Verification of Programs Manipulating Complex Dynamic Data Structures

Jiří Šimáček*
Department of Intelligent Systems
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
isimacek@fit.vutbr.cz

Abstract

We develop a verification method based on a novel use of tree automata to represent heap configurations to allow verification of important properties—such as no null-pointer dereferences, absence of memory leaks, etc.—for programs manipulating complex dynamically linked data structures. In our approach, a heap is split into several "separated" parts such that each of them can be represented by a tree automaton. The automata can refer to each other allowing the different parts of the heaps to mutually refer to their boundaries. Moreover, we allow for a hierarchical representation of heaps by allowing alphabets of the tree automata to contain other, nested tree automata. Program instructions can be easily encoded as operations on our representation structure. This allows verification of programs based on symbolic state-space exploration together with refinable abstraction within the so-called abstract regular tree model checking. A motivation for the approach is to combine advantages of automata-based approaches (higher generality and flexibility of the abstraction) with some advantages of separation-logic-based approaches (efficiency).

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Alternation and Nodeterminism*

Keywords

hypergraphs, heaps, pointers, verification, shape analysis, finite tree automata, forest automata

*Recommended by thesis supervisor: Prof. Tomáš Vonar
Defended at Faculty of Information Technology, Brno University of Technology on October 29, 2012.

© Copyright 2013. All rights reserved. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from STU Press, Vazovova 5, 811 07 Bratislava, Slovakia.

Šimáček, J. Verification of Programs Manipulating Complex Dynamic Data Structures. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 5, No. 1 (2013) 25-33

1. Introduction

Traditional approaches for ensuring quality of computer systems such as code review or testing are nowadays reaching their inherent limitations due to the growing complexity of the current computer systems. That is why, there is an increasing demand for more capable techniques. One of the ways how to deal with this situation is to use suitable formal verification approaches.

In case of software, one especially critical area is that of ensuring safe memory usage in programs using dynamic memory allocation. The development of such programs is quite complicated, and many programming errors can easily arise here. Worse yet, the bugs within memory manipulation often cause an unpredictable behaviour, and they are often very hard to find. Indeed, despite the use of testing and other traditional means of quality assurance, many of the memory errors make it into the production versions of programs causing them to crash unexpectedly by breaking memory protection or to gradually waste more and more memory (if the error causes memory leaks). Consequently, using formal verification is highly desirable in this area.

Formal verification of programs with dynamically linked data structures is, however, very demanding since these programs are infinite-state. One of the most promising ways of dealing with infinite state verification is to use symbolic verification in which infinite sets of reachable configurations are represented finitely using a suitable formalism. In case of programs with dynamically linked data structures, the use of symbolic verification is complicated by the fact that their configurations are graphs, and representing infinite sets of graphs is particularly complicated (compared to objects like words or trees).

Many different verification approaches for programs manipulating dynamically linked data structures have been proposed so far. Some of them are based on logics [18, 21, 20, 5, 12, 19, 9, 23, 22, 8, 16, 11], others are based on using automata [7, 6, 10], upward closed sets [1, 3], as well as other formalisms. The approaches differ in their generality, efficiency, and degree of automation. Among the fully automatic ones, the works [5, 22] present an approach based on separation logic (see [20]) that is quite scalable due to using local reasoning. However, their method is limited to programs manipulating various kinds of lists. There are other works based on separation logic which also consider trees or even more complex data structures, but they either expect the input program to be in some

special form (e.g., [12]) or they require some additional information about the data structures which are involved (as in [19, 17]). Similarly, even the other existing approaches that are not based on separation logic often suffer from the need of non-trivial user aid in order to successfully finish the verification task (see, e.g., [18, 21]). On the other hand, the work [7] proposed an automata-based method which is able to handle fully automatically quite complex data structures, but it suffers from several drawbacks such as a monolithic representation of memory configurations which does not allow this approach to scale well.

