Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 266 (2010) 93–107
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures

Jonathan Heinen[1] Thomas Noll[2] Stefan Rieger[3]

Software Modeling and Verification Group
RWTH Aachen University
Aachen, Germany

Abstract

We present a novel abstraction framework for heap data structures that uses graph grammars, more precisely context-free hyperedge replacement grammars, as an intuitive formalism for efficiently modeling dynamic data structures. It aims at extending finite-state verification techniques to handle pointer-manipulating programs operating on complex dynamic data structures that are potentially unbounded in their size. We demonstrate how our framework can be employed for analysis and verification purposes by instantiating it for binary trees, and by applying this instantiation to the well-known Deutsch-Schorr-Waite traversal algorithm. Our approach is supported by a prototype tool, enabling the quick verification of essential program properties such as heap invariants, completeness, and termination.

Keywords: Hyperedge Replacement Grammars, Abstraction, Verification, Model Checking, Pointer Programs

1 Introduction and Related Work

For analyzing pointer programs operating on dynamically allocated data structures, abstraction techniques are indispensable. Accordingly, a wide variety of techniques have been developed to tackle this problem, such as *shape analysis* [5,24], *predicate abstraction* [2,8,18], *regular model checking* [7], and *separation logic* [17,22], which has lately become a very active field of research.

As every state of the heap can be considered as a graph (interpreting cells as vertices and pointers as edges), it is quite natural to model the dynamic behavior of pointer programs by means of *graph transformations*. This analogy is the basis of verification methods which analyze the behavior of graph transformation systems

[1] Email: heinen@cs.rwth-aachen.de
[2] Email: noll@cs.rwth-aachen.de
[3] Email: rieger@cs.rwth-aachen.de

doi:10.1016/j.entcs.2011.07.001

[3,4,12]. In particular, there is a strand of research in which pointer manipulations are represented by graph transformation rules [1,9,19,21].

However, almost all of these approaches suffer from the necessity to develop an *abstract* transformation for each operation modifying a data structure, in a way that is dependent on the employed abstraction method to ensure the finiteness of the state space. As the detailed analysis of [16] exemplifies, such abstract versions of the concrete pointer operations are generally hard to find.

To tackle this problem, we have developed a novel framework – *Juggrnaut (Just Use Graph GRammars for Nicely Abstracting Unbounded sTructures)* – which allows to automatically derive abstract versions of pointer-manipulating operations, and which has been introduced in [23]. Our approach is to model states of the heap as *hypergraphs*, and to represent both pointer operations and abstraction mappings by *hypergraph transformations*. More concretely we employ *hyperedge replacement grammars* [10] to specify data structures and their abstractions. The key idea is to use the replacement operations which are induced by the grammar rules in two directions. By a *backward* application of some rule, a subgraph of the heap can be condensed into a single nonterminal hyperedge, thus obtaining an *abstraction* of the heap. By applying rules in *forward* direction, certain parts of the heap which have been abstracted before can be *concretized* again. Later we will see that the introduction of concretization steps will avoid the necessity for explicitly defining the effect of pointer-manipulating operations on abstracted parts of the heap: it is obtained "for free" by combining forward rule application (if required), the concrete pointer operation, and re-abstraction by backward rule application.

Another grammar-based approach to heap abstraction is presented in [13]. However, it only supports tree data structures. In contrast, the expressivity of hyperedge replacement grammars does not only allow us to define abstractions for tree-shaped data, but also for strongly connected structures such as cyclic lists [23].

Verification of the Deutsch-Schorr-Waite tree traversal algorithm has already been considered in [7,15]. We thought it to be an interesting example for proving the practical feasibility of our approach and for comparing our results with others. For more details please consult Section 4.

Thus our approach is unique in that it offers a new, descriptive way for specifying abstractions on a wide range of heap data structures. It supports dynamic memory allocation (entailing unbounded heap sizes) and destructive updates. In addition it is easily extendable to concurrent programs with dynamic thread creation along the lines of [16].

The remainder of this paper is organized as follows. Sections 2 and 3 pick up the required preliminaries from [23], respectively presenting the concept of hyperedge replacement grammars and our heap abstraction technique (with a new application example). Section 4 constitutes the main part of this paper and describes the verification of the Deutsch-Schorr-Waite tree traversal algorithm using our prototype tool. Finally, Section 5 concludes the presentation.

2 Hyperedge Replacement

As the formal basis of the Juggrnaut framework we employ hyperedge replacement grammars as they provide sufficient expressive strength for our application but are context-free. In the following we introduce some notations that will be useful in the specification of our framework.

Given a set S, $S^\star$ is the set of all finite sequences (strings) over S including the empty sequence ε. For $s \in S^\star$ the length of s is denoted by $|s|$, the set of all elements of s is written $[s]$, and by $s(i)$ we refer to the ith component of s. Given a tuple $t = (A, B, C, ...)$ we write A_t, B_t etc. for the components if their names are clear from the context.

The domain of a function f is denoted by $dom(f)$. For two functions f and g with $dom(f) \cap dom(g) = \emptyset$ we define $f \cup g$ by $(f \cup g)(x) = f(x)$ if $x \in dom(f)$ and $(f \cup g)(x) = g(x)$ if $x \in dom(g)$. For a set $S \subseteq dom(f)$ the function $f \restriction S$ is the restriction of f to S. Every $f : A \to B$ is implicitly defined on sets $f : 2^A \to 2^B$ and on sequences $f : A^\star \to B^\star$ by point-wise application. By $f[a \mapsto b]$ we denote the function update which is defined as usual. The identity function on a set S is $\mathbf{id}_S$.

Let Σ be a finite ranked alphabet where $rk : \Sigma \to \mathbb{N}$ assigns to each symbol $a \in \Sigma$ its rank $rk(a)$. We partition Σ into a set of *nonterminals* $N_\Sigma \subseteq \Sigma$ and a set of *terminals* $T_\Sigma = \Sigma \setminus N_\Sigma$. We will use capital letters for nonterminals and lower case letters for terminal symbols. We assume that both the rk function and the partitioning are implicitly given with Σ. Hyperedge replacement grammars operate on hypergraphs, which allow hyperedges connecting an arbitrary number of vertices.

