Shape Analysis through Predicate Abstraction and Model Checking

Dennis Dams and Kedar S. Namjoshi

Bell Labs, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974.
{dennis,kedar}@research.bell-labs.com

Abstract. We propose a new framework, based on predicate abstraction and model checking, for *shape analysis* of programs. Shape analysis is used to statically collect information — such as possible reachability and sharing — about program stores. Rather than use a specialized abstract interpretation based on shape graphs, we instantiate a generic and automated abstraction procedure with shape predicates from a correctness property. This results in a predicate-discovery procedure that identifies predicates relevant for correctness, using an analysis based on weakest preconditions, and creates a finite state abstract program. The correctness property is then checked on the abstraction with a model checking tool. To enable this process, we calculate weakest preconditions for common shape properties, and present heuristics for accelerating convergence.

Exploring abstract state spaces with model checkers enables one to tap into a wealth of techniques and highly optimized implementations for state space exploration, and to analyze properties that go beyond invariances. We illustrate this simple and flexible framework with the analysis of some "classical" list manipulation programs, using our implementation of the abstraction algorithm, and the SPIN and COSPAN model checkers for state space exploration.

1 Introduction

Shape analysis is used to statically determine global properties of the program heap. Examples of such properties are "points-to" reachability between objects, the existence of cycles, or sharing within the heap. Typically, such analyses are based on abstract interpretations [7] of heaps with various kinds of *shape graphs*. This paper presents a new framework for shape analysis, which is based on Schmidt and Steffen's observation that static analysis is model checking of an abstract interpretation [30]. The exploitation of this paradigm renders our framework different in several key ways.

A major difference is the way in which abstract interpretation is performed. We use a *generic* abstraction algorithm to calculate an abstraction of the program, relative to a given shape property. Starting with the shape predicates in the property, the algorithm iteratively, and in a goal-directed manner, discovers other predicates that are relevant to the property, by computing weakest preconditions (*wp*) [12]. It also constructs an abstraction where these predicates are represented with boolean variables: the process is an instance of *predicate abstraction* [14]. The correctness of the abstract interpretation is guaranteed by the algorithm. An advantage is that non-shape predicates (such as arithmetic inequalities), which improve the precision of abstraction, are included in a straightforward manner. One of the main contributions of this paper is the calculation of weakest preconditions for second-order shape predicates like reachability, the identification of other predicates that arise in this process, and the calculation of their *wp*'s. We have implemented a weakest precondition calculator for shape predicates relative to C-like program constructs, including heuristics to accelerate convergence.

The abstract program computed by this algorithm is analyzed with a *generic* model checker. This enables one to tap into a wealth of techniques and highly optimized implementations for state space exploration. Furthermore, one can check for temporal properties that go beyond invariances. Our calculator generates the abstracted program in formats that are accepted by the explicit state model checker SPIN [18] and the BDD-based model checker COSPAN [16]. We demonstrate our approach on a "classical" list reversal program. A detailed description of the experiments, including input and result files, can be found at [32].

```
typedef struct node
  {struct node *n; int data;} Node;
typedef Node *List;

List insert(List x, int a)
{
List t;
n1: t=(List) malloc(sizeof(Node));
n2: t->data=a;
n3: t->n=x;
n4: x=t;
ne: return x;
}
```

```
n1: {reach[;n](x,k), false}
    t=(List) malloc (sizeof(Node));
n2: {reach[&(t->n);n](x,k), (t==k)}
    t->data=a;
n3: {reach[&(t->n);n](x,k)\/(t==k)}
    t->n=x;
n4: {reach[;n](t,k)}
    x=t;
ne: {reach[;n](x,k)}
    return x;
```

Fig. 1. List insertion procedure (left) and *wp* calculations (right)

Overview We introduce our framework through the analysis of a C program for inserting an element at the head of a list, shown in Figure 1. We are interested in checking the property that the insertion process does not, inadvertently, make one of the existing list members unreachable from x. Define the predicate $\mathit{reach}[A;F](i,j)@l$ to be true of those program states where control is at location l and, in the heap, it is possible to reach address j from address i, following only those fields in F and avoiding all addresses in set A. We write $\mathit{reach}[;F](i,j)$ if A is empty, and when $F = \{n\}$ it is written as n. The above property can then be expressed formally by the linear temporal logic formula $(\forall k : \mathsf{G}(\mathit{reach}[;n](x,k)$ $@n1 \Rightarrow \mathsf{G}(\mathit{true}@ne \Rightarrow \mathit{reach}[;n](x,k)@ne)))$.