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

For the above purpose, we propose a novel approach of representing sets of heaps via *tree automata* (TA). In our representation, a heap is split in a canonical way into several *tree components* whose roots are the so-called *cut-points*. Cut-points are nodes pointed to by program variables or having several incoming edges. The tree components can refer to the roots of each other, and hence they are "separated" much like heaps described by formulae joined by the separating conjunction in separation logic [20]. Using this decomposition, sets of heaps with a bounded number of cut-points are then represented by a new class of automata called *forest automata* (FA) that are basically tuples of TA accepting tuples of trees whose leaves can refer back to the roots of the trees. Moreover, we allow alphabets of FA to contain *nested FA*, leading to a *hierarchical encoding of heaps*, allowing us to represent even sets of heaps with an unbounded number of cut-points (e.g., sets of DLL). Intuitively, a nested FA can describe a part of a heap with a bounded number of cut-points (e.g., a DLL segment), and by using such an automaton as an alphabet symbol an unbounded number of times, heaps with an unbounded number of cut-points are described. Finally, since FA are not closed under union, we work with sets of forest automata, which are an analogy of disjunctive separation logic formulae.

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, 4]. The use of separation allows for a further increase in efficiency compared to a monolithic automata-based encoding proposed in [7].

Outline. The rest of this text is organised as follows. Section 2 contains preliminary notions that we build on. In Section 3, we give an informal presentation of forest automata which are formally described in Section 4 and Section 5. In Section 6 we briefly describe our verification procedure for programs manipulating dynamically linked data structures. Our experimental results are presented in Section 7. The conclusion and possible future directions are discussed in Section 8.

More details regarding this topic may also be found in [13, 14, 15].

2. Preliminaries

In this section, we provide basic notions which are used throughout the rest of this text.

2.1 Alphabets and Trees

A *ranked alphabet* Σ is a set of symbols together with a ranking function $\# : \Sigma \to \mathbb{N}$. For $a \in \Sigma$, the value $\#a$ is called the *rank* of a. For any $n \geq 0$, we denote by Σ_n the set of all symbols of rank n from Σ. Let ε denote the empty sequence. A *tree* t over a ranked alphabet Σ is a partial mapping $t : \mathbb{N}^* \to \Sigma$ that satisfies the following conditions: (1) $dom(t)$ is a finite prefix-closed subset of $\mathbb{N}^*$ and (2) for each $v \in dom(t)$, if $\#t(v) = n \geq 0$, then $\{i \mid vi \in dom(t)\} = \{1, \ldots, n\}$. Each sequence $v \in dom(t)$ is called a *node* of t. For a node v, we define the ith *child* of v to be the node vi, and the ith *subtree* of v to be the tree t' such that $t'(v') = t(viv')$ for all $v' \in \mathbb{N}^*$. A *leaf* of t is a node v which does not have any children, i.e., there is no $i \in \mathbb{N}$ with $vi \in dom(t)$. We denote by T_Σ the set of all trees over the alphabet Σ.

2.2 Tree Automata

A (finite, non-deterministic) *tree automaton* (abbreviated sometimes 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 $((q_1, \ldots, q_n), a, q)$ where $q_1, \ldots, q_n, q \in Q, a \in \Sigma$, and $\#a = n$. We use equivalently $(q_1, \ldots, q_n) \xrightarrow{a} q$ and $q \xrightarrow{a} (q_1, \ldots, q_n)$ to denote that $((q_1, \ldots, q_n), a, q) \in \Delta$. The two notations correspond to the bottom-up and top-down representation of tree automata, respectively. (Note that we can afford to work interchangeably with both of them since we work with non-deterministic tree automata, which are known to have an equal expressive power in their bottom-up and top-down representations.) In the special case when $n = 0$, we speak about the so-called *leaf rules*, which we sometimes abbreviate as $\xrightarrow{a} q$ or $q \xrightarrow{a}$.

Let $\mathcal{A} = (Q, \Sigma, \Delta, F)$ be a TA. A *run* of $\mathcal{A}$ over a tree $t \in T_\Sigma$ is a mapping $\pi : dom(t) \to Q$ such that, for each node $v \in dom(t)$ of rank $\#t(v) = n$ where $q = \pi(v)$, if $q_i = \pi(vi)$ for $1 \leq i \leq n$, then Δ has a rule $(q_1, \ldots, q_n) \xrightarrow{t(v)} q$. We write $t \stackrel{\pi}{\Longrightarrow} q$ to denote that π is a run of $\mathcal{A}$ over t such that $\pi(\varepsilon) = q$. We use $t \Longrightarrow q$ to denote that $t \stackrel{\pi}{\Longrightarrow} q$ for some run π. The *language* accepted by a state q is defined by $\mathcal{L}_\mathcal{A}(q) = \{t \mid t \Longrightarrow q\}$, while the language of a set of states $S \subseteq Q$ is defined as $\mathcal{L}_\mathcal{A}(S) = \bigcup_{q \in S} \mathcal{L}_\mathcal{A}(q)$. When it is clear which TA $\mathcal{A}$ we refer to, we only write $\mathcal{L}(q)$ or $\mathcal{L}(S)$. The language of $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \mathcal{L}_\mathcal{A}(F)$. We also extend the notion of a language to a tuple of states $(q_1, \ldots, q_n) \in Q^n$ by letting