Definition 2.1 A *(labeled) hypergraph* over Σ is a tuple $H = (V, E, att, \ell, ext)$ where V is a set of vertices and E a set of hyperedges, $att : E \to V^\star$ maps each hyperedge to a sequence of attached vertices, $\ell : E \to \Sigma$ is an hyperedge-labeling function, and $ext \in V^\star$ a sequence of pairwise distinct external vertices (we omit this component if it is the empty word ε).

We require that for all $e \in E$: $|att(e)| = rk(\ell(e))$. The set of all hypergraphs over Σ is denoted by HGraph_Σ.

Thus hyperedges, which are separate objects in the graph, are mapped to sequences of attached vertices. The connections between hyperedges and vertices are called *tentacles*. We will not distinguish between *isomorphic* copies of a hypergraph. Two hypergraphs H_1 and H_2 are isomorphic if they are identical modulo renaming of vertices and hyperedges. A hypergraph H is called *substantial* if $V_H \neq [ext_H]$ or $|E_H| > 1$.

To facilitate notation later on we introduce the notion of a *handle* which is a hypergraph consisting of only one hyperedge attached to its external vertices.

Definition 2.2 Given $X \in \Sigma$ with $rk(X) = n$, an X*-handle* is the hypergraph $X^\bullet = (\{v_1, ..., v_n\}, \{e\}, [e \mapsto v_1...v_n], [e \mapsto X], v_1...v_n) \in \mathrm{HGraph}_\Sigma$.

Definition 2.3 A *hyperedge replacement grammar* (HRG) over Σ is a set G of *(production) rules*, each of the form $X \to H$ with $X \in N_\Sigma$ and $H \in \mathrm{HGraph}_\Sigma$

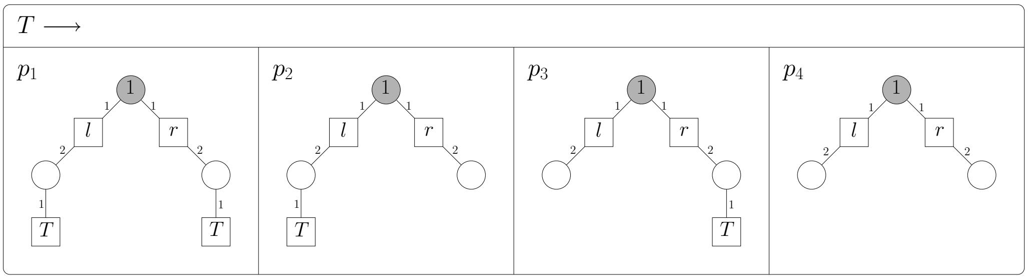


Fig. 1. HRG for Binary Trees

where $|ext_H| = rk(X)$.

We denote the set of hyperedge replacement grammars over Σ by HRG_Σ and assume that there are no isomorphic production rules, i.e., rules with identical left-hand and isomorphic right-hand sides. By $G^X \subseteq G$ we denote the set of X-rules of G (that is, all rules in G with X as left-hand side).

Fig. 1 gives a HRG generating fully branched[4] binary trees. Vertices are depicted as circles, hyperedges as squares, and the numbered lines between squares and circles are the tentacles. The only nonterminal is T (of rank one), and the letters l and r (of rank two) are respectively used to model the left- and right-pointers. The small numbers close to the connecting hyperedges represent the order of the connected vertices and the vertices shaded in gray are the external vertices.

Each HRG rule specifies for some nonterminal X a replacement hypergraph H that will replace (the hyperedge labeled by) X when the rule $X \to H$ is applied. When a hyperedge e labeled by a nonterminal is replaced, the external vertices of the replacement graph are matched with the attached vertices of e. Thus a hyperedge replacement represents a *local* change in the graph structure.

Definition 2.4 Let $G \in \mathrm{HRG}_\Sigma$, $H \in \mathrm{HGraph}_\Sigma$, $p = X \to K \in G$ and $e \in E_H$ with $\ell(e) = X$. Let $E_{H-e} := E_H \setminus \{e\}$. We assume w.l.o.g. that $V_H \cap V_K = E_H \cap E_K = \emptyset$ (otherwise the components in K have to be renamed). The *substitution of e by K*, $J \in \mathrm{HGraph}_\Sigma$, is then defined by

$$\begin{aligned} V_J &= V_H \cup (V_K \setminus [ext_K]) & E_J &= E_{H-e} \cup E_K \\ \ell_J &= (\ell_H \restriction E_{H-e}) \cup \ell_K & ext_J &= ext_H \\ att_J &= mod \circ ((att_H \restriction E_{H-e}) \cup att_K) \end{aligned}$$

$$\text{with } mod = \mathbf{id}_{V_J}[ext_K(1) \mapsto att_H(e)(1), ..., ext_K(rk(e)) \mapsto att_H(e)(rk(e))]$$

We write $H \overset{e/K}{\Longrightarrow} J$ for the above replacement. $H \overset{e,G}{\Longrightarrow} J$ denotes that there is a rule $X \to K \in G$ such that $H \overset{e/K}{\Longrightarrow} J$. $H \overset{G}{\Longrightarrow} J$ is used if there is an $e \in E_H$ and $H \overset{e,G}{\Longrightarrow} J$. The reflexive-transitive closure and the inverse of $\overset{G}{\Longrightarrow}$ are denoted by $\overset{G}{\Longrightarrow}^\star$ and $\overset{G}{\Longrightarrow}^{-1}$, respectively.

[4] A tree node has either both successors or none.