The predicate $reach[;n](x,k)$ holds at ne iff its *weakest precondition* (*wp*) [12] holds at $n4$. While weakest preconditions for simple constructs can be computed by syntactic substitution, this is not true for second order predicates like reachability. We show in Section 3 how to compute *wp* for reachability: for the present discussion, it suffices to know that the *wp* for this predicate simplifies to the expected value: $reach[;n](t,k)$. The abstraction algorithm (Section 2) computes *wp*'s for individual predicates in this goal-directed, "backward" manner until the initial location $n1$ is reached. The results are shown enclosed in $\{\ldots\}$ in Figure 1 (right) with *wp*'s for predicates at $n3$ separated by a comma at $n2$. The key point to note is how the *wp* calculations identify further predicates that are relevant to the correctness property.

```
b0 <-> reach[;n](x,k)
b1 <-> reach[;n](t,k)
b2 <-> reach[&(t->n);n](x,k)
b3 <-> (t==k)
```

```
n1: b2 := b0, b3 := false;
n2: b2 := b2, b3 := b3;
n3: b1 := b2 \/ b3;
n4: b0 := b1;
ne:
```

Fig. 2. Predicate-boolean correspondence (left) and abstract program (right)

The abstract program, and the predicate-boolean correspondence, is shown in Figure 2. The abstract actions are calculated by substituting boolean variables for predicates in the results of the *wp* calculations. For instance, the update for $b1$ on edge $n3$ to $n4$ is given by $b2 \lor b3$, which is the result of substitution on the *wp* for its corresponding predicate $reach[;n](t,k)$. Note that booleans are only guaranteed to have correct values where it matters — e.g. $b1$ may have the (clearly wrong) value *true* at $n2$, but it is set to its correct value at location $n3$, just before it is used to calculate $b0$. The abstraction algorithm ensures this property which, in turn, ensures the correctness of the abstraction.

The adjusted correctness property for the abstract program, $\mathsf{G}(b0@n1 \Rightarrow \mathsf{G}(true@ne \Rightarrow b0@ne))$ can be established using a model checker. For this example, the property can be established directly from the *wp* calculations; however, this is difficult to do for the analysis of programs with loops. In general, the *wp* calculations serve to transform the program *locally*, while the model checking determines *global* properties. As the abstraction is always *conservative* in nature, a property that holds of the abstraction also holds of the source program. Hence, the source program is also correct.

The various components of the framework are described in detail in subsequent sections.

2 Abstraction by Iterated Weakest Preconditions

In Predicate Abstraction [14], the abstract program is defined over a set of boolean variables which represent source program predicates. Determining a set of predicates relevant to showing a correctness property is undecidable in general

[17]; however, several (semi-) algorithms exist [26, 6, 3, 22]. We use a modified form of the algorithm from [26]. This algorithm *simultaneously* derives a set of relevant predicates and computes the abstract actions, through an iterated weakest precondition calculation.

The original algorithm operates on programs modeled by a set of actions defined as guarded commands. A sequential procedure body can be easily translated to this notation by using a variable, say "*pc*", to represent the control location. With this encoding, however, the scheme in [26] examines every action at each iteration, resulting in several unnecessary calculations. Thus, we encode the control transition in the action name itself, and tailor the algorithm to only inspect relevant actions. A control flow edge from location m to n labeled with *guard* g and *update* a is turned into the action $s_{m,n} : g \rightarrow a$ (note that a is deterministic). For an update $s : g \rightarrow a$, $wp(s, p) \equiv g \wedge wp(a, p)$[1]. The other major difference is that we allow for user guidance of the abstraction in the form of abstraction hints.

Our modified algorithm is presented in Figure 3. The algorithm iteratively computes a set of pairs (p, n), where p is a predicate and n is a control location. The pair (p, n) asserts that predicate p holds at location n. The data structures used are a set of pairs, S, and a set of newly generated pairs, N. The parameters to the algorithm are: (a) a correctness property, written in a universal temporal logic such as LTL or ACTL*, (b) an iteration bound k, and (c) a set of *approximation hints*.

In the main loop (step 2), the algorithm processes unmarked pairs in a breadth-first manner. For each unmarked pair (p, n), and every action $s_{m,n}$, the algorithm computes $wp(s_{m,n}, p)$. By the semantics of wp, the truth of predicate p at node n after executing statement $s_{m,n}$ is given by the value of $wp(s_{m,n}, p)$ at node m. This, in turn, is determined by the values of its constituent predicates at node m. These predicates are extracted, and processed in the next iteration. From the undecidability result, there can be no termination guarantee in general, so a user-supplied bound k is used to limit the number of iterations. The approximation hints are used to introduce new predicates that may accelerate the termination of the loop – an example is provided in Section 5. Every predicate p generated during the algorithm has a corresponding boolean variable, called b_p. The substitution of predicates in a formula f with their corresponding booleans results in a formula $\overline{f}$ – as a general rule, we represent the abstract version of a concrete object o by $\overline{o}$. *Simplification* is used in step 2 to reduce the number of newly generated pairs for faster termination: correctness does not depend on the power, but only on the soundness of the simplifier.

