Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm

Alexey Loginov[1], Thomas Reps[1], and Mooly Sagiv[2]

[1] Comp. Sci. Dept., University of Wisconsin; {alexey,reps}@cs.wisc.edu
[2] School of Comp. Sci., Tel-Aviv University; msagiv@post.tau.ac.il

Abstract. This paper reports on the automated verification of the *total correctness* (partial correctness and termination) of the Deutsch-Schorr-Waite (DSW) algorithm. DSW is an algorithm for traversing a binary tree without the use of a stack by means of destructive pointer manipulation. Prior approaches to the verification of the algorithm involved applications of theorem provers or hand-written proofs. TVLA's abstract-interpretation approach made possible the automatic symbolic exploration of all memory configurations that can arise. With the introduction of a few simple core and instrumentation relations, TVLA was able to establish the partial correctness and termination of DSW.

1 Introduction

The Deutsch-Schorr-Waite (DSW) algorithm provides a way to traverse a tree without the use of a stack by temporarily—but systematically—stealing pointer fields of the tree's nodes to serve in place of the stack that one ordinarily needs during, e.g., an in-order traversal.[3] The benefits of being able to perform a tree traversal without the use of a stack are best seen in the context of garbage collection: such an algorithm can be employed during the *mark* phase of garbage collection, when the scarcity of available memory can preclude the use of either an explicit stack for traversing a tree, or a recursive tree traversal (which would use an implicit stack of activation records).

The subtlety of the algorithm (and the complexity of analyzing it) is due to the fact that, during the traversal, the algorithm visits each node of the tree three times, and performs a kind of pointer rotation on each node visit [10]. By the time the algorithm finishes, it has restored the original values of each node's left-child and right-child pointers, thus restoring the original tree.

Richard Bornat singles out the algorithm as a key test for formal methods: "The [Deutsch-]Schorr-Waite algorithm is the first mountain that any formalism for pointer analysis should climb." [2] Past approaches have involved hand-written proofs of complicated invariants to verify the partial correctness of the algorithm. Even with some automation, these efforts were usually laborious: a proof performed in 2002 with the help of the Jape proof editor took 152 pages! [1] The key advantage of TVLA's abstract-interpretation approach over proof-theoretic approaches is that a relatively small number of concepts are involved in defining an abstraction of the structures that can arise on

[3] The variant of the algorithm that we analyzed works correctly when applied to a directed acyclic graph (DAG). While our current analysis applies only when the input is a binary tree, §7 discusses how this limitation can be addressed.

any execution, and verification is then carried out automatically by symbolic exploration of all memory configurations that can arise. In particular, we defined the abstraction using a few simple instrumentation relations—eight key formulas—each containing only two atomic subformulas.

The contributions of this work can be summarized as follows:

- We defined an abstraction (in the canonical-abstraction framework used by TVLA) that captures sufficient invariants of DSW to demonstrate partial correctness and termination.
- We used the fact that each tree node passes through four states (induced by the original state and the three visits to each node) to define a *state-dependent* abstraction, which requires fewer structures to represent the memory configurations that can arise in DSW than would be necessary without state dependence.
- We used the abstraction to establish the partial correctness of DSW via automatic symbolic exploration of all memory configurations.
- We used the *state-dependent* abstraction to establish a bound on the number of iterations of the algorithm's loop, thus establishing that DSW terminates.

2 Program Analysis using 3-Valued Logic

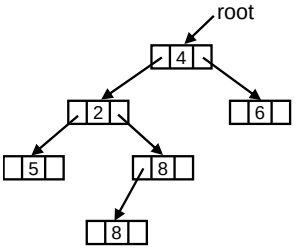

Fig. 1. A possible concrete store for a binary tree.

In this section we give a brief overview of the framework of parametric shape analysis via three-valued logic. For more details, the reader is referred to [17].

Program states are represented using *first-order logical structures*, which consist of a collection of *individuals*, together with an *interpretation* for a finite vocabulary of finite-arity relation symbols, $\mathcal{R}$. An interpretation is a truth-value assignment for each relation symbol for every appropriate-arity tuple of individuals. To ensure termination, the framework puts a bound on the number of distinct logical structures that can arise during analysis by grouping individuals that are indistinguishable according to a special subset of unary relations, $\mathcal{A}$. The grouping of nodes is referred to as *canonical abstraction* and the set $\mathcal{A}$ is referred to as the set of *abstraction relations*.

The application of canonical abstraction typically transforms a logical structure S into a *3-valued logical structure* $S^{\#}$, in which the third value, $\mathbf{1/2}$, denotes the possibility of having either $\mathbf{0}$ (false) or $\mathbf{1}$ (true) in S. A program state is updated and queried via logical formulas, which are interpreted over the three-valued structure $S^{\#}$ using a straightforward extension of Kleene's 2-valued semantics.

Because of canonical abstraction, individuals in a 3-valued structure can represent more than one individual in a given 2-valued structure; such individuals are referred to as *summary individuals*. In general, a 3-valued logical structure can represent an infinite set of 2-valued structures.

Program states are encoded in terms of *core relations*, $\mathcal{C} \subseteq \mathcal{R}$. Core relations are part of the underlying semantics of the language to be analyzed; they record atomic properties of stores. For instance, Tab. 1 gives the definition of a C binary-tree datatype,

```
typedef struct node {
  struct node *left;
  int data;
  struct node *right;
} *Tree;
```

(a)

Relation	Intended Meaning
$x(v)$	Does pointer variable `x` point to heap cell v?
left(v_1, v_2)	Does the `left` field of v_1 point to v_2? (Is v_2 the left child of v_1?)
right(v_1, v_2)	Does the `right` field of v_1 point to v_2? (Is v_2 the right child of v_1?)

(b)

Table 1. (a) Declaration of a binary-tree datatype in C. (b) Core relations used for representing the stores manipulated by programs that use type `Tree`.

and lists the core relations that would be used to represent the stores manipulated by programs that use type `Tree`, such as the store in Fig. 1. Unary relations represent pointer variables, and binary relations *left* and *right* represent the `left` and `right` fields of a `Tree` node. Fig. 2(a) shows 2-valued structure S_2, which represents the store of Fig. 1 using the relations of Tab. 1.

p	Intended Meaning	Defining Formula
down(v_1, v_2)	Do the `left` or `right` fields of v_1 point to v_2? (Is v_2 a child of v_1?)	*left*$(v_1, v_2) \vee$ *right*(v_1, v_2)
$t_{down}(v_1, v_2)$	Is v_2 reachable from v_1 along `left` and `right` fields?	*down*$^*(v_1, v_2)$
$r_x(v)$	Is v reachable from pointer variable `x` along `left` and `right` fields?	$\exists\, v_1 : x(v_1) \wedge t_{down}(v_1, v)$

Table 2. Defining formulas of instrumentation relations commonly employed in analyses of programs that use type `Tree`. There is a separate relation r_x for every program variable `x`.

