# Counterexample Validation and Interpolation-Based Refinement for Forest Automata

FIT BUT Technical Report Series

*Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, and Tomáš Vojnar*

Technical Report No. FIT-TR-2016-03
Faculty of Information Technology, Brno University of Technology

Last modified: November 15, 2016

**NOTE:** This technical report contains an extended version of a paper with the same name, which is currently under submission.

# Counterexample Validation and Interpolation-Based Refinement for Forest Automata

Lukáš Holík, Martin Hruška, Ondřej Lengál, Adam Rogalewicz, and Tomáš Vojnar

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

**Abstract.** In the context of shape analysis, counterexample validation and abstraction refinement are complex and so far not sufficiently resolved problems. We provide a novel solution to both of these problems in the context of fully-automated and rather general shape analysis based on forest automata. Our approach is based on backward symbolic execution on forest automata, allowing one to derive automata-based interpolants and refine the automata abstraction used. The approach allows one to distinguish true and spurious counterexamples and guarantees progress of the abstraction refinement. We have implemented the approach in the FORESTER tool and present promising experimental results.

## 1 Introduction

In [14,16], *forest automata* (FAs) were proposed as a formalism for representing sets of heap graphs within a fully-automated and scalable *shape analysis* of programs with complex *dynamic linked data structures*. FAs were implemented in the FORESTER tool and successfully used to verify programs over a wide range of data structures, such as different kinds of lists (singly- and doubly-linked, circular, nested, and/or having various additional pointers), different kinds of trees, as well as skip lists. FAs have the form of tuples of *tree automata* (TAs), allowing abstract transformers corresponding to heap operations to have a *local impact* (i.e., to change just a few component TAs instead of the entire heap representation), leading to scalability. To handle complex nested data structures, FAs may be *hierarchically nested*, i.e., lower-level FAs can be used as (automatically derived) alphabet symbols of higher-level FAs.

Despite FORESTER managed to verify a number of programs, it suffered from two important deficiencies. Namely, due to using abstraction and the lack of mechanisms for checking validity of possible counterexamples, it could report *spurious errors*, and, moreover, it was unable to refine the abstraction using the spurious counterexample. Interestingly, as discussed in the related work section, this problem is common for many other approaches to shape analysis, which may perhaps be attributed to the complexity of heap abstractions. In this paper, we tackle the above problem by providing a novel method for *validation of possible counterexample traces* as well as a *counterexample guided abstraction refinement* (CEGAR) loop for shape analysis based on FAs.

Our counterexample validation is based on *backward symbolic execution* of a candidate counterexample trace on the level of FAs (with no abstraction on the FAs) while checking *non-emptiness of its intersection* with the forward symbolic execution (which was abstracting the FAs). For that, we have to revert not only abstract transformers corresponding to program statements but also various meta-operations that are used in the forward symbolic execution and that significantly influence the way sets of heap configurations are represented by FAs. In particular, this concerns *folding* and *unfolding* of

nested FAs (which we call *boxes*) as well as *splitting*, *merging*, and *reordering* of component TAs, which is used in the forward run for the following two reasons: to prevent the number of component TAs from growing and to obtain a canonic FA representation.

If the above meta-operations were not reverted, we would not only have problems in reverting some program statements but also in intersecting FAs obtained from the forward and backward run. Indeed, the general problem of checking emptiness of intersection of FAs that may use different boxes and different component TAs (i.e., intuitively, different decompositions of the represented heap graphs) is open. When we carefully revert the mentioned operations, it, however, turns out that the FAs obtained in the forward and backward run use *compatible* decomposition and hierarchical structuring of heap graphs, and so checking emptiness of their intersection is possible. Even then, however, the intersection is not trivial as the boxes obtained in the backward run may represent smaller sets of sub-heaps, and hence we cannot use boxes as symbols and instead have to perform the intersection *recursively* on the boxes as well.

Our abstraction on FAs is a modification of the so-called *predicate language abstraction* [10]. This particular abstraction collapses those states of component TAs that have non-empty intersection with the same predicate languages, which are obtained from the backward execution. We show that, in case the intersection of the set of configurations of the above described forward and backward symbolic runs is empty, we can derive from it an *automata interpolant* allowing us to get more predicate languages and to refine the abstraction such that progress of the CEGAR loop is guaranteed (in the sense that we do not repeat the same abstract forward run).

We have implemented the proposed approach in FORESTER and tested it on a number of small but challenging programs. Despite there is, of course, a lot of space for further optimisations, the experimental results are very encouraging. FORESTER can now not only verify correct programs with complex dynamic data structures but also reliably report errors in such programs. For some classes of dynamic data structures (notably skip lists), FORESTER is, to the best of our knowledge, the only tool that can provide both sound verification as well as reliable error reporting in a fully automated analysis (i.e., no manually provided heap predicates, no invariants, etc.). Moreover, for some classes of programs (e.g., various kinds of doubly-linked lists, trees, and nested lists), the only other tool that we are aware to be able to provide such functionality is our older automata-based tool [7], which is, however, far less scalable due to the use of a monolithic heap encoding based on a single TA. Finally, the refinement mechanism we introduced allowed us to verify some programs that were before out of reach of FORESTER due to handling finite domain data stored in the heap (which can be used by the programs themselves or introduced by tagging selected elements in dynamic data structures when checking properties such as sortedness, reordering, etc.).

## 2 Related Work

Many different approaches to shape analysis have been proposed, using various underlying formalisms, such as logics [17,24,26,20,9,23], automata [7,12,14,16,8], graphs [11,13], or graph grammars [15]. Apart from the underlying formalisms, the approaches differ in their degree of automation, in the heap structures they can handle, and in their scalability. The shape analysis based on forest automata proposed in [16] that we build on in this paper belongs among the most general, fully automated approaches, still having decent scalability.

As noted also in the recent work [2], a common weakness of the current approaches to shape analysis is a lack of proper support for checking spuriousness of counterexample traces, possibly followed by automated refinement of the employed abstraction. This is exactly the problem that we tackle in this paper. Below, we characterize previous attempts on the problem and compare our approach with them.

The work [4] adds a CEGAR loop on top of the TVLA analyzer [24], which is based on *3-valued predicate logic with transitive closure*. The refinement is, however, restricted to adding more pointer variables and/or data fields of allocated memory cells to be tracked only (together with combining the analysis with classic predicate analysis on data values). The analysis assumes the other necessary heap predicates (i.e., the so-called core and instrumentation relations in terms of [24]) to be fixed in advance and not refined. The work [19] also builds on TVLA but goes further by learning more complex instrumentation relations using inductive logic programming. The core relations are still fixed in advance though. Compared with both of these works, we do not assume any predefined fixed predicates. Moreover, the approach of [19] is not CEGAR-based—it refines the abstraction whenever it hits a possible counterexample in which some loss of precision happened, regardless of whether the counterexample is real or not.

In [22], a CEGAR-based approach was proposed for automated refinement of the so-called *Boolean heap abstraction* using disjunctions of universally quantified Boolean combinations of first-order predicates with free variables and transitive closure. Unlike our work, the approach assumes the analyzed programs to be annotated by procedure contracts and representation invariants of data structures. New predicates are inferred using finite-trace weakest preconditions on the annotations, and hence new predicates with reachability constraints can only be inferred via additional heuristic widening on the inferred predicates. Moreover, the approach is not appropriate for handling nested data structures, such as lists of lists, requiring nested reachability predicates.

In the context of approaches based on *separation logic*, several attempts to provide counterexample validation and automated abstraction refinement have appeared. In [3], the SLAYER analyzer was extended by a method to check spuriousness of counterexample traces via bounded model checking and SMT. Unlike our work, the approach may, however, fail in recognising that a given trace represents a real counterexample. Moreover, the associated refinement can only add more predicates to be tracked from a pre-defined set of such predicates. In [2], another counterexample analysis for the context of separation logic was proposed within a computation loop based on the Impact algorithm [18]. The approach uses bounded backwards abduction to derive so-called spatial interpolants and to distinguish between real and spurious counterexample traces. It allows for refinement of the predicates used but only by extending them by data-related properties. The basic predicates describing heap shapes are provided in advance and fixed. Another work based on backwards abduction is [5]. The work assumes working with a parametrized family of predicates, and the refinement is based on refining the parameter. Three concrete families of this kind are provided, namely, singly-linked lists in which one can remember bigger and bigger multisets of chosen data values, remember nodes with certain addresses, or track ordering properties. The basic heap predicates are again fixed. The approach does not guarantee recognition of spurious and real counterexamples nor progress of the refinement.

Unlike our approach, none of the so-far presented works is based on automata, and all of the works require some fixed set of shape predicates to be provided in advance.

Among *automata-based approaches*, counterexample analysis and refinement was used in [7] (and also in some related, less general approaches like [6]). In that case, however, a single tree automaton was used to encode sets of memory configurations, which allowed standard abstraction refinement from abstract regular (tree) model checking [10] to be used. On the other hand, due to using a single automaton, the approach did not scale well and had problems with some heap transformations.

The basic formalism of forest automata using fixed abstraction and user-provided database of boxes was introduced in [14]. We later extended the basic framework with automatic learning of boxes in [16]. The work [1] added ordering relations into forest automata to allow verification of programs whose safety depends on relations among data values from an unbounded domain. In [14,16], we conjectured that counterexample validation and abstraction refinement should be possible in the context of forest automata too. However, only now, do we show that this is indeed the case, but also that much more involved methods than those of [10] are needed.

## 3 Forest Automata and Heaps

We consider sequential non-recursive C programs, operating on a set of pointer variables and the heap, using standard statements and control flow constructs. Heap cells contain zero or several pointer or data fields.

