Property-Directed Shape Analysis

S. Itzhaky[1], N. Bjørner[2], T. Reps[3,4], M. Sagiv[1], and A. Thakur[3]

[1] Tel Aviv University, Tel Aviv, Israel
[2] Microsoft Research, USA
[3] University of Wisconsin–Madison, USA
[4] GrammaTech, Inc., USA

Abstract. This paper addresses the problem of automatically generating quantified invariants for programs that manipulate singly and doubly linked-list data structures. Our algorithm is *property-directed*—i.e., its choices are driven by the properties to be proven. The algorithm is able to establish that a correct program has no memory-safety violations—e.g., null-pointer dereferences, double frees—and that data-structure invariants are preserved. For programs with errors, the algorithm produces concrete counterexamples.

More broadly, the paper describes how to integrate IC3 with full predicate abstraction. The analysis method is complete in the following sense: if an inductive invariant that proves that the program satisfies a given property is expressible as a Boolean combination of a given set of predicates, then the analysis will find such an invariant. To the best of our knowledge, this method represents the first shape-analysis algorithm that is capable of (i) reporting concrete counterexamples, or alternatively (ii) establishing that the predicates in use are not capable of proving the property in question.

1 Introduction

The goal of our work is to automatically generate quantified invariants for programs that manipulate singly-linked and doubly-linked list data structures. For a correct program, the invariant generated ensures that the program has no memory-safety violations, such as null-pointer dereferences, and that data-structure invariants are preserved. For a program in which it is possible to have a memory-safety violation or for a data-structure invariant to be violated, the algorithm produces a concrete counterexample. Although in this paper we mainly discuss memory-safety properties and data-structure invariants, the technique can be easily extended to other correctness properties (see §5).

To the best of our knowledge, our method represents the first shape-analysis algorithm that is capable of (i) reporting concrete counterexamples, or alternatively (ii) establishing that the abstraction in use is not capable of proving the property in question. This result is achieved by combining several existing ideas in a new way:

- The algorithm uses a predicate-abstraction domain [12] in which quantified predicates express properties of singly and doubly linked lists. In contrast to most recent work, which uses restricted forms of predicate abstraction—such as Cartesian abstraction [1]—our algorithm uses full predicate abstraction (i.e., the abstraction uses arbitrary Boolean combinations of predicates).

– The abstraction predicates and language semantics are expressed in recently developed *reachability logics*, AF^R and EA^R, respectively, which are decidable using a reduction to SAT [17].
– The algorithm is property-directed—i.e., its choices are driven by the memory-safety properties to be proven. In particular, the algorithm is based on IC3 [3], which we here refer to as *property-directed reachability* (PDR).

PDR integrates well with full predicate abstraction: in effect, the analysis obtains the same precision as the best abstract transformer for full predicate abstraction, without ever constructing the transformers explicitly. In particular, we cast PDR as a *framework* that is parameterized on

– the logic $\mathcal{L}$ in which the semantics of program statements are expressed, and
– the finite set of predicates that define the abstract domain $\mathcal{A}$ in which invariants can be expressed. An element of $\mathcal{A}$ is an arbitrary Boolean combination of the predicates.

Furthermore, our PDR framework is *relatively complete with respect to the given abstraction*. That is, the analysis is guaranteed to terminate and either (i) verifies the given property, (ii) generates a concrete counterexample to the given property, or (iii) reports that the abstract domain is not expressive enough to establish the proof. Outcome (ii) is possible because the "frame" structure maintained during PDR can be used to build a trace formula; if the formula is satisfiable, the model can be presented to the user as a concrete counterexample. Moreover, if the analysis fails to prove the property or find a concrete counterexample (outcome (iii)), then there is no way to express an inductive invariant that establishes the property in question using a Boolean combination of the abstraction predicates. Note that outcome (iii) is a much stronger guarantee than what other approaches provide in such cases when they neither succeed nor give a concrete counterexample.

Key to instantiating the PDR framework for shape analysis was a recent development of the AF^R and EA^R logics for expressing properties of linked lists [17]. AF^R is used to define abstraction predicates, and EA^R is used to express the language semantics. AF^R is a decidable, alternation-free fragment of first-order logic with transitive closure (FO^{TC}). When applied to list-manipulation programs, atomic formulas of AF^R can denote reachability relations between memory locations pointed to by pointer variables, where reachability corresponds to repeated dereferences of *next* or *prev* fields. One advantage of AF^R is that it does not require any special-purpose reasoning machinery: an AF^R formula can be converted to a formula in "effectively propositional" logic, which can be reduced to SAT solving. That is, in contrast to much previous work on shape analysis, our method makes use of *a general purpose SMT solver*, Z3 [5] (rather than specialized tools developed for reasoning about linked data structures, e.g., [25, 6, 2, 11]).

The main restriction in AF^R is that it allows the use of a relation symbol f^* that denotes the transitive closure of a function symbol f, but only limited use of f itself. Although this restriction can be somewhat awkward, it is mainly a concern for the analysis designer (and the details have already been worked out in [17]). As a language

Name	Description	Mnemonic
$x = y$	equality	
$x\langle f\rangle y$	$x\text{->}f = y$	
$x\langle f^*\rangle y$	an f path from x to y	
$f.ls\ [x, y]$	unshared f linked-list segment between x and y	
$alloc(x)$	x points to an allocated element	St
$f.stable(h)$	any f-path from h leads to an allocated element	St
$f/b.rev\ [x, y]$	reversed f/b linked-list segment between x and y	R
$f.sorted\ [x, y]$	sorted f list segment between x and y	S

Table 1. Predicates for expressing various properties of linked lists whose elements hold data values. x and y denote program variables that point to list elements or `null`. f and b are parameters that denote pointer fields. (The mnemonics are referred to in Table 6.)

for expressing invariants, AF^R provides a fairly natural abstraction, which means that analysis *results* should be understandable by non-experts (see §2).[5]

Our work represents the first algorithm for shape analysis that either (i) succeeds, (ii) returns a concrete counterexample, or (iii) returns an abstract trace showing that the abstraction in use is not capable of proving the property in question. The specific contributions of our work include

- A framework, based on the PDR algorithm, for finding an inductive invariant in a certain logic fragment (abstract domain) that allows one to prove that a given pre-/post-condition holds or find a concrete counter-example to the property, or, in the case of a negative result, the information that there is no inductive invariant expressible in the abstract domain (§3).
- An instantiation of the framework for finding invariants of programs that manipulate singly-linked or doubly-linked lists. This instantiation uses AF^R to define a simple predicate-abstraction domain, and is the first application of PDR to establish quantified invariants of programs that manipulate linked lists (§4).
- An empirical evaluation showing the efficacy of the PDR framework for a set of linked-list programs (§5).