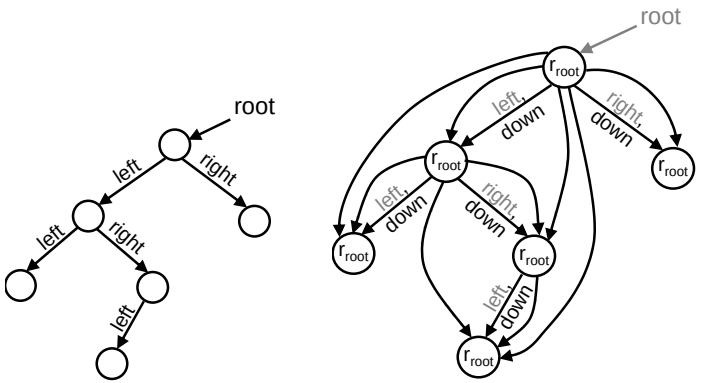

Fig. 2. A logical structure S_2 that represents the store shown in Fig. 1 in graphical form: (a) S_2 with relations of Tab. 1. (b) S_2 with relations of Tabs. 1 and 2 (relations of Tab. 1 appear in grey). Unlabeled (curved) arcs between nodes represent the t_{down} relation. Self-loops of the t_{down} relation (corresponding to the reflexive tuples) have been omitted to reduce clutter.

The abstraction function on which an analysis is based, and hence the precision of the analysis defined, can be tuned by (i) choosing to equip structures with additional *instrumentation relations* to record derived properties, and (ii) varying which of the unary core and unary instrumentation relations are used as the set of abstraction relations. The set of instrumentation relations is denoted by $\mathcal{I}$. Each arity-k relation symbol is defined by an *instrumentation-relation defining formula* with k free variables. Instrumentation relation symbols may appear in the defining formulas of other instrumentation relations as long as there are no circular dependences.

Tab. 2 lists some instrumentation relations that are important for the analysis of programs that use type `Tree`. Instrumentation relations that involve reachability properties, such as relation $r_x(v)$, often play a crucial role in the definitions of abstractions. These relations have the effect of keeping disjoint subtrees summarized separately. Fig. 2(b) shows 2-valued structure S_2, which represents the store of Fig. 1 using the core relations of Tab. 1, as well as the instrumentation relations of Tab. 2.

If all unary relations are abstraction relations, the canonical abstraction of 2-valued logical structure S_2 is S_3, shown in Fig. 3, with all tree nodes not pointed to by `root` represented by the summary individual at the bottom. In S_2, nodes in the left subtree of `root`'s target are indistinguishable from those in its right subtree according to $\mathcal{A}$ (consisting of relations $x(v)$ and $r_x(v)$ for each program variable `x`). S_3 represents all trees with two or more elements, with the root node pointed to by program variable `root`.

The following graphical notation is used for depicting 3-valued logical structures:

- Individuals are represented by circles containing (non-0) values for unary relations. Summary individuals are represented by double circles.
- A unary relation p corresponding to a pointer-valued program variable is represented by a solid arrow from p to the individual u for which $p(u) = 1$, and by the absence of a p-arrow to each node u' for which $p(u') = 0$. (If $p = 0$ for all individuals, the relation name p is not shown.)
- A binary relation q is represented by a solid arrow labeled q between each pair of individuals u_i and u_j for which $q(u_i, u_j) = 1$, and by the absence of a q-arrow between pairs u'_i and u'_j for which $q(u'_i, u'_j) = 0$.
- Relations with value $1/2$ are represented by dotted arrows.

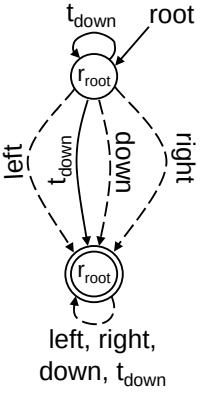

Fig. 3. A 3-valued structure S_3 that is the canonical abstraction of structure S_2. In addition to S_2, S_3 represents any tree of size 2 or more that is pointed to by program variable `root`.

For each kind of statement in the programming language, the concrete semantics is defined by *relation-update formulas* for core relations. The structure transformers for the abstract semantics are defined by the same relation-update formulas for core relations and *relation-maintenance formulas* for instrumentation relations. The latter are generated automatically via *finite differencing* [15]. Abstract interpretation collects a set of 3-valued structures at each program point. It is implemented as an iterative procedure that finds the least fixed point of a certain set of equations [17]. When the fixed point is

reached, the structures that have been collected at a program point describe a superset of all the execution states that can arise there.

Not all logical structures represent admissible stores. To exclude structures that do not, we impose integrity constraints. For instance, relation $x(v)$ of Tab. 1 captures whether pointer variable `x` points to memory cell v; x would be given the attribute "unique", which imposes the integrity constraint that x can hold for at most one individual in any structure: $\forall v_1, v_2 \colon x(v_1) \wedge x(v_2) \Rightarrow v_1 = v_2$. This formula evaluates to $\mathbf{1}$ in any 2-valued logical structure that corresponds to an admissible store. Integrity constraints contribute to the concretization function (γ) for our abstraction [23]. Integrity constraints are enforced by *coerce*, a clean-up operation that may "sharpen" a 3-valued logical structure by setting an indefinite value ($\mathbf{1/2}$) to a definite value ($\mathbf{0}$ or $\mathbf{1}$), or discard a structure entirely if an integrity constraint is definitely violated by the structure (e.g., if it cannot represent any admissible store).

2.1 Analyzing Programs that Manipulate (Only) Trees

When analyzing a program in which each data structure at every point is a tree (a property that we will call *treeness*), it is possible to take advantage of this fact to reduce the (abstract) state space that is explored. This is achieved by having the analysis perform a semantic reduction after each step to filter out non-trees that may have crept into the representation. When the analysis relies on the program to maintain treeness, to guarantee that the results are sound, the analysis must check that treeness is preserved at every step. We address the latter obligation first. The techniques described below are applicable whenever one wishes to analyze programs in which all input, output, and intermediate data structures are trees. We call such analyses *tree-specific shape analyses*; our DSW analysis is an example of a particular tree-specific shape analysis. (Other work in which tree-specific shape analyses have been developed include [4, 7, 8].)

Checking that Treeness is Maintained. The analyzer checks that treeness is maintained by asserting certain logical formulas that capture the conditions under which the execution of a program statement could result in a violation of treeness. Before the computation of a transfer function, the logical formulas of corresponding assertions are evaluated. If a formula *possibly fails to hold*, i.e., does not evaluate to $\mathbf{1}$, then an error report is issued and the analysis is terminated.

For purposes of this paper, a binary tree is a structure containing no cycles and no nodes with multiple incoming `left` or `right` pointers. (Our definition disallows the sharing of subtrees, and thus is more restrictive than the traditional definition that merely requires there to be at most one path between any pair of nodes. This is not an inherent limitation of TVLA; if the sharing of subtrees is to be permitted, the restriction on sharing can be relaxed—see footnote 5.)

Given a data structure that satisfies the data-structure invariants for a binary tree, only one type of statement has the potential to transform the data structure into one that violates some of those properties, namely, a statement of the form `x->sel = y` (where `sel` can be `left` or `right`), which creates a new `sel`-connection in the data structure. Two logical formulas capture the conditions that guarantee that the application of the transformer for a statement of the form `x->sel = y` maintains treeness.