Configurations of the considered programs consist of memory-allocated data and an assignment of variables. *Heap memory* can be viewed as a (directed) graph whose nodes correspond to allocated memory cells. Every node contains a set of named pointer and data fields. Each pointer field points to another node (we model the `NULL` and undefined locations as special memory nodes pointed by variables `NULL` and `undef`, respectively), and the same holds for pointer variables of the program. Data fields of memory nodes hold a data value. We use the term *selector* to talk both about pointer and data fields. For simplification, we model data variables as pointer variables pointing to allocated nodes that contain a single data field with the value of the variable, and therefore consider only pointer variables hereafter.

We represent heap memory by partitioning it into a tuple of trees, the so-called *forest*. The leaves of the trees contain information about roots of which trees they should be merged with to recover the original heap. Our *forest automata* symbolic representations of sets of heaps is based on representing sets of forests using tuples of tree automata.

Let us now formalize these ideas. In the following, we use $f : A \rightharpoonup B$ to denote a partial function from $A$ to $B$ (also viewed as a total function $f : A \to (B \cup \{\top\})$, assuming that $\top \notin B$). We also assume a bounded data domain $\mathbb{D}$.

*Graphs and Heaps.* Let $\Gamma$ be a finite set of *selectors* and $\Omega$ be a finite set of *references* s.t. $\Omega \cap \mathbb{D} = \emptyset$. A *graph* $g$ over $\langle \Gamma, \Omega \rangle$ is a tuple $\langle V_g, \mathit{next}_g \rangle$ where $V_g$ is a finite set of *nodes* and $\mathit{next}_g : \Gamma \to (V_g \rightharpoonup (V_g \cup \Omega \cup \mathbb{D}))$ maps each selector $a \in \Gamma$ to a partial mapping $\mathit{next}_g(a)$ from nodes to nodes, references, or data values. References and data values are treated as special terminal nodes that are not in the set of regular nodes, i.e., $V_g \cap (\Omega \cup \mathbb{D}) = \emptyset$. For a graph $g$, we use $V_g$ to denote the nodes of $g$, and for a selector $a \in \Gamma$, we use $a_g$ to denote the mapping $\mathit{next}_g(a)$. Given a finite set of variables $\mathbb{X}$, a *heap* $h$ over $\langle \Gamma, \mathbb{X} \rangle$ is a tuple $\langle V_h, \mathit{next}_h, \sigma_h \rangle$ where $\langle V_h, \mathit{next}_h \rangle$ is a graph over $\langle \Gamma, \emptyset \rangle$ and $\sigma_h : \mathbb{X} \to V_h$ is a (total) map of variables to nodes.

*Forest representation of heaps.* A graph $t$ is a *tree* if its nodes and pointers (i.e., not references nor data fields) form a tree with a unique root node, denoted $root(t)$. A *forest* over $\langle \Gamma, \mathbb{X} \rangle$ is a pair $\langle t_1 \cdots t_n, \sigma_f \rangle$ where $t_1 \cdots t_n$ is a sequence of trees over $\langle \Gamma, \{\overline{1}, \ldots, \overline{n}\} \rangle$ and $\sigma_f$ is a (total) mapping $\sigma_f : \mathbb{X} \to \{\overline{1}, \ldots, \overline{n}\}$. The elements in $\{\overline{1}, \ldots, \overline{n}\}$ are called *root references* (note that $n$ must be the number of trees in the forest). A forest $\langle t_1 \cdots t_n, \sigma_f \rangle$ over $\langle \Gamma, \mathbb{X} \rangle$ represents a heap over $\langle \Gamma, \mathbb{X} \rangle$, denoted $\otimes \langle t_1 \cdots t_n, \sigma_f \rangle$, obtained by taking the union of the trees of $t_1 \cdots t_n$ (assuming w.l.o.g. that the sets of nodes of the trees are disjoint), connecting root references with the corresponding roots, and mapping every defined variable $x$ to the root of the tree indexed by $x$. Formally, $\otimes \langle t_1 \cdots t_n, \sigma_f \rangle$ is the heap $h = \langle V_h, \mathit{next}_h, \sigma_h \rangle$ defined by (i) $V_h = \bigcup_{i=1}^{n} V_{t_i}$, and (ii) for $a \in \Gamma$ and $v \in V_{t_k}$, if $a_{t_k}(v) \in \{\overline{1}, \ldots, \overline{n}\}$ then $a_h(v) = root(t_{a_{t_k}(v)})$ else $a_h(v) = a_{t_k}(v)$, and finally (iii) for every $x \in \mathbb{X}$, $\sigma_h(x) = root(t_{\sigma_f(x)})$.

### 3.1 Forest Automata

A forest automaton is essentially a tuple of tree automata accepting a set of tuples of trees that represents a set of graphs via their forest decomposition, associated with a mapping of variables to root references.

*Tree automata.* A (finite, non-deterministic) *tree automaton* (TA) over $\langle \Gamma, \Omega \rangle$ is a triple $A = (Q, q_0, \Delta)$ where $Q$ is a finite set of *states* (we assume $Q \cap (\mathbb{D} \cup \Omega) = \emptyset$), $q_0 \in Q$ is the *root state* (or initial state), denoted $root(A)$, and $\Delta$ is a set of *transitions*. Each transition is of the form $q \to \overline{a}(q_1, \ldots, q_m)$ where $m \geq 0$, $q \in Q$, $q_1, \ldots, q_m \in (Q \cup \Omega \cup \mathbb{D})$[1], and $\overline{a} = a^1 \cdots a^m$ is a sequence of different symbols from $\Gamma$.

Let $t$ be a tree over $\langle \Gamma, \Omega \rangle$, and let $A = (Q, q_0, \Delta)$ be a TA over $\langle \Gamma, \Omega \rangle$. A *run* of $A$ over $t$ is a total map $\rho : V_t \to Q$ where $\rho(root(t)) = q_0$ and for each node $v \in V_t$ there is a transition $q \to \overline{a}(q_1, \ldots, q_m)$ in $\Delta$ with $\overline{a} = a^1 \cdots a^m$ such that $\rho(v) = q$ and for all $1 \leq i \leq m$, we have (i) if $q_i \in Q$, then $a_t^i(v) \in V_t$ and $\rho(a_t^i(v)) = q_i$, and (ii) if $q_i \in \Omega \cup \mathbb{D}$, then $a_t^i(v) = q_i$. We define the *language* of $A$ as $L(A) = \{t \mid \text{there is a run of } A \text{ over } t\}$, and the language of a state $q \in Q$ as $L(A, q) = L((Q, q, \Delta))$.

*Forest automata.* A *forest automaton* (FA) over $\langle \Gamma, \mathbb{X} \rangle$ is a tuple of the form $F = \langle A_1 \cdots A_n, \sigma \rangle$ where $A_1 \cdots A_n$, with $n \geq 0$, is a sequence of TAs over $\langle \Gamma, \{\overline{1}, \ldots, \overline{n}\} \rangle$ whose sets of states $Q_1$, ..., $Q_n$ are mutually disjoint, and $\sigma : \mathbb{X} \to \{\overline{1}, \ldots, \overline{n}\}$ is a mapping of variables to root references. A forest $\langle t_1 \cdots t_n, \sigma_f \rangle$ over $\langle \Gamma, \mathbb{X} \rangle$ is *accepted* by $F$ iff $\sigma_f = \sigma$ and there are runs $\rho_1, \ldots, \rho_n$ such that for all $1 \leq i \leq n$, $\rho_i$ is a run of $A_i$ over $t_i$. The *language* of $F$, denoted as $L(F)$, is the set of heaps over $\langle \Gamma, \mathbb{X} \rangle$ obtained by applying $\otimes$ on forests accepted by $F$.

*Cut-points and the dense form.* A *cut-point* of a heap $h$ is its node that is either pointed by some variable or is a target of more than one selector edge. The roots of forests that are not cut-points in the represented heaps are called *false roots*. A forest automaton is *dense* if its accepted forests do not have false roots. Each forest automaton can be transformed into a set of dense forest automata that together have the same language

[1] For simplicity, data values and references are used as special leaf states accepting the data values and references they represent, instead of having additional leaf transitions to accept them.

as the original. This property is a part of canonicity, which can be achieved by normalization, introduced in [14] for the purpose of checking entailment of forest automata. A transformation to the dense form is essential in the symbolic execution of a program.

### 3.2 Boxes

Forest automata, as defined in Sec. 3.1, can represent heaps with cut-points of an unbounded in-degree as, e.g., in singly-linked lists (SLLs) with head/tail pointers (indeed there can be any number of references from leaf nodes to a certain root). The basic definition of FAs cannot, however, deal with heaps with an unbounded number of cut-points since this would require an unbounded number of TAs within FAs. An example of such a set of heaps is the set of all doubly-linked lists (DLLs) of an arbitrary length, where each internal node is a cut-point. The solution provided in [14] is to allow FAs to use other nested FAs, called *boxes*, as symbols to "hide" recurring subheaps and in this way eliminate cut-points. The alphabet of a box itself may also include boxes, these boxes are, however, required to form a finite hierarchy—they cannot be recursively nested. The language of a box is a set of heaps over two special variables, `in` and `out`, which correspond to the input and the output port of the box. For simplicity of presentation, we give only a simplified version of boxes; see [14] for a more general definition that allows boxes with an arbitrary number of output ports.