2 A Motivating Example

To illustrate the analysis, we use the procedure `insert`, shown in Fig. 1, that inserts a new element pointed to by `e` into the non-empty, singly-linked list pointed to by `h`. `insert` is annotated with a pre-condition and a post-condition.

Table 1 shows a set of predicates for expressing properties of linked lists whose elements hold data values. The predicates above the horizontal line in Table 1 are inspired by earlier work on shape analysis [13] and separation logic [24].

Given an input procedure, optionally annotated with a pre-condition *Pre* and post-condition *Post* (expressed as formulas over the same vocabulary of predicates); the goal of the analysis is to compute an invariant for the head of each loop [6] expressed as a CNF formula over the predicates given in Table 1 (and their negations).

[5] By a "non-expert", we mean someone who has no knowledge of either the analysis algorithm, or the abstraction techniques used inside the algorithm.

[6] The current implementation supports procedures with only a single loop; however, this restriction is not an essential limitation of our technique.

```
void insert(List e, List h, List x) {
  Requires: h ≠ null ∧ h⟨n+⟩x ∧ x⟨n*⟩null ∧ e ≠ null ∧ e⟨n⟩null ∧ ¬h⟨n*⟩e
  Ensures: h ≠ null ∧ h⟨n*⟩e ∧ e⟨n⟩x ∧ x⟨n*⟩null
  p = h;
  q = null;
  while (p != x && p != null) {
    q = p;
    p = p->n;
  }
  q->n = e;
  e->n = p;
}
```

Fig. 1. A procedure to insert the element pointed to by `e` into the non-empty, singly-linked list pointed by `h`.

The task is not trivial because (i) a loop invariant may be more complex than a program's pre-condition or post-condition, and (ii) it is infeasible to enumerate all the potential invariants expressible as CNF formulas over the predicates shown in Table 1. For instance, there are 6 variables in `insert` (including `null`), and hence $2^{6\times6\times6}$ clauses can be created from the 36 possible instantiations of each of the 6 binary predicates in Table 1. Therefore, the number of candidate invariants that can be formulated with these predicates is more than $2^{2^{6\times6\times6}}$. It would be infeasible to investigate them all explicitly.

Our analysis algorithm is based on *property-directed reachability* [3]. It starts with the trivial invariant **true**, which is repeatedly refined until it becomes inductive.[7] On each iteration, a concrete counterexample to inductiveness is used to refine the invariant by excluding predicates that are implied by that counterexample.

When applied to the procedure in Fig. 1, our analysis algorithm terminated in about 24 seconds, and inferred the following 13-clause loop invariant:

$$\begin{array}{llll} q \neq e & \wedge (h\langle n^*\rangle x \wedge p = x \rightarrow h\langle n^*\rangle q) & \wedge (p = x \rightarrow q\langle n\rangle p) \\ \wedge\ (\neg e\langle n\rangle e) & \wedge (q\langle n^*\rangle p \rightarrow q\langle n\rangle p) & \wedge (h\langle n^*\rangle p \vee p = \text{null}) \\ \wedge\ (e\langle n\rangle \text{null}) & \wedge (x = \text{null} \vee p\langle n^*\rangle x) & \wedge (q \neq x \vee p \neq \text{null}) \\ \wedge\ \neg h\langle n^*\rangle e & \wedge (p = \text{null} \rightarrow h\langle n^*\rangle q) & \wedge (p = q \vee q\langle n\rangle p) \\ & \wedge (h\langle n^*\rangle q \wedge h\langle n^*\rangle x \rightarrow h\langle n\rangle q \vee q\langle n^*\rangle x) & \end{array} \quad (1)$$

This loop invariant also guarantees that the code is memory safe. It is also possible to apply the analysis to infer sufficient conditions for memory safety using true post-conditions.

Our analysis is also capable of finding concrete counterexamples when the procedure violates the specification. For example, when the conjunct "$x \neq h$" is added to the precondition in Fig. 1 and "$e \neq$ `null`" is removed, the algorithm returns the counterexample trace shown in Fig. 2. Not surprisingly, e is `null` in the first state at the loop head (Fig. 1(a)). The loop body executes once, at which point we reach the loop

[7] An invariant I is inductive at the entry to a loop if whenever the code of the loop body is executed on an arbitrary state that satisfies both I and the loop condition, the result is a state that satisfies I.

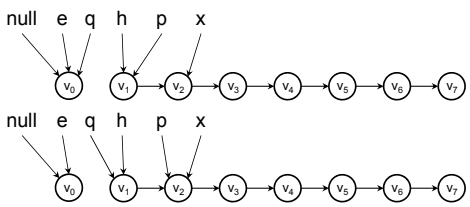

Fig. 2. A two-state counterexample trace obtained from the algorithm when it is applied to a version of Fig. 1 in which the conjunct $x \neq h$ was added to the precondition and $e \neq \texttt{null}$ was removed. (a) First state at the loop head; (b) second state at the loop head, at which point the loop exits, and a null-dereference violation subsequently occurs.

Algorithm 1: PDR$_{\mathcal{A}}(\mathit{Init}, \rho, \mathit{Bad})$

```
1  R[−1] := false
2  R[0] := true
3  N := 0
4  while true do
5      if there exists 0 ≤ i < N
             such that R[i] = R[i + 1]
       then
6          return valid
7      (r, A) := Check_A(Bad, R[N])
8      if r = unsat then
9          N := N + 1
10         R[N] := true
11     else
12         reduce_A(N, A)
```

Algorithm 2: reduce$_{\mathcal{A}}(j, A)$

```
1  (r, A1) := Check_A(Init, A)
2  if r = sat then
3      σ := Model(Init ∧ ρ^{N−j} ∧ (Bad)'^{×(N−j)})
4      if σ is None then error "abstraction failure"
5      else error "concrete counterexample(σ)"
6  while true do
7      (r, A2) :=
8          Check_A((Init)' ∨ (R[j − 1] ∧ ρ), (A)')
9      if r = unsat then break
10     else reduce_A(j − 1, A2)
11 for i = 0 . . . j do
12     R[i] := R[i] ∧ (¬A1 ∨ ¬A2)
```

head in the state shown in Fig. 1(b). The loop then exits, and there is a null-dereference violation on `e` in the statement `e->next = p`.

3 Property-Directed Reachability

In this section, we present an adaptation of the IC3 algorithm that uses predicate abstraction. In this paper, by *predicate abstraction* we mean the technique that performs verification using a given *fixed* set of abstraction predicates [9], and not techniques that incorporate automatic refinement of the abstraction predicates; e.g. CEGAR. The PDR algorithm shown in Alg. 1 is parameterized by a given finite set of predicates $\mathcal{P}$ expressed in a logic $\mathcal{L}$. The requirements on the logic $\mathcal{L}$ are:

R1 $\mathcal{L}$ is decidable (for satisfiability).
R2 The transition relation for each statement of the programming language can be expressed as a two-vocabulary $\mathcal{L}$ formula.