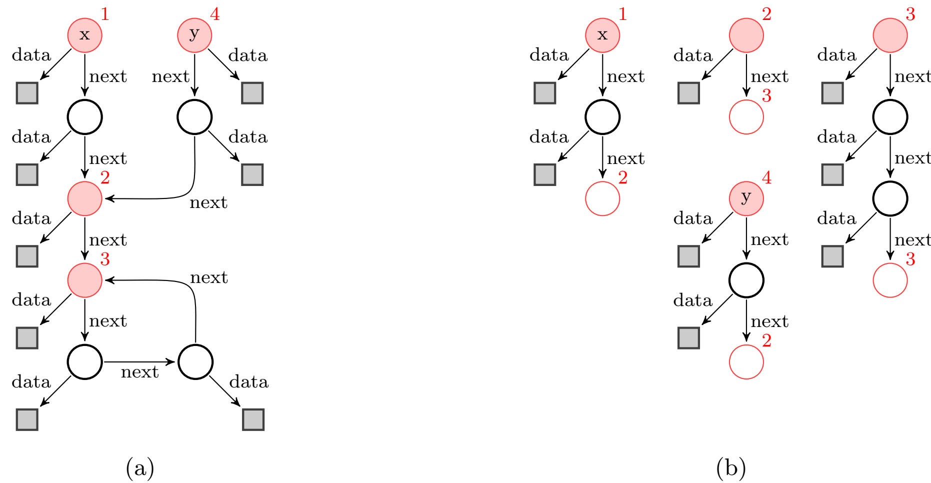

Figure 1: (a) A heap graph with cut-points highlighted in red, (b) the canonical tree decomposition of the heap with x ordered before y

$\mathcal{L}((q_1,\ldots,q_n)) = \mathcal{L}(q_1) \times \cdots \times \mathcal{L}(q_n)$. The language of a set of n-tuples of sets of states $S \subseteq (2^Q)^n$ is the union of languages of elements of S, the set $\mathcal{L}(S) = \bigcup_{E \in S} \mathcal{L}(E)$. We say that X accepts y to express that $y \in \mathcal{L}(X)$.

3. From Heaps to Forests

In this section, we outline in an informal way our proposal of hierarchical forest automata and the way how sets of heaps can be represented by them. For the purpose of the explanation, *heaps* may be viewed as oriented graphs whose nodes correspond to allocated memory cells and edges to pointer links between these cells. The nodes may be labelled by non-pointer data stored in them (assumed to be from a finite data domain) and by program variables pointing to the nodes. Edges may be labelled by the corresponding selectors.

In what follows, we restrict ourselves to *garbage free heaps* in which all memory cells are reachable from pointer variables by following pointer links. However, this is not a restriction in practice since the emergence of garbage can be checked for each executed program statement. If some garbage arises, an error message can be issued and the symbolic computation stopped. Alternatively, the garbage can be removed and the computation continued.

It is easy to see that each heap graph can be *decomposed* into a set of *tree components* when the leaves of the tree components are allowed to reference back to the roots of these components. Moreover, given a total ordering on program variables and selectors, each heap graph may be decomposed into a tuple of tree components in a *canonical way* as illustrated in Figure 1 (a) and (b). In particular, one can first identify the so-called *cut-points*, i.e., nodes that are either pointed to by a program variable or that have several incoming edges. Next, the cut-points can be canonically numbered using a depth-first traversal of the heap graph starting from nodes pointed to by program variables in the order derived from the order of the program variables and respecting the order of selectors. Subsequently, one can split the heap graph into tree components rooted at particular cut-points. These components should contain all the nodes reachable from their root while not passing through any cut-point, plus a copy of each reachable cut-point, labelled by its number. Finally, the tree components can then be canonically ordered according to the numbers of the cut-points representing their roots.