The first formula captures the precondition for *down* to remain acyclic:

$$\forall v_1, v_2 \colon x(v_1) \wedge y(v_2) \Rightarrow \neg t_{down}(v_2, v_1) \tag{1}$$

The second formula captures the precondition for the statement to avoid introducing sharing:[4]

$$\forall v_1, v_2 \colon y(v_2) \Rightarrow \neg \mathit{down}(v_1, v_2)^5 \tag{2}$$

Semantic Reduction for Trees. After each application of an abstract transformer, we perform a semantic reduction to filter out non-trees that may have crept into the abstract structures computed by the transformer. The reduction is implemented as an application of *coerce* to enforce integrity constraints that express data-structure invariants.

For instance, relation *down* is given the attributes "acyclic" and "invfunction". The "acyclic" attribute of *down* results in the automatic generation of the following integrity constraint:

$$\forall v_1, v_2 \colon t_{down}(v_1, v_2) \wedge t_{down}(v_2, v_1) \Rightarrow v_1 = v_2 \tag{3}$$

The "invfunction" attribute of *down* results in the automatic generation of the following integrity constraint:

$$\forall v_1, v_2 \colon (\exists v \colon \mathit{down}(v_1, v) \wedge \mathit{down}(v_2, v)) \Rightarrow v_1 = v_2 \tag{4}$$

Operation *coerce* is applied at certain steps of the algorithm, e.g., after the application of an abstract transformer, to enforce Constraints (3) and (4), along with a few others, to help prevent the analysis from admitting non-trees, and thereby possibly losing precision.

3 Deutsch-Schorr-Waite Tree-Traversal Algorithm

The original Deutsch-Schorr-Waite algorithm reverses the direction of `left` and `right` pointers, as it traverses the tree [18]. It attaches two bits, `mark` and `tag`, to each node. The `mark` bit serves to prevent multiple visits to nodes on a cycle or in shared subtrees. The `tag` bit records whether, during the traversal of reversed pointers, a node was reached from its left or right child.

In [10], Lindstrom gave a variant that eliminated the need for both bits, provided the input data structure contains no cycles. His insight was that one could treat the visit step at an internal node as a kind of pointer-rotation operation, and that completion of the tree-traversal could be established having the algorithm watch for a distinguished

[4] As explained in §3, we ensure that `x->sel` is `NULL` prior an assignment of the form `x->sel = y`, so the assignment indeed creates a new `sel`-connection.

[5] If we relaxed the restriction on the sharing of subtrees, then, in place of Formula (2), we would employ a slightly more complex formula that precludes the possibility of creating two paths between a pair of tree nodes v_1 and v_4 (one path that existed prior to the statement, and the other that was created due to the introduction of the new `sel` edge from `x` to `y`):
$\forall v_1, v_2, v_3, v_4 \colon t_{down}(v_1, v_4) \wedge t_{down}(v_1, v_2) \wedge x(v_2) \wedge y(v_3) \Rightarrow \neg t_{down}(v_3, v_4)$

```
[1] void traverse(Tree *root)
[2] { Tree *prev, *cur, *next;

[3]   if (root == NULL)
[4]     return;
[5]   prev = -1;
[6]   cur = root;
[7]   while (1) {
        // Save left subtree
[8]     next = cur->left;




        // Rotate pointers
[9]     cur->left = cur->right;
[10]    cur->right = prev;
        // Move forward
[11]    prev = cur;
[12]    cur = next;
[13]    if (cur == -1)
          // Traversal completed
[14]      break;
[15]    if (cur == NULL) {
          // Swap prev and cur
[16]      cur = prev;
[17]      prev = NULL;
[18]    }
[19]  }
[20]}
```

(a)

```
void traverse(Tree *root)    [1]
{ Tree *prev, *cur,          [2]
       *next, *tmp;          [3]
  if (root == NULL)          [4]
    return;                  [5]
  prev = SENTINEL;           [6]
  cur = root;                [7]
  while (1) {                [8]
    // Save the left subtree
    next = cur->left;        [9]
    // Rotate pointers
    tmp = cur->right;       [10]
    // Maintain treeness
    cur->right = NULL;      [11]
    cur->right = prev;      [12]
    cur->left = NULL;       [13]
    cur->left = tmp;        [14]
    // Move forward
    prev = cur;             [15]
    cur = next;             [16]
    if (cur == SENTINEL)    [17]
      // Traversal completed
      break;                [18]
    if (cur == NULL) {      [19]
      // Swap prev and cur
      cur = prev;           [20]
      prev = NULL;          [21]
    }                       [22]
  }                         [23]
}                           [24]
```

(b)

Fig. 4. (a) Original version of the Deutsch-Schorr-Waite algorithm (adapted from [10]). (b) Modified version of the Deutsch-Schorr-Waite algorithm that was analyzed using TVLA. (The differences appear in bold.)

value that serves as a kind of sentinel. In this paper, we actually consider the Lindstrom variant, but continue to refer to it as Deutsch-Schorr-Waite (DSW). Another connection between our analysis (of the Lindstrom variant) and the original version of DSW is discussed briefly in §7.

Fig. 4 shows two versions of the Deutsch-Schorr-Waite algorithm. The left-hand column shows a version adapted from [10], also known as Lindstrom scanning. The right-hand column shows a slightly modified version of the algorithm that we used in our work. There are two differences between the two versions.

First, the constant `-1` on lines [5] and [13] has been replaced with `SENTINEL`, where `SENTINEL` is assumed to be a reference to a distinguished node that is not part of the input tree. In TVLA, pointer values can either equal `NULL` (corresponding to the situation in which the pointer does not point to any heap object) or point to a heap object

that was allocated by `malloc`. In this sense, TVLA follows the semantics of Java, in which new non-`NULL` pointer values can be generated only via memory-allocation operations.

Second, a purely local transformation (involving the introduction of one temporary variable `tmp`) has been applied to lines [9]–[10]:

```
                                    [10] tmp = cur->right;
                                    // Maintain treeness
[9] cur->left = cur->right;   ⟹    [11] cur->right = NULL;
[10] cur->right = prev;             [12] cur->right = prev;
                                    [13] cur->left = NULL;
                                    [14] cur->left = tmp;
```

This really involved three transformations:

1. Assignment statements of the form `x->sel1 = y->sel2` have been normalized to statement sequences `tmp = y->sel2; x->sel1 = tmp` (see lines [10] and [14] of Fig. 4(b)).
2. Assignment statements of the form `x->sel = y` have been normalized to statement sequences `x->sel = NULL; x->sel = y` (see lines [11]–[12] and [13]–[14] of Fig. 4(b)). This ensures that statements of the form `x->sel = y` can never destroy existing `sel`-paths in the data structure, thus simplifying the task of maintaining information about the reachability of tree nodes from program variables.
3. Assignments `cur->right = NULL` and `cur->right = prev` have been moved to lines [11] and [12] (before assignments to `cur->left`). This change prevents the right child of `cur`'s target from temporarily having two incoming edges after the assignment to `cur->left` on line [14].[6] The resulting algorithm maintains the invariant that the nodes of the input tree always make up one or two data structures that satisfy the binary-tree properties: after the assignment on line [14] of Fig. 4(b), the nodes of the input tree make up two trees, one rooted at `next`'s target, and the other rooted at `cur`'s target; the original root is a descendant of `cur`'s target.

Transformations 1 and 2 above are simple normalizations that one could expect to find in a translation of programs written in a high-level language into a lower-level intermediate representation. Transformation 3 prevents the temporary sharing of `cur`'s right subtree (it would otherwise briefly become `cur`'s left and `cur`'s right subtree). We could relax our restriction on sharing and analyze the version of the algorithm that does not include transformation 3 (§7 discusses how we would approach this task), but we chose to verify total correctness and preservation of treeness for the slightly modified version of the DSW algorithm shown in Fig. 4(b). Because of transformation 3, the techniques of §2.1 apply in the analysis of this version; we now describe this version in detail.