The *language* of a grammar $G \in \mathrm{HRG}_\Sigma$ consists of all terminal graphs (that is, graphs that have only hyperedges with terminal labels) derivable from a given starting graph $H \in \mathrm{HGraph}_\Sigma$, i.e., $L(G, H) = \{K \in \mathrm{HGraph}_{T_\Sigma} \mid H \overset{G}{\Longrightarrow}^{\star} K\}$.

With regard to the applications it is important not to have nonterminals in the grammar from which no terminal graph is derivable ($\forall X \in N_\Sigma : L(G, X^\bullet) \neq \emptyset$). Such grammars are called *productive*. Any HRG can be transformed into an equivalent productive grammar if its language is non-empty.

We are interested in (heap) graph abstractions for analysis and verification, which need to be effectively computable. To ensure termination of the abstraction procedure (which applies grammar rules in backward direction) we have to require that $G \in \mathrm{HRG}_\Sigma$ is *growing*, i.e., for all $X \to H \in G$ the hypergraph H is substantial. This is no restriction since every HRG can be transformed into a growing HRG that generates the same language [10]. For more details please refer to [23]. One easily sees that the HRG in Fig. 1 is growing.

3 Abstraction of Heap States

In this section we give a sketch of the essential results [23] as they are required for understanding the framework. Moreover we present a new application example.

For using HRGs as an abstraction mechanism for pointer-manipulating programs we have to represent heaps as hypergraphs. This is done by introducing two types of terminal hyperedges: hyperedges labeled with program variables (which we include in the terminal alphabet) are of rank one, hyperedges of rank two – labeled with record selectors – are representing pointers in the heap. Formally, we let $T_\Sigma = \mathit{Var}_\Sigma \uplus \mathit{Sel}_\Sigma$ where $rk(\mathit{Var}_\Sigma) = \{1\}$ and $rk(\mathit{Sel}_\Sigma) = \{2\}$. Finally there are nonterminal hyperedges of arbitrary rank that are used in the abstraction and that stand for (a set of) entire subgraphs.

Definition 3.1 A *heap configuration* over Σ is a hypergraph $H \in \mathrm{HGraph}_\Sigma$ such that $\forall x \in \mathit{Var}_\Sigma : |\{e \in E_H \mid \ell_H(e) = x\}| \leq 1$ and $\mathit{ext}_H = \varepsilon$ where Var_Σ and Sel_Σ satisfy the constraints mentioned above. We denote the set of all heap configurations over Σ by $\mathbf{HC}_\Sigma$. A heap configuration H is called *concrete* if $H \in \mathbf{HC}_{T_\Sigma}$.

Since heap objects that are not (indirectly) reachable from program variables do not play any role in program semantics we delete them using a *garbage collector*. When computing the reachability of vertices we handle nonterminal hyperedges conservatively as undirected hyperedges connecting all attached vertices.

We employ a simple pointer programming language to model operations on the heap. It supports *pointer assignment* ($\alpha := \alpha'$, where $\alpha \in \{x, x.s\}$ and $\alpha' \in \{x, x.s, nil\}$ for $x \in \mathit{Var}_\Sigma$ and $s \in \mathit{Sel}_\Sigma$), *dynamic object creation* (new(α)), *conditional jumps* (**if** β **goto** n, where the Boolean expression β allows pointer comparison and boolean connectives), and *unconditional jumps* (**goto** n). Please note that the programming language does not support arbitrary dereferencing depths. This is no restriction since this feature can be emulated by a sequence of assignments. An object deletion command is omitted for simplicity since garbage collection is implicitly assumed. For more details on the programming language and its semantics refer to [23]. An example program can be found in Fig. 5.

When modeling the semantics of assignments we assume that all hyperedges which are connected to vertices referenced by variables, are labeled by terminal symbols. Since the dereferencing depth of pointer expressions is limited to one, this will avoid the necessity for defining the effect of pointer-manipulating operations on abstracted parts of the heap. If there is a nonterminal hyperedge e connected to vertex v that is referenced directly by a variable x we record e together with the index i of the tentacle pointing to v as *violation point* (e, i). Configurations without violation points are called *admissible*. See Fig. 2 for an inadmissible heap configuration.

Definition 3.2 Let $H = (V, E, att, \ell) \in \mathbf{HC}_\Sigma$. The set of *violation points*, $\mathcal{VP}(H) \subseteq E \times \mathbb{N}$, is given by:

$$(e, i) \in \mathcal{VP}(H) \;\Leftrightarrow\; \ell(e) \in N_\Sigma \wedge (\exists e' \in E, v \in V : \ell(e') \in Var_\Sigma \wedge att(e') = att(e)(i))$$

The set of admissible heap configurations is $\mathbf{aHC}_\Sigma = \{H \in \mathbf{HC}_\Sigma \mid \mathcal{VP}(H) = \emptyset\}$.

Based on the concepts presented so far we can formalize the notion of an abstraction function $\mathfrak{A}_G$, called *heap abstractor*. According to the principle that abstraction is performed by backward application of rules, $\mathfrak{A}_G$ returns some irreducible, admissible successor of the current heap configuration with respect to the inverse derivation relation $\overset{G}{\Longrightarrow}^{-1}$.

Definition 3.3 Let $G \in \mathrm{HRG}_\Sigma$. A *heap abstractor* over G is a function $\mathfrak{A}_G$: $\mathbf{aHC}_\Sigma \to \mathbf{aHC}_\Sigma$ such that

$$\mathfrak{A}_G(H) \in \{K \in \mathbf{aHC}_\Sigma \mid K \overset{G}{\Longrightarrow}^{\star} H \text{ s.t. } \nexists J \in \mathbf{aHC}_\Sigma \text{ with } J \overset{G}{\Longrightarrow} K\}.$$

Note that heap abstraction mappings are only uniquely defined if $\overset{G}{\Longrightarrow}^{-1}$ is confluent which, together with its well-foundedness that is implied since the HRG is growing, yields unique normal forms. In general the abstractor should minimize the size of a heap configuration. Also note that this definition immediately implies the *correctness* of our abstraction in the sense that every concrete heap configuration can be re-generated from its abstraction:

Corollary 3.4 *Under the above assumptions, $H \in L(G, \mathfrak{A}_G(H))$ for every $H \in$ $\boldsymbol{aHC}_{T_\Sigma}$.*

Clearly our abstraction function only abstracts heap parts that are consistent with the grammar (e.g., that are binary trees). This, however, does *not* mean that the Juggrnaut framework is unable to handle inconsistencies, i.e., parts in the heap that violate the data structure definitions (for our tree example this could for example mean a back-hyperedge from a leaf to the root). These parts remain concrete while other heap parts are abstracted.