A *nested forest automaton* over $\langle \Gamma, \mathbb{X} \rangle$ is an FA over $\langle \Gamma \cup \mathcal{B}, \mathbb{X} \rangle$ where $\mathcal{B}$ is a finite set of *boxes*. A *box* $B$ over $\Gamma$ is a nested FA $\langle A_1 \cdots A_n, \sigma_\Box \rangle$ over $\langle \Gamma, \{\mathtt{in}, \mathtt{out}\} \rangle$ such that $\sigma_\Box(\mathtt{in}) \neq \sigma_\Box(\mathtt{out})$ and $A_1 \cdots A_n$ do not contain an occurrence of $B$ (even a nested one). Unless stated otherwise, the FAs in the rest of the paper are nested.

In the case of a nested FA $F$, we need to distinguish between its language $L(F)$, which is a set of heaps over $\langle \Gamma \cup \mathcal{B}, \mathbb{X} \rangle$, and its *semantics* $[\![F]\!]$, which is a set of heaps over $\langle \Gamma, \mathbb{X} \rangle$ that emerges when all boxes in the heaps of the language are recursively *unfolded* in all possible ways. Formally, given heaps $h$ and $h'$, the heap $h'$ is an *unfolding* of $h$ if there is an edge $(B, u, v) \in \mathit{next}_h$ with a box $B = \langle A_1 \cdots A_n, \sigma_\Box \rangle$ in $h$, such that $h'$ can be constructed from $h$ by substituting $(B, u, v)$ with some $h_B \in [\![B]\!]$ such that $\sigma_\Box(\mathtt{in}) = u$ and $\sigma_\Box(\mathtt{out}) = v$. The substitution is done by removing $(B, u, v)$ from $h$ and uniting the heap-graph of $h$ with that of $h_B$. We then write $h \rightsquigarrow_{(B,u,v)/h_B} h'$, or only $h \rightsquigarrow h'$ if the precise edge $(B, u, v)$ and heap $h_B$ are not relevant. We use $\rightsquigarrow^*$ to denote the reflexive transitive closure of $\rightsquigarrow$. The *semantics* of $F$, written as $[\![F]\!]$, is the set of all heaps $h'$ over $\langle \Gamma, \mathbb{X} \rangle$ for which there is a heap $h$ in $L(F)$ such that $h \rightsquigarrow^* h'$.

## 4 Program Semantics

The dynamic behaviour of a program is defined by its control flow graph, a mapping $p : \mathbb{T} \to (\mathbb{L} \times \mathbb{L})$ where $\mathbb{T}$ is a set of program statements, and $\mathbb{L}$ is a set of program locations. Statements are partial functions $\tau : \mathbb{H} \rightharpoonup \mathbb{H}$ where $\mathbb{H}$ is the set of heaps over the selectors $\Gamma$ and variables $\mathbb{X}$ occurring in the program, which are used as representations of program configurations. The initial configuration is $h_{\mathsf{init}} = \langle \emptyset, \emptyset, \emptyset \rangle$. We assume that statements are indexed by their line of code, so that no two statements of a program are equal. If $p(\tau) = (\ell, \ell')$, then the program $p$ can move from $\ell$ to $\ell'$ while modifying

the heap $h$ at location $\ell$ into $\tau(h)$. We assume that $\mathbb{X}$ contains a special variable pc that always evaluates to a location from $\mathbb{L}$, and that every statement updates its value according to the target location. Note that a single program location can have multiple succeeding program locations (which corresponds, e.g., to conditional statements), or no successor (which corresponds to exit points of a program). We use $src(\tau)$ to denote $\ell$ and $tgt(\tau)$ to denote $\ell'$ in the pair above. Every program $p$ has a designated location $\ell_{\mathsf{init}}$ called its *entry point* and $\ell_{\mathsf{err}} \in \mathbb{L}$ called the error location[2].

A *program path* $\pi$ in $p$ is a sequence of statements $\pi = \tau_1 \cdots \tau_n \in \mathbb{T}^*$ such that $src(\tau_1) = \ell_{\mathsf{init}}$, and, for all $1 < i \leq n$, it holds that $src(\tau_i) = tgt(\tau_{i-1})$. We say that $\pi$ is *feasible* iff $\tau_n \circ \cdots \circ \tau_1(h_{\mathsf{init}})$ is defined. The program $p$ is safe if it contains no feasible program path with $tgt(\tau_n) = \ell_{\mathsf{err}}$. In the following, we fix a program $p$ with locations $\mathbb{L}$, variables $\mathbb{X}$, and selectors $\Gamma$.

## 5 Symbolic Execution with Forest Automata

Safety of the program $p$ is verified using symbolic execution in the domain $\mathbb{F}$ of forest automata over $\langle \Gamma, \mathbb{X} \rangle$. The program is executed symbolically by iterating abstract execution of program statements and a generalization step. These high-level operations are implemented as sequences of atomic operations and splitting. Atomic operations are functions of the type $o : \mathbb{F} \rightharpoonup \mathbb{F}$. Splitting splits a forest automaton $F$ into a set $\mathcal{S}$ of forest automata such that $[\![F]\!] = \bigcup_{F' \in \mathcal{S}} [\![F']\!]$. Splitting is necessary for some operations since forest automata are not closed under union, i.e., some sets of heaps expressible by a finite union of forest automata are not expressible by a single forest automaton.

To show an example of sets of heaps not expressible using a single FA, assume that the statement `x = y->sel` is executed on a forest automaton that encodes cyclic singly linked lists of an arbitrary length where `y` points to the head of the list. If the list is of length 1, then `x` will, after execution of the statement, point to the same location as `y`. If the list is longer, `x` and `y` will point to different locations. In the former case, the configuration has a single tree component, with both variables pointing to it. In the latter case, the two variables point to two different components. These two configurations cannot be represented using a single forest automaton.

The symbolic execution explores the program's *abstract reachability tree* (ART). Elements of the tree are forest automata corresponding to sets of reachable configurations at particular program locations. The tree is rooted by the forest automaton $F_{\mathsf{init}}$ s.t. $[\![F_{\mathsf{init}}]\!] = \{h_{\mathsf{init}}\}$. Every other node is a result of an application of an atomic operation or a split on its parent, and the applied operation is recorded on the tree edge between the two. The atomic operation corresponds to one of the following: symbolic execution of an effect of a program statement, generalization, or an auxiliary meta-operation that modifies the FAs while keeping its semantics (e.g., connects or cuts its components). Splitting appears in the tree as a node with several children connected via edges labelled by a special operation *split*. The said operations are described in more detail in Sec. 7.

The tree is expanded starting from the root as follows: First, a symbolic configuration in the parent node is generalized by iterating the following three operations:

[2] For simplification, we assume checking the error line (un-)reachability property only, which is, anyway, sufficient in most practical cases. For detection of garbage (which is not directly expressible as line reachability), we can extend the formalism and check for garbage after every command, and if a garbage is found, we jump to $\ell_{\mathsf{err}}$.

(i) transformation to the dense form, (ii) application of regular abstraction over-approximating sets of sub-graphs between cut-points of the represented heaps, (iii) folding boxes to decrease the number of cut-points in the represented heaps, until fixpoint. The transformation into the dense form is performed in order to obtain the most general abstraction in the subsequent step. A configuration where one more loop of the transformation-abstraction-folding sequence has no further effect is called *stable*. Operations implementing effects of statements are then applied on stable configurations. Exploration of a branch is terminated if its last configuration is entailed by a symbolic configuration with the same program location reached previously elsewhere in the tree.

A *symbolic path* is a path between a node and one of its descendants in the ART, i.e., a sequence of FAs and operations $\omega = F_0 o_1 F_1 \dots o_n F_n$ such that $F_i = o_i(F_{i-1})$. A *forward run* is a symbolic path where $F_0 = F_{\mathsf{init}}$. We write $\omega_i$ to denote the prefix of $\omega$ ending by $F_i$ and ${}_i\omega$ to denote its suffix from $F_i$. A forward run that reaches $\ell_{\mathsf{err}}$ is called an *abstract counterexample*. We associate every operation $o$ with its *exact semantics* $\hat{o}$, defined as $\hat{o}(H) = \bigcup_{h \in H} \{\tau(h)\}$ if $o$ implements the program statement $\tau$, and as the identity for all other operations (operations implementing generalization, splitting, etc.), for a set of heaps $H$. The *exact execution* of $\omega$ is a sequence $h_0 \cdots h_n$ such that $h_0 \in [\![F_0]\!]$ and $h_i \in \hat{o}(\{h_{i-1}\}) \cap [\![F_i]\!]$ for $0 < i \leq n$. We say that $\omega$ is *feasible* if it has an exact execution, otherwise it is *infeasible/spurious*. The atomic operations are either semantically precise, or over-approximate their exact semantics, i.e., it always holds that $\hat{o}([\![F]\!]) \subseteq [\![o(F)]\!]$. Therefore, if the exploration of the program's ART finds no abstract counterexample, there is no exact counterexample, and the program is safe.

The regular abstraction mentioned above is based on over-approximating sets of reachable configurations using some of the methods described later in Sec. 9. The analysis starts with some initial abstraction function, which may, however, be too rough and introduce spurious counterexamples. The main contribution of the present paper is that we are able to analyse abstract counterexamples for spuriousness using the so-called *backward run* (cf. Sec. 8), and if the counterexamples are indeed spurious, we can *refine* the abstraction used to avoid the given spurious error symbolic path, and continue with the analysis, potentially further repeating the analyse-refine steps. We will describe the backward run and abstraction refinement shortly in the following section and give a more thorough description in Sec. 8 and Sec. 9.