Then for a particular program, we are given:

– A finite set of predicates $\mathcal{P} = \{p_i \in \mathcal{L}\}, 1 \leq i \leq n$.

– The transition relation of the system as a two-vocabulary formula $\rho \in \mathcal{L}$.
– The initial condition of the system, *Init* $\in \mathcal{L}$.
– The formula specifying the set of bad states, *Bad* $\in \mathcal{L}$.

Let $\mathcal{A}$ be the full predicate abstraction domain over the predicates $\mathcal{P}$. That is, each element $A \in \mathcal{A}$ is an *arbitrary* Boolean combination of the predicates $\mathcal{P}$. $A \in \mathcal{A}$ is inductive with respect to *Init* and ρ if and only if *Init* $\rightarrow A$ and $A \wedge \rho \rightarrow (A)'$. $(\varphi)'$ renames the vocabulary of constant symbols and relation symbols occurring in φ from $\{c, \ldots, r, \ldots\}$ to $\{c', \ldots, r', \ldots\}$. φ is $(\varphi)'$ stripped of primes.

If the logic $\mathcal{L}$ is propositional logic, then Alg. 1 is an instance of IC3 [3]. Our presentation is a simplification of more advanced variants [3, 7, 14]. For instance, the presentation omits inductive generalization, although our implementation does implement inductive generalization (see §5). Furthermore, this simplified presentation brings out the fact that the PDR algorithm is really an analysis *framework* that is parameterized on the set of abstraction predicates $\mathcal{P}$.

The algorithm employs an unbounded array R, where each *frame* $R[i] \in \mathcal{A}$ over-approximates the set of concrete states after executing the loop at most i times. The algorithm maintains an integer N, called the *frame counter*, such that the following invariants hold for all $0 \leq i < N$:

1. *Init* is a subset of all $R[i]$, i.e., *Init* $\rightarrow R[i]$.
2. The safety requirements are satisfied, i.e., $R[i] \rightarrow \neg$*Bad*.
3. Each of the $R[i+1]$ includes the states in $R[i]$, i.e., $R[i] \rightarrow R[i+1]$.
4. The successors of $R[i]$ are included in $R[i+1]$, i.e., for all σ, σ' if $\sigma \models R[i]$ and $\langle \sigma, \sigma' \rangle \models \rho$, then $\sigma' \models R[i+1]$.

We illustrate the workings of the algorithm using a simple example, after which we explain the algorithm in detail.

Example 1. Consider the program `while (x != y) x = x.n;` with precondition *Init* $\stackrel{\text{def}}{=} y \neq \text{null} \wedge x\langle n^+ \rangle y$. We wish to prove absence of null-dereference; that is, *Bad* $\stackrel{\text{def}}{=} x \neq y \wedge x = \text{null}$.

Table 2 shows a trace of PDR running with this input; each line represents a SAT query carried out by PDR$_{\mathcal{A}}$ (line 7) or by reduce$_{\mathcal{A}}$ (line 8). At each stage, if the result (r) is "unsat", then either we unfold one more loop iteration ($N := N + 1$) or we learn a new clause to add to $R[j]$ of the previous step, as marked by the "↗" symbol. If the result is "sat", the resulting model is used to further refine an earlier clause by recursively calling reduce$_{\mathcal{A}}$.

On the first row, we start with $R[0] = \mathbf{true}$, so definitely $R[0] \wedge$ *Bad* is satisfiable, for example with a model where $x = y = \text{null}$. The algorithm checks if this model represents a reachable state at iteration 0 (see the second row), and indeed it is not—the result is "unsat" and the unsat-core is $y = \text{null}$ (*Init* $\wedge\, y = \text{null}$ is not satisfiable). Therefore, we infer the negation, $y \neq \text{null}$, and add that to $R[0]$. The algorithm progresses in the same manner—e.g., after two more lines, $R[0] = (y \neq \text{null} \wedge x \neq \text{null})$, and so on. Eventually, the loop terminates when $R[i] = R[i+1]$ for some i; in this example, the algorithm terminates because $R[1] = R[2]$. The resulting invariant is $R[2] \equiv (y \neq \text{null} \wedge x\langle n^* \rangle y)$, a slight generalization of Pre in this case. □

j	**Formula**	**Model**	$A := \beta_{\mathcal{A}}(Model)$	**Inferred**
0	$R[0] \wedge \mathit{Bad}$	$(\text{null}, 1)\ 1 \mapsto \text{null}$	$A := x = \text{null} \wedge x \neq y \wedge \neg x\langle n^* \rangle y \wedge y\langle n^* \rangle x$	$x \neq \text{null}$
-1	$((\mathit{Init})' \vee (R[-1] \wedge \rho)) \wedge (A)'$	**unsat**		$\nearrow$
0	$R[0] \wedge \mathit{Bad}$	**unsat**		
1	$R[1] \wedge \mathit{Bad}$	$(\text{null}, 1)\ 1 \mapsto \text{null}$	$A := x = \text{null} \wedge x \neq y \wedge \neg x\langle n^* \rangle y \wedge y\langle n^* \rangle x$	$-$
0	$((\mathit{Init})' \vee (R[0] \wedge \rho)) \wedge (A)'$	$(1, 1)\ 1 \mapsto \text{null}$	$A := x = y \neq \text{null} \wedge x\langle n^* \rangle y \wedge y\langle n^* \rangle x$	$x \neq y$
-1	$((\mathit{Init})' \vee (R[-1] \wedge \rho)) \wedge (A)'$	**unsat**		$\nearrow$
1	$R[1] \wedge \mathit{Bad}$	$(\text{null}, 1)\ 1 \mapsto \text{null}$	$A := x = \text{null} \wedge x \neq y \wedge \neg x\langle n^* \rangle y \wedge y\langle n^* \rangle x$	$-$
0	$((\mathit{Init})' \vee (R[0] \wedge \rho)) \wedge (A)'$	$(1, 2)\ 1, 2 \mapsto \text{null}$	$A := x \neq y \wedge x, y \neq \text{null} \wedge \neg x\langle n^* \rangle y \wedge \neg y\langle n^* \rangle x$	$x\langle n^* \rangle y$
-1	$((\mathit{Init})' \vee (R[-1] \wedge \rho)) \wedge (A)'$	**unsat**		$\nearrow$
1	$R[1] \wedge \mathit{Bad}$	$(\text{null}, 1)\ 1 \mapsto \text{null}$	$A := x = \text{null} \wedge x \neq y \wedge \neg x\langle n^* \rangle y \wedge y\langle n^* \rangle x$	$x\langle n^* \rangle y$
0	$((\mathit{Init})' \vee (R[0] \wedge \rho)) \wedge (A)'$	**unsat**		$\nearrow$
1	$R[1] \wedge \mathit{Bad}$	**unsat**		
2	$R[2] \wedge \mathit{Bad}$	$(\text{null}, 1)\ 1 \mapsto \text{null}$	$A := x = \text{null} \wedge x \neq y \wedge \neg x\langle n^* \rangle y \wedge y\langle n^* \rangle x$	$x\langle n^* \rangle y$
1	$((\mathit{Init})' \vee (R[1] \wedge \rho)) \wedge (A)'$	**unsat**		$\nearrow$
	$R[1] = R[2]$	**valid**		

Table 2. Example run with $\mathit{Init} \stackrel{\text{def}}{=} y \neq \text{null} \wedge x\langle n^+ \rangle y$, $\mathit{Bad} \stackrel{\text{def}}{=} x \neq y \wedge x = \text{null}$, and $\rho \stackrel{\text{def}}{=} (x' = n(x))$. The output invariant is $I := x\langle n^* \rangle y$.