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

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

Note, however, that FA are not closed under union. Even for FA having the same number of components, uniting the TA component-wise may yield an FA overapproximating the union of the sets of heaps represented by the original FA (cf. Section 4). Thus, we represent unions of FA explicitly as *sets of FA* (SFA), which is similar to dealing with disjunctions of conjunctive separation logic formulae. However, as we will see, inclusion on the sets of heaps represented by SFA is still easily decidable.

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

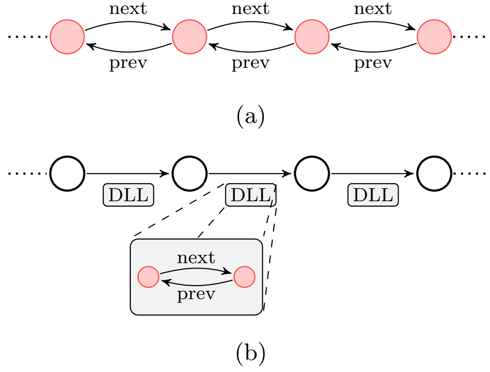

Figure 2: (a) A part of a DLL, (b) a hierarchical encoding of the DLL

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

Sets of heap hypergraphs corresponding either to the top level of the representation or to boxes of different levels can then be decomposed into (hyper)tree components and represented using *hierarchical FA* whose alphabet can contain nested FA.[1] Intuitively, FA appearing in the alphabet of some superior FA play a role similar to that of inductive predicates in separation logic.[2] We restrict ourselves to automata that form a finite and strict hierarchy (i.e., there is no circular use of the automata in their alphabets).

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

[1]Since graphs are a special case of hypergraphs, in the following, we will work with hypergraphs only.
[2]For instance, we use a nested FA to encode a DLL segment of length 1. In separation logic, the corresponding induction predicate would represent segments of length 1 or more. In our approach, the repetition of the segment is encoded in the structure of the top-level FA.

4. Forest Automata

We now formalise the notion of hypergraphs, their forest representation, and the notion of forest automata.

4.1 Hypergraphs

A *ranked alphabet* is a finite set Γ of symbols associated with a map $\# : \Gamma \to \mathbb{N}$. The value $\#(a)$ is called the *rank* of $a \in \Gamma$. We use $\#(\Gamma)$ to denote the maximum rank of a symbol in Γ. A ranked alphabet Γ is a *hypergraph alphabet* if it is associated with a total ordering $\preceq_\Gamma$ on its symbols. For the rest of the section, we fix a hypergraph alphabet Γ.

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

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

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

A *path* in a hypergraph $G = (V, E, P)$ is a sequence $\langle v_0, a_1, v_1, \dots, a_n, v_n \rangle$, $n \geq 0$, where for all $1 \leq i \leq n$, v_i is an a_i-successor of v_{i-1}. G is called *deterministic* iff $\forall (v, a, \overline{v}), (v, a', \overline{v}') \in E$: $a = a' \implies \overline{v} = \overline{v}'$. G is called *well-connected* iff each node $v \in V$ is reachable through some path from some input port of G.

As we have already briefly mentioned in Section 3, in hypergraphs representing heaps, input ports correspond to nodes pointed to by program variables or to input nodes of components, and output ports correspond to output nodes of components. Figure 1 (a) shows a hypergraph with two input ports corresponding to the variables `x` and `y`. The hyperedges are labelled by selectors `data` and `next`. All the hyperedges are of arity 1.

4.2 Forest Representation of Hypergraphs

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

Formally, an (unordered, oriented, Γ-labelled) *tree* $T = (V, E)$ consists of a set of vertices and hyperedges defined

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

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

- For every $i \in \{1, \ldots, n\}$, $T_i = (V_i, E_i)$ is a tree that is labelled by the alphabet $(\Gamma \cup \{1, \ldots, n\})$.
- R is a (forest) port specification consisting of a set of *input ports* $I_R \subseteq \{1, \ldots, n\}$, a set of *output ports* $O_R \subseteq \{1, \ldots, n\}$, and a total ordering $\preceq_R$ of $I_R \cup O_R$.
- For all $i, j \in \{1, \ldots, n\}$, (1) if $i \neq j$, then $V_i \cap V_j = \emptyset$, (2) $\#(i) = 0$, and (3) a vertex v with $(v, i) \in E_j$ is not a source of any other edge (it is a leaf). We call such vertices *root references* and denote by $rr(T_i, j)$ the set of all root references to T_j in T_i, i.e., $rr(T_i, j) = \{v \in V_i \mid (v, j) \in E_i\}$. We also define $rr(T_i) = \bigcup_{j=1}^{n} rr(T_i, j)$.