For each tree node *n*, the body of the `while` loop is executed three times with `cur` pointing to *n*. Each time that *n* is considered, its `left` and `right` pointers are rotated in a counter-clockwise fashion on lines [10]–[14] of Fig. 4(b) (cf. lines [9] and [10] of

[6] Only the assignment `cur->right = NULL` needs to be moved to achieve the desired effect. We moved both assignments for clarity.

Fig. 4(a)). After the third such execution, the original values for the `left` and `right` pointers are re-established, as we explain below.

Before the first execution of lines [10]–[14] of Fig. 4(b) with `cur` pointing to *n*, no nodes in the subtrees rooted at *l* or *r* (*n*'s left and right subtrees in the original tree) have been visited, and no `left` or `right` pointers of nodes in the subtrees rooted at *l* or *r* have been modified. In this situation, we say that *n* is in *state 0*. Fig. 5(a) illustrates this situation.

A pointer to node *l*, the left child of *n* prior to the rotation of *n*'s `left` and `right` pointers, is saved in `next` on line [9]. After the rotation, the traversal continues by moving into the (sub)tree rooted at `next`, i.e., *l* (see lines [15] and [16]). When `cur` becomes null, the values of `cur` and `prev` are swapped on lines [20] and [21]. This causes the traversal to backtrack to the most recently visited node that had a right subtree in the original tree.

When the traversal backtracks to *n*, the algorithm reaches lines [10]–[14] of Fig. 4(b) for the second time with `cur` pointing to *n*. At this point, all nodes in *l*'s subtree and no nodes in *r*'s subtree have been visited. The `left` and `right` pointers of nodes in *l*'s subtree have been rotated three times and restored to their original values. No `left` or `right` pointers of nodes in *r*'s subtree have been modified. In this situation we say that *n* is in *state 1*. Fig. 5(b) illustrates this situation.

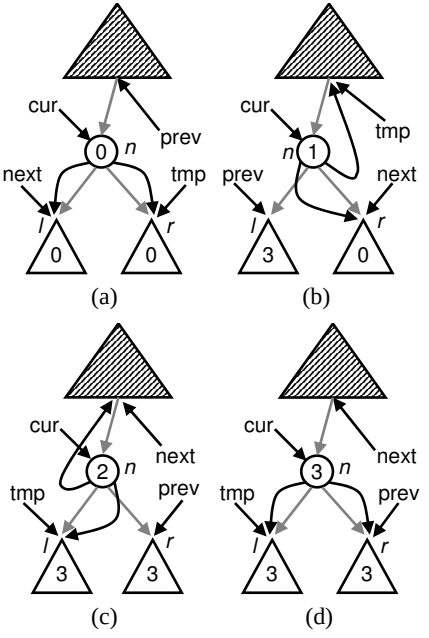

Fig. 5. States of the subtree of *n* with `cur` pointing to *n*: (a) after the first execution of statement on line [10] of Fig. 4(b), *n* is in state 0; (b) after the second execution of statement on line [10] of Fig. 4(b), *n* is in state 1; (c) after the third execution of statement on line [10] of Fig. 4(b), *n* is in state 2; (d) after the third execution of statement on **line [14]** of Fig. 4(b), *n* is in state 3. Grey edges represent the original values of the `left` and `right` fields.

A pointer to node *r*, the left child of *n* prior to the second rotation of *n*'s pointers, is saved in `next`. After the rotation, the traversal continues by moving into the (sub)tree rooted at *r* (see lines [15] and [16]). Once again, the algorithm backtracks when `cur` is null. When the traversal backtracks to *n*, the algorithm reaches lines [10]–[14] of Fig. 4(b) for the third (and final) time with `cur` pointing to *n*. At this point, all nodes in *l*'s and *r*'s subtrees have been visited. The `left` and `right` pointers of nodes in both subtrees have been rotated three times and restored to their original values. In this situation we say that *n* is in *state 2*. Fig. 5(c) illustrates this situation.

After the subsequent execution of lines [10]–[14] of Fig. 4(b) with `cur` pointing to *n*, *n*'s `left` and `right` pointers are restored to their original values. At this point, all nodes in the subtree rooted at *n* have been visited, and all `left` and `right` pointers

in the subtree have been rotated three times and restored to their original values. In this situation we say that *n* is in *state 3*. Fig. 5(d) illustrates this situation.

The algorithm traverses the tree *in order*, visiting each node *n* three times: (1) while following the original `left` pointers from *n*'s parent through *n* into *l*'s subtree, (2) while backtracking from *l*'s subtree to *n* and then traversing *r*'s subtree, and (3) while backtracking from *r*'s subtree through *n* to *n*'s parent in the original tree.

Fig. 6 depicts the states of the tree nodes that are not in the subtree pointed to by `cur`. All ancestors (in the original tree) of `cur`'s target are in state 1 or 2, indicating that the left (1) or right (2), subtree is currently being traversed. If `cur`'s target lies in the left subtree of an ancestor, then that ancestor must be in state 1, otherwise it must be in state 2. The triangular shapes at left represent all nodes that occur earlier than `cur`'s target in an in-order traversal of the tree. For each of these nodes there exists an ancestor of `cur`'s target, such that the node is in the left subtree of the ancestor, and `cur`'s target is in the right subtree of the ancestor. All nodes in that category are in state 3; they have been visited three times, and their `left` and `right` pointers have been reset to their original values. The triangular shapes at right represent all nodes that occur later than `cur`'s target in an in-order traversal of the tree. For each of these nodes there exists an ancestor of `cur`'s target, such that the node is in the right subtree of the ancestor, and `cur`'s target is in the left subtree of the ancestor. All nodes in that category are in state 0; they have not been visited, and their `left` and `right` pointers still have their original values.

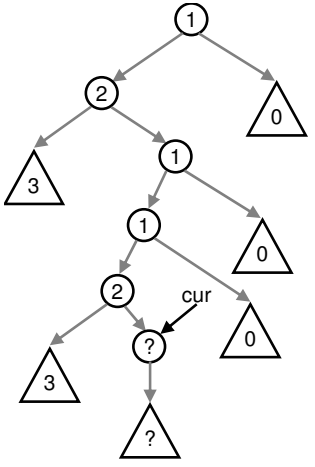

Fig. 6. States of tree nodes that are outside of the subtree pointed to by `cur`. (Grey edges represent the original values of the `left` and `right` fields.)

4 A Shape Abstraction for Verifying DSW

Consider the problem of establishing that the Deutsch-Schorr-Waite algorithm shown in Fig. 4(b) is partially correct. This is an assertion that compares the state of a store at the end of the procedure with its state at the start.

Partial correctness of DSW means (i) the tree produced at exit must be identical to the input tree, and (ii) every node must be visited. We will come back to property (ii) when we discuss the total correctness of DSW in §5. Property (i) can be specified as follows:

$$\forall v_1, v_2 \colon \mathit{left}(v_1, v_2) \Leftrightarrow \mathit{left}^0(v_1, v_2) \tag{5}$$

$$\forall v_1, v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_1, v_2), \tag{6}$$

where left^0 and right^0 denote the initial values of relations *left* and *right*, respectively. Additionally, a correct traversal routine must neither lose nodes of the input tree, nor gain new ones. However, this property is implied by properties (5) and (6).