### 5.1 Counterexample Analysis and Abstraction Refinement

Assume that the forward run $\omega = F_0 o_1 F_1 \cdots o_n F_n$ is spurious. Then there must be an index $i > 0$ such that the symbolic path ${}_i\omega$ is feasible but ${}_{i-1}\omega$ is not. This means that the operation $o_i$ over-approximated the semantics of $\omega$ and introduced into $[\![F_i]\!]$ some heaps that are not in $\hat{o}_i([\![F_{i-1}]\!])$ and that are *bad* in the sense that they make ${}_i\omega$ feasible. An *interpolant for* $\omega$ is then a forest automaton $I_i$ representing the bad heaps of $[\![F_i]\!]$ that were introduced into $[\![F_i]\!]$ by the over-approximation in $o_i$ and are disjoint from $\hat{o}_i([\![F_{i-1}]\!])$. Formally,

1. $[\![I_i]\!] \cap \hat{o}_i([\![F_{i-1}]\!]) = \emptyset$ and
2. $\omega_i$ is infeasible from all $h \in [\![F_i]\!] \setminus [\![I_i]\!]$.

In the following, we describe how to use backward run, which reverts operations of the forward run on the semantic level, to check spuriousness of an abstract counterexample. Moreover, we show how to derive interpolants from backward runs reporting

spurious counterexamples, and how to use those interpolants to refine the operation of abstraction so that it will not introduce the bad configurations in the same way again. A *backward run* for $\omega$ is the sequence $\overline{\omega} = \overline{F}_0 \cdots \overline{F}_n$ such that

1. $\overline{F}_n = F_n$ and
2. $[\![\overline{F}_{i-1}]\!] = \hat{o}_i^{-1}([\![\overline{F}_i]\!]) \cap [\![F_{i-1}]\!]$, that is, $\overline{F}_{i-1}$ represents the *weakest precondition* of $[\![\overline{F}_i]\!]$ w.r.t. $\hat{o}_i$ that is *localized* to $[\![F_{i-1}]\!]$.

If there is an $\overline{F}_i$ such that $[\![\overline{F}_i]\!] = \emptyset$ (and, consequently, $[\![\overline{F}_0]\!] = \emptyset, \ldots, [\![\overline{F}_{i-1}]\!] = \emptyset$), the forward run is spurious. In such a case, an interpolant $I_i$ for $\omega$ can be obtained as $\overline{F}_{i+1}$ where $i+1$ is the smallest index such that $[\![\overline{F}_{i+1}]\!] \neq \emptyset$. We elaborate on the implementation of the backward run in Sec. 8.

We note that our use of interpolants differs from that of McMillan [21] in two aspects. First, due to the nature of our backward run, we compute an interpolant over-approximating the source of the suffix of a spurious run, not the effect of its prefix. Second, for simplicity of implementation in our prototype, we do not compute a sequence of localized interpolants but use solely the interpolant obtained from the beginning of the longest feasible suffix of the counterexample for a global refinement. It would also, however, be possible to use the sequence $\overline{F}_i, \ldots, \overline{F}_n$ as localized interpolants.

In Sec. 9, we show that using the interpolant $I_i$, it is possible to refine regular abstraction $o_i$ (the only over-approximating operation) to exclude the spurious run. The *progress guarantees* for the next iterations of the CEGAR loop are then the following:

1. for any FA $F$ such that $[\![F]\!] \subseteq [\![F_{i-1}]\!]$ that is compatible with $F_{i-1}$ (as defined in Sec. 6) it holds that $[\![o_i(F)]\!] \cap [\![I_i]\!] = \emptyset$,
2. forward runs $\omega' = F'_0 o_1 F'_1 \cdots o_n F'_n$ such that for all $1 \leq j \leq n$, $[\![F'_i]\!] \subseteq [\![F_i]\!]$ and $F'_i$ is compatible with $F_i$ are excluded from the ART.

The compatibility intuitively means that boxes are folding the same sub-heaps of represented heaps and that the TA components are partitioning them in the same way.

## 6 Intersection of Forest Automata

The previous section used intersection of semantics of forest automata to detect spuriousness of a counterexample. In this section, we give an algorithm that computes an under-approximation of the intersection of semantics of a pair of FAs, and later give conditions (which are, in fact, met by the pairs of FAs in our backward run analysis) on the intersected FAs to guarantee that the computed intersection is precise.

A simple way to compute the intersection of semantics of two FAs, denoted as $\cap$, is component-wise, that is, for two FAs $F = \langle A_1 \cdots A_n, \sigma \rangle$ and $F' = \langle A'_1 \cdots A'_n, \sigma \rangle$, we compute the FA $F \cap F' = \langle (A_1 \cap A'_1) \cdots (A_n \cap A'_n), \sigma \rangle$—note that the assignments need to be equal. The tree automata product construction for our special kind of tree automata synchronizes on data values and on references. That is, a pair $(a, b)$ that would be computed by a classical product construction where $a$ or $b$ is a reference or a data value is replaced by $a$ if $a = b$, and removed otherwise.

The above algorithm is, however, incomplete, i.e., it only guarantees $[\![F \cap F']\!] \subseteq [\![F]\!] \cap [\![F']\!]$. To increase the precision, we take into account the semantics of the boxes in the product construction, yielding a construction denoted using $\sqcap$. When synchronising two rules in the TA product, we recursively call intersection of forest automata. That is,

we compute the FA $F \sqcap F'$ in a similar way as $\cap$, but replace the tree automata product $A \cap A'$ by its variant $A \sqcap A'$. For $A = (Q, q_0, \Delta)$ and $A' = (Q', q'_0, \Delta')$, it computes the TA $A \sqcap A' = (Q \times Q', (q_0, q'_0), \Delta \sqcap \Delta')$ where $\Delta \sqcap \Delta'$ is built as follows:

$$\Delta \sqcap \Delta' = \big\{(q, q') \to \overline{a} \sqcap \overline{a}'((q_1, q'_1), \ldots, (q_m, q'_m)) \mid q \to \overline{a}(q_1, \ldots, q_m) \in \Delta, \\ q' \to \overline{a}'(q'_1, \ldots, q'_m) \in \Delta'\big\}.$$

Suppose $\overline{a} = a_1 \cdots a_m$, $\overline{a}' = a'_1 \cdots a'_m$, and that there is an index $0 \leq i \leq m$ such that if $j \leq i$, $a_j$ and $a'_j$ are not boxes, and if $i < j$, $a_j$ and $a'_j$ are boxes. The vector of symbols $\overline{a} \sqcap \overline{a}'$ is created as $(a_1 \sqcap a'_1) \cdots (a_m \sqcap a'_m)$ if $a_i \sqcap a'_i$ is defined for all $i$'s, othewise the transition is not created. The symbol $a_i \sqcap a'_i$ is defined as follows:

1. for $j \leq i$, $a_j \sqcap a'_j$ is defined as $a_j$ if $a_j = a'_j$ and is undefined otherwise,
2. for $j > i$, $a_j \sqcap a'_j$ is the intersection of FAs (both $a_j$ and $a'_j$ are boxes, i.e., FAs).

*Compatibility of forest automata.* For a forest automaton $F = \langle A_1 \cdots A_n, \sigma \rangle$, its version with marked components is the FA $F^D = \langle A_1 \cdots A_n, \sigma \cup \sigma_{\mathsf{root}} \rangle$ where $\sigma_{\mathsf{root}}$ is the mapping $\{\mathsf{root}_1 \mapsto 1, \ldots, \mathsf{root}_n \mapsto n\}$. The *root variables* $\mathsf{root}_i$ are fresh variables that point to the roots of the tree components in $L(F)$. $[\![F^D]\!]$ then contains the same heaps as $[\![F]\!]$, but the roots of the components from $L(F)$ remain visible as they are explicitly marked by the root variables. In other words, the root variables track how the forest decomposition of heaps in $L(F)$ partitions the heaps from $[\![F]\!]$. By removing the root variables of $h^D \in [\![F^D]\!]$, we get the original heap $h \in [\![F]\!]$. We call $h^D$ the *component decomposition of* $h$ *by* $F$.

Using the notion of component decomposition, we further introduce a notion of the *representation* of a heap by an FA. Namely, the *representation* of a box-free heap $h$ by an FA $F$ with $h \in [\![F]\!]$ records how $F$ represents $h$, i.e., (i) how $F$ decomposes $h$ into components, and (ii) how its sub-graphs enclosed in boxes are represented by the boxes. Formally, the representation of $h$ by $F$ is a pair $\mathit{repre} = (h^D, \{\mathit{repre}_1, \ldots, \mathit{repre}_n\})$ such that $h^D$ is the component decomposition of $h$ by $F$, and $\mathit{repre}_1, \ldots, \mathit{repre}_n$ are obtained from the sequence of unfoldings

$$h_0 \rightsquigarrow_{(B_1, u_1, v_1)/g_1} h_1 \rightsquigarrow_{(B_2, u_2, v_2)/g_2} \cdots \rightsquigarrow_{(B_n, u_n, v_n)/g_n} h_n$$

with $h_0 = h^D$ and $h_n \in L(F^D)$, such that for each $1 \leq i \leq n$, $\mathit{repre}_i$ is (recursively) the representation of $g_i$ in $B_i$.

We write $[\![\mathit{repre}]\!]$ to denote $\{h\}$, and, for a set of representations $R$, we let $[\![R]\!] = \bigcup_{\mathit{repre} \in R} [\![\mathit{repre}]\!]$. The set of *representations accepted by a forest automaton* $F$ is the set $\mathit{Repre}(F)$ of all representations of heaps from $[\![F]\!]$ by $F$. We say that a pair of FAs $F$ and $F'$ is *(representation) compatible* iff $[\![F]\!] \cap [\![F']\!] = [\![\mathit{Repre}(F) \cap \mathit{Repre}(F')]\!]$. The compatibility of a pair of FAs intuitively means that for every heap from the semantic intersection of the two FAs, at least one of its representations is shared by them.