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

- $V = \bigcup_{i=1}^{n} V_i \setminus rr(T_i)$,
- $E = \bigcup_{i=1}^{n} \{(v, a, \overline{v}') \mid a \in \Gamma \wedge \exists (v, a, \overline{v}) \in E_i \ \forall 1 \leq j \leq \#(a)$: if $\exists (\overline{v}.j, k) \in E_i$ with $k \in \{1, \ldots, n\}$, then $\overline{v}'.j = root(T_k)$, else $\overline{v}'.j = \overline{v}.j\}$,
- $I_P = \{root(T_i) \mid i \in I_R\}$,
- $O_P = \{root(T_i) \mid i \in O_R\}$,
- $\forall u, v \in I_P \cup O_P$ such that $u = root(T_i)$ and $v = root(T_j)$:

$$u \preceq_P v \iff i \preceq_R j.$$

4.3 Forest Automata

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

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

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

- For all $1 \leq i \leq n$, $\mathcal{A}_i = (Q_i, \Sigma, \Delta_i, F_i)$ is a TA with $\Sigma = \Sigma_\Gamma \cup \{1, \ldots, n\}$ and $\#(i) = 0$.
- R is defined as for forests, i.e., it consists of input and output ports $I_R, O_R \subseteq \{1, \ldots, n\}$ and a total ordering $\preceq_R$ on $I_R \cup O_R$.

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

5. Hierarchical Forest Automata

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

5.1 Hierarchical Hypergraphs

We first introduce hypergraphs with hyperedges labelled by the so-called boxes which are sets of hypergraphs (defined up to isomorphism[3]). A hypergraph G with hyperedges labelled by boxes encodes a set of hypergraphs. The hypergraphs encoded by G can be obtained by replacing every hyperedge of G labelled by a box by some hypergraph from the box. The hypergraphs within the boxes may themselves have hyperedges labelled by boxes, which gives rise to a hierarchical structure (which we require to be of a finite depth).

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

[3]Dealing with hypergraphs (and later also automata) defined up to isomorphism avoids a need to deal with classes instead of sets.

ranked alphabet containing all Υ-labelled boxes such that $\mathbb{B}[\Upsilon] \cap \Upsilon = \emptyset$. The operator $\mathbb{B}$ gives rise to a hierarchy of alphabets $\Gamma_0, \Gamma_1, \ldots$ where:

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

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

5.2 Semantics of Hierarchical Hypergraphs

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

If H is plain, then $[\![H]\!]$ contains just H itself. If H is of level $j > 0$, then hypergraphs from $[\![H]\!]$ are obtained in such a way that hyperedges labelled by boxes $B \in \Gamma_j$ are substituted in all possible ways by plain components from $[\![B]\!]$. The substitution is similar to an ordinary hyperedge replacement used in graph grammars. When an edge e is substituted by a component C, the input port of C is identified with the source node of e, and the output ports of C are identified with the target nodes of e. The correspondence of the output ports of C and the target nodes of e is defined using the order of the target nodes in e and the ordering of ports of C. The edge e is finally removed from H.

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

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

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

Figure 2 (b) shows a hierarchical hypergraph of level 1 whose semantics is the (hyper)graph of Figure 2 (a).