Let us consider the example grammar depicted in Fig. 1 again. Given a binary tree, we are able to abstract it fully (that is, to the nonterminal symbol T) using the given grammar. We are though not yet able to abstract *paths* in a tree. This is important if for instance a variable points to a leaf or there is a back-hyperedge from the leaf which would leave the path (of potentially unbounded length) fully expanded and thus may lead to an infinite state space. To tackle this problem we introduce the additional rules depicted in figure Fig. 3.a.

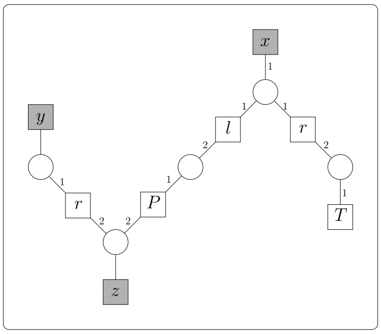


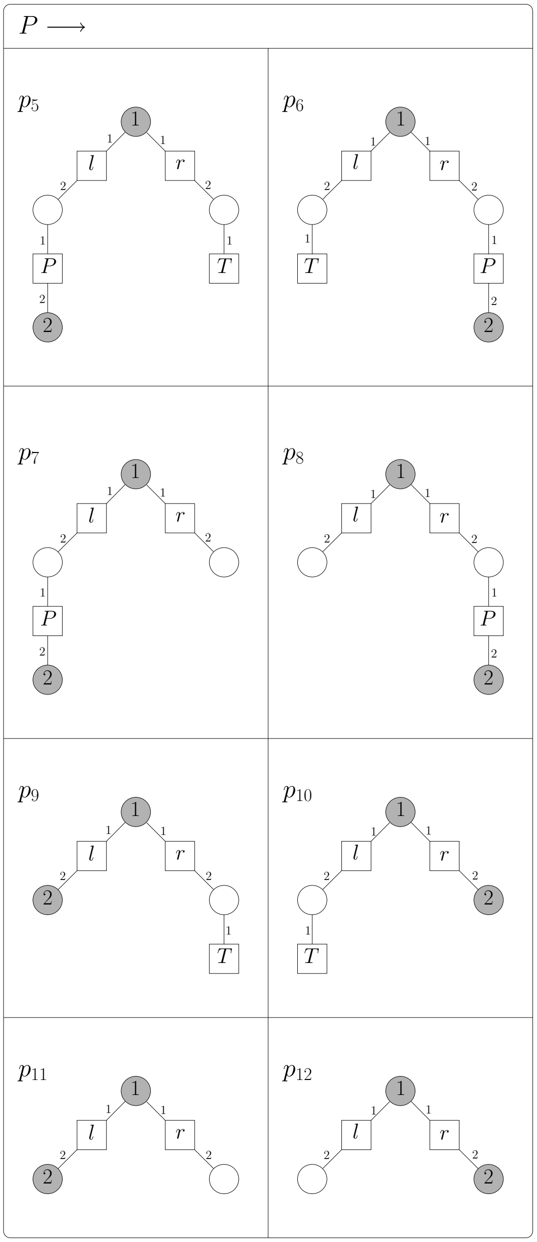
Fig. 2: Inadmissible Configuration

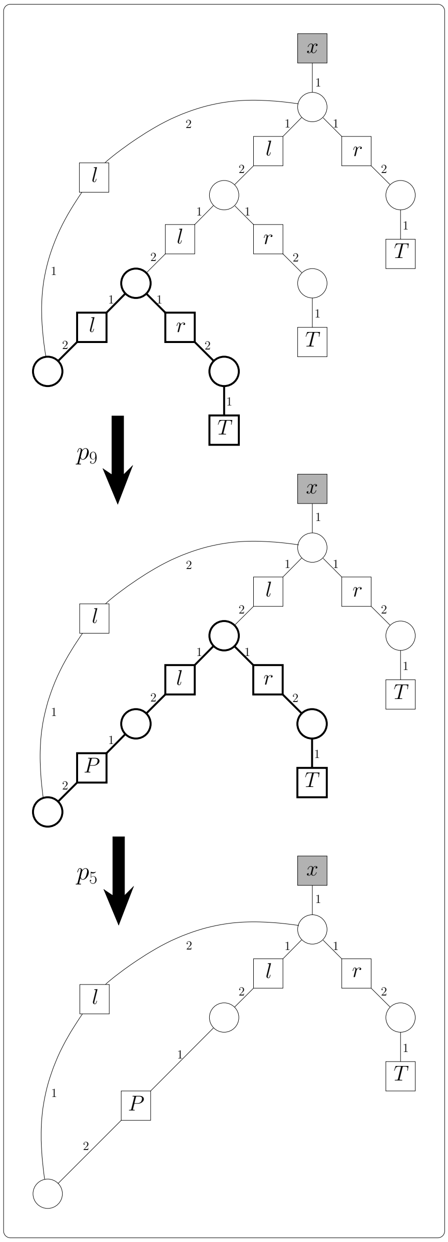
The new nonterminal P of rank two represents tree paths of arbitrary length. In Fig. 3.b an example heap is shown where some parts are already abstract (the T-labeled leaves). We can now apply rule p_9 backwards (the redex is highlighted in the graph) and obtain the graph in the middle. To this graph we can apply rule p_5. Here we could again apply rule p_5 but the resulting configuration would be inadmissible (as a P-hyperedge would be adjacent to the vertex referenced by x). Thus abstraction terminates.

