useless states and subsequently by using, for instance, techniques for simulation-based reduction of nondeterministic automata).

Executing the Actual Effect of Pointer Manipulating Statements on FA. We now describe the execution of particular pointer manipulating statements on FA. Null and undefined values are represented by special pointer variables `null` and `undefined`. During the whole execution, these variables point to two designated cut-points with no successors. If a statement requests (1) an access to or a change of a successor of these nodes, or (2) a value or a change of data stored in these nodes, then we have encountered a null or undefined pointer dereference, which is followed by examining the error trace as described in Appendix C. Below there is a brief description of how the particular pointer manipulating statements can be executed over FA provided that they are not working with undefined or null pointers in a forbidden way:

- `x = y`, `x = null`: `x` is removed from all labels in which it appears. Then we label by `x` each transition that is labelled by `y` (or by `null`, respectively).
- `x = malloc()`: `x` is removed from all labels in which it appears. Then a new tree automaton accepting exactly a single tree encoding a single heap node pointed by x and having undefined successors is added.
- `x = y->s`: `x` is removed from all labels in which it appears. Then the TA which accepts nodes pointed by `y` is split at the selector `s`, and the transitions of the newly created TA leading to its accepting state, at which `y->s` is accepted, are labelled by `x`.
- `x->s = y`: The TA that accepts nodes pointed by `x` is split at the selector `s`. Then the transition $(\ldots, q_s, \ldots) \xrightarrow{a} q_f$ where q_f is an accepting state and q_s accepts a reference to the newly created TA accepting the `s`-subtrees of the trees accepted by the original TA is substituted by the transitions $(\ldots, q_r, \ldots) \xrightarrow{a} q_f$ and $\xrightarrow{r} q_r$ where r is a reference to the TA accepting trees whose root is labelled by `y`.
- `x->data = d`: Each transition $(q_1, \ldots, q_n) \xrightarrow{a} q_f$ of the TA accepting the nodes pointed by `x` where q_f is the accepting state is replaced by $(q_1, \ldots, q_n) \xrightarrow{a'} q_f$ where a' is obtained from a by changing the data value to d.
- `free(x)`: The TA that accepts nodes pointed by `x` is split at all outgoing selectors. Then the modified TA accepting the nodes pointed by `x` is removed from the given FA, and `x` is added to the labels associated with the special `undef` node.
- Tests over pointer variables and data stored in cells pointed to by variables are evaluated by examining labels of rules leading to accepting states of the TA that accept trees whose roots represent nodes pointed by the concerned pointer variables. These rules contain all the needed information (e.g., testing equality of pointer variables means that these variables should be associated with the same root node). After each test, the original SFA is split into two SFA—one of them accepts the trees that satisfy the tested condition, and the other one accepts the trees which do not satisfy it.

C Backward Symbolic Execution

In what follows, we discuss how a program path that is suspected to lead to an error may be executed backwards in order to check whether it really leads to an error or whether it corresponds to a spurious error (implying a need to refine the abstraction used).

First, let us briefly recall some essentials of the forward symbolic execution of a program over FA. The statements of the program are processed sequentially. The symbolic execution of a statement over a set of configurations represented by a CFA can cause the computation to branch either due to several possible outcomes of some operation over the given set configurations and/or due to the procedure of transforming an FA resulting from the operation into an SCFA (which is then further processed element-wise, hence the branching). The branching creates an execution tree whose nodes are labelled by CFA and the appropriate control line. The nodes of the tree where the computation can still continue are kept in memory together with information on how to reverse the effect of the statements that generated the particular CFA (e.g., after a statement assigning some data field, i.e., `x->data = d`, we remember the original value of the data field `x->data`, etc.).

Once the forward symbolic execution hits an erroneous configuration, the program trace suspected to lead to an error can be extracted by traversing the execution tree from the leaf node in which the (possible) error was detected towards the root. Now, it is necessary to check whether the path really leads to an error or whether a spurious counterexample arising just due to the applied abstraction was detected. In the latter case, the abstraction is to be refined.