The challenge is that the abstraction has to track the "unintended" use of pointers for stack simulation with sufficient precision to verify that at the end of the algorithm their correct usage has been reestablished. Canonical abstraction with just the properties listed in Tabs. 1 and 2 is an insufficiently precise abstraction to demonstrate that the tree's edges are restored.

The key relations for establishing properties (5) and (6) at the end of the program are those that capture the relationships of pointers that arise between tree nodes during the traversal. The following set of unary relations capture properties of nodes in state 0 (before any changes to the nodes' `left` and `right` pointers) or state 3 (after the nodes' `left` and `right` pointer values have been restored):

$$eq_{l,l^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{left}(v_1, v_2) \Leftrightarrow \mathit{left}^0(v_1, v_2) \tag{7}$$

$$eq_{r,r^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_1, v_2) \tag{8}$$

Unary relations $eq_{l,l^0}(v_1)$ and $eq_{r,r^0}(v_1)$ distinguish individuals that represent tree nodes whose `left`, respectively `right`, pointers have their initial values. We can now use $\forall v \colon eq_{l,l^0}(v)$ in place of Formula (5) and $\forall v \colon eq_{r,r^0}(v)$ in place of Formula (6) when asserting the partial correctness of DSW.

The following set of unary relations capture properties of nodes in state 1, after one visit to those nodes, i.e., one rotation of the `left` and `right` pointers:

$$eq_{l,r^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{left}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_1, v_2) \tag{9}$$

$$re_{r,l^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{left}^0(v_2, v_1) \tag{10}$$

$$re_{r,r^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_2, v_1) \tag{11}$$

Unary relation $eq_{l,r^0}(v_1)$ distinguishes individuals that represent tree nodes whose `left` field points to their right (in the input tree) subtree. Unary relations $re_{r,l^0}(v_1)$ and $re_{r,r^0}(v_1)$ (*re* is a mnemonic for *reverse*) distinguish individuals that represent tree nodes n whose `right` fields point to their parents in the input tree (assuming that n is the left child in the case of $re_{r,l^0}(v_1)$ and right child, otherwise).

The following set of unary relations capture properties of nodes in state 2, after two visits to those nodes, i.e., two rotations of the `left` and `right` pointers:

$$eq_{r,l^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{right}(v_1, v_2) \Leftrightarrow \mathit{left}^0(v_1, v_2) \tag{12}$$

$$re_{l,l^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{left}(v_1, v_2) \Leftrightarrow \mathit{left}^0(v_2, v_1) \tag{13}$$

$$re_{l,r^0}(v_1) \stackrel{\text{def}}{=} \forall v_2 \colon \mathit{left}(v_1, v_2) \Leftrightarrow \mathit{right}^0(v_2, v_1) \tag{14}$$

Unary relation $eq_{r,l^0}(v_1)$ distinguishes individuals that represent tree nodes whose `right` field points to their left (in the input tree) subtree. Unary relations $re_{l,l^0}(v_1)$ and $re_{l,r^0}(v_1)$ distinguish individuals that represent tree nodes n whose `left` fields point to their parents in the input tree (assuming that n is the left child in the case of $re_{l,l^0}(v_1)$ and right child, otherwise).

Let us give the intuition behind the use of the relations defined by Formulas (7)–(14) for the partial-correctness verification of DSW, which involves establishing that all `left` and `right` pointers have their initial values at the end of DSW.

These relations maintain the relationship between the current and the original values of `left` and `right` pointers. Prior to the first rotation of pointers for node *n*, *n* has entries **1** for the state-0 relations (Formulas (7) and (8)), which say that there has been no change from *n*'s starting pointer values. These entries allow the analysis to conclude that after the current iteration's rotation of *n*'s pointers, *n* should have entry **1** for state-1 relations, Formula (9) and Formulas (10) or (11). Similarly, the **1** entries for the state-1 relations for node *n* help establish the **1** entries for its state-2 relations (Formula (12) and Formulas (13) or (14)) after the second rotation of *n*'s pointers. Finally, the **1** entries for the state-2 relations for node *n* help establish the **1** entries for its state-3 relations Formulas (7) and (8) after the third rotation of *n*'s pointers.

In our initial attempt to establish the partial correctness of DSW, we added all relations of Formulas (7)–(14) to the set of abstraction relations, $\mathcal{A}$. This attempt failed (we terminated the analysis after several days of computation) because of the vast abstract state space that needed to be explored. To pare down the abstract state space, we observed that not all node distinctions introduced by the relations of Formulas (7)–(14) were necessary. For instance, note that any leaf node in state 0 or state 3 satisfies (among other relations) Formula (9), which defines eq_{l,r^0}—nominally a state-1 relation—because it has no outgoing `left` or `right` pointers, while an internal tree node in state 0 or state 3 does not satisfy it. As a result, eq_{l,r^0} prevents canonical abstraction from summarizing a leaf node in state 0 or 3 with an internal node in one of those states. The resulting abstraction has a larger-than-necessary state space because we only need to ensure that tree nodes in state 1 have their `left` field pointing to their original right subtree, i.e., have the property defined by the relation eq_{l,r^0}.

To remove such unnecessary distinctions, we introduce the concept of a *state-dependent* abstraction. The first component of such an abstraction is a collection of unary core *state relations*, $state_0(v)$, $state_1(v)$, $state_2(v)$, and $state_3(v)$.[7] Every time the rotation of `left` and `right` pointers of the tree node pointed to by `cur` is completed (after line [14] of Fig. 4(b)), the node's state is changed to the next state. (The state relations carry no semantics with respect to the pointer values of nodes; they simply record the "visit counts" for each node.) As the second component of the abstraction, we introduce state-relation-guarded versions of the relations of Formulas (7)–(14):

$$s_0\_eq_{l,l^0}(v_1) \stackrel{\text{def}}{=} state_0(v_1) \wedge eq_{l,l^0}(v_1) \tag{15}$$

$$s_0\_eq_{r,r^0}(v_1) \stackrel{\text{def}}{=} state_0(v_1) \wedge eq_{r,r^0}(v_1) \tag{16}$$

$$s_1\_eq_{l,r^0}(v_1) \stackrel{\text{def}}{=} state_1(v_1) \wedge eq_{l,r^0}(v_1) \tag{17}$$

$$s_1\_re_{r,l^0}(v_1) \stackrel{\text{def}}{=} state_1(v_1) \wedge re_{r,l^0}(v_1) \tag{18}$$

$$s_1\_re_{r,r^0}(v_1) \stackrel{\text{def}}{=} state_1(v_1) \wedge re_{r,r^0}(v_1) \tag{19}$$

$$s_2\_eq_{r,l^0}(v_1) \stackrel{\text{def}}{=} state_2(v_1) \wedge eq_{r,l^0}(v_1) \tag{20}$$

$$s_2\_re_{l,l^0}(v_1) \stackrel{\text{def}}{=} state_2(v_1) \wedge re_{l,l^0}(v_1) \tag{21}$$

$$s_2\_re_{l,r^0}(v_1) \stackrel{\text{def}}{=} state_2(v_1) \wedge re_{l,r^0}(v_1) \tag{22}$$

[7] The state relations are *not* added to the set of abstraction relations, $\mathcal{A}$.

$$s_3\_eq_{l,l^0}(v_1) \stackrel{\text{def}}{=} state_3(v_1) \wedge eq_{l,l^0}(v_1) \tag{23}$$

$$s_3\_eq_{r,r^0}(v_1) \stackrel{\text{def}}{=} state_3(v_1) \wedge eq_{r,r^0}(v_1) \tag{24}$$

We replace the relations of Formulas (7)–(14) in the set of abstraction relations, $\mathcal{A}$, with Formulas (15)–(24). The resulting abstraction allows the grouping of nodes that have different values for the relation eq_{l,r^0}, for example, as long as these nodes are not in state 1.