Though an abstractor always yields an admissible configuration, an assignment may lead to an inadmissible heap since variables can navigate through the heap. To restore admissibility we employ *partial* concretization by applying rules in forward direction (hyperedge replacement). Derivation stops as soon as the resulting configuration is admissible, in order to minimize the degree of concretization.

This *partial concretization*, however, raises additional requirements for the production rules. To see this, let us again consider the binary tree example with the path abstraction rules from Fig. 3.a. Now consider the inadmissible graph depicted in Fig. 2 that could easily arise from an assignment $z := y.r$. Here applying any of the rules p_5, p_6, p_7, p_8 would still yield an inadmissible configuration. This can be continued infinitely often and thus does not solve the problem.

To circumvent this problem we considered using *Greibach Normal Form* (GNF) for hyperedge replacement grammars [11] (a generalization of Double Greibach Normal Form of context-free string grammars). For a HRG in GNF a single rule application for each violation point suffices to establish admissibility since external nodes are only adjacent to terminal hyperedges. Although the GNF is effectively computable for a given HRG it unfortunately may become quite large [11]. We decided to introduce a more general concept that uses *redundant* rules instead. For our binary tree grammar these additional rules (not extending the language of the HRG) are shown in Fig. 4. The idea is now to use rules p_{13}, p_{14}, p_{15} and p_{16} instead of rules p_5, p_6, p_7 and p_8 when we need to concretize “bottom-up”, i.e., when the second attachment vertex of a P-hyperedge is referenced by a variable. If this is the case for the first attachment vertex we use the latter rules (if both cases apply we use the rules from both sets in succession).





a) Rules for Path Abstraction b) Path Abstraction Example

Fig. 3. Path Abstraction

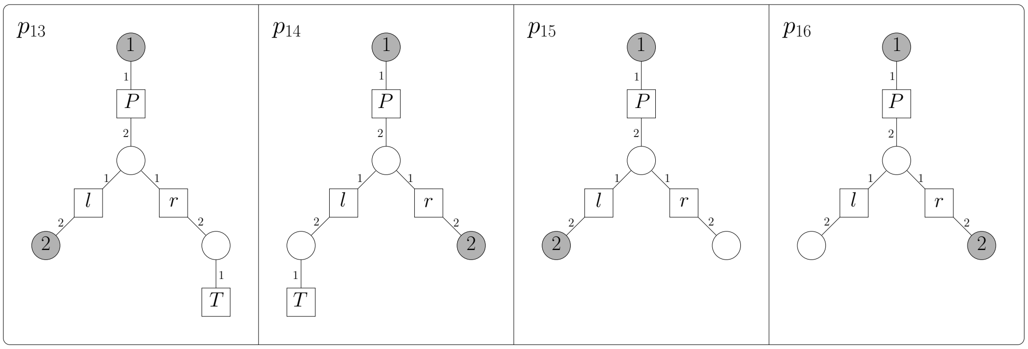


Fig. 4. P-Rules for Concretization "from below"

To this aim, [23] introduces a new class of HRGs – *heap abstraction grammars* – that formally captures this concept. The idea is that, for each nonterminal $X \in N_\Sigma$ and each $i \in \{1, \ldots, rk(X)\}$, there exists a subset G_i^X of rules which can be used for concretizing a nonterminal hyperedge from the ith attached vertex (when this vertex is referenced by a variable) while preserving the graph language $(L(G_i^X \cup \{Y \to H \in G \mid Y \neq X\}, X^\bullet) = L(G, X^\bullet))$.

For our tree example, e.g., we have $G_1^P = \{p_5, ..., p_{12}\}$ and $G_2^P = \{p_9, ..., p_{16}\}$. Note that the rules $p_9, ..., p_{12}$ are included in both subsets. Using this concept, we can now define a mapping that removes all violation points.

Definition 3.5 Let $G \in \mathrm{HRG}_\Sigma$ be a heap abstraction grammar. The *heap concretizer*, $\mathfrak{C}_G : \mathbf{HC}_\Sigma \to 2^{\mathbf{aHC}_\Sigma}$, is then defined as follows:

$$\mathfrak{C}_G(H) = \begin{cases} \mathfrak{C}_G(\{K \in \mathbf{HC}_\Sigma \mid H \overset{e,G_i^X}{\Longrightarrow} K\}) & \text{if } \exists (e,i) \in \mathcal{VP}(H) \wedge \ell_H(e) = X \\ \{H\} & \text{if } H \in \mathbf{aHC}_\Sigma \end{cases}$$

Note that the order of rule applications is not important since HRGs are confluent [10] and that in contrast to a heap abstractor (Def. 3.3), the heap concretizer is uniquely defined as it yields *all* (first) reachable admissible configurations. The cardinality of the resulting set of admissible heap configurations depends on the number of rules in the grammar (more precisely on the cardinality of the appropriate G_i^X).

Combining the concepts introduced before we are now able to abstractly transform the heap based on the program statements. For the more involved commands (assignment, **new**) four steps are necessary: we first *execute the actual command* and apply *garbage collection*. Then *partial concretization* (the only nondeterministic step) restores admissibility, followed by the *re-abstraction*. Again, the details are described in [23].

The correctness proof for our abstraction technique relates concrete and abstract computations in the following way: whenever a concrete heap $H \in \mathbf{aHC}_{T_\Sigma}$ is transformed into $H' \in \mathbf{aHC}_{T_\Sigma}$ and its abstraction $\mathfrak{A}_G(H) \in \mathbf{aHC}_\Sigma$ is (abstractly) transformed into $H'' \in \mathbf{aHC}_\Sigma$, then $H' \in L(G, H'')$. That is, every concrete computation has its abstract counterpart, and thus our abstraction constitutes a safe approximation of the system.