We now examine some other issues that arise in applying this algorithm. An *initial condition*, *init*, can be encoded by introducing a new transition $s_{n0,n1}$: *init* $\rightarrow$ *skip*, between a new initial location ($n0$) and the old one ($n1$). An abstraction computed without hints can be used only to prove properties that depend solely on predicates generated by the iterated wp calculations. While this may

[1] This reflects the semantics that an action is executed at a state only if its guard holds at that state.

1. Initially, S contains all pairs from the correctness property, together with (q, m), for each predicate q that occurs in the guard of some action $s_{m,n} : g \to a$. Initially, the abstract actions are defined as $\overline{s}_{m,n} : \overline{g} \to skip$. All pairs in S are unmarked, and N is empty.
2. At each iteration, as long as there is an unmarked pair in S, and the iteration bound k is not reached, do the following.
   (a) For each unmarked pair (p, n), mark it as examined, and:
      i. if there is an approximation hint mapping (p, n) to an expression h, then add the equivalence $b_p \equiv \overline{h}$ at node n and insert $\{(q, n) \mid q \in pred(h)\}$ into N.
      ii. else, consider action $s_{m,n} : g \to a$, for each m, and:
         A. compute $wp(s_{m,n}, p)$, and *simplify* it to obtain a formula f.
         B. for each q in $pred(f)$, add (q, m) to the set N.
         C. add $b_p := \overline{f}$ to the update of abstract action $\overline{s}_{m,n}$.
   (b) Add all pairs in N to S.
3. After step 2 has terminated, *over-approximate* the booleans corresponding to any unmarked predicates, using either an approximation hint, or non-determinism $(\{false, true\})$, or $\top$.

Fig. 3. The predicate abstraction algorithm

be sufficient in some cases (e.g., list insertion) it is necessary, in general, to use hints. The approximation hint for $p@n$ can be any expression h that is "more abstract" than p. This is formalized as $p \preceq_3 h$, where $\preceq_3$ is the "abstraction order" relation of 3-valued logic[2]. For instance, $(x > 3)$ may be approximated by *if* $(x \leq 0)$ *then false else* $\top$.

The algorithm always computes a conservative approximation to the source program, where a program state (s, n) is related to an abstract state (s', n') if, and only if, the control locations n, n' are identical, and for every predicate p such that the pair (p, n) is considered during abstraction, $p(s) \preceq_3 b_p(s')$. This algorithm is also complete in the sense shown in [26, 2].

As stated, this algorithm is intra-procedural. It can be extended to handle procedure calls by the process described in [1]. For example, $wp(x := F(y), p(x))$ can be calculated by determining $wp(body(F), p(r))(y)$, where r is the return value of the body of F. This introduces additional predicates within the body of F. The call to F is then replaced by a call to the abstract version of F, i.e., $b_p := \overline{F}(\{b_q \mid q \in pred(wp(body(F), p(r)))\})$.

3 Weakest Preconditions for Shape Analysis

To utilize the above abstraction algorithm for shape analysis, we need to calculate weakest preconditions for common shape properties such as *reachability*, *cyclicity*, and *sharing*. The definitions of these properties, and their weakest precondition calculations, are both based on a *memory model*.

[2] $x \preceq_3 x$ for all x, and $x \preceq_3 \top$, for all x. $\top$ and $\{false, true\}$ are identified for this purpose.

3.1 Memory Model

Heap and stack contents are modeled by an unbounded array M, indexed by the integers, together with a finite subset of the integers, called *alloc*, which records the allocated addresses. A structure field name, such as n, is represented with a function, called $\hat{n}$, which maps the address of the structure to the address where field n is 'stored' – we assume field names are globally distinct. An expression e has two attributes relative to M: its *address*, denoted by $addr_M(e)$, and its *value*, denoted by $val_M(e)$. The rules for calculating these attributes, and the interpretations of basic program statements, are given in Figure 4. In these rules, attributes are written as a pair (address,value), $\perp$ represents an undefined result, and we have simplified matters by having *malloc* allocate a single memory location.