Some terminology used in the PDR algorithm:

- $\texttt{Model}(\varphi)$ returns a model σ satisfying φ if it exists, and None if it doesn't.
- The abstraction of a model σ, denoted by $\beta_{\mathcal{A}}(\sigma)$, is the cube of predicates from $\mathcal{P}$ that hold in σ: $\beta_{\mathcal{A}}(\sigma) = \bigwedge\{p \mid \sigma \models p, p \in \mathcal{P}\} \wedge \bigwedge\{\neg q \mid \sigma \models \neg q, q \in \mathcal{P}\}$.
- Let $\varphi \in \mathcal{L}$ is a formula in the unprimed vocabulary, $A \in \mathcal{A}$ is a value in the unprimed or primed vocabulary. $\mathit{Check}_{\mathcal{A}}(\varphi, A)$ returns a pair (r, A_1) such that
  - if $\varphi \wedge A$ is satisfiable, then $r = \mathsf{sat}$ and A_1 is the abstraction of a concrete state in the unprimed vocabulary. That is, if the given A is in the unprimed vocabulary, then $\beta_{\mathcal{A}}(\sigma)$ for some $\sigma \models \varphi \wedge A$; else if A is in the primed vocabulary, then $A_1 = \beta_{\mathcal{A}}(\sigma)$ for some $(\sigma, \sigma') \models \varphi \wedge A$.
  - if $\varphi \wedge A$ is unsatisfiable, then $r = \mathsf{unsat}$, and A_1 is a predicate such that $A \rightarrow A_1$ and $\varphi \wedge A_1$ is unsatisfiable. The vocabulary of A_1 is the same as that of A. If A is in the primed vocabulary (as in line 8 of Alg. 2), $\mathit{Check}_{\mathcal{A}}$ drops the primes from A_1 before returning the value.

  A valid choice for A_1 in the unsatisfiable case would be $A_1 = A$ (and indeed the algorithm would still be correct), but ideally A_1 should be the weakest such predicate. For instance, $\mathit{Check}_{\mathcal{A}}(\mathbf{false}, A)$ should return $(\mathsf{unsat}, \mathbf{true})$. In practice, when $\varphi \wedge A$ is unsatisfiable, the A_1 returned is an unsat core of $\varphi \wedge A$ constructed exclusively from conjuncts of A. Such an unsat core is a Boolean combination of predicates in $\mathcal{P}$, and thus is an element of $\mathcal{A}$.

We now give a more detailed explanation of Alg. 1. Each $R[i], i \geq 0$ is initialized to $\mathbf{true}$ (lines 2 and 10), and $R[-1]$ is $\mathbf{false}$. N is initialized to 0 (line 3). At line 5, the algorithm checks whether $R[i] = R[i+1]$ for some $0 \leq i < N$. If true, then an inductive invariant proving unreachability of *Bad* has been found, and the algorithm returns valid (line 6).

At line 7, the algorithm checks whether $R[N] \wedge \mathit{Bad}$ is satisfiable. If it is unsatisfiable, it means that $R[N]$ excludes the states described by *Bad*, and the frame counter

N is incremented (line 9). Otherwise, $A \in \mathcal{A}$ represents an abstract state that satisfies $R[N] \wedge \mathit{Bad}$. PDR then attempts to reduce $R[N]$ to try and exclude this abstract counterexample by calling reduce$_{\mathcal{A}}(N, A)$ (line 12).

The reduce algorithm (Alg. 2) takes as input an integer $j, 0 \leq j \leq N$, and an abstract state $A \in \mathcal{A}$ such that there is a path starting from A of length $N-j$ that reaches *Bad*. Alg. 2 tries to strengthen $R[j]$ so as to exclude A. At line 1, reduce first checks whether $\mathit{Init} \wedge A$ is satisfiable. If it is satisfiable, then there is an abstract trace of length $N-j$ from *Init* to *Bad*, using the transition relation ρ. The call to `Model` at line 3 checks whether there exists a concrete model corresponding to the abstract counterexample. ρ^k denotes k unfoldings of the transition relation ρ; ρ^0 is $\mathbf{true}$. $(\mathit{Bad})'^{\times k}$ denotes k applications of the renaming operation $(\cdot)'$ to *Bad*. If no such concrete model is found, then the abstraction was not precise enough to prove the required property (line 4); otherwise, a concrete counterexample to the property is returned (line 5).

Now consider the case when $\mathit{Init} \wedge A$ is unsatisfiable on line 1. $A_1 \in \mathcal{A}$ returned by the call to $\mathit{Check}_{\mathcal{A}}$ is such that $\mathit{Init} \wedge A_1$ is unsatisfiable; that is, $\mathit{Init} \rightarrow \neg A_1$.

The while-loop on lines 6–10 checks whether the $(N-j)$-length path to *Bad* can be extended backward to an $(N-j+1)$-length path. In particular, it checks whether $R[j-1] \wedge \rho \wedge (A)'$ is satisfiable. If it is satisfiable, then the algorithm calls reduce recursively on $j-1$ and A_2 (line 10). If no such backward extension is possible, the algorithm exits the while loop (line 9). Note that if $j = 0$, $\mathit{Check}_{\mathcal{A}}(R[j-1] \wedge \rho, A)$ returns $(\mathsf{unsat}, \mathbf{true})$, because $R[-1]$ is set to $\mathbf{false}$.

The conjunction of $(\neg A_1 \vee \neg A_2)$ to $R[i], 0 \leq i \leq j$, in the loop on lines 11–12 eliminates abstract counterexample A while preserving the required invariants on R. In particular, the invariant $\mathit{Init} \rightarrow R[i]$ is maintained because $\mathit{Init} \rightarrow \neg A_1$, and hence $\mathit{Init} \rightarrow (R[i] \wedge (\neg A_1 \vee \neg A_2))$. Furthermore, A_2 is the abstract state from which there is a (spurious) path of length $N-j$ to *Bad*. By the properties of $\mathit{Check}_{\mathcal{A}}$, $\neg A_1$ and $\neg A_2$ are each disjoint from A, and hence $(\neg A_1 \vee \neg A_2)$ is also disjoint from A. Thus, conjoining $(\neg A_1 \vee \neg A_2)$ to $R[i], 0 \leq i \leq j$ eliminates the spurious abstract counterexample A. Lastly, the invariant $R[i] \rightarrow R[i+1]$ is preserved because $(\neg A_1 \vee \neg A_2)$ is conjoined to all $R[i], 0 \leq i \leq j$, and not just $R[j]$.

