A Reachability Predicate for Analyzing Low-Level Software

Shaunak Chatterjee[1], Shuvendu K. Lahiri[2], Shaz Qadeer[2], and Zvonimir Rakamarić[3]

[1] Indian Institute of Technology, Kharagpur
[2] Microsoft Research
[3] University of British Columbia

Abstract. Reasoning about heap-allocated data structures such as linked lists and arrays is challenging. The *reachability* predicate has proved to be useful for reasoning about the heap in type-safe languages where memory is manipulated by dereferencing object fields. Sound and precise analysis for such data structures becomes significantly more challenging in the presence of low-level pointer manipulation that is prevalent in systems software.

In this paper, we give a novel formalization of the reachability predicate in the presence of internal pointers and pointer arithmetic. We have designed an annotation language for C programs that makes use of the new predicate. This language enables us to specify properties of many interesting data structures present in the Windows kernel. We present preliminary experience with a prototype verifier on a set of illustrative C benchmarks.

1 Introduction

Static software verification has the potential to improve programmer productivity and reduce the cost of producing reliable software. By finding errors at the time of compilation, these techniques help avoid costly software changes late in the development cycle and after deployment. Many successful tools for detecting errors in systems software have emerged in the last decade [2,16,10]. These tools can scale to large software systems; however, this scalability is achieved at the price of precision. Heap-allocated data structures are one of the most significant sources of imprecision for these tools. Fundamental correctness properties, such as control and memory safety, depend on intermediate assertions about the contents of data structures. Therefore, imprecise reasoning about the heap usually results in a large number of annoying false warnings increasing the probability of missing the real errors.

The *reachability predicate* is important for specifying properties of *linked* data structures. Informally, a memory location v is reachable from a memory location u in a heap if either $u = v$ or u contains the address of a location x and v is reachable from x. Automated reasoning about the reachability predicate is difficult for two reasons. First, reachability cannot be expressed in first-order logic, the input language of choice for most modern and scalable automated theorem provers. Second, it is difficult to precisely specify the update to the reachability predicate when a heap location is updated.

Previous work has addressed these problems in the context of a reachability predicate suitable for verifying programs written in high-level languages such as Java and C# [22,18,1,17,5]. This predicate is inadequate for reasoning about low-level software, which commonly uses programming idioms such as internal pointers (addresses of object fields) and pointer arithmetic to move between object fields. We illustrate this point with several examples in Section 2.

The goal of our work is to build a scalable verifier for systems software that can reason precisely about heap-allocated data structures. To this end, we introduce in this paper a new reachability predicate suitable for verifying low-level programs written in C. We describe how to automatically compute the precise update for the new predicate and a method for reasoning about it using automated first-order theorem provers. We have designed a specification language that uses our reachability predicate, allows succinct specification of interesting properties of low-level software, and is conducive to modular program verification. We have implemented a modular verifier for annotated C programs called HAVOC (Heap-Aware Verifier Of C). We report on our preliminary encouraging experience with HAVOC on a set of small but interesting C programs.

1.1 Related Work

HAVOC is a static assertion checker for C programs in the same style that ESC/Java [15] is a static checker for Java programs, and Spec# [4] is a static checker for C# programs. However, HAVOC is different in that it deals with the low-level intricacies of C and provides reachability as a fundamental primitive in its specification language. The ability to specify reachability properties also distinguishes HAVOC from other assertion checkers for C such as CBMC [9] and SATURN [23]. The work of McPeak and Necula [20] allows reasoning about reachability, but only indirectly using ghost fields in heap-allocated objects. These ghost fields must be updated manually by the programmer whereas HAVOC provides the update to its reachability predicate automatically.

There are several verifiers that do allow the verification of properties based on the reachability predicate. TVLA [19] is a verification tool based on abstract interpretation using 3-valued logic [22]. It provides a general specification logic combining first-order logic with reachability. Recently, they have also added an axiomatization of reachability in first-order logic to the system [18]. However, TVLA has mostly been applied to Java programs and, to our knowledge, cannot handle the interaction of reachability with pointer arithmetic.