Program variable x: $(\alpha(x), M[\alpha(x)])$, where α maps program variables to addresses
Structure access $e.n$: $(\hat{n}(addr_M(e)), M[\hat{n}(addr_M(e))])$
Address expression $\&e$: $(\perp, addr_M(e))$
Dereference $*e$: $(val_M(e), M[val_M(e)])$, if $val_M(e) \in alloc$, else error
Numeric constant c: $(\perp, c)$
Arithmetic operation $op(e1, \ldots, en)$: $(\perp, op(val_M(e1), \ldots, val_M(en)))$
Pointer addition $e + i$: $(\perp, val_M(e) + i)$

Guard g: $val_M(g)$
Ordinary assignment $e1 := e2 : M[addr_M(e1)] := val_M(e2)$
Memory allocation $e := malloc : M[addr_M(e)] := a; alloc := alloc \cup \{a\}$,
for some $a \notin alloc$
Memory de-allocation $free(e) : alloc := alloc \setminus \{val_M(e)\}$

Fig. 4. The memory model

The *wp* of a predicate p relative to a statement s is computed by translating both p and s in terms of M, calculating *wp* in the standard way for array updates [15], and translating the result back to the syntax of program expressions (see e.g. [5]). For an array update statement $s : M[a] := v$, $wp(s, p(M))$ is given by $p(M')$, where M' is the array defined by $M'[i] = M[i]$ for $i \neq a$ and $M'[a] = v$. This new array is also denoted by $M' = M[a \leftarrow v]$. Distributing M' into p results in an expression in terms of M.

For example, consider the predicate $p \equiv (x = 0)$, for a program variable x, and statement $s : *u := 10$. The assignment is interpreted as a memory update resulting in $M' \equiv M[val_M(u) \leftarrow 10]$, and $wp(s,p)$ is given by $(val_M(x') = 0)$. Distributing M' into *val* results in $(\textit{if } (val_M(u) = \alpha(x)) \textit{ then } 10 \textit{ else } val_M(x)) = 0$, which simplifies to $(val_M(u) \neq \alpha(x)) \wedge (val_M(x) = 0)$. Translating back to program syntax gives $(u \neq \&x) \wedge (x = 0)$ as the weakest precondition. This process of translating back and forth from M thus correctly takes into account *aliasing* effects. It is tedious to carry out such calculations by hand, but they are easily automated, as described in the following section.

3.2 Weakest Preconditions for Shape Predicates

We have, so far, considered *wp* calculations for simple kinds of predicates, albeit taking into account complex aliasing effects. We are primarily interested in more global, second-order shape properties. The key property is $reach[A; F](i, j, M)$. Informally, this says that there is a sequence of steps in M from address i to address j, which avoids all addresses in A, and uses only the fields in F. We define this precisely below as a least fixpoint. A *step* refers to a memory dereference. For example, if z is a variable of type `Node` (see Figure 1), the location where the address of the next node is stored is $\hat{n}(\alpha(z))$ (the value of $\&((*z).n)$). But the address of the next element itself is given by $M[\hat{n}(\alpha(z))]$, which results from a memory dereference.

For a set F of field names, let $F^*(w, y, A)$ hold iff y is reachable from w using only field accesses from F (e.g. $x.a.b.c$), while avoiding addresses in A. This is defined as follows:

$$\begin{aligned} F^*(w, y, A) \;\equiv\;& alloc(y) \;\wedge \\ & (\mu Z, x : alloc(x) \;\wedge\; (x = y \;\vee\; (\neg A(x) \;\wedge\; (\exists a : a \in F : Z(\hat{a}(x))))))(w) \end{aligned}$$

We can then define *reach* by

$$\begin{aligned} & reach[A; F](w, b, M) \;\equiv \\ & (\mu Z, x : alloc(x) \;\wedge\; (\exists k : F^*(x, k, A) : (k = b \;\vee\; (\neg A(k) \;\wedge\; Z(M[k])))))(w) \end{aligned}$$

Note the explicit dereferencing step $M[k]$ in this definition.

The *wp* for *reach* is calculated for an update $M' = M[i \leftarrow c]$ by substituting M' for M in this fixpoint expression, and simplifying the result. The definitions of the other shape predicates and their *wp*'s are shown in Figure 5. Their *wp*'s are calculated using the *wp* for *reach*; their derivations are available at [32]. Informally, the *wp* for reachability says that it is possible to reach b from x after an update $M' = M[i \leftarrow c]$ provided that, in the previous state, either: (i) it is possible to reach b from x avoiding addresses in $A \cup \{i\}$, or (ii) i is not in A, and there are paths from x to i, and from c to b that avoid A. In the first case, the memory update does not invalidate the path and, in the second case, the memory update serves to link two paths into the desired path from x to b. The other *wp* expressions have similar informal readings.