5 Establishing that DSW Terminates

We can establish that DSW terminates using the unary state relations of §4 via a simple progress monitor, which we describe below.

For each state relation s, we create a copy of s, which is used to save the values of relation s at the start of the currently-processed loop iteration (after line [8] of Fig. 4(b)). We give the new relations the superscript *lh* to indicate that they hold the *loop-head* values. The first abstract operation of each iteration of the loop takes a snapshot of the current states of nodes: $state_i^{lh}(v) \leftarrow state_i(v)$, for each $i \in [0..3]$ and each binding of v to individuals in the abstract structure being processed. Additionally, it asserts that `cur` does not point to a tree node in state 3 at the head of the loop.

The last operation of every loop iteration performs a progress test by asserting the following formula:

$$\begin{array}{rl} \exists\, v\colon & \left(state_0^{lh}(v) \wedge state_1(v) \vee state_1^{lh}(v) \wedge state_2(v) \vee state_2^{lh}(v) \wedge state_3(v)\right) \\ \bigwedge \forall\, v_1 \neq v\colon & (state_0^{lh}(v_1) \Leftrightarrow state_0(v_1)) \wedge (state_1^{lh}(v_1) \Leftrightarrow state_1(v_1)) \wedge \\ & (state_2^{lh}(v_1) \Leftrightarrow state_2(v_1)) \wedge (state_3^{lh}(v_1) \Leftrightarrow state_3(v_1)) \end{array}$$

The assertion ensures that one node's state makes forward progress (the first line of the assertion) and that no other node changes state (the second and third lines of the assertion).

Together with the assertion that `cur` does not point to a tree node in state 3 at the start of the loop, the above progress monitor establishes that each tree node is visited exactly three times, thus establishing that the algorithm terminates, as well as the fact that every node is, in fact, visited by the algorithm (property (ii) of partial correctness).

6 Experimental Evaluation

We applied TVLA to the DSW algorithm shown in Fig. 4(b) and analyzed it using the abstraction defined in §4. As input for the algorithm, we supplied the 3-valued structure S_7 shown in Fig. 7, which is essentially the structure S_3 from Fig. 3 refined with values for relations introduced in §4. Additionally, S_7 contains a special *sentinel* node that is not part of the input tree; it is referenced by program variable `SENTINEL`. In Fig. 7, as well as Fig. 8, relations $left^0$ and $right^0$ are omitted to reduce clutter. Their values are identical to *left* and *right*, respectively. We have also omitted the values for state-1 and state-2 relations eq_{l,r^0}, re_{r,l^0}, re_{r,r^0}, eq_{r,l^0}, re_{l,l^0}, and re_{l,r^0}. They have value $\mathbf{1/2}$

for the non-sentinel nodes of both figures and value **1** for the sentinel nodes. Because we are performing tree-specific shape analysis, both figures only represent concrete structures that satisfy the treeness integrity constraints (see §2.1).

Fig. 8 shows the unique structure S_8 collected by the analysis at the exit node. The definite **1** values for relations eq_{l,l^0} and eq_{r,r^0} (defined by Formulas (7) and (8)) for each individual of S_8 establish that the outgoing `left` and `right` pointers of every tree node are restored, thus establishing partial correctness property (i), i.e., that the tree produced at exit is identical to the input tree. The absence of violations of the progress monitor defined in §5 establishes that DSW terminates, as well as the fact that every node is visited (partial correctness property (ii)).

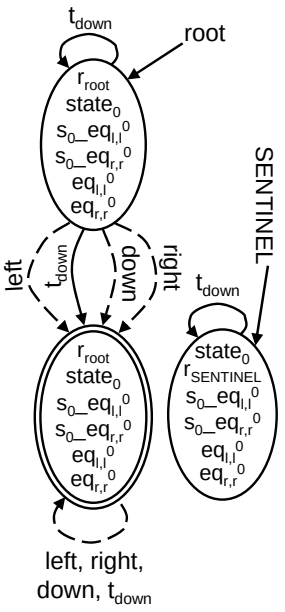

Fig. 7. A 3-valued structure S_7 that represents all trees of size 2 or more.

The analysis took just under nine hours on a 3GHz Linux PC and used 150MB of memory. While the authors have a number of ideas for performance optimizations for the research system, the main goal was to demonstrate the feasibility of automatic symbolic exploration of heap-manipulating programs with vast (abstract) state spaces.

The cost of verifying that DSW terminates is negligible (when compared to the cost that DSW is partially correct) because the progress monitor does not increase the size of the reachable state space. The number of distinct abstract structures that were collected at all program points exceeded 80,000. The number of structures at some program points exceeded 11,000. This number is not surprising, if we consider that some of these structures contained 15 individuals. (At intermediate steps, the analysis explored abstracts structures with up to 21 individuals!) However, 80,000 is well below the limit imposed by the number of distinct 3-valued structures, $2^{2^{20}}$, which represents the number of subsets of individuals with every possible vector of unary abstraction-relation values. (There are 20 unary abstraction relations: pointer relations $x(v)$ and reachability relations $r_x(v)$ for each of the five pointer-valued program variables, as well as ten relations of Formulas (15)–(24).) Fig. 9 shows a sample abstract structure S_9 that arises before line [11] of Fig. 4(b). In S_9, as in all other structures that arise at that point, the state relations and state-relation-guarded relations defined by Formulas (15)–(24), have precise values for all individuals.

In summary, our experiment showed that, using the abstraction defined in §4, an automatic analysis can maintain enough precision to identify sufficient invariants to demonstrate both partial correctness and termination of DSW.

7 Discussion and Future Work

The analysis carried out by TVLA performs fully-automatic state-space exploration. However, one has to bring to bear some expertise in specifying TVLA analyses. The

concept of tree-specific shape analysis (see §2.1) is of general utility. It can be reused for any analysis in which all input, output, and intermediate data structures are trees. The instrumentation relations defined by Formulas (9)–(14), which capture pointer relationships of tree nodes, and core state relations $\mathit{state}_0(v), \ldots, \mathit{state}_3(v)$, which are used to control the precision of the abstraction, are specific to the problem of verifying the total correctness of DSW.

A key difference between our approach and theorem-prover-based approaches is that we do not need to specify loop invariants. Instead, we need to specify a collection of node distinctions (or node relationships), such as the relations $\mathit{eq}_{l,r^0}(v_1)$ and $\mathit{re}_{r,l^0}(v_1)$ of Formulas (9)–(14); these allow the node distinctions specified to be observable by the analysis. Given the appropriate node distinctions, abstract interpretation automatically infers the invariants satisfied by the program.