Formally, the output of PDR$_{\mathcal{A}}(\mathit{Init}, \rho, \mathit{Bad})$ is captured by the following theorem:

Theorem 1. *Given (i) the set of abstraction predicates $\mathcal{P} = \{p_i \in \mathcal{L}\}, 1 \leq i \leq n$ where $\mathcal{L}$ is a decidable logic, and the full predicate abstraction domain $\mathcal{A}$ over $\mathcal{P}$, (ii) the initial condition Init $\in \mathcal{L}$, (iii) a transition relation ρ expressed as a two-vocabulary formula in $\mathcal{L}$, and (iv) a formula Bad $\in \mathcal{L}$ specifying the set of bad states, $PDR_{\mathcal{A}}(\mathit{Init}, \rho, \mathit{Bad})$ terminates, and reports either*

1. valid *if there exists $A \in \mathcal{A}$ s.t. (i) Init $\rightarrow A$, (ii) A is inductive, and (iii) $A \rightarrow \neg$Bad,*
2. *a concrete counterexample trace, which reaches a state satisfying Bad, or*
3. *an abstract trace, if the inductive invariant required to prove the property cannot be expressed as an element of $\mathcal{A}$.* □

The proof of Theorem 1 in [18] is based on the observation that, when "abstraction failure" is reported by reduce$_{\mathcal{A}}(j, A)$, the set of models $\sigma_i \models R[i]$ $(j \leq i < N)$ represents an abstract error trace.

Inductive Generalization. Each $R[i]$ is a conjunction of clauses $\varphi_1 \wedge \cdots \wedge \varphi_m$. If we detect that some ψ_j comprising a subset of literals of φ_j, it holds that $R[i] \wedge \rho \wedge \psi_j \models (\psi_j)'$, then ψ_j is *inductive relative to* $R[i]$. In this case, it is safe to conjoin ψ_j to $R[k]$ for $k \leq i+1$. Spurious counter-examples can also be purged if they are inductively blocked. The advantages of this method are explained thoroughly by Bradley [3].

4 Property-Directed Reachability for Linked-List Programs

In this section, we describe how $\text{PDR}_{\mathcal{A}}(\mathit{Init}, \rho, \mathit{Bad})$ described in Alg. 1 can be instantiated for verifying linked-list programs. The key insight is the use of the recently developed reachability logics for expressing properties of linked lists [17].

4.1 Reachability Logics

We use two related logics for expressing properties of linked data structures:

- AF^R is a decidable fragment of first-order logic with transitive closure (FO^{TC}), which is an alternation-free quantified logic. This logic is used to express the abstraction predicates $\mathcal{P}$, and pre- and post-conditions. It is closed under negation, and decidable for both satisfiability and validity.
- EA^R allows there to be universal quantifiers inside of existential ones. It is used to define the transition formulas of statements that allocate new nodes and dereference pointers. This logic is not closed under negation, and is only decidable for satisfiability. We count on the fact that transition formulas are only used in a positive form in the satisfiability queries in Alg. 1.

Although AF^R is used as the language for defining the predicates in $\mathcal{P}$, the *wlp* rules go slightly outside of AF^R, producing EA^R formulas (see Table 5 below).

Definition 1. ***(EA^R)*** *A* ***term****, t, is a variable or constant symbol. An* ***atomic formula*** *is one of the following: (i) $t_1 = t_2$; (ii) $r(t_1, t_2, \ldots, t_a)$ where r is a relation symbol of arity a (iii) A* ***reachability constraint*** *$t_1 \langle f^* \rangle t_2$, where f is a function symbol. A* ***quantifier-free formula*** *(QF^R) is a boolean combination of atomic formulas. A* ***universal formula*** *begins with zero or more universal quantifiers followed by a quantifier-free formula. An* ***alternation-free formula*** *(AF^R) is a boolean combination of universal formulas.* ***EA^R*** *consists of formulas with quantifier-prefix $\exists^*\forall^*$.*

In particular, $QF^R \subset AF^R \subset EA^R$. □

Technically, EA^R forbids *any* use of an individual function symbol f; however, when f defines an acyclic linkage chain—as in acyclic singly linked and doubly linked lists—f can be defined in terms of f^* by using universal quantification to express that an element is the closest in the chain to another element. This idea is formalized by showing that for all α and β, $f(\alpha) = \beta \leftrightarrow E_f(\alpha, \beta)$ where E_f is defined as follows:

$$E_f(\alpha, \beta) \stackrel{\text{def}}{=} \alpha\langle f^+ \rangle \beta \wedge \forall \gamma : \alpha\langle f^+ \rangle \gamma \rightarrow \beta\langle f^* \rangle \gamma, \tag{2}$$

where $\alpha\langle f^+ \rangle \beta \stackrel{\text{def}}{=} \alpha\langle f^* \rangle \beta \wedge \alpha \neq \beta$. However, because of the quantifier in Eqn. (2), the right-hand side of Eqn. (2) can only be used in a context that does not introduce a quantifier alternation (so that the formula remains in a decidable fragment of FO^{TC}).

Name	Formula
$x\langle f\rangle y$	$E_f(x,y)$
$f.ls\ [x,y]$	$\forall\alpha,\beta : x\langle f^*\rangle\alpha \wedge \alpha\langle f^*\rangle y \wedge \beta\langle f^*\rangle\alpha \rightarrow (\beta\langle f^*\rangle x \vee x\langle f^*\rangle\beta)$
$f.stable(h)$	$\forall\alpha : h\langle f^*\rangle\alpha \rightarrow alloc(\alpha)$
$f/b.rev\ [x,y]$	$\forall\alpha,\beta : \left(\begin{array}{c}\alpha \neq \texttt{null} \wedge \beta \neq \texttt{null}\\ \wedge\ x\langle f^*\rangle\alpha \wedge \alpha\langle f^*\rangle y \wedge x\langle f^*\rangle\beta \wedge \beta\langle f^*\rangle y\end{array}\right) \rightarrow \big(\alpha\langle f^*\rangle\beta \leftrightarrow \beta\langle b^*\rangle\alpha\big)$
$f.sorted\ [x,y]$	$\forall\alpha,\beta : \left(\begin{array}{c}\alpha \neq \texttt{null} \wedge \beta \neq \texttt{null}\\ \wedge\ x\langle f^*\rangle\alpha \wedge \alpha\langle f^*\rangle\beta \wedge \beta\langle f^*\rangle y\end{array}\right) \rightarrow \mathrm{dle}(\alpha,\beta)$

Table 3. AF^R formulas for the derived predicates shown in Table 1. f and b denote pointer fields. dle is an uninterpreted predicate that denotes a total order on the data values. The intention is that dle(α,β) holds whenever $\alpha\texttt{->}d \leq \beta\texttt{->}d$, where d is the data field. We assume that the semantics of dle are enforced by an appropriate total-order background theory.