The *wp*'s for these shape predicates have a remarkable *closure* property, in that they are all expressed in terms of other shape predicates in the same set. Closure ensures that only these types of shape predicates are generated by the abstraction algorithm, making it possible to spot patterns that indicate where approximation is needed. Such pattern-driven approximation is used to analyze a list reversal program in Section 5.

We use predicates with the same names to state program properties: e.g., $reach[A; F](e1, e2)$, where A is a set of program expressions, and $e1, e2$ are program expressions. The translation into the memory model (as a prelude to computing *wp*) is given by $reach[val_M(A); F](val_M(e1), val_M(e2), M)$.

• $reach[A;F](x,b,M)$: it is possible to reach address b from address x in 0 or more steps. The wp is given by

$$reach[Ai;F](x,b,M) \ \lor \ (\neg A(i) \ \land \ reach[Ai;F](x,i,M) \ \land \ reach[Ai;F](c,b,M))$$

• $reachp[A;F](x,b,M)$: it is possible to reach address b from address x in 1 or more steps. This is defined by $alloc(x) \ \land \ (\exists k : F^*(x,k,A) : \neg A(k) \ \land \ reach[A;F](M(k),b,M))$, with wp:

$$reachp[Ai;F](x,b,M) \ \lor \ (reach[Ai;F](c,b,M) \ \land \ \neg A(i) \ \land \ reach[Ai,F](x,i,M))$$

• $dshared[A;F](x,y,M)$: there exists a non-null node reachable from both x and y. This is defined by $(\exists v : v \neq \mathsf{NULL} : reach[A;F](x,v,M) \ \land \ reach[A;F](y,v,M))$, with wp:

$$\begin{aligned}
&dshared[Ai;F](x,y,M) \ \lor \\
&(\neg A(i) \ \land \ reach[Ai;F](x,i,M) \ \land \ dshared[Ai;F](y,c,M)) \ \lor \\
&(\neg A(i) \ \land \ reach[Ai;F](y,i,M) \ \land \ dshared[Ai;F](x,c,M)) \ \lor \\
&(\neg A(i) \ \land \ (c \neq \mathsf{NULL}) \ \land \ reach[Ai;F](x,i,M) \ \land \ reach[Ai;F](y,i,M))
\end{aligned}$$

• $cyclic[A;F](x,M)$: x reaches a node that is involved in a cycle. This is defined by $(\exists v : reach[A;F](x,v,M) \ \land \ reachp[A;F](v,v,M))$, with wp:

$$\begin{aligned}
&cyclic[Ai;F](x,M) \ \lor \\
&(\neg A(i) \ \land \ reach[Ai;F](x,i,M) \ \land \ cyclic[Ai;F](c,M)) \ \lor \\
&(\neg A(i) \ \land \ reach[Ai;F](x,i,M) \ \land \ reach[Ai;F](c,i,M))
\end{aligned}$$

Fig. 5. Weakest preconditions for shape predicates for $M' = M[i \leftarrow c]$. In these formulas, we use Ai to represent $A \cup \{i\}$

As an example, in the program from Section 1, consider the predicate $reach[;n](x,k)$, and the assignment $x := t$, where t and x are of type `List`. The translated predicate is given by $reach[;n](val_M(x), val_M(k), M)$, while the assignment results in the memory update $M' = M[\alpha(x) \leftarrow val_M(t)]$. Substituting M' for M gives $reach[;n](val_{M'}(x), val_{M'}(k), M'))$. This simplifies, using the wp for $reach$, to $reach[\alpha(x);n](val_M(t), val_M(k), M) \ \lor \ (true \ \land \ \underline{reach[;n](val_M(t), \alpha(x), M)} \ \land \ reach[;n](val_M(t), val_M(k), M))$. From the definition of type `List`, it is clear that it is not possible to reach the address of x from t. Thus, the underlined term simplifies to $false$, and the result is: $reach[\alpha(x);n](val_M(t),\ val_M(k), M)$. The avoiding address $\alpha(x)$ is superfluous for the same reason, so it can be removed, giving the result (in program syntax) as $reach[;n](t,k)$.

4 A Predicate Calculator for Shape Analysis

We have implemented a prototype predicate abstractor for shape analysis, based on the set of predicates discussed above, in OCaml. The input to this tool consists of a flow-chart program with C-style instructions and variable types including all basic types as well as struct and pointer types. Nested structs, unions, and array types are not currently accepted by the tool but their inclusion poses no technical difficulties. The tool does not yet handle procedure calls. Along with the program, a set of predicate-location pairs is given, as e.g. extracted from the property to be checked. These serve as the starting point for the *wp* calculations. Finally, an iteration bound and a (possibly empty) collection of approximation hints are given as input.