Caduceus [14] is a modular verifier for C programs. It allows the programmer to write specifications in terms of arbitrary recursive predicates, which are axiomatized in an external theorem prover. It then allows the programmer to interactively verify the generated verification conditions in that prover. HAVOC only allows the use of a fixed set of reachability predicates but provides much more automation than Caduceus. All the verification conditions generated by

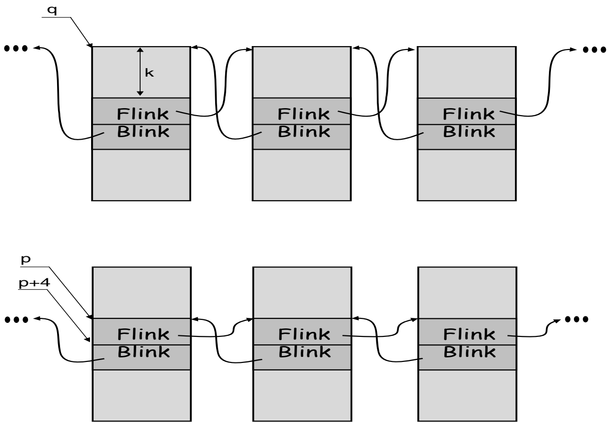

Fig. 1. Doubly-linked lists in Java and C

Havoc are discharged automatically using SMT (satisfiability modulo-theories) provers. Unlike Caduceus, Havoc understands internal pointers and the use of pointer arithmetic to move between fields of an object.

Calcagno et al. have used separation logic to reason about memory safety and absence of memory leaks in low-level code [7]. They perform abstract interpretation using rewrite rules that are tailored for "multi-word lists", a fixed predicate expressed in separation logic. Our approach is more general since we provide a family of reachability predicates, which the programmer can compose arbitrarily for writing richer specifications (possibly involving quantifiers); the rewriting involved in the generation and validation of verification conditions is taken care of automatically by Havoc. Their tool can infer loop invariants but handles procedures by inlining. In contrast, Havoc performs modular reasoning, but does not infer loop invariants.

2 Motivation

Consider the two doubly-linked lists shown in Figure 1. The list at the top is typical of high-level object-oriented programs. The linking fields `Flink` and `Blink` point to the *beginning* of the successor and predecessor objects in the list. In each iteration of a loop that iterates over the linked list, the iterator variable points to the beginning of a list object whose contents are accessed by a simple field dereference. Existing work would allow properties of this linked list to be specified using the two reachability predicates $\mathtt{R}_{\mathtt{Flink}}$ and $\mathtt{R}_{\mathtt{Blink}}$, each of which is a binary relation on *objects*. For example, $\mathtt{R}_{\mathtt{Flink}}(a, b)$ holds for objects a and b if $a.\mathtt{Flink}^i = b$ for some $i \geq 0$.

The list at the bottom is typical of low-level systems software. Such a list is constructed by embedding a structure `LIST_ENTRY` containing the two fields, `Flink` and `Blink`, into the objects that are supposed to be linked by the list.

```
typedef struct _LIST_ENTRY {
  struct _LIST_ENTRY *Flink;
  struct _LIST_ENTRY *Blink;
} LIST_ENTRY;
```

The linking fields, instead of pointing to the beginning of the list objects, point to the beginning of the embedded linking structure. In each iteration of a loop that iterates over such a list, the iterator variable contains a pointer to the beginning of the structure embedded in a list object. A pointer to the beginning of the list object is obtained by performing pointer arithmetic captured with the following C macro.

```
#define CONTAINING_RECORD(a, T, f) \
            (T *) ((int)a - (int)&((T *)0)->f)
```

This macro expects an internal pointer `a` to a field `f` of an object of type `T` and returns a typed pointer to the beginning of the object.

There are two good engineering reasons for this ostensibly dangerous programming idiom. First, it becomes possible to write all list manipulation code for operations such as insertion and deletion separately in terms of the type `LIST_ENTRY`. Second, it becomes easy to have one object be a part of several different linked lists; there is a field of type `LIST_ENTRY` in the object corresponding to each list. For these reasons, this idiom is common both in the Windows and the Linux operating system[1].

Unfortunately, this programming idiom cannot be modeled using the predicates $\mathtt{R}_{\mathtt{Flink}}$ and $\mathtt{R}_{\mathtt{Blink}}$ described earlier. The fundamental reason is that these lists may link objects via pointers at a potentially non-zero offset into the objects. Different data structures might use different offsets; in fact, the