5.3 Hierarchical Forest Automata

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

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

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

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

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

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

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

6. Verification Based on Forest Automata

A fundamental property of our newly proposed formalism of forest automata is that (1) the language inclusion can be checked efficiently (see [15]), and (2) C program statements manipulating pointers can be easily encoded as operations modifying FA (see again [15]). Due to this and

due to the fact that FA are based on tree automata, we can build on the concept of abstract regular tree model checking (see [7]) to obtain a new symbolic verification procedure for the considered class of programs. The procedure then works as follows: The algorithm maintains a set of visited program configurations and a set of program configurations which need to be processed. At the beginning, the set of visited program configurations is empty, and the set of program configurations waiting to be processed contains the initial configuration of the program to be analysed which consists of the initial assignment of program variables, the empty heap, and the program counter pointing to the first instruction of the program. Then, the algorithm iteratively picks one waiting program configuration and performs a symbolic execution of the appropriate program statement. This essentially means that one takes a forest automaton representing a set of heaps and transforms it into a new forest automaton. The set of heaps represented by the newly obtained forest automaton reflects the change within the heap caused by the execution of the given program statement. In addition to that, one can also apply *abstraction* in order to be able to obtain sets of all reachable configurations, which are typically infinite, in a finite number of steps. In the next step, the algorithm checks whether the newly created program configuration is covered by the set of already visited program configurations by means of testing inclusion of languages represented by forest automata. If the newly obtained symbolic configuration is not covered by the set of visited program configurations, it is inserted into the set of waiting program configurations. The process then continues by picking another waiting configuration. During the symbolic execution, the algorithm checks whether the verified code behaves properly, i.e., it does not dereference invalid pointers, it does not produce memory leaks, etc. If the program does not operate properly, the procedure is immediately terminated, and an error is reported. If the set of waiting configurations becomes empty, the procedure terminates and outputs that the program is safe.

When an error is encountered, it remains to find out whether it is reachable within the original program, or it was encountered due to an excessive abstraction. In order to check, whether the error is indeed reachable, one can execute the corresponding trace without the abstraction. If such trace cannot be executed, then the set of reachable program configurations is over-approximated too much, and the abstraction needs to be refined. The refinement can be done globally which is, however, not very efficient. A better solution is to use the counterexample-guided abstraction refinement as introduced in the framework of abstract regular tree model checking (see again [7]). For that to work, one needs to be able to execute the error trace backwards (see [14] for more details).

Our approach has been implemented in a prototype tool called *Forester* as a `gcc` plug-in. This allows us to demonstrate that the proposed approach is very promising as the tool can successfully handle multiple highly non-trivial case studies (for some of which we are not aware of any other tool that could handle them fully automatically).

7. Experimental Results

We have experimentally compared the performance of our tool with that of Space Invader [5], the first fully automated tool based on separation logic, Predator [11], a new fully automated tool based in principle on separation logic (although it represents sets of heaps using graphs), and also with the ARTMC tool [7] based on abstract regular tree model checking[4]. We tested the tool on sample programs with various types of lists (singly-linked, doubly-linked, cyclic, nested), trees, and their combinations. Basic memory safety properties—in particular, absence of null and undefined pointer dereferences, double free operations, and absence of garbage—were checked. We have run our tests on a machine with an Intel T9600 (2.8GHz) CPU and 4GiB of RAM. The comparison with Space Invader and Predator was done on examples with lists only since Invader and Predator do not handle trees. The higher flexibility of our automata abstraction shows up, for example, in the test case with a list of sublists of lengths 0 or 1 for which Space Invader does not terminate. Our technique handles this example smoothly (without any need to add any special inductive predicates that could decrease the performance or generate false alarms). Predator can also handle this test case, but to achieve that, the algorithms implemented in it must have been manually extended to use a new kind of list segment of length 0 or 1, together with an appropriate modification of the implementation of Predator's join and abstraction operations[5]. On the other hand, the ARTMC tool can, in principle, handle more general structures than we can currently handle such as trees with linked leaves. However, the representation of heap configurations used in ARTMC is much heavier which causes ARTMC not to scale that well.

Table 1 summarises running times (in seconds) of the four tools on our case studies. The value T means that the running time exceeded 30 minutes, o.o.m. means that the tool ran out of memory, and the value Err stands for a failure of symbolic execution. The names of experiments in the table contain the name of the data structure handled by the program. In particular, "SLL" stands for singly-linked lists, "DLL" for doubly linked lists (the prefix "C" means cyclic), "tree" for binary trees, "tree+parents" for trees with parent pointers. Nested variants of SLL are named as "SLL of" and the type of the nested list. In particular, "SLL of 0/1 SLLs" stands for SLL of nested SLL of length 0 or 1. "SLL+head" stands for a list where each element points to the head of the list, "SLL of 2CDLLs" stands for SLL whose implementation of lists used in the Linux kernel with restricted pointer arithmetic [11] which we can also handle. All experiments start with a random creation and end with a disposal of the specified structure. If some further operation is performed in between the creation phase and the disposal phase, it is indicated in brackets. In the experiment "tree+stack", a randomly created tree is disposed using a stack in a top-down manner such that we always dispose a root of a subtree and save its subtrees into the stack. "DSW" stands for the Deutsch-Schorr-Waite tree traversal (the Lindstrom variant). Forester has been provided a set of suitable boxes for each of the test cases for the comparison in Table 1.