Recently, a machine-learning technique has been used to identify key instrumentation relations automatically [11]. In the future, we would like to see if it can be used to identify the key relations for verifying DSW, namely the relations of Formulas (9)–(14).

Although the instrumentation relations introduced in §4 are tailored for establishing the correctness of DSW, the concept of state-dependent abstractions is likely to be of general utility. In fact, simpler versions of state-dependent abstractions have arisen in past work. For example, the unary relation *inOrder* was used to establish the partial correctness of sorting [9]. The state-dependent abstractions defined in this paper are prepared to deal with more than just two states (initial and final, as is the case for the relation *inOrder*), and use the value of the state as a guard to reduce the number of distinct properties recorded for individuals, thereby reducing the size of the (abstract) state space that is explored.

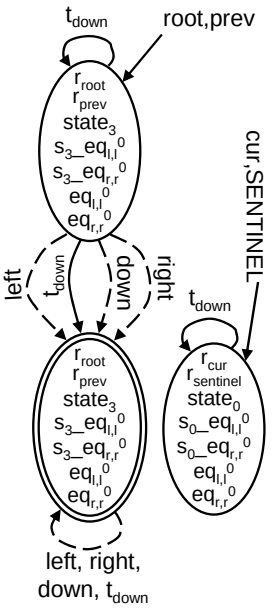

Fig. 8. A 3-valued structure S_8 collected at exit of DSW.

There is an interesting analogy between the explicit state-tracking that the original DSW algorithm performs via the `mark` and `tag` bits, and the state relations of our abstraction. (In some sense, the state relations introduced for purposes of analysis impose a DSW-like view of the world to track the actions of the Lindstrom variant of the algorithm.)

While we chose to apply a transformation that ensures that the algorithm maintains treeness (transformation 3 of §3), it is possible to verify the unmodified algorithm (Fig. 4(a)) by introducing the following instrumentation relation:

$$\mathit{isLocallyShared}(v) \stackrel{\text{def}}{=} \exists\, v_1 : \mathit{left}(v_1, v) \wedge \mathit{right}(v_1, v)$$

Relation *isLocallyShared* (which has value 0 for all nodes in the input 3-valued structure, indicating that the input is a valid binary tree) allows us to relax the restriction on sharing by tracking where sharing occurs rather than requiring its absence. To be

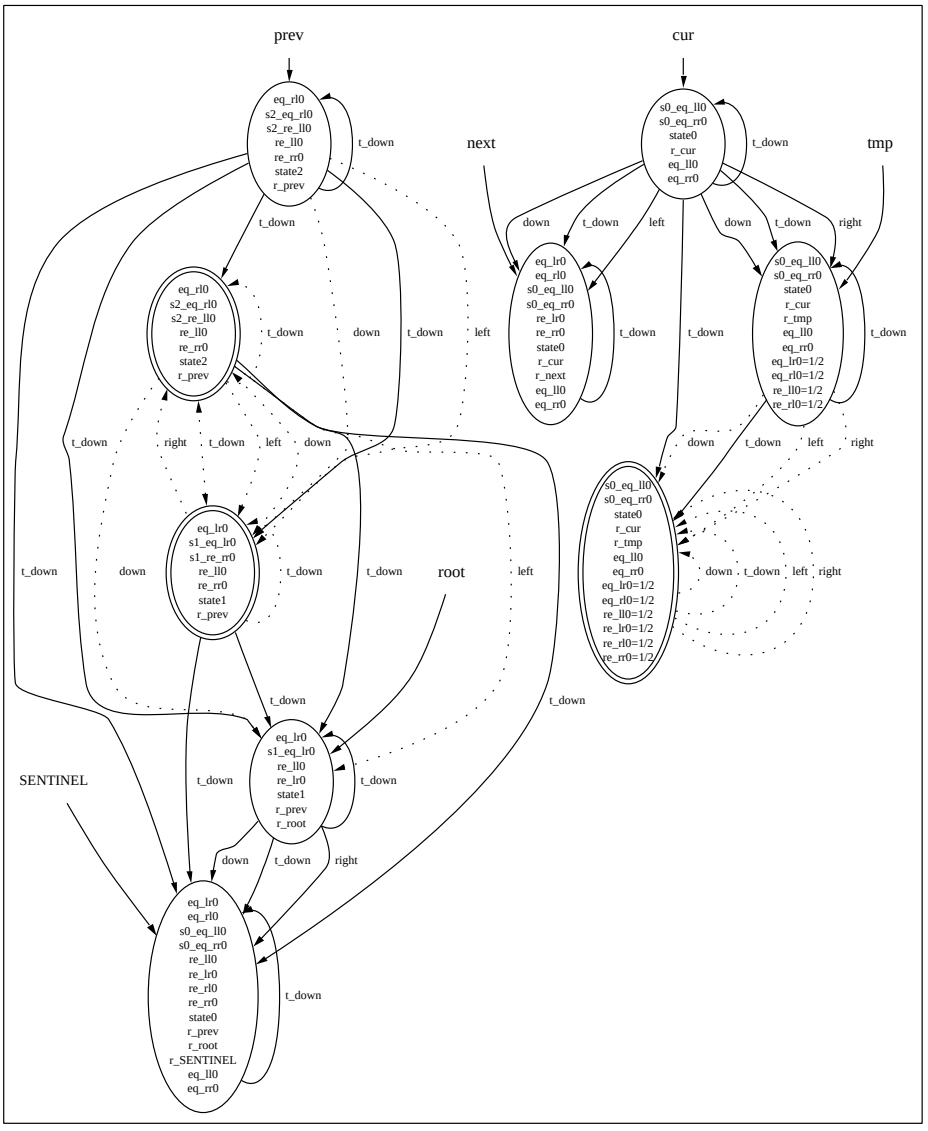

Fig. 9. A 3-valued structure S_9 that arises prior to the first rotation of pointers of the node *n* pointed to by `cur` (before line [11] of Fig. 4(b)). Relations $left^0$ and $right^0$ are omitted from the figure. Initially, node *n* was the right child of the node pointed to by `prev`. The latter node is now the root of a tree with leaf `SENTINEL` (the original root is the parent of `SENTINEL`). No nodes in *n*'s subtree have been visited; that subtree has not been modified from its initial state.

applicable to the version of the algorithm that does not include transformation 3, the tree-specific shape analysis of §2.1 can be generalized to handle the limited class of DAGs that arise in lines [9]–[10] of Fig. 4(a) as follows:

1. The precondition for the absence of sharing (Formula (2)) would be removed.
2. The integrity constraints that forbid structures that contain sharing would be modified to include an *isLocallyShared* guard to permit the kind of local sharing that arises in Fig. 4(a). E.g., Constraint (4) becomes:

$$\forall\, v_1, v_2 \colon (\exists\, v \colon \neg \mathit{isLocallyShared}(v) \wedge \mathit{down}(v_1, v) \wedge \mathit{down}(v_2, v)) \Rightarrow v_1 = v_2.$$

The DSW algorithm shown in Fig. 4(b) (as well as the algorithm shown in Fig. 4(a)) does not work correctly when applied to a data structure that contains a cycle: the traversal terminates prematurely and not all of the edges are properly restored. However, the algorithm works correctly when applied to a DAG: a node *n* with k paths from the root to *n* is visited $3k$ times, rather than 3 times. (Note, however, that k can be exponential in the size of the graph.) Given a bound on k, we may be able to verify the correctness of DSW for DAGs, if we relax the restriction on sharing and introduce $3k$ state relations and the corresponding state-relation-guarded relations. However, unless k is very small it is not likely that the reachable state space can be explored with our computing resources. In the general case, in which the input is a DAG with no bound on k, the partial-correctness result can be obtained by having the state relations of nodes wrap around: a visit to a node in state 3 results in changing the node's state to 1. While this change would be sufficient to establish that the outgoing `left` and `right` pointers of every DAG node are restored and that every node is visited, the analysis would no longer be able to establish termination using the simple progress monitor of §5.