The tool's main challenge is in simplifying the "raw" formulas that are obtained as *wp*'s. These may be large due to case distinctions for aliasing. By rewriting newly generated predicates to simpler ones, semantical equivalence with already-computed predicates can often be detected. This reduces the overall number of generated predicates, which is essential in a practical application of the algorithm. Also, it renders the predicates more readable, which facilitates the identification of good approximations. We have implemented a variety of rules that aim to simplify individual predicates like *reach* and their arguments, and pointer (in)equalities. For example, $reach[A; F](null, e)$ rewrites to *false* regardless of A, F, and e; in $reach[A; F](e_1, e_2)$, e_2 can be removed from the avoid set A; and $\&(x \to n) \neq \&y$ is true when y is a variable. Another essential class of rewrite rules is formed by *type reasoning*, further discussed below. Furthermore, we apply several standard rules, including the Davis-Putnam-Logemann-Loveland procedure [11], to simplify boolean expressions. The tool includes automatic conversion of abstract programs to S/R or Promela format, the resp. input formalisms for the model checkers COSPAN and SPIN. The correctness property to be verified can be added to the abstract program by using assertions or temporal logic.

4.1 Type Reasoning

Suppose that x and y are program variables of the *List* type from our examples. A typical predicate that may occur during the manipulation of *wp*'s is $reach[A; n](x, \&y)$. Regardless of the avoid set, this predicate is false, as can be seen by reasoning about types, as follows. Variable x itself is of type *List*. By dereferencing x, we get an object of struct-type *Node*, in which selection of the n field yields a *List* again. So the only types that are reachable from *List* are *List* and *Node*. The type of $\&y$ however is pointer to *List*. So $\&y$ cannot be reachable from x.

Reachability between types is formalized by a predicate *typreach*. For a set F of field names, $typreach[F](t_1, t_2)$ expresses that from an object of address type t_1 it is possible to reach an object of address type t_2 if only selection of fields from F is allowed. It is possible to prove the key correctness property that if $reach[A; F](e_1, e_2)$, then $typreach[F](t_1, t_2)$ for any address expressions e_1

and e_2 with types t_1 and t_2 resp. This property allows the simplifier to replace a predicate of the form $reach[A;F](e_1,e_2)$by *false* if $typreach[F](t_1,t_2)$ fails to hold. (Otherwise, nothing is replaced.) Type reasoning is also used to simplify equalities; e.g. $(x == y)$ is false if the types of x and y differ.

5 An Example

We illustrate our approach on a program for in-place reversal of singly-linked lists, also considered in e.g. [4, 24, 33, 28, 29]. The core of the program is given in Figure 6.

```
List x, y, t;  /* x is an acyclic list */

n1: y = NULL;
n2: while (x != NULL) {
n3:   t = y;
n4:   y = x;
n5:   x = x->n;
n6:   y->n = NULL;
n7:   y->n = t; };
n8:
```

Fig. 6. The list reversal program

Initially, x is the list to be reversed. It is traversed head to tail, reversing the next-pointers (n) one by one. At the start of every iteration of the while loop, x is the rest of the list to be reversed and y is the initial segment that has been reversed so far. Variable t is auxiliary; at any point during the reversal, it points to one of the first two nodes of the already-reversed segment. Using shape analysis, we want to verify that y is an acyclic list after the reversal (using the negation of predicate $p_1 = cyclic[;n]y@n8$), given the precondition that x is acyclic ($p_2 = cyclic[;n]x@n1$).

We run our tool on (the flow-chart description of) the program together with predicates p_1 and p_2, but without any approximation. By choosing a relatively large iteration bound, predicates are generated that occur in *wp*'s obtained by propagating p_1 backwards through the sequence of statements of the while loop for a several iterations (p_2, being at $n1$, does not generate any new predicates). By manually inspecting these predicates, we can get an idea of the appropriate approximations to be applied. Setting the iteration bound to 30, 185 predicate/location pairs are calculated corresponding to about 3 backward iterations through the while loop. All predicates are of type *cyclic*, *reach*, or equality; they differ in their arguments. The following is an excerpt from the tool's output, showing some predicates relevant at location $n5$.

```
n5: reach[(&(y->n),&(*(x).n->n));n](t,*(x).n)
n5: reach[(&(y->n),&(*(x).n->n),&(*(*(x).n).n->n));n](t,*(x).n)
n5: reach[(&(y->n),&(*(x).n->n),&(*(*(x).n).n->n));n](t,*(*(x).n).n)
```

The difference between these instantiations of the *reach* predicate is in the number of "$\rightarrow n$ dereferences" of x. Clearly, the predicates will keep growing along this pattern for every next iteration of *wp*'s around the while loop, due to the assignment $x = x \rightarrow n$ after $n5$. An approximation that weeds out this growth is found by observing that the union of all these predicates implies that *from t, an address can be reached that can also be reached by following k n-fields starting from* $*(x).n$*, for some* k. A similar property can be expressed by a single *dshared* predicate — hence we use that as an approximation. If $\mathit{dshared}[;n](t,x)$ is false, then so is every of the predicates above. So we will try to cut off the growth pattern by approximating the first of those predicates, $\mathit{reach}[(\&(y \rightarrow n), \&(*(x).n \rightarrow n));n](t,*(x).n)$ at $n5$, by "if $\neg\mathit{dshared}[;n](t,x)$ then *false* else $\top$".