8. Conclusions and Future Directions

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

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

Table 1: A comparison of Forester with other existing tools

example	Forester	Invader	Predator	ARTMC
SLL (delete)	0.01	0.10	0.01	0.50
SLL (reverse)	< 0.01	0.03	< 0.01	
SLL (bubblesort)	0.02	Err	0.02	
SLL (insertsort)	0.02	0.10	0.01	
SLL (mergesort)	0.07	Err	0.13	
SLL of CSLLs	0.07	T	0.12	
SLL+head	0.01	0.06	0.01	
SLL of 0/1 SLLs	0.02	T	0.03	
$\text{SLL}_{\text{Linux}}$	< 0.01	T	< 0.01	
DLL (insert)	0.02	0.08	0.03	0.40
DLL (reverse)	0.01	0.09	0.01	1.40
DLL (insertsort1)	0.20	0.18	0.15	1.40
DLL (insertsort2)	0.06	Err	0.03	
CDLL	< 0.01	0.09	< 0.01	
DLL of CDLLs	0.18	T	0.13	
SLL of $\text{2CDDLs}_{\text{Linux}}$	0.03	T	0.19	
tree	0.06			3.00
tree+stack	0.02			
tree+parents	0.10			
tree (DSW)	0.16			o.o.m

We have presented a new method for verification of heap manipulating programs. In particular, we use forest automata to encode sets of heaps and exploit the fact that the set of C statements that we need to support can be easily symbolically executed over FA. Moreover, the fact that FA are built over tree automata allows us to build a verification procedure based on the framework of abstract regular tree model checking. Further, we have described how more complex data structures—such as doubly-linked lists—can be verified using hierarchically nested forest automata. Finally, we have implemented the above mentioned approach in a prototype tool called Forester. We have performed an experimental evaluation on a set of benchmarks consisting of C programs manipulating various dynamically allocated data structures in order to compare Forester to other similar tools. The obtained results confirmed that our approach is quite competitive in practice.

As of what concerns the future work, it would be interesting to extend our approach such that it could track some information about the data stored within dynamically linked data structures. This would allow one to verify algorithms in which the memory safety depends, for instance, on the fact that a certain sequence is sorted. As an example, we can mention programs manipulating red-black trees in which case one needs to distinguish red and black trees. Apart from that, tracking of the data stored inside the dynamically linked data structure would allow our tool to also check properties concerning that data.

Another line of research is a generalisation of our approach to concurrent programs. Here, an especially interesting case is that of lockless concurrent data structures, which are extremely difficult to understand and validate.

Acknowledgements. This work was supported by the Czech Ministry of Education (COST OC 10009, MEB 021023, and MSM 0021630528), the Czech Science Foundation (102/09/H042, 201/09/P531, and P103/10/0306), the EU/Czech IT4Innovations Centre of Excellence (ED 1.1.00/02.0070), the European Science Foundation (ESF COST action IC 0901), the French National Research Agency (ANR-09-SEGI-016 VERIDYC), and the Brno University of Technology (FIT-S-10-1, FIT-S-11-1, FIT-S-12-1).

References

[1] P. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic Abstraction for Programs with Dynamic Memory Heaps. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 341–354, Berlin, Heidelberg, 2008. Springer Verlag.

[2] P. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. Computing Simulations over Tree Automata: Efficient Techniques for Reducing Tree Automata. In *Proc. of TACAS*, number 4963 in LNCS, pages 93–108, Berlin, Heidelberg, 2008. Springer Verlag.

[3] P. Abdulla, J. Cederberg, and T. Vojnar. Monotonic Abstraction for Programs with Multiply-Linked Structures. In *Proc. of RP*, volume 6945 of *LNCS*, pages 125–138, Berlin, Heidelberg, 2011. Springer Verlag.