More precisely, assume that we have a suspected error trace—a branch of the execution tree of the form $\mathcal{F}_0, \ldots, \mathcal{F}_n$ where for each $0 \leq i \leq n$, $\mathcal{F}_i$ is a CFA encoding a set of possible configurations of the heap. Here, $\mathcal{F}_0$ represents the set of initial configurations (in our case, usually the empty heap), and $\mathcal{F}_n$ encodes a set of configurations that include some bad configurations. This is, there is a nonempty set $Bad \subseteq [\![\mathcal{L}_H(\mathcal{F}_n)]\!]$ of hypergraphs that are erroneous meaning that they, e.g., represent heaps with garbage, heaps where a value of a variable to be dereferenced is `null` or `undef`, heap configurations reached at a designated error location, etc. Assume that we have a CFA $\mathcal{F}_{Bad}$ such that with $[\![\mathcal{L}_H(\mathcal{F}_{Bad})]\!] = Bad$. Such a CFA is directly obtained when a designated error line is hit; in the case of the generic pointer manipulation errors, a CFA encoding configurations that would cause such an error are produced by the symbolic execution of the appropriate statement (as one of the cases of its execution). Of course, sometimes, one can obtain an SCFA for the bad configurations, but this can be processes element-wise. Further, for every $1 \leq i \leq n$, the CFA $\mathcal{F}_i$ was obtained from $\mathcal{F}_{i-1}$ by symbolically executing some program statement in the following four steps described in Section 5 and also in Appendix B, followed by abstraction:

1. materialisation (unfolding boxes represented by SCFA at the relevant FA transitions),
2. introduction of additional root-points into the represented heaps (and hence component TA on the level of FA),
3. performing the actual update,
4. iterative folding and normalisation yielding an SCFA.

If we are only interested in determining whether the obtained program path is a real or spurious counterexample, we execute the trace again but without abstraction. If after the n-th step we end up with an SCFA representing a set that contains bad configurations, then the counterexample is real, otherwise it is spurious. However, to be able to refine abstraction within the framework of abstract regular tree model checking in a way driven by the counterexample, we need a backward execution which allows us to determine the exact point in the symbolic execution where a use of abstraction introduced configurations from which a spurious counterexample was generated.[25]

As usual in CEGAR, a backward execution of a program trace computes a chain $\mathcal{F}_{Bad} = \mathcal{F}'_n, \ldots, \mathcal{F}'_j$ of CFA representing sets of configurations where $j < n$ and either $j \geq 0$ is the greatest index such that $[\![\mathcal{L}_H(\mathcal{F}_j)]\!] \cap [\![\mathcal{L}_H(\mathcal{F}'_j)]\!] = \emptyset$ or $j = 0$ if there is no such index. For each $j < i \leq n$, $\mathcal{F}'_{i-1}$ is derived from $\mathcal{F}'_i$ by applying backwards the steps by which $\mathcal{F}_i$ was obtained from $\mathcal{F}_{i-1}$, but without abstraction. If $j = 0$ and $[\![\mathcal{L}_H(\mathcal{F}'_0)]\!] \cap [\![\mathcal{L}_H(\mathcal{F}_0)]\!] \neq \emptyset$, then the backward execution reached the initial configurations which means that the error trace is feasible in the verified program. If $[\![\mathcal{L}_H(\mathcal{F}'_j)]\!] \cap [\![\mathcal{L}_H(\mathcal{F}_j)]\!] = \emptyset$ for some $j > 0$, then the counterexample is spurious, and $\mathcal{F}'_{j+1}$ represents some configurations introduced by abstraction that caused the discovery of the spurious counterexample. In that case, we may use the pair $\mathcal{F}'_{j+1}, \mathcal{F}_{j+1}$ to refine the abstraction to prevent this spurious error trace from appearing in further verification by abstract regular tree model checking [6].