A Predicate Abstraction Domain that uses AF^R. The abstraction predicates used for verifying properties of linked list programs were introduced informally in Table 1. Table 3 gives the corresponding formal definition of the predicates as AF^R formulas. Note that all four predicates defined in Table 3 are quantified. (The quantified formula for E_f is given in Eqn. (2).) In essence, we use a template-based approach for obtaining quantified invariants: the discovered invariants have a quantifier-free structure, but the atomic formulas can be quantified AF^R formulas.

We now show that the EA^R logic satisfies requirements R1 and R2 for the PDR algorithm stated in §3.

Decidability of EA^R. To satisfy requirement R1 stated in §3, we have to show that EA^R is decidable for satisfiability.

EA^R is decidable for satisfiability because any formula in this logic can be translated into the "effectively propositional" decidable logic of $\exists^*\forall^*$ formulas described by Piskac et al. [22]. EA^R includes relations of the form f^* (the reflexive transitive closure of a function symbol f), but only allows limited use of f itself.

Every EA^R formula can be translated into an $\exists^*\forall^*$ formula using the following steps [17]: (i) add a new uninterpreted relation R_f, which is intended to represent reflexive transitive reachability via f; (ii) add the consistency rule Γ_{linOrd} shown in Table 4, which asserts that R_f is a partial order, i.e., reflexive, transitive, acyclic, and linear;[8] and (iii) replace all occurrences of $t_1\langle f^*\rangle t_2$ by $R_f(t_1,t_2)$. (By means of this translation step, acyclicity is built into the logic.)

Proposition 1 (Simulation of EA^R). *Consider EA^R formula φ over vocabulary $\mathcal{V} = \langle\mathcal{C},\mathcal{F},\mathcal{R}\rangle$. Let $\varphi' \stackrel{\text{def}}{=} \varphi[R_f(t_1,t_2)/t_1\langle f^*\rangle t_2]$. Then φ' is a first-order formula over vocabulary $\mathcal{V}' = \langle\mathcal{C},\emptyset,\mathcal{R}\cup\{R_f : f\in\mathcal{F}\rangle$, and $\Gamma_{linOrd}\wedge\varphi'$ is satisfiable if and only if the original formula φ is satisfiable.*

This proposition is the dual of [16, Proposition 3, Appendix A.1] for validity of $\forall^*\exists^*$ formulas.

[8] Note that the order is a partial order and not a total order, because not every pair of elements must be ordered.

$\forall\alpha : R_f(\alpha,\alpha)$	reflexivity
$\wedge\ \forall\alpha,\beta,\gamma : R_f(\alpha,\beta) \wedge R_f(\beta,\gamma) \rightarrow R_f(\alpha,\gamma)$	transitivity
$\wedge\ \forall\alpha,\beta : R_f(\alpha,\beta) \wedge R_f(\beta,\alpha) \rightarrow \alpha = \beta$	acyclicity
$\wedge\ \forall\alpha,\beta,\gamma : R_f(\alpha,\beta) \wedge R_f(\alpha,\gamma) \rightarrow (R_f(\beta,\gamma) \vee R_f(\gamma,\beta))$	linearity

Table 4. A universal formula, Γ_{linOrd}, which asserts that all points reachable from a given point are linearly ordered.

Command C	*wlp*(C, Q)
`assume` φ	$\varphi \rightarrow Q$
`x = y`	$Q[y/x]$
`x = y->f`	$y \neq \texttt{null} \wedge \exists\alpha : (E_f(y,\alpha) \wedge Q[\alpha/x])$
`x->f = null`	$x \neq \texttt{null} \wedge Q[\alpha\langle f^*\rangle\beta \wedge (\neg\alpha\langle f^*\rangle x \vee \beta\langle f^*\rangle x)/\alpha\langle f^*\rangle\beta]$
`x->f = y`	$x \neq \texttt{null} \wedge Q[\alpha\langle f^*\rangle\beta \vee (\alpha\langle f^*\rangle x \wedge y\langle f^*\rangle\beta)/\alpha\langle f^*\rangle\beta]$
`x = malloc()`	$\exists\alpha : \neg\mathit{alloc}(\alpha) \wedge Q[(\mathit{alloc}(\beta) \vee (\beta = \alpha \wedge \beta = x))/\mathit{alloc}(\beta))]$
`free(x)`	$\mathit{alloc}(x) \wedge Q[(\mathit{alloc}(\beta) \wedge \beta \neq x)/\mathit{alloc}(\beta))]$

Table 5. Rules for *wlp* for atomic commands. *alloc* stands for a memory location that has been allocated and not subsequently freed. $E_f(y,\alpha)$ is the universal formula defined in Eqn. (2). $Q[y/x]$ denotes Q with all occurrences of atomic formula x replaced by y.

Axiomatic specification of concrete semantics in *EA*^{R}. To satisfy requirement R2 stated in §3, we have to show that the transition relation for each statement *Cmd* of the programming language can be expressed as a two-vocabulary formula $\rho \in \mathit{EA}^R$. Let *wlp*(*Cmd*, Q) be the weakest liberal precondition of command *Cmd* with respect $Q \in \mathit{EA}^R$. Then, the transition formula for command *Cmd* is *wlp*(*Cmd*, *Id*), where *Id* is a two-vocabulary formula that specifies that the input and the output states are identical, i.e.,

$$\mathit{Id} \stackrel{\text{def}}{=} \bigwedge_{c\in\mathcal{C}} c = c' \wedge \bigwedge_{f\in\mathcal{F}} \forall\alpha,\beta : \alpha\langle f^*\rangle\beta \Leftrightarrow \alpha\langle f'^*\rangle\beta.$$

To show that the concrete semantics of linked list programs can be expressed in EA^R, we have to prove that EA^R is closed under *wlp*; that is, for all commands *Cmd* and $Q \in \mathit{EA}^R$, *wlp*(*Cmd*, Q) $\in \mathit{EA}^R$.

Table 5 shows rules for computing *wlp* for atomic commands. Note that pointer-related rules in Table 5 each include a memory-safety condition to detect `null`-dereferences. For instance, the rule for "`x->f = y`" includes the conjunct "$x \neq$ `null`"; if, in addition, we wish to detect accesses to unallocated memory, the rule would be extended with the conjunct "$\mathit{alloc}(x)$".

The following lemma establishes the soundness and completeness of the *wlp* rules.