**Lemma 1.** *For a pair $F$ and $F'$ of compatible FAs, it holds that $[\![F \sqcap F']\!] = [\![F]\!] \cap [\![F']\!]$.*

## 7 Implementation of the Forward Run

This section describes the operations that are used to implement the forward symbolic execution over FAs. To be able to implement the backward run, we will need to maintain compatibility between the forward run and the so-far constructed part of the backward run. Therefore, we will present the operations used in the forward run mainly from the point of view of their effect on the representation of heaps (in the sense of Sec. 6). Then, in Sec. 8, we will show how this effect is inverted in the backward run such that, when starting from compatible configurations, the inverted operations preserve compatibility of the configurations in the backward run with their forward run counterparts.

We omit most details of the way the operations are implementated on the level of manipulations with rules and states of FAs. We refer the reader to [14,25] for the details. We note that when we talk about removing a component or inserting a component in an FA, this also includes renaming references and updating assignments of variables. When a component is inserted at position $i$, all references to $\overline{j}$ with $j > i$ are replaced by $\overline{i+1}$, including the assignment $\sigma$ of variables. When a component is removed from position $i$, all references to $\overline{j}$ with $j > i$ are replaced by references to $\overline{j-1}$.

*Splitting.* Splitting has already been discussed in Sec. 5. It splits the symbolic execution into several branches such that the union of the FAs after the split is semantically equal to the original FA. The split is usually performed when transforming an FA into several FAs that have only one variant of a root rule of some of their components. From the point of view of a single branch of the ART, splitting is an operation, denoted further as $split$, that transforms an FA $F$ into an FA $F'$ s.t. $[\![F']\!] \subseteq [\![F]\!]$ and $Repre(F') \subseteq Repre(F)$. Therefore, $F$ is compatible with $F'$.

*Operations modifying component decomposition.* This class of operations is used to implement transformation of FAs to the dense form and as pre-processing steps before the operations of folding, unfolding, and symbolic implementation of program statements. They do not modify the semantics of forest automata, but change the component decomposition of the represented heaps.

- *Connecting of components.* When the $j$-th component $A_j$ of a forest automaton $F$ accepts trees with false roots, then $A_j$ can be connected to the component that refers to it. Indeed, as such roots are not cut-points, a reference $\overline{j}$ to them can appear only in a single component, say $A_k$, and at most once in every tree from its language (because a false root can have at most one incoming edge). For simplicity, assume that $A_j$ has only one root state $q$ that does not appear on the right-hand sides of rules. The connection is done by adding the states and rules of $A_j$ to $A_k$, replacing the reference $\overline{j}$ in the rules of $A_k$ by $q$. The $j$-th component is then removed from $F$. The previous sequence of actions is denoted as the operation $connect[j, k, q]$ below.
- *Cutting of a component.* Cutting divides a component with an index $j$ into two. The part of the $j$-th component containing the root will accept tree prefixes of the original trees, and the new $k$-th component will accept their remaining sub-trees. The cutting is done at a state $q$ of $A_j$, which appears exactly once in each run (the FA is first transformed to satisfy this). Occurrences of $q$ at the right-hand sides of

rules are replaced by the reference $\overline{k}$ to the new component, and $q$ becomes the root state of the new component. We denote this operation by $cut[j, k, q]$.
- *Swapping of components.* The operation $swap[j, k]$ swaps the $j$-th and the $k$-th component (and renames references and assignments accordingly).

*Folding of boxes.* The folding operation assumes that the concerned FA is first transformed into the form $F = \langle A_{\mathtt{in}} A_2 \cdots A_{n-1} A_{\mathtt{out}} A'_1 \cdots A'_m, \sigma \rangle$ by a sequence of splitting, cutting, and swapping. The tuple of TAs $A_{\mathtt{in}} A_2 \cdots A_{n-1} A_{\mathtt{out}}$ will then be folded into a new box $B$ with $A_{\mathtt{in}}$ as its input component and $A_{\mathtt{out}}$ as its output. Moreover, the operation is given sets of selectors $S_{\mathtt{in}}, S_{\mathtt{out}}$ of roots of components in $A_{\mathtt{in}}$ and $A_{\mathtt{out}}$ that are to be folded into $B$. The box $B = \langle A^B_{\mathtt{in}} A_2 \cdots A_{n-1} A^B_{\mathtt{out}}, \{\mathtt{in} \mapsto 1, \mathtt{out} \mapsto n\} \rangle$ arises from $F$ by taking $A_{\mathtt{in}} A_2 \cdots A_{n-1} A_{\mathtt{out}}$ and by removing selectors that are not in $S_{\mathtt{in}}$ and $S_{\mathtt{out}}$ from root rules of $A_{\mathtt{in}}$ and $A_{\mathtt{out}}$ to obtain $A^B_{\mathtt{in}}$ and $A^B_{\mathtt{out}}$ respectively.

Folding returns the forest automaton $F' = \langle A'_{\mathtt{in}} A'_{\mathtt{out}} A'_1 \cdots A'_m, \sigma' \rangle$ that arises from $F$ as follows. All successors of the roots accepted in $A_{\mathtt{in}}$ and $A_{\mathtt{out}}$ reachable over selectors from $S_{\mathtt{in}}$ and $S_{\mathtt{out}}$ are removed in $A'_{\mathtt{in}}$ and $A'_{\mathtt{out}}$ respectively (since they are enclosed in $B$). The root of the trees of $A'_{\mathtt{in}}$ gets an additional edge labelled by $B$, leading to the reference $\overline{n}$ (the output port), and the components $A_2 \cdots A_{n-1}$ are removed (since they are also enclosed in $B$). This operation is denoted as $fold[n, S_{\mathtt{in}}, S_{\mathtt{out}}, B]$.

*Unfolding of boxes.* Unfolding is called as a preprocessing step before operations that implement program statements in order to expose the selectors accessed by the statement. It is called after a sequence of cutting, splitting, and swapping that changes the forest automaton into the form $F' = \langle A'_{\mathtt{in}} A'_{\mathtt{out}} A'_1 \cdots A'_m, \sigma' \rangle$ where trees of $A'_{\mathtt{in}}$ have a reference $\overline{2}$ to $A'_{\mathtt{out}}$ accessible by an edge going from the root and labelled by the box $B$ that is to be unfolded. Furthermore, assume that the box $B$ is of the form $\langle A^B_{\mathtt{in}} A_2 \cdots A_{n-1} A^B_{\mathtt{out}}, \{\mathtt{in} \mapsto 1, \mathtt{out} \mapsto n\} \rangle$ and the input and the output ports have outgoing selectors from the sets $S_{\mathtt{in}}$ and $S_{\mathtt{out}}$ respectively. The operation returns the forest automaton $F$ that arises from $F'$ by inserting components $A^B_{\mathtt{in}} A_2 \cdots A_{n-1} A^B_{\mathtt{out}}$ in between $A'_{\mathtt{in}}$ and $A'_{\mathtt{out}}$, removing the $B$ successor of the root in $A'_{\mathtt{in}}$, merging $A^B_{\mathtt{in}}$ with $A'_{\mathtt{in}}$, and $A^B_{\mathtt{out}}$ with $A'_{\mathtt{out}}$. The merging on the TA level consists of merging root transitions of the TAs. We denote this operation as $unfold[n, S_{\mathtt{in}}, S_{\mathtt{out}}, B]$.

*Symbolic execution of program statements.* We will now discuss our symbolic implementation of the most essential statements of a C-like programming language. We assume that the operations are applied on an FA $F = \langle A_1 \cdots A_n, \sigma \rangle$.

- $\mathtt{x} := \mathtt{malloc}()$: A new $(n+1)$-th component $A_{new}$ is appended to $F$ s.t. it contains one state and one transition with all selector values set to $\sigma(\mathtt{undef})$. The assignment $\sigma(\mathtt{x})$ is set to $\overline{n+1}$.
- $\mathtt{x} := \mathtt{y}\text{->}\mathtt{sel}$ and $\mathtt{y}\text{->}\mathtt{sel} := \mathtt{x}$: If $\sigma(\mathtt{y}) = \sigma(\mathtt{undef})$, the operation moves to the error location. Otherwise, by splitting, cutting, and unfolding, $F$ is transformed into the form where $A_{\sigma(\mathtt{y})}$ has only one root rule and the rule has a $\mathtt{sel}$-successor that is a root reference $\overline{j}$. The statement $\mathtt{x} := \mathtt{y}\text{->}\mathtt{sel}$ then changes $\sigma(\mathtt{x})$ to $\overline{j}$, and $\mathtt{y}\text{->}\mathtt{sel} := \mathtt{x}$ changes the reference $\overline{j}$ in $A_{\sigma(\mathtt{y})}$ to $\sigma(\mathtt{x})$.
- $\mathtt{assume}(\mathtt{x} \sim \mathtt{y})$ where $\sim \in \{==, !=\}$: This statement tests the equality of $\sigma(\mathtt{x})$ and $\sigma(\mathtt{y})$ and stops the current branch of the forward run if the result does not match $\sim$.

- `assume(x->data` $\sim$ `y->data)` where $\sim$ is some data comparison: We start by unfolding and splitting $F$ into the form where $A_{\sigma(\mathtt{x})}$ and $A_{\sigma(\mathtt{y})}$ have only one root rule with exposed `data` selector. The data values at the `data` selectors are then compared and the current branch of the forward run is stopped if they do not satisfy $\sim$. The operation moves to the error locations if $\sigma(\mathtt{x})$ or $\sigma(\mathtt{x})$ are equal to $\sigma(\mathtt{undef})$.
- `free(x)`: The component $A_{\sigma(\mathtt{x})}$ is removed, and all references to $\sigma(\mathtt{x})$ are replaced by $\sigma(\mathtt{undef})$.