Altogether we are able to generate all reachable abstract states for a pointer program and to use this abstract state space to verify properties that – if fulfilled – also hold in the concrete case due to our over-approximation.

4 An Application of Juggrnaut

The Juggrnaut framework has been implemented in a prototype tool that we are employing to verify properties of pointer-manipulating algorithms. It explores the abstract state space exhaustively and can be used to evaluate LTL-formulae on the generated state space. In this paper we will focus on a variant of the Deutsch-Schorr-Waite (DSW) traversal algorithm, which traverses a binary tree by means of destructive pointer manipulation without using a stack [25]. It has already been verified in [15] and [7]. The authors of [15] prove various properties of the algorithm like structural invariance, correctness, completeness and termination while [7] concentrates on pointer safety and shape invariants. In this section we will demonstrate that the Juggrnaut tool is able to verify these properties based on the binary tree grammar introduced in the previous section.

The DSW algorithm has originally been developed to permit garbage collection without using stacks. It is applied to list structures with sharing and cycles. To ensure termination, visited nodes are marked with various flags. Here we use a variant of the original algorithm as proposed by Lindstrom [14] which operates on acyclic structures, eliminating the need of marking nodes by rotating pointers instead of just inverting them as proposed for the original algorithm. Our version of the DSW tree-traversal algorithm is depicted in Fig. 5. It is essentially the same as the one given in [15] with the difference that it can directly be used by our tool without any modification or instrumentation.

```
 1 if root = nil goto 15;
 2 new(sen);
 3 prev := sen;
 4 cur := root;
 5 next := cur.l;
   // rotate pointers
 6 cur.l := cur.r;
 7 cur.r := prev;
   // move forward
 8 prev := cur;
 9 cur := next;
   // traversal complete ?
10 if (cur = sen) goto 15;
11 if (cur ≠ nil) goto 5;
   // swap prev and cur
12 cur := prev;
13 prev := nil;
14 goto 5;
```

Fig. 5: The DSW Algorithm

Pointer Safety, Shape Invariants, and Structural Invariance

Among the most common errors in software systems are pointer errors arising from dereferencing null pointers. The Juggrnaut tool can find pointer errors on-the-fly while generating the abstract state space. The same holds for shape invariants such as sharing or cycle properties. For the DSW algorithm pointer safety can be checked in less than a second since this is the time needed for state space generation (see Table 1, column "no marking").

Verifying structural invariance can be done in a similar fashion. In Fig. 6.a the initial state is shown. It is already abstract and represents arbitrary (fully branched)

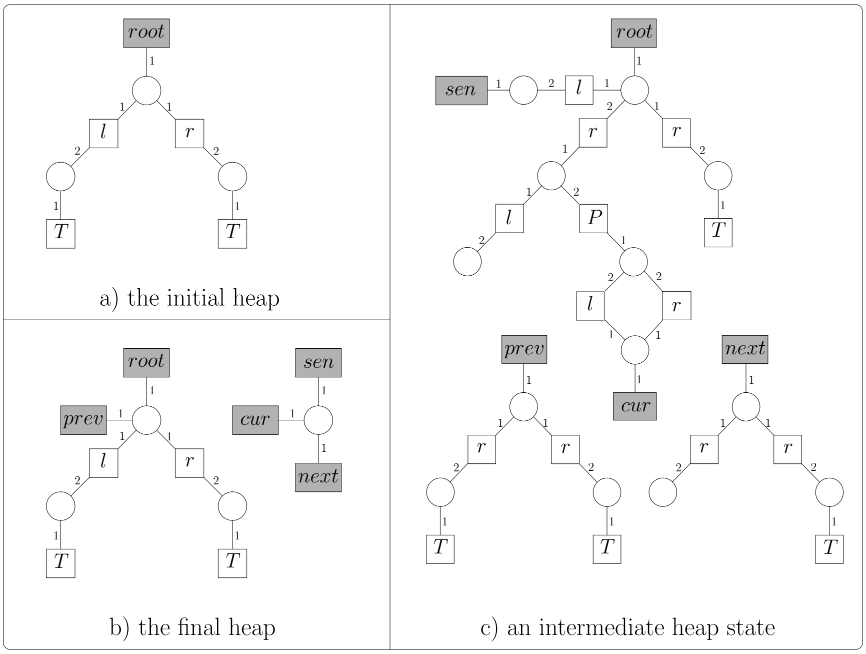


Fig. 6. Abstract Heap Structures Generated by the DSW Algorithm

binary trees where each leaf has a minimal depth of two[5]. When exploring the abstract state space arising from symbolic execution of the DSW algorithm we obtain a single final state (in which the program has terminated) which is depicted in Fig. 6.b. Except for the auxiliary variables and the sentinel vertex that is inserted by the algorithm, the abstract tree referenced by the root variable is the same as in the initial state. From this we can conclude that the DSW algorithm retains the tree structure. We do not yet know, however, whether the result is exactly the same as the input, i.e., whether every *concrete* vertex in *any* possible tree remains at the same position in that tree (correctness).

Note that, as already mentioned before, Juggrnaut can handle programs violating the data structure definition. In Fig. 6 an intermediate state arising during state space exploration is shown where this is the case. In fact the DSW algorithm alters the tree structure while traversing the tree by rotating pointers. Upon termination, however, the result is again a tree. In [15] such inconsistencies had to be resolved by changing the code of the algorithm or alternatively by adapting the abstraction. Such auxiliary measures are not necessary for the Juggrnaut tool.