This is the only pattern that occurs in the list of predicates, and indeed several other predicates at $n5$ can be approximated in terms of $\mathit{dshared}[;n](t,x)$ as well. The pattern also occurs in a sequence of equality predicates. At $n5$ these equations are in terms of y and x, and it is not immediately clear how to approximate them. But if we consider the same pattern at $n4$, where the y's get replaced by x's due to the *wp* over the assignment $y = x$, it shows up as follows.

```
n4: (x==*(x).n)
n4: (x==*(*(x).n).n)
n4: (x==*(*(*(x).n).n).n)
```

It is clear that none of the predicates can be true under the given precondition that x is not cyclic. So here we bring in the predicate $\mathit{cyclic}[;n]x$ by approximating $x = (*(x).n)$ by "if $\neg\mathit{cyclic}[;n]x$ then *false* else $\top$".

Having added 5 approximations, introducing the two new predicates mentioned above, we rerun the tool. We choose a larger iteration bound (40) so that not only all points will be reached where the approximations apply, but also the two new predicates introduced by them are propagated backwards far enough so that any patterns that they themselves may generate become apparent. This time, the pattern (the same as before: growing $\rightarrow n$-dereferences) occurs in 3 new predicates. One of them is a *dshared* predicate and can be approximated, at $n5$, in terms of the predicate $\mathit{dshared}[;n](t,x)$ from above. The other 2 stem from the predicate $\mathit{cyclic}[;n]x$ and can be approximated by that same predicate, at $n4$.

The third run of the tool, although started with iteration bound 40 again, converges after 29 iterations with the message that no predicates remain to be examined. A total number of 33 relevant predicates have been found at that point, 3 of which are suggested by the approximations, of which there are 8 altogether. Each run of the tool takes about 1 second.

Another way to identify suitable predicates for approximation is through the analysis of counter-examples that are produced by the model checkers in case the current abstraction is too coarse. Error-trace analysis boils down to solving satisfiability questions over predicate formulae, and thus we might benefit here from work on *decidable* logics to reason about heaps or arrays [4, 19, 21, 31].

Model checking Next, we instruct the tool to produce the corresponding abstracted list reversal program in both S/R and Promela formats, and use the COSPAN and SPIN model checkers to independently verify the original cyclicity property. In both cases, the checking is done in the order of hundredths of a second, within minimal amounts of memory (0.1MB with SPIN and less with COSPAN[3]). The Promela version has 34 reachable states, each 48 bytes in size. For the S/R version, 32 states are reached[4] and the constructed BDD's have 2454 nodes. Both verifications confirm that the property holds. Removing the precondition that x is acyclic results in failure, showing that it is necessary.

In case of the list-insertion example from the Introduction, the tool converges after 4 iterations without the need for any approximations. So in this case the abstraction is fully automatic. The resulting S/R and Promela models have 12 and 7 reachable states resp., and verification is again done in a fraction of a second with minimal amounts of memory in both cases.

6 Related Work and Conclusions

A synthesis and generalization of several existing algorithms for shape analysis is presented in [29]. Their algorithm constructs a shape graph invariant, expressed in 3-valued logic, by an abstract interpretation of program actions. The invariant is based on two core predicates: $x(v)$ (the node for variable x) and $n(v1, v2)$ (a link from $v1$ to $v2$ via field n). To improve precision, user-supplied instrumentation predicates have to be used, including shape predicates and also non-shape predicates such as $\leq$. Precision can also be improved by a *focus* operation that turns undefined values into non-determinism, or by *materializing* new elements (e.g., to distinguish between reachability in 0, 1, or more steps). A *coerce* operation eliminates inconsistent parts of an invariant. The implementation (TVLA) [24] includes a *blur* operation, which weakens an invariant.