The updates are followed by checking that all components are reachable from program variables in order to detect garbage. If some component is not reachable, the execution either moves to the error location, or—if the analysis is set to ignore memory leaks—removes the unreachable component and continues with the execution.

*Regular Abstraction.* Regular abstraction is described in Sec. 9. It is preceded by a transformation to the dense form by connecting and splitting the FA.

## 8 Inverting Operations in the Backward Run

We now present how we compute the weakest localized preconditions (*inversions* for short) of the operations from Sec. 7 in the backward run. As mentioned in Sec. 7, it is crucial that compatibility with the forward run is preserved. Let $F_i = o(F_{i-1})$ appear in the forward run and $\overline{F}_i$ be an already computed configuration in the backward run s.t. $F_i$ and $\overline{F}_i$ are compatible. We will describe how to compute $\overline{F}_{i-1}$ such that it is also compatible with $F_{i-1}$.

Inverting most operations is straightforward. The operation $cut[j, k, q]$ is inverted by $connect[k, j, q_k]$ where $q_k$ is the root state of $A_k$, $swap[j, k]$ is inverted by $swap[k, j]$, and *split* is not inverted, i.e., $\overline{F}_{i-1} = \overline{F}_i$.

One of the more difficult cases is $connect[j, k, q]$. Assume for simplicity that $k$ is the index of the last component of $F_{i-1}$. Connecting can be inverted by cutting, but prior to that, we need to find *where* the $k$-th component of $\overline{F}_i$ should be cut. To find the right place for the cut, we will use the fact that the places of connection are marked by the state $q$ in the FA $F_i$ from the forward run. We use the tree automata product $\sqcap$ from Sec. 6, which propagates the information about occurrences of $q$ to $\overline{F}_i$, to compute the product of the $k$-th component of $F_i$ and the $k$-th component of $\overline{F}_i$. We replace the $k$-th component of $\overline{F}_i$ by the product, which results in an intermediate FA $\overline{F}'_i$. The product states with the first component $q$ now mark the places where the forward run connected the components (they were leaves referring to the $k$-th component). This is where the backward run will cut the components to revert the connecting. Before that, though, we replace the mentioned product states with $q$ by a new state $q'$. This replacement does not change the language because $q$ was appearing exactly once in every run (because in the forward run, it is the root state of the connected component that does not appear on the right-hand sides of rules), therefore, a product state with $q$ can appear at most once in every run of the product too. Finally, we compute $\overline{F}_{i-1}$ as $cut[k, j, q'](\overline{F}'_i)$.

Folding is inverted by unfolding and vice versa. Namely, $fold[n, S_{\mathtt{in}}, S_{\mathtt{out}}, B]$ is inverted by $unfold[n, S_{\mathtt{in}}, S_{\mathtt{out}}, B]$ and $unfold[n, S_{\mathtt{in}}, S_{\mathtt{out}}, B]$ by $fold[n, S_{\mathtt{in}}, S_{\mathtt{out}}, B']$

where the box $B'$ folded in the backward run might be semantically smaller than $B$ (since the backward run is returning with a subset of configurations of the forward run).

Regular abstraction is inverted using the intersection construction from Sec. 6. That is, if $o_i$ is a regular abstraction, then $\overline{F}_{i-1} = \overline{F}_i \sqcap F_{i-1}$.

Finally, inversions of abstract statements compute the FA $\overline{F}_{i-1} = \langle \bar{A}'_1 \cdots \bar{A}'_n, \bar{\sigma}' \rangle$ from $\overline{F}_i = \langle \bar{A}_1 \cdots \bar{A}_m, \bar{\sigma} \rangle$ and $F_{i-1} = \langle A_1 \cdots A_n, \sigma \rangle$ as follows:

- $\mathtt{x} = \mathtt{malloc}()$: We obtain $\overline{F}_{i-1}$ from $\overline{F}_i$ by removing the $j$-th TA, for $\bar{\sigma}(\mathtt{x}) = \overline{j}$. The value of $\bar{\sigma}'(\mathtt{x})$ is set to $\sigma(\mathtt{x})$.
- $\mathtt{x} := \mathtt{y}\text{->}\mathtt{sel}$: Inversion is done by setting $\bar{\sigma}'(\mathtt{x})$ to the value of $\sigma(\mathtt{x})$ from $F_{i-1}$.
- $\mathtt{y}\text{->}\mathtt{sel} := \mathtt{x}$: The target of the $\mathtt{sel}$-labelled edge from the root of $A_{\bar{\sigma}'(\mathtt{y})}$ is set to its target in $A_{\sigma(\mathtt{y})}$.
- $\mathtt{assume}(...)$: Tests do not modify FAs and as we are returning with a subset of configurations from the forward run, they do not need to be inverted, i.e., $\overline{F}_{i-1} = \overline{F}_i$.
- $\mathtt{free}(\mathtt{x})$: First, the component of $F_{i-1}$ at the index $\sigma(\mathtt{x})$, which was removed in the forward run, is inserted at the same position in $\overline{F}_i$, and $\bar{\sigma}'(\mathtt{x})$ is set to that position. Then we must invert the rewriting of root references pointing to $\sigma(\mathtt{x})$ to $\sigma(\mathtt{undef})$ done by the forward run. For this, we compute the $\sqcap$ forest automata product from Sec. 6 with $F_{i-1}$, but modified so that instead of discarding reached pairs $(\sigma(\mathtt{undef}), \sigma(\mathtt{x}))$, it replaces them by $\sigma(\mathtt{x})$. Intuitively, the references to $\mathtt{x}$ are still present at $F_{i-1}$, so their occurrences in the product mark the occurrences of references to $\mathtt{undef}$ that were changed to point to $\mathtt{undef}$ by $\mathtt{free}(\mathtt{x})$. The modified product therefore redirects the marked root references to $\mathtt{undef}$ back to $\mathtt{x}$.

*The role of compatibility in the backward run.* Inversions of regular abstraction, component connection, and $\mathtt{free}(\mathtt{x})$, use the TA product construction $\sqcap$ from Sec. 6. The precision of all intersection and product computations in the backward run depends on the compatibility of the backward and forward run. Inverting the program statements also depends on the compatibility of the backward and forward run. Particularly, inversions of $\mathtt{x} := \mathtt{y}\text{->}\mathtt{sel}$ and $\mathtt{y}\text{->}\mathtt{sel} := \mathtt{x}$ use indices of components from $F_{i-1}$. They therefore depend on the property that heaps from $\overline{F}_i$ are decomposed into components in the same way. The compatibility is achieved by inverting every step of folding and unfolding, and every operation of connecting, cutting, and swapping of components.

## 9 Regular Abstractions over Forest Automata

Our abstraction over FAs is based on automata abstraction from the framework of *abstract regular tree model checking* (ARTMC) [10]. This framework comes with two abstractions for tree automata, *finite height abstraction* and *predicate abstraction*. Both of them are based on merging states of a tree automaton that are equivalent according to a given equivalence relation. Formally, given a tree automaton $A = (Q, q_0, \Delta)$, its abstraction is the TA $\alpha(A) = (Q/_\sim, [q_0]_\sim, \Delta_\sim)$ where $\sim$ is an equivalence relation on $Q$, $Q/_\sim$ is the set of $\sim$'s equivalence classes, $[q_0]_\sim$ denotes the equivalence class of $q_0$, and $\Delta_\sim$ arises from $\Delta$ by replacing occurrences of states in transitions by their equivalence classes. It holds that $|Q/_\sim| \leq |Q|$ and $L(A) \subseteq L(\alpha(A))$.

*Finite height abstraction* is a function $\alpha_h$ that merges states with languages equivalent up to a given tree height $h$. Formally, it merges states of $A$ according to the equivalence relation $\sim^h$ defined as follows: $q_1 \sim^h q_2 \Leftrightarrow L^{\leq h}(A, q_1) = L^{\leq h}(A, q_2)$ where $L^{\leq h}(A, q)$ is the language of tree prefixes of trees from of $L(A, q)$ up to the height $h$.

*Predicate language abstraction* is a function $\alpha^{[\mathcal{P}]}$ parameterized by a set of predicate languages $\mathcal{P} = \{P_1, \ldots, P_n\}$ represented by tree automata. States are merged according to the equivalence $q \sim_{\mathcal{P}} q'$ which holds for the two states if their languages $L(A, q)$ and $L(A, q')$ intersect with the same subset of predicate languages from $\mathcal{P}$.

*Abstraction on forest automata.* We extend the abstractions from ARTMC to FAs by applying the abstraction over TAs to the components of the FAs. Formally, let $\alpha$ be a tree automata abstraction. For an FA $F = \langle A_1 \cdots A_n, \sigma \rangle$, we define $\alpha(F) = \langle \alpha(A_1) \cdots \alpha(A_n), \sigma \rangle$. Additionally, in the case of predicate abstraction, which uses automata intersection to annotate states by predicate languages, we use the intersection operator $\sqcap$ from Sec. 6, which descends recursively into boxes, and it is thus more precise from the point of view of the semantics of FAs. Since the abstraction only over-approximates languages of the individual components, it holds that $[\![F]\!] \subseteq [\![\alpha(F)]\!]$ and $\mathit{Repre}(F) \subseteq \mathit{Repre}(\alpha(F))$—and so $F$ and $\alpha(F)$ are compatible.