Completeness and Termination

To handle completeness, i.e., to show that every vertex is visited at least once and to prove termination of the DSW algorithm we have to prove the following LTL-properties (where the quantified variables range over all objects in the heap, and “final” is true for a state if it has no outgoing transitions):

[5] It is the most general initial heap. Verifying the other cases works analogously and is omitted for simplicity here.

	no marking	single marking	ext. marking	TVLA [15]
Initial/Final States	1	89	482	
Number of States	10,254	3,329,400	19,229,960	$> 80,000$
Number of Transitions	11,599	3,787,263	21,857,376	
State Space Generation (min:sec)	<0:01	4:47	34:03	
Memory Consumption	40 MB	386 MB	1,740 MB	150 MB
Pointer Safety/Shape Invariants	**on-the-fly**	on-the-fly	on-the-fly	
Structural Invariance	**on-the-fly**	on-the-fly	on-the-fly	
Completeness (min:sec)	-	**0:10**	0:48	
Termination (min:sec)	-	**0:21**	2:05	
Correctness (min:sec)	-	-	**2:21**	
Total Time (State Space Gen. + all Properties)			**0:39:17**	**<9:00:00**

Table 1
State Space Generation with the Juggrnaut Tool

Completeness: $\forall x : \neg(\text{cur} \neq x \ \ \mathbf{U} \ \ \text{final})$
This formula states that it cannot happen that the variable cur, pointing to the current vertex during tree traversal, is never pointing to x until a final state is reached. In other words, it has to point at least once to x.

Termination: $\forall x : \mathbf{FG}(\text{cur} \neq x)$
Here we state that the vertex referenced by x is visited only finitely often. (From some point onwards it is not referenced by cur anymore.)

The problem of the above formulae is the universal quantification over all heap objects. How can we keep track of their identity when abstracting from them? The idea is to symbolically identify a vertex by introducing a distinguished variable pointing to it. As this cannot be done for all vertices at the same time (otherwise abstraction could not be applied), the choice of this vertex has to be nondeterministic. Thus we generate all possible abstract heaps in which x is pointing to a (different) vertex. If we are able to prove that the above formulae (without quantification) hold for all of those *marked* configurations as initial states, then we reached our goal. In [6] the same idea is applied to verify a list-reversal program. Also [20] uses a similar approach.

Definition 4.1 Let H_0 be the initial heap from Fig. 6.a. The set of *marked initial states* $\text{Init}_G^{H_0}$ is the set of abstract states representing all possible markings of unfoldings (derivable hypergraphs) of the start heap.

$$\text{Init}_G^{H_0} := \mathfrak{A}_G(\text{Marked}_G^{H_0})$$

$$\text{with Marked}_G^{H_0} := \{(V, E \uplus \{x\}, att', \ell[x \mapsto x]) \mid (V, E, att, \ell) \in L(G, H_0), dom(att') = dom(att) \cup \{x\},\ att'(x) \in V\}$$

Formally we take all concrete graphs that are derivable from H_0 ($= L(G, H_0)$; there are generally infinitely many), set the variable x to each possible location in a new copy of each graph, and abstract the whole set. Thus we obtain a set of initial states which is finite in our case since the Juggrnaut tool generates all possible abstract marked initial states by nondeterministically descending into an abstract tree, concretizing where necessary and abstracting where possible. Figure 7 shows one of those 89 (abstract) marked initial states. As one can see, the tree is partially expanded in the proximity of x while the path abstraction collapsed the path from the root. The correctness of the marking approach follows from our abstraction technique (see Def. 3.3).

Corollary 4.2 *The concrete marked states can be regenerated from* $\text{Init}_G^{H_0}$, *i.e.,* $\bigcup\{L(G, H) \mid H \in \text{Init}_G^{H_0}\} = \text{Marked}_G^{H_0}$.

Employing the LTL model checker integrated into our tool we can now verify completeness and termination of the DSW algorithm. Due to the more complex situation the state space is now much larger than before but its generation still takes less than 5 minutes. When the state space exploration is finished, however, we can verify the LTL-formulae above in 10 and 21 seconds, respectively.

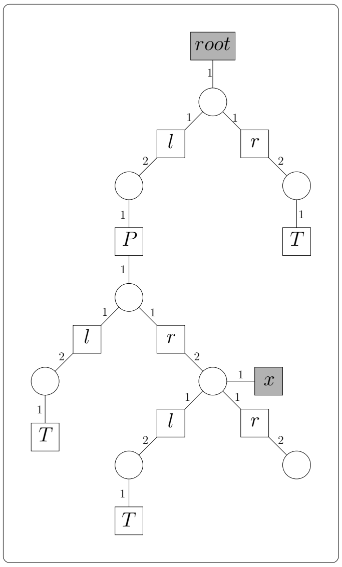


Fig. 7: Singly-Marked Heap

Correctness of the DSW Algorithm

The approach presented in the previous section still does not suffice to show correctness. The correctness property can inductively be formalized by the following LTL-formula:

$$\begin{aligned}\forall x\ \exists x_l\ \exists x_r :\quad & x.l = x_l \ \wedge\ x.r = x_r \\ & \wedge\ (x = \text{root} \rightarrow \mathbf{G}(x = \text{root})) \\ & \wedge\ \mathbf{G}(\text{final} \rightarrow (x.l = x_l \ \wedge\ x.r = x_r))\end{aligned}$$

stating that every concrete heap cell is at the same position in the tree after program termination. The second line represents the induction basis. To verify this formula we need an extended marking that also keeps track of the successors of x by introducing two additional auxiliary variables x_l and x_r. Except for this modification it works the same as the single marking from Def. 4.1.

Additional Experimental Results

Our experimental results are listed in Table 1. The benchmarks were obtained on a 2.33 GHz Intel Xeon machine (64 Bit Linux). The boldface entries represent the shortest possible time (why use an extended marking if the single marking suffices and is much faster?). "On-the-fly" means that there is no separate model checking run necessary; the properties are automatically verified during state space generation. The authors of [15] executed their tool TVLA on a 3 GHz Linux machine. With the extended marking we obtained 482 initial states from which we generated about 19 million abstract states for the DSW algorithm. The state space exploration and the model checking procedure for all properties together took still less than 40 minutes.