In practice, one would rarely be interested in using such an algorithm to traverse a DAG because of the potentially exponential cost. In most applications, one is likely to want to process each node once (e.g., in depth-first order) and visit each node a constant number of times. This can be achieved by equipping the nodes with two bits to record the visit count (a number from 0 to 3). All nodes reachable from a node with visit count 3 must have been visited three times. If `cur` is set to point to a node with visit count 3, the direction of the traversal can be reversed by swapping the values of `cur` and `prev`, thus terminating the exploration of the node's subgraph. By relaxing the restriction on sharing, it should be possible to verify the total correctness of the modified algorithm.

8 Related Work[8]

The general form of the Deutsch-Schorr-Waite algorithm works correctly for arbitrary graphs [18]. (Unlike the algorithm we used in our work, which was taken from [10], the general form is not constant-space because it uses mark and tag bits.) We divide the discussion of related work according to the kind of data structures to which the analyzed algorithm can be applied.

DSW on Arbitrary Graphs. The first formal proofs of the partial correctness of DSW were performed manually by Morris [14] and Topor [20]. In [19], Suzuki automated some steps of the partial-correctness verification of the algorithm by introducing decision procedures that could handle heap-manipulating programs. More recently, Bornat

[8] The discussion of [14, 20, 19] relies on what is reported in [22, 13].

used the Jape proof editor [3] to construct a partial-correctness proof of DSW [2]. The resulting proof used 152 pages [1].

Our automated approach provides the obvious benefit of disposing with the need to provide manual proofs, which require significant investments of time and expertise. However, even in the presence of a powerful theorem prover, proof-based approaches rely on the user to provide loop invariants that are sufficient to establish the property being verified. For instance, the properties of nodes and their subtrees that are described in §3 (see Figs. 5 and 6 and the corresponding text) would have to be specified as loop invariants. As discussed in §7, our obligation is simpler: we have to specify instrumentation relations that act as *ingredients* for a loop invariant; the analysis automatically synthesizes a loop invariant—in the form of a collection of 3-valued structures that overapproximate the set of concrete structures that actually arise—by means of state-space exploration.

Yang [21] and Mehta and Nipkow [13] gave manually-constructed, but machine-checkable, proofs of the partial correctness of DSW. The two approaches share the goal of making formal reasoning about heap-manipulating programs more natural. The former approach uses the logic of Bunched Implications [5] (a precursor formalism to Separation Logic [16]), which permits the user to reason with Hoare triples in the presence of complicated aliasing relationships. The latter approach uses Isabelle/HOL to construct formal proofs that are human-readable. These approaches improve the usability of proof-based techniques. However, they still lack the automation of our approach.

DSW on Trees and DAGs. Yelowitz and Duncan were the first to present a termination argument for the Deutsch-Schorr-Waite algorithm [22]. They analyzed Knuth's version of the algorithm [6], which uses tag bits but does not work correctly for graphs that contain a cycle. It does, however, work for DAGs, as does the version we used, taken from [10]. The termination argument involved the use of program invariants to prove bounds on the number of executions of statements in the loop. In §5, we showed how to use the *state relations* defined in §4 in a simple progress monitor for the algorithm's loop to establish that DSW terminates (on trees). As was the case for partial correctness, our task is reduced to establishing appropriate distinctions between nodes. Given the state relations, the complete state-space exploration shows no violation of the progress monitor and establishes a bound (namely, three) on the number of visits to each tree node; consequently, the algorithm must terminate.

Several previous papers reported on automatic verification of weaker properties of the Deutsch-Schorr-Waite algorithm, namely that the algorithm has no unsafe pointer operations or memory leaks, and that the data structure produced at the end is, in fact, a binary tree [15, 12, 7]. The authors first established these properties in [15]. ([12] contains a typo stating that that work establishes partial correctness; however, [12] reused the TVLA specification from [15], and establishes the same properties as [15].) Finally, [7] extended the framework of [17] with grammars, which provide convenient syntactic sugar for expressing shape properties of data structures. That work relied on the use of grammars, instead of instrumentation relations, to express tree properties and the absence of memory leaks.

References

1. R. Bornat. Proofs of pointer programs in Jape. "Available at http://www.dcs.qmul.ac.uk/ ~richard/pointers/".
2. R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, July 2000.
3. R. Bornat and B. Sufrin. Animating formal proofs at the surface: The Jape proof calculator. *The Computer Journal*, 43:177–192, 1999.
4. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Dept. of Computer Science, Cornell University, January 1990.
5. S. Ishtiaq and P. O'Hearn. Bi as an assertion language for mutable data structures. In *Symp. on Principles of Programming Languages*, pages 14–26, January 2001.
6. D. Knuth. *The Art of Computer Programming – Vol. 1, Fundamental Algorithms*. Addison-Wesley, 1973.
7. O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *European Symp. On Programming*, pages 124–140, April 2005.
8. T. Lev-Ami, N. Immerman, and M. Sagiv. Fast and precise abstraction for shape analysis. To appear in *Proc. Computer-Aided Verification*, August 2006.
9. T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Int. Symp. on Software Testing and Analysis*, pages 26–38, August 2000.
10. G. Lindstrom. Scanning list structures without stacks or tag bits. *Information Processing Letters*, 2(2):47–51, June 1973.
11. A. Loginov, T. Reps, and M. Sagiv. Abstraction refinement via inductive learning. In *Proc. Computer-Aided Verification*, pages 519–533, July 2005.
12. R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *Static Analysis Symp.*, pages 265–279, August 2004.
13. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *Automated Deduction — CADE-19*, pages 121–135, July 2003.
14. J. Morris. Verification-oriented language design. Tech. Report TR-7, Computer Science Div., University of California–Berkeley, December 1972.
15. T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas with applications to program analysis. In *European Symp. On Programming*, pages 380–398, April 2003.
16. J. Reynolds. Separation Logic: A logic for shared mutable data structures. In *Symp. on Logic in Computer Science*, pages 55–74, July 2002.
17. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
18. H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.
19. N. Suzuki. *Automatic Verification of Programs with Complex Data Structures*. PhD thesis, Dept. of Computer Science, Stanford University, February 1976.
20. R. Topor. The correctness of the Schorr-Waite list marking algorithm. Tech. Report MIP-R-104, School of Artificial Intelligence, University of Edinburgh, July 1974.
21. H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, June 2001.
22. L. Yelowitz and A. Duncan. Abstractions, instantiations, and proofs of marking algorithms. In *Symp. on Artificial Intelligence and Programming Languages*, pages 13–21, August 1977.
23. G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. To appear in *ACM Transactions on Computational Logic (TOCL)*.