Lemma 1. *Consider a command C of the form defined in Table 5 and postcondition Q. Then, $\sigma \models \mathit{wlp}(C, Q)$ if and only if the execution of C on σ can yield a state σ' such that $\sigma' \models Q$.*

This lemma is the dual of [16, Prop. 1, App. A.1] for validity of $\forall^*\exists^*$ formulas.

Benchmark	Memory-safety + data-structure integrity					Additional properties				
	$\mathcal{A}$	Time	N	# calls to Z3	# clauses	$\mathcal{A}$	Time	N	# calls to Z3	# clauses
create		1.37	3	28	3		8.19	4	96	7
delete		14.55	4	61	6		9.32	3	67	7
deleteAll	St	6.77	3	72	6	St	37.35	7	308	12
filter		2.37	3	27	4		55.53	5	94	5
insert		26.38	5	220	16		25.25	4	155	13
prev		0.21	2	3	0		11.64	4	118	6
last		0.33	2	3	0		7.49	3	41	4
reverse		5.35	5	128	4		146.42	6	723	11
sorted insert	S	41.07	3	48	7	S	51.46	4	134	10
sorted merge		26.69	4	87	10	S	256.41	5	140	14
make doubly-linked		18.91	3	44	5	R	1086.61	5	112	8

Table 6. Experimental results. Column $\mathcal{A}$ signifies the set of predicates used (blank = only the top part of Table 1; S = with the addition of the *sorted* predicate family; R = with the addition of the *rev* family; St = with the addition of the *stable* family, where *alloc* conjuncts are added in *wlp* rules). Running time is measured in seconds. N denotes the highest index for a generated element $R[i]$. The number of clauses refers to the inferred loop invariant.

Weakest liberal preconditions of compound commands $C_1; C_2$ (sequencing) and $C_1 | C_2$ (nondeterministic choice) are defined in the standard way, i.e.,

$$wlp(C_1; C_2, Q) \stackrel{\text{def}}{=} wlp(C_1, wlp(C_2, Q)) \qquad wlp(C_1 | C_2, Q) \stackrel{\text{def}}{=} wlp(C_1, Q) \wedge wlp(C_2, Q)$$

Consider a program with a single loop "`while` *Cond* `do` *Cmd*". Alg. 1 can be used to prove whether or not a precondition $Pre \in AF^R$ before the loop implies that a postcondition $Post \in AF^R$ holds after the loop, if the loop terminates: we supply Alg. 1 with $Init \stackrel{\text{def}}{=} Pre$, $\rho \stackrel{\text{def}}{=} Cond \wedge wlp(Cmd, Id)$ and $Bad \stackrel{\text{def}}{=} \neg Cond \wedge \neg Post$. Furthermore, memory safety can be enforced on the loop body by setting $Bad \stackrel{\text{def}}{=} (\neg Cond \wedge \neg Post) \vee (Cond \wedge \neg wlp(Cmd, \mathbf{true}))$.

5 Experiments

To evaluate the usefulness of the analysis algorithm, we applied it to a collection of sequential procedures that manipulate singly and doubly-linked lists (see Table 6). For each program, we report the predicates used, the time (in seconds), the number of PDR frames, the number of calls to Z3, and the size of the resulting inductive invariant, in terms of the number of clauses. All experiments were run on a 1.7GHz Intel Core i5 machine with 4GB of RAM, running OS X 10.7.5. We used version 4.3.2 of Z3 [5], compiled for a 64-bit Intel architecture (using gcc 4.2 and LLVM).

For each of the benchmarks, we verified that the program avoids `null`-dereferences, as well as that it preserves the data-structure invariant that the inputs and outputs are acyclic linked-lists. In addition, for some of the benchmarks we were also able to verify some additional correctness properties. While full functional correctness, or even partial correctness, is hard to achieve using predicate abstraction, we were able to use simple formulas to verify several interesting properties that go beyond

Benchmark	Property checked
create	Some memory location pointed to by `x` (a global variable) that was allocated prior to the call, is not reachable from the list head, `h`.
delete	The argument `x` is no longer reachable from `h`.
deleteAll	An arbitrary non-null element `x` of the list becomes non-allocated.
filter	Two arbitrary elements `x` and `y` that satisfy the filtering criterion and have an `n`-path between them, maintain that path.
insert	The new element `e` is reachable from `h` and is the direct predecessor of the argument `x`.
last	The function returns the last element of the list.
prev	The function returns the element just before `x`, if one exists.
reverse	If `x` comes before `y` in the input, then `x` should come after `y` in the output.
sorted insert	The list rooted at `h` remains sorted.
make doubly-linked	The resulting `p` is the inverse of `n` within the list rooted at `h`.

Table 7. Some correctness properties that can be verified by the analysis procedure. For each of the programs, we have defined suitable *Pre* snf *Post* formulas in AF^R.

		Automatic bug finding			
Benchmark	Bug description	Time	N	# calls to Z3	c.e. size
insert	Precondition is too weak (omitted $e \neq$ `null`)	4.46	1	17	8
filter	Potential `null` dereference	6.30	1	21	3
	Typo: list head used instead of list iterator	103.10	3	79	4
reverse	Corrupted data structure: a cycle is created	0.96	1	9	2

Table 8. Results of experiments with buggy programs. Running time is measured in seconds. N denotes the highest index for a generated element $R[i]$. "C.e. size" denotes the largest number of individuals in a model in the counterexample trace.

memory-safety properties and data-structure invariants. Table 7 describes the properties we checked for the various examples. As seen from columns 3, 4, 8, and 9 of the entries for `delete` and `insert` in Table 6, trying to prove *stronger* properties can sometimes result in *fewer* iterations being needed, resulting in a *shorter* running time. In the remainder of the examples, handling additional properties beyond memory-safety properties and data-structure invariants required more processing effort, which can be attributed mainly to the larger set of symbols (and hence predicates) in the computation.

Bug Finding. We also ran our analysis on programs containing deliberate bugs, to demonstrate the utility of this approach to bug finding. In all of the cases, the method was able to detect the bug and generate a concrete trace in which the safety or correctness properties are violated. The output in that case is a series of concrete states $\sigma_0..\sigma_N$ where each σ_i contains the set of heap locations, pointer references, and program variables at step i. The experiments and their results are shown in Table 8. We found both the length of the trace and the size of the heap structures to be very small. Their small size makes the traces useful to present to a human programmer, which can help in locating and fixing the bug.

Observations. It is worth noting that for programs where the proof of safety is trivial—because every access is guarded by an appropriate conditional check, such as in `prev` and `last`—the algorithm terminates almost immediately with the correct invariant

true. This behavior is due to the property-directedness of the approach, in contrast with abstract interpretation, which always tries to find the least fixed point, regardless of the desired property.