*Abstraction refinement.* The finite height abstraction may be refined by simply increasing the height $h$. Advantages of finite height abstraction include its relative simplicity and the fact that the refinement does not require counterexample analysis. A disadvantage is that the refinement by increasing the height is quite rough. Moreover, the cost of computing in the abstract domain rises quickly with increasing the height of the abstraction as exponentially more concrete configurations may be explored before the abstraction closes the analysis of a particular branch. The finite height abstraction was used—in a specifically fine-tuned version—in the first versions of FORESTER [14,16], which successfully verified a number of benchmarks, but the refinement was not sufficiently flexible to prove some more challenging examples.

Predicate abstraction, upon which we build in this paper, offers the needed additional flexibility. It can be refined by adding new predicates to $\mathcal{P}$ and it gives strong guarantees about excluding counterexamples. In ARTMC, interpolants in the form of tree automata $I_i$ are extracted from spurious counterexamples in the way described in Sec. 5.1. The interpolant is then used to refine the abstraction so that the spurious run is excluded from the program's ART.

The guarantees shown to hold in [10] on the level of TAs are the following. Let $A$ and $I = (Q, q_0, \Delta)$ be two TAs and let $\mathcal{P}(I) = \{L(I, q) \mid q \in Q\}$ denote the set of languages of states of $I$. Then, if $L(A) \cap L(I) = \emptyset$, it is guaranteed that $L(\alpha^{[\mathcal{P}(I)]}(A)) \cap L(I) = \emptyset$. That is, when the abstraction is refined with languages of all states of $I$, it will exclude $L(I)$—unless applied on a TA whose language is already intersecting $L(I)$.

We can generalize the result of [10] to forest automata in the following way, implying the progress guarantees of CEGAR described in Section 5.1. For a forest automaton $F = \langle A_1 \cdots A_n, \sigma \rangle$, let $\mathcal{P}(F) = \bigcup_{i=1}^{n} \mathcal{P}(A_i)$.

**Lemma 2.** *Let $F$ and $I$ be FAs s.t. $I$ is compatible with $\alpha^{[\mathcal{P}]}(F)$ and $[\![F]\!] \cap [\![I]\!] = \emptyset$. Then $[\![\alpha^{[\mathcal{P} \cup \mathcal{P}(I)]}(F)]\!] \cap [\![I]\!] = \emptyset$.*

We note that the lemma still holds if $\mathcal{P}(I)$ is replaced by $\mathcal{P}(A_i)$ only where $A_i$ is the $i$-th component of $I$ and $L(A_i \sqcap A_i') = \emptyset$ for the $i$-th component $A_i'$ of $\alpha^{[\mathcal{P}]}(F)$.

## 10 Experiments

We have implemented our counterexample analysis and abstraction refinement as an extension of FORESTER and evaluated it on a set of C programs manipulating singly-

**Table 1.** Results of experiments.

| Program | Status | LoC | Time [s] | Refnm | Preds | Program | Status | LoC | Time [s] | Refnm | Preds |
|---|---|---|---|---|---|---|---|---|---|---|---|
| SLL (delete) | safe | 33 | 0.02 | 0 | 0 | DLL (rev) | safe | 39 | 0.70 | 0 | 0 |
| SLL (bubblesort) | safe | 42 | 0.02 | 0 | 0 | CDLL | safe | 32 | 0.02 | 0 | 0 |
| SLL (insersort) | safe | 36 | 0.04 | 0 | 0 | DLL (insersort) | safe | 42 | 0.56 | 0 | 0 |
| SLLOfCSLL | safe | 47 | 0.02 | 0 | 0 | DLLOfCDLL | safe | 54 | 1.76 | 0 | 0 |
| **SLL01** | safe | 70 | 1.20 | 1 | 1 | **DLL01** | safe | 73 | 0.65 | 2 | 2 |
| **CircularSLL** | safe | 49 | 3.57 | 3 | 3 | **CircularDLL** | safe | 52 | 37.22 | 18 | 24 |
| **OptPtrSLL** | safe | 59 | 1.90 | 3 | 3 | **OptPtrDLL** | safe | 62 | 1.87 | 5 | 5 |
| **QueueSLL** | safe | 71 | 11.32 | 10 | 10 | **QueueDLL** | safe | 74 | 44.68 | 14 | 14 |
| **GBSLL** | safe | 64 | 0.84 | 3 | 3 | **GBDLL** | safe | 71 | 1.89 | 4 | 4 |
| **GBSLLSent** | safe | 68 | 0.85 | 3 | 3 | **GBDLLSent** | safe | 75 | 2.19 | 4 | 4 |
| **RGSLL** | safe | 72 | 14.41 | 22 | 38 | **RGDLL** | safe | 76 | 78.76 | 26 | 26 |
| **WBSLL** | safe | 62 | 0.84 | 5 | 5 | **WBDLL** | safe | 71 | 1.37 | 7 | 7 |
| **SortedSLL** | safe | 76 | 227.12 | 15 | 15 | **SortedDLL** | safe | 82 | 36.67 | 11 | 11 |
| **EndSLL** | safe | 45 | 0.07 | 2 | 2 | **EndDLL** | safe | 49 | 0.10 | 3 | 3 |
| **TreeRB** | error | 130 | 0.08 | 0 | 0 | **TreeWB** | error | 125 | 0.05 | 0 | 0 |
| TreeCnstr | safe | 52 | 0.31 | 0 | 0 | **TreeCnstr** | error | 52 | 0.03 | 0 | 0 |
| TreeOfCSLL | safe | 109 | 0.57 | 0 | 0 | **TreeOfCSLL** | error | 109 | 0.56 | 1 | 3 |
| TreeStack | safe | 58 | 0.20 | 0 | 0 | **TreeStack** | error | 58 | 0.01 | 0 | 0 |
| TreeDsw | safe | 72 | 1.87 | 0 | 0 | **TreeDsw** | error | 72 | 0.02 | 0 | 0 |
| TreeRootPtr | safe | 62 | 1.43 | 0 | 0 | **TreeRootPtr** | error | 62 | 0.17 | 2 | 6 |
| SkipList | safe | 84 | 3.36 | 0 | 0 | **SkipList** | error | 84 | 0.08 | 1 | 1 |

and doubly-linked list, trees, skip-lists, and their combinations. We were able to analyse all of them fully automatically without any need to supply manually crafted predicates nor any other manual aid. The test cases are described in detail in App. A.

We present our experimental results in Table 1. The table gives for each test case its name, information whether the program is safe or contains an error, the number of lines of code, the time needed for the analysis, the number of refinements, and, finally, the number of predicates learnt during the abstraction refinement. The experiments were performed on a computer with Intel Core i5@2.50 GHz CPU and 8 GiB of memory running the Debian Sid OS with the Linux kernel.

Some of the test cases consider dynamic data structures without any data stored in them, some of them data structures storing finite-domain data. Such data can be a part of the data structure itself, as, e.g., in red-black trees, they can arise from some finite data abstraction, or they are also sometimes used to mark some selected nodes of the data structure when checking the way the data structure is changed by a given algorithm (e.g., one can check whether an arbitrarily chosen successive pair of nodes of a list marked red and green is swapped when the list is reversed—see e.g. [10]).

As the results show, some of our test cases do not need refinement. This is because the predicate abstraction is *a priori* restricted in order to preserve the forest automata "interconnection graph" [16], which roughly corresponds to the reachability relation among variables and cut-points in the heaps represented by a forest automaton (an approach used already with the finite height abstraction in former versions of FORESTER).

Table 1 also provides a comparison with the previous version of FORESTER from [16]. In particular, the highlighted cases are not manageable by that versions of FORESTER. These cases can be split into two classes. In the first class there are safe programs where the initial abstraction is too coarse and introduces spurious counterexamples, and the abstraction thus needs to be refined. The other class consists of programs containing a real

error (which could not be confirmed without the backward run). The times needed for analysis are comparable in both versions of FORESTER.

To illustrate a typical learnt predicate, let us consider the test case *GBSLL*. This program manipulates a list with nodes storing two data values, green and blue, for which it holds that a green node is always followed by a blue one. The program also contains a tester code to test this property. FORESTER first learns two predicates describing particular violations of the property: (1) a green node is at the end of the list and (2) there are two green nodes in a row. After that, FORESTER derives a general predicate representing all lists with the needed invariant, i.e, a green node is followed by a blue one. The program is then successfully verified.

Another example comes from the analysis of the program *TreeCSLL*, which creates and deletes a tree where every tree node is also the head of a circular list. It contains an undefined pointer dereference error in the deletion of the circular lists. FORESTER first finds a spurious error (an undefined pointer dereference too) in the code that creates the circular lists. In particular, the abstraction introduces a case in which a tree node that is also the head of a list needs not be allocated, and an attempt of accessing its next selector causes an undefined pointer dereference error. This situation is excluded by the first refinement, after which the error within the list deletion is correctly reported. Notice that, in this case, the refinement learns a property of the shape, not a property over the stored data values. The ability to learn shape as well as data properties (as well as properties relating shape with data) using a uniform mechanism is one the features of our method which distinguishes it from most of the related work.

## 11 Discussion and Future Work

Both the described forward and backward symbolic execution are quite fast. We believe that the efficiency of the backward run (despite the need of computing expensive automata products) is to a large degree because it inverts unfolding (by folding). Backward run is therefore carried out with configurations encoded in a compact folded form.