To be able to revert Steps 1 to 4 of the symbolic execution of a program statement in order to obtain $\mathcal{F}'_{i-1}$ from $\mathcal{F}'_i$, we remember certain information along the computation path. More precisely, we remember where we performed folding and unfolding (to revert Step 1 and Step 4). Additionally, we remember certain information to revert the actual update (Step 3) as we mentioned already at the beginning of this appendix and also the actions which were taken in order to transform the obtained FA into CFA (Step 4).

Now, assume that we have computed $\mathcal{F}'_{i+1}$ in the backward execution and we want to revert the effect of the i-th statement. First of all, we have to iteratively revert the iterative folding and normalisation (the need to iterate stems from possibly unfolding multiple boxes folded in the forward execution). The folding is reverted by simply unfolding the corresponding SFA. The normalisation is reverted by splitting certain component TA (in cases where some root-points were removed in the forward execution) and by an inverse reordering of the component TA. Next, we revert the effect of the actual update, which is typically achieved by using the stored additional information about what in the original configurations was changed by the actual update. Step 2 only introduces additional roots in the forward execution which are redundant wrt. the canonical representation. Therefore, it can be reverted by performing the standard normalisation procedure which converts the given FA into a CFA. Finally, we have to revert the effect of unfolding performed in Step 1, which is done by folding wherever needed.

[25] We note that in the current implementation where we use the finite-height abstraction [6], we only check whether the counterexample is spurious or not. If it is, we simply globally refine abstraction by increasing the abstraction height by one. However, we argue here that the backward execution is possible, which is crucial for using the more advanced predicate language abstraction (which is to be implemented in Forester in the near future).

Checking emptiness of the intersection $[\![\mathcal{L}_H(\mathcal{F}_{i-1})]\!] \cap [\![\mathcal{L}_H(\mathcal{F}'_{i-1})]\!]$ is an issue by itself. For two general hierarchical FA $\mathcal{F} = (\mathcal{A}_1, \ldots, \mathcal{A}_n, I, O)$ and $\mathcal{F}' = (\mathcal{B}_1, \ldots, \mathcal{B}_n, I, O)$, we do not know yet whether the intersection emptiness problem is decidable. However, we solve this problem analogically as in the case of checking language inclusion by using a safe approximation. This is, we compute the automaton $\mathcal{F} \cap \mathcal{F}' = (\mathcal{A}_1 \cap \mathcal{B}_1, \ldots, \mathcal{A}_2 \cap \mathcal{B}_2, I, O)$ where $\mathcal{A}_i \cap \mathcal{B}_i$ is the usual intersection of two tree automata. It obviously holds that $[\![\mathcal{L}_H(\mathcal{F} \cap \mathcal{F}')]\!] \subseteq [\![\mathcal{L}_H(\mathcal{F})]\!] \cap [\![\mathcal{L}_H(\mathcal{F}')]\!]$. Notice that by having a method that underapproximates the intersection, the only thing that can happen is that emptiness of the intersection $[\![\mathcal{L}_H(\mathcal{F}_i)]\!] \cap [\![\mathcal{L}_H(\mathcal{F}'_i)]\!]$ is detected sooner during the backward execution (for a larger i) than it should be (or it is detected in cases where it should not be detected at all). The only possibly consequence of this is that we attempt to refine abstraction instead of signalling a real counterexample or that we refine the abstraction in a wrong way. Both of the cases can cause the computation not to terminate, but it cannot lead to introducing false positives neither false negatives.

Moreover, again like in the case of language inclusion, for many practical cases, the above way of computing intersection of hierarchical FA gives precise results. Namely, it is precise under the same conditions under which our language inclusion check is precise (see Appendix A.5). We note that for SCFA that appear in our case studies, the conditions from Appendix A.5 hold and thus intersection can be computed precisely on them.