We experimented with different refinements of inductive-generalization (§3). Our algorithm could in many cases succeed without it, but without the most basic version that just pushes each clause (without removing literals), we observed runs with up to $N = 40$ iterations. On the other hand, the more advanced versions of inductive generalization did not help us: trying to remove literals resulted in a large number of expensive (and useless) solver calls; and blocking spurious counter-examples using inductive generalization also turned out to be quite expensive in our setting.

We also noticed that the analysis procedure is sensitive to the number of abstraction predicates used. In particular, using predicates whose definitions involve quantifiers can affect the running time considerably. When the predicate families $f.sorted\ [x, y]$ and $f/b.rev\ [x, y]$ are added to $\mathcal{A}$, running times can increase substantially (about 20-60 times). This effect occurred even in the case of `sorted merge`, where we did not attempt to prove an additional correctness property beyond safety and integrity—and indeed there were no occurrences of the added predicates in the loop invariant obtained. As can be seen from Table 6, the PDR algorithm *per se* is well-behaved, in the sense that the number of calls to Z3 increased only modestly with the additional predicates. However, each call to Z3 took a lot more time.

6 Related Work

The literature on program analysis is vast, and the subject of shape analysis alone has an extensive literature. Thus, in this section we are only able to touch on a few pieces of prior work that relate to the ideas used in this paper.

Predicate abstraction. Houdini [8] is the first algorithm of which we are aware that aims to identify a loop invariant, given a set of predicates as candidate ingredients. However, Houdini only infers *conjunctive* invariants from a given set of predicates. Santini [29, 28] is a recent algorithm for discovering invariants expressed in terms of a set of candidate predicates. Like our algorithm, Santini is based on full predicate abstraction (i.e., it uses arbitrary Boolean combinations of a set of predicates), and thus is strictly more powerful than Houdini. Santini could make use of the predicates and abstract domain described in this paper; however, unlike our algorithm, Santini would not be able to report counterexamples when verification fails. Other work infers quantified invariants [27, 15] but does not support the reporting of counterexamples. Templates are used in many tools to define the abstract domains used to represent sets of states, by fixing the form of the constraints permitted. Template Constraint Matrices [26] are based on inequalities in linear real arithmetic (i.e., polyhedra), but leave the linear coefficients as symbolic inputs to the analysis. The values of the coefficients are derived in the course of running the analysis. In comparison, a coefficient in our use of EA^R corresponds to one of the finitely many constants that appear in the program, and we instantiated our templates prior to using PDR.

As mentioned in §1, PDR meshes well with full predicate abstraction: in effect, the analysis obtains the benefit of the precision of the abstract transformers for full predicate abstraction, without ever constructing the abstract transformers explicitly. PDR

also allows a predicate-abstraction-based tool to create concrete counterexamples when verification fails.

Abstractions based on linked-list segments. In this paper, our abstract domain is based on formulas expressed in AF^R, which has very limited capabilities to express properties of stretches of data structures that are not pointed to by a program variable. This feature is similar to the self-imposed limitations on expressibility used in a number of past approaches, including (a) canonical abstraction [25]; (b) a prior method for applying predicate abstraction to linked lists [21]; (c) an abstraction method based on "must-paths" between nodes that are either pointed to by variables or are list-merge points [19]; and (d) domains based on separation logic's list-segment primitive [6, 2] (i.e., "ls$[x, y]$" asserts the existence of a possibly empty list segment running from the node pointed to by x to the node pointed to by y). Decision procedures have been used in previous work to compute the best transformer for individual statements that manipulate linked lists [30, 23].

STRAND and elastic quantified data automata. Recently, Garg et al. developed methods for obtaining quantified invariants for programs that manipulate linked lists via an abstract domain of *quantified data automata* [10, 11]. To create an abstract domain with the right properties, they use a weakened form of automaton—so-called *elastic* quantified data automata—that is unable to observe the details of stretches of data structures that are not pointed to by a program variable. (Thus, an elastic automaton has some of the characteristics of the work based on linked-list segments described above.) An elastic automaton can be converted to a formula in the decidable fragment of STRAND over lists [20].

Other work on IC3/PDR. Our work represents the first application of PDR to programs that manipulate dynamically allocated storage. We chose to use PDR because it has been shown to work extremely well in other domains, such as hardware verification [3, 7]. Subsequently, it was generalized to software model checking for program models that use linear real arithmetic [14] and linear rational arithmetic [4]. Cimatti and Griggio [4] employ a quantifier-elimination procedure for linear rational arithmetic, based on an approximate pre-image operation. Our use of a predicate-abstraction domain allows us to obtain an approximate pre-image as the unsat core of a single call to an SMT solver (line 8 of Alg. 2).

7 Conclusion

Compared to past work on shape analysis, our approach (i) is based on full predicate abstraction, (ii) makes use of standard theorem proving techniques, (iii) is capable of reporting concrete counterexamples, and (iv) is based on property-directed reachability. The experimental evaluation in §5 illustrates these four advantages of our approach. The algorithm is able to establish memory-safety and preservation of data-structure invariants for all of the examples, using only the simple predicates given in Table 1. This result is surprising because earlier work on shape analysis that employed the same predicates [13] failed to prove these properties. One reason is that [13] only uses positive and negative combinations of these predicates, whereas our algorithm uses arbitrary Boolean combinations of predicates.

References

1. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *TACAS*, 2001.
2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
3. A. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.
4. A. Cimatti and A. Griggio. Software model checking via IC3. In *CAV*, 2012.
5. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
6. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
7. N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, 2011.
8. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for Esc/Java. In *FME*, 2001.
9. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, 2002.
10. P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *CAV*, 2013.
11. P. Garg, P. Madhusudan, and G. Parlato. Quantified data automata on skinny trees: An abstract domain for lists. In *SAS*, 2013.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
13. L. Hendren. *Parallelizing Programs with Recursive Data Structures*. PhD thesis, Cornell Univ., Ithaca, NY, Jan 1990.
14. K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.
15. K. Hoder, L. Kovács, and A. Voronkov. Invariant generation in vampire. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 60–64. Springer, 2011.
16. S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. Technical report, IMDEA, Madrid, Spain, 2011. Available at software.imdea.org/∼ab/Publications/cav2013tr.pdf.
17. S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *CAV*, 2013.
18. S. Itzhaky, N. Bjørner, T. Reps, M. Sagiv, and A. Thakur. Property-directed shape analysis. TR 1807, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, May 2014.
19. T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, 2006.
20. P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using STRAND. In *SAS*, 2011.
21. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, 2005.
22. R. Piskac, L. de Moura, and N. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *J. Autom. Reasoning*, 44(4):401–424, 2010.
23. A. Podelski and T. Wies. Counterexample-guided focus. In *POPL*, 2010.
24. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.
25. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
26. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In R. Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2005.

27. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234, 2009.
28. A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Attaining most-precise inductive invariants. TR-1790, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, Apr. 2013.
29. A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Automating abstract interpretation. *Electr. Notes Theor. Comp. Sci.*, 2013.
30. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.