[4] P. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When Simulation Meets Antichains (on Checking Language Inclusion of NFAs). In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 158–174, Berlin, Heidelberg, 2010. Springer Verlag.

[5] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. OâĂŹhearn, and H. Yang. Shape analysis for composite data structures. In *Proc. of CAV*, pages 178–192, Berlin, Heidelberg, 2007. Springer Verlag.

[6] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. *Formal Methods in System Design*, 38(2):158–192, Apr. 2011.

[7] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*, pages 52–70, Berlin, Heidelberg, 2006. Springer Verlag.

[8] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-abduction. In *Proc. of PLDI*, volume 44 of *ACM SIGPLAN Notices*, pages 289–300, New York, NY, USA, 2009. ACM Press.

[9] B.-Y. Chang, X. Rival, and G. Necula. Shape Analysis with Structural Invariant Checkers. In *Proc. of SAS*, volume 4634 of *LNCS*, pages 384–401, Berlin, Heidelberg, 2007. Springer Verlag.

[10] J. Deshmukh, E. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS*, volume 3920 of *LNCS*, pages 27–41, Berlin, Heidelberg, 2006. Springer Verlag.

[11] K. Dudka, P. Peringer, and T. Vojnar. Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures using Separation Logic. In *Proc. of CAV*, volume 6806 of *LNCS*, pages 372–378, Berlin, Heidelberg, 2011. Springer Verlag.

[12] B. Guo, N. Vachharajani, and D. August. Shape Analysis with Inductive Recursion Synthesis. In *Proc. of PLDI*, volume 42 of *ACM SIGPLAN Notices*, pages 256–265, New York, NY, USA, 2007. ACM Press.

[13] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. In *Proc. of CAV*, volume 6806 of *LNCS*, pages 424–440, Berlin, Heidelberg, 2011. Springer Verlag.

[14] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. Technical report, Faculty of Information Technology, BUT, 2011.

[15] P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest automata for verification of heap manipulation. *Formal Methods in System Design*, pages 1–24, 2012.

[16] P. Madhusudan, G. Parlato, and X. Qiu. Decidable Logics Combining Heap Structures and Data. In *Proc. of POPL*, volume 46 of *ACM SIGPLAN Notices*, pages 611–622, New York, NY, USA, 2011. ACM Press.

[17] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic Numeric Abstractions for Heap-manipulating Programs. In *Proc. of POPL*, volume 45 of *ACM SIGPLAN Notices*, pages 211–222, New York, NY, USA, 2010. ACM Press.

[18] A. Møller and M. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI*, volume 36 of *ACM SIGPLAN Notices*, pages 221–231, New York, NY, USA, 2001. ACM Press.

[19] H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *Proc. of VMCAI*, volume 4349 of *LNCS*, pages 251–266, Berlin, Heidelberg, 2007. Springer Verlag.

[20] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[21] S. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.

[22] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV*, volume 5123 of *LNCS*, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] K. Zee, V. Kuncak, and M. Rinard. Full Functional Verification of Linked Data Structures. In *Proc. of PLDI*, volume 43 of *ACM SIGPLAN Notices*, pages 349–361, New York, NY, USA, 2008. ACM Press.

Selected Papers by the Author

L. Holík, J. Šimáček. Optimizing an LTS-Simulation Algorithm. *Computing and Informatics*, 2010(7):1337–1348, 2010.

L. Holík, O. Lengál, J. Šimáček, and T. Vojnar. Efficient Inclusion Checking on Explicit and Semi-Symbolic Tree Automata. *In Proc. of ATVA*, volume 6996 of LNCS, pages 243–258, 2011. Springer-Verlag.

O. Lengál, J. Šimáček, and T. Vojnar. VATA: A Library for Efficient Manipulation of Non-Deterministic Tree Automata. *In Proc. of TACAS*, volume 7214 of LNCS, pages 79–94, 2012. Springer-Verlag.

P. Habermehl, L. Holík, J. Šimáček, A. Rogalewicz, and T. Vojnar. Forest Automata for Verification of Heap Manipulation. *Formal Methods in System Design*, 41(1):83–106, 2012. Springer-Verlag.