The TVLA tool by Loginov, Reps, and Sagiv has also been applied on DSW [15]. According to the authors it ran almost 9 hours on a similar machine for obtaining the same verification results even though it already had been optimized specifically for the DSW algorithm. The Juggrnaut tool however has just been fed with a grammar for binary trees and applies no optimizations for state space reduction yet (such as stuttering equivalence). The authors of [7] have verified pointer safety and probably[6] some shape invariants for the DSW algorithm using abstract regular model checking within 57s (on a 3,2 GHz Xeon Machine). For the same task the Juggrnaut tool needs less than a second.

5 Conclusions and Future Work

We have presented the Juggrnaut framework for the analysis and verification of pointer-manipulating programs operating on dynamic data structures. The abstraction mechanism is parametrized via a hyperedge replacement graph grammar that models the data structure(s) used in the program. We have shown how heap states can abstractly be represented and how abstract state spaces can be generated. Our theoretical results have been implemented in a prototype tool which allows the exploration of abstract state spaces and the evaluation of LTL formulae.

The results obtained by applying Juggrnaut to the Deutsch-Schorr-Waite traversal algorithm are very promising and encourage us to employ our framework to other algorithms and data structures as well. We believe that due to the introduction of a graphical "abstraction modeling language", the Juggrnaut approach is more intuitive than other methods in this domain.

For the future in addition to further optimizations of our tool such as implementing state space reduction techniques we are planning to extend our heap logic to formulate more involved properties and integrate it with the model checker. Furthermore we will analyze how data structure definitions – as they occur in many programming languages – can be used for automatically generating an appropriate abstraction grammar. Finally we will investigate in which ways other graph grammar formalisms are applicable within Juggrnaut.

[6] It is not clear whether this was done for the DSW algorithm; the authors only say that it has been done for *some* case studies.

References

[1] Bakewell, A., D. Plump and C. Runciman, *Specifying pointer structures by graph reduction*, in: *Applications of Graph Transformations with Industrial Relevance '03*, LNCS **3062** (2004), pp. 30–44.

[2] Balaban, I., A. Pnueli and L. D. Zuck, *Shape analysis by predicate abstraction*, in: *VMCAI '05*, LNCS **3385** (2005), pp. 164–180.

[3] Baldan, P., A. Corradini and B. König, *Verifying Finite-State Graph Grammars: An Unfolding-Based Approach*, in: *CONCUR '04*, LNCS **3170** (2004), pp. 83–98.

[4] Baldan, P. and B. König, *Approximating the behaviour of graph transformation systems*, in: *ICGT '02*, LNCS **2505** (2002), pp. 14–29.

[5] Beyer, D., T. A. Henzinger and G. Théoduloz, *Lazy shape analysis*, in: *CAV '06*, LNCS **4144** (2006), pp. 532–546.

[6] Bouajjani, A., P. Habermehl, P. Moro and T. Vojnar, *Verifying programs with dynamic 1-selector-linked list structures in regular model checking*, in: *TACAS '05*, LNCS **3440** (2005), pp. 13–29.

[7] Bouajjani, A., P. Habermehl, A. Rogalewicz and T. Vojnar, *Abstract regular tree model checking of complex dynamic data structures*, in: *SAS '06*, LNCS **4134** (2006), pp. 52–70.

[8] Dams, D. and K. S. Namjoshi, *Shape analysis through predicate abstraction and model checking*, in: *VMCAI '03*, LNCS **2575** (2003), pp. 310–323.

[9] Dodds, M. and D. Plump, *Extending C for checking shape safety*, in: *GT-VC '05*, ENTCS **154(2)** (2006), pp. 95–112.

[10] Drewes, F., H.-J. Kreowski and A. Habel, *Hyperedge replacement graph grammars*, in: G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations*, World Scientific, 1997 pp. 95–162.

[11] Engelfriet, J., *A Greibach Normal Form for Context-Free Graph Grammars*, in: *ICALP '92*, LNCS **623** (1992), pp. 138–149.

[12] Kastenberg, H. and A. Rensink, *Model checking dynamic states in GROOVE*, in: *SPIN '06*, LNCS **3925** (2006), pp. 299–305.

[13] Lee, O., H. Yang and K. Yi, *Automatic verification of pointer programs using grammar-based shape analysis*, in: *ESOP '05*, LNCS **3444** (2005), pp. 124–140.

[14] Lindstrom, G., *Scanning list structures without stacks or tag bits*, Inf. Process. Lett. **2** (1973), pp. 47–51.

[15] Loginov, A., T. W. Reps and M. Sagiv, *Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm*, in: *SAS '06*, LNCS **4134** (2006), pp. 261–279.

[16] Noll, T. and S. Rieger, *Verifying dynamic pointer-manipulating threads*, in: *FM '08*, LNCS **5014** (2008), pp. 84–99.

[17] O'Hearn, P. W., H. Yang and J. C. Reynolds, *Separation and information hiding*, in: *POPL '04* (2004), pp. 268–280.

[18] Podelski, A. and T. Wies, *Boolean heaps*, in: *SAS '05*, LNCS **3672** (2005), pp. 268–283.

[19] Rensink, A., *Canonical graph shapes*, in: *ESOP '04*, LNCS **2986** (2004), pp. 401–415.

[20] Rensink, A., *Model checking quantified computation tree logic*, in: *CONCUR 06*, LNCS **4137** (2006), pp. 110–125.

[21] Rensink, A. and D. Distefano, *Abstract graph transformation*, in: *SVV '05*, ENTCS **157(1)**, 2006.

[22] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *LICS '02* (2002), pp. 55–74.

[23] Rieger, S. and T. Noll, *Abstracting complex data structures by hyperedge replacement*, in: *ICGT '08*, LNCS **5214** (2008), pp. 69–83.

[24] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3–valued logic*, ACM Trans. Program. Lang. Syst. **24** (2002), pp. 217–298.

[25] Schorr, H. and W. M. Waite, *An efficient machine-independent procedure for garbage collection in various list structures*, Commun. ACM **10** (1967), pp. 501–506.