FORESTER was not able to terminate on a few tree benchmarks. For a program manipulating a red-black tree using the rebalancing procedures, the initial forward run did not terminate. For another tree-based implementation of a set that includes a tester code checking full functional correctness, the CEGAR did not learn the right predicates despite many refinements. The non-termination of the forward run is probably related to the initial restrictions of the predicate abstraction. Restricting the abstraction seems to be harmful especially in the case of tree structures. If the abstraction remembers unnecessary fine information about tree branches, the analysis will explore exponentially many variants of tree structures with different branches satisfying different properties. The scenario where CEGAR seems to be unable to generalize is related to the splitting of the symbolic execution. The symbolic runs are then too specialised and CEGAR learns a large number of too specialised predicates from them (which are sometimes irrelevant to the "real" cause of the error).

A closer examination and resolution of these issues is a part of our future work. Allowing the abstraction more freedom is mostly an implementation issue, although nontrivial to achieve in the current implementation of FORESTER. Resolving the issue of splitting requires to cope with the domain of forest automata not being closed under union. This is possible, e.g., by modifying the definition of the FA language, which

currently uses the Cartesian product of sets of trees, so that it would connect tree components based on reachability relation between them (instead of taking all elements of the Cartesian product). Another possibility would be to use sets of forest automata instead of individual ones as the symbolic representation of sets of heaps.

## References


1. Abdulla, P.A., Holík, L., Jonsson, B., Lengál, O., Trinh, C.Q., Vojnar, T.: Verification of heap manipulating programs with ordered data by extended forest automata. Acta Informatica 53(4), 357–385 (2016), http://dx.doi.org/10.1007/s00236-015-0235-0
2. Albarghouthi, A., Berdine, J., Cook, B., Kincaid, Z.: Spatial interpolants. In: Proc. of ESOP'15. LNCS, vol. 9032. Springer (2015)
3. Berdine, J., Cox, A., Ishtiaq, S., Wintersteiger, C.: Diagnosing abstraction failure for separation logic-based analyses. In: Proc. of CAV'12. LNCS, vol. 7358. Springer (2012)
4. Beyer, D., Henzinger, T.A., Théoduloz, G.: Lazy shape analysis. In: Proceedings of the 18th International Conference on Computer Aided Verification—CAV'06. LNCS, vol. 4144, pp. 532–546. Springer (2006)
5. Botinčan, M., Dodds, M., Magill, S.: Refining existential properties in separation logic analyses. Tech. Rep. arXiv:1504.08309 (2015)
6. Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In: Proc. of TACAS'05. LNCS, vol. 3440. Springer (2005)
7. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: Proc. of SAS'06. LNCS, vol. 4134. Springer (2006)
8. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. Formal Methods in System Design 38(2), 158–192 (2011)
9. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis—ATVA'12. LNCS, vol. 7561, pp. 167–182. Springer (2012)
10. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular (tree) model checking. International Journal on Software Tools for Technology Transfer 14(2), 167–191 (2012)
11. Chang, B.Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Proceedings of the 14th International Static Analysis Symposium—SAS'07. LNCS, vol. 4634, pp. 384–401. Springer (2007)
12. Deshmukh, J., Emerson, E., Gupta, P.: Automatic Verification of Parameterized Data Structures. In: Proc. of TACAS'06. LNCS, vol. 3920. Springer (2006)
13. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Proceedings of the 20th International Static Analysis Symposium—SAS'13. LNCS, vol. 7935, pp. 215–237. Springer (2013)
14. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest automata for verification of heap manipulation. Formal Methods in System Design 41(1), 83–106 (2012)
15. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph grammar abstraction for unbounded heap structures. In: Proceedings of the 3rd International Workshop on Harnessing Theories for Tool Support in Software—TTSS'09. ENTCS, vol. 266, pp. 93–107. Elsevier (2010)
16. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully automated shape analysis based on forest automata. In: Proceedings of the 25th International Conference on Computer Aided Verification—CAV'13. LNCS, vol. 8044, pp. 740–755. Springer (2013)

17. Jensen, J.L., Jørgensen, M.E., Schwartzbach, M.I., Klarlund, N.: Automatic verification of pointer programs using monadic second-order logic. In: Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation—PLDI'97. pp. 226–234. ACM (1997)
18. Jhala, R., McMillan, K.: Lazy abstraction with interpolants. In: Proc. of CAV'06. LNCS, vol. 4144. Springer (2006)
19. Loginov, A., Reps, T., Sagiv, M.: Abstraction refinement via inductive learning. In: Proceedings of the 17th International Conference on Computer Aided Verification—CAV'05. LNCS, vol. 3576, pp. 519–533. Springer (2005)
20. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: Proceedings of the 37th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'10. pp. 211–222. ACM (2010)
21. McMillan, K.L.: Interpolation and SAT-based model checking. In: CAV'03. LNCS, vol. 2725, pp. 1–13. Springer (2003)
22. Podelski, A., Wies, T.: Counterexample-guided focus. In: Proceedings of the 37th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'10. pp. 249–260. ACM (2010)
23. Qin, S., He, G., Luo, C., Chin, W.N., Chen, X.: Loop invariant synthesis in a combined abstract domain. Journal of Symbolic Computation 50, 386–408 (2013)
24. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
25. Šimáček, J.: Harnessing Forest Automata for Verification of Heap Manipulating Programs. Ph.D. thesis, Grenoble Alpes University, France (2012), `https://tel.archives-ouvertes.fr/tel-00805794`
26. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Proceedings of the 20th International Conference on Computer Aided Verification—CAV'08. LNCS, vol. 5123, pp. 385–398. Springer (2008)

## A Description of the Test Cases

This section describes the test cases used in the experimental evaluation in Sec. 10. Note that we use a limited set of integer values since we do not support integer abstraction. SLL and DLL denote singly- and doubly-linked lists respectively.

The cases described in the following list satisfy some (regular-expressible) invariant, which we check in our analysis. Moreover, we also verify memory safety properties (absence of `null`/undefined pointer dereference, invalid free, and presence of garbage) in all test cases.

- *(SLL/DLL)01*: The nodes of the list may or may not point to an external node, which, if present, is unique for each list item. We check the invariant that each node has a pointer set to `null` or to an address of an external node.
- *Circular(SLL/DLL)*: A circular linked list consisting of nodes with integer values. The head of the list has the dedicated value $0$. The rest of nodes with their integer values form a non-decreasing sequence. We verify that the successor of an arbitrary node can have a smaller value only when the next node is the head of the list.
- *OptPtr(SLL/DLL)*: Each node of the list has an integer value, a pointer to the next node, and an optional pointer to an external node. When constructing the list, an integer value of every node is chosen nondeterministically. When the integer value $0$ or $1$ is chosen, the optional pointer points to the node itself. On the other hand, when $2$ is chosen, a new external node is allocated and its address is assigned to the optional pointer. We verify the relation of integer values and optional pointers for all nodes.
- *Queue(SLL/DLL)*: We create a list with nodes containing the integers 0, 1, 2, and 3. The list can form sequences $0$, $01$, $012$, $0123^*$. A particular sequence is created during construction of the list nondeterministically. We remember which sequence was actually created by an auxiliary integer variable. Then we traverse the list and check that the sequence formed by the list corresponds to the value of the auxiliary variable.
- *GB(SLL/DLL)*: We create a list containing green and blue nodes. The colors arbitrarily alternate but it holds that a green node is always followed by a blue node.
- *GB(SLL/DLL)Sen*: This case is similar to the previous one but instead of terminating the list with the `null` value, the list is terminated using a dedicated sentinel node.
- *RG(SLL/DLL)*: The list contains an arbitrary prefix of white nodes, one red node followed by a green one, and an arbitrary suffix of white nodes. The list is reversed and it is checked whether the green node is followed by the red node.
- *WB(SLL/DLL)*: Exactly one blue node is inserted into a list of white nodes of an arbitrary length. Then the list is traversed and it is checked that the number of blue nodes is one.
- *Sorted(SLL/DLL)*: This test case contains a sorted list of nodes with integer values $0$ and $1$. A node with value $1$ is added at an arbitrary position that keeps the order of nodes in the list. Finally, it is checked that the list is still ordered.
- *End(SLL/DLL)*: The last element of list has a special integer value.
- *SkipList*: Construction and traversal of a skip list.

- *TreeRB*: We construct a red black tree and then go through the tree checking (regular) invariants of this data structure. The created tree has an arbitrary height, and the nodes may have both, one, or no child allocated. A transposition of nodes needed to preserve the data structure's invariants is done continuously during construction of the tree when a new node is added. This operation is complex since it requires relocation of nodes in several levels of trees. The nodes also have to have parent pointers due to the transposition.
- *TreeWB*: We construct a tree that has all nodes white except exactly one blue node. The blue node is at an arbitrary position. We traverse the tree and check that there is a single blue node.
- *TreeWBAllPaths*: A tree with all nodes being white is constructed nondeterministically. We transform the created tree to the form where each path from the root to any leaf contains exactly one blue node. The blue nodes are placed arbitrarily in the tree with respect to the described invariant of the structure. Then we start an arbitrary number of arbitrary walks from the root of tree to a leaf and check that each of these walks contains exactly one blue node.

The following test cases from our benchmark are checked only for memory safety properties.

- *TreeCnstr*: The construction of an arbitrary binary tree.
- *TreeDsw*: We construct a binary tree and perform the Deutsch-Schorr-Waite traversal.
- *TreeCSLL*: We construct a binary tree where each node points to a circular singly linked list of an arbitrary length. Then we traverse the whole tree and all nested lists.
- *TreeRootPtr*: The construction and traversal of a binary tree with nodes containing root pointers.
- *TreeStack*: The construction of a binary tree which is subsequently destroyed using stack implemented by a singly-linked list.