Although the exact relationship between our algorithms is—as yet—unclear, some general comments can be made. First, the abstract program computed by our algorithm can be used to construct shape graph invariants—this is, in fact, done implicitly by the model checking procedure during reachability analysis (in addition, non-invariance properties can also be checked.) Secondly, operations similar to focus, coerce, and blur, all of which have to do with the precision of the reachability computation, are implemented in model checkers. Determining how well these generic techniques work for the particular problem of shape analysis is an intriguing question for future work but, in the examples we have considered, the model checking was not an issue.

One of the chief differences is the backward, goal-directed nature of our abstraction method, and the corresponding lack of distinction between core and

[3] SPIN always seems to take at least 0.1MB due to overhead or a built-in lower bound.
[4] The difference in the number of reachable states is due to different ways of modeling.

instrumentation predicates. In fact, the iterated *wp* calculations, starting with predicates from the property, naturally identify other relevant predicates, including all of the needed instrumentation predicates. On the other hand, the TVLA tool analyzes the list reversal example fully automatically, in contrast to our use of user-supplied approximation hints. However, we believe that it is possible to automate the heuristics we have used for identifying approximations, so that programs such as this are handled fully automatically.

In [25, 27], *wp* for reachability is calculated, but no other shape predicates are considered. Predicate abstraction, combined with model checking, has been used in analyses of some heap properties: points-to analysis [1], correctness of concurrent garbage collectors [10, 9], and loop invariants [13]. These papers, however, do not handle shape properties.

In conclusion, we believe that the separation of concerns between abstraction and state space exploration that is proposed in the new framework opens up several possibilities. The abstraction method serves to discover the predicates that are relevant for proving a given property. The *wp* calculations perform abstraction *locally*, leaving the *global* state space exploration to a model checker. One can thus take advantage of highly optimized model checking implementations, and the wide variety of logics and system models to which they apply. Our initial experience has, we hope, demonstrated the promise of this method, while raising several interesting questions for theoretical investigations and experimental improvements.

Our ongoing efforts are focused on mechanizing the heuristics for approximations. As the verification problem for shape properties is undecidable, we cannot hope for a fully automated procedure that works on all instances. However, there have been successful attempts [23] to automate similar approximation heuristics using recognition of pattern growth in (regular) expressions, based on the framework of *widening* [7, 8]. Also, we can potentially benefit from the design of theorem proving tools, such as ACL2 [20], which successfully recognize induction patterns in many cases.

References

1. T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.
2. T. Ball, A. Podelski, and S. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, volume 2280 of *LNCS*, 2002.
3. T. Ball and S. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*, 2001.
4. M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *ESOP*, volume 1576 of *LNCS*, pages 2–19, 1999.
5. R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 102–126, 2000.
6. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, 2000.

7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.
8. Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 269–295, 1992.
9. S. Das and D. Dill. Successive approximation of abstract transition relations. In *LICS*, 2001.
10. S. Das, D. Dill, and S. Park. Experience with predicate abstraction. In *CAV*, volume 1633 of *LNCS*, 1999.
11. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Assoc. Computing Machinery*, 7:201–215, 1960.
12. E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *C.ACM*, 18, 1975.
13. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, 2002.
14. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, 1997.
15. D. Gries. *The Science Of Programming*. Springer-Verlag, 1981.
16. R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *CAV*, volume 1102 of *LNCS*, 1996.
17. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
18. G. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
19. J.L. Jensen, M.E. Jørgensen, N. Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 226–236, 1997.
20. M. Kaufmann, P. Manolios, and J.S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
21. N. Klarlund and M.I. Schwartzbach. Graphs and decidable transductions based on edge constraints (extended abstract). In *Colloquium on Trees in Algebra and Programming*, pages 187–201, 1994.
22. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS*, volume 2031 of *LNCS*, 2001.
23. D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized networks of processes. *Theoretical Computer Science*, 256:113–144, 2001.
24. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, volume 1824 of *LNCS*, 2000.
25. J. Morris. (1) A general axiom of assignment (2) Assignment and linked data structures. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, 1981.
26. K.S. Namjoshi and R.P. Kurshan. Syntactic program transformations for automatic abstraction. In *CAV*, volume 1855 of *LNCS*, 2000.
27. G. Nelson. Verifying reachability invariants of linked structures. In *POPL*, 1983.
28. N. Rinetzky and S. Sagiv. Interprocedural shape analysis for recursive programs. In *Computational Complexity*, pages 133–149, 2001.
29. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.

30. D.A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. In *SAS*, volume 1503 of *LNCS*, 1998.
31. A. Stump, C.W. Barrett, D.L. Dill, and J.R. Levitt. A decision procedure for an extensional theory of arrays. In *LICS*, pages 29–37, 2001.
32. `http://www.cs.bell-labs.com/~kedar/shape_analysis.html`.
33. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *POPL*, pages 27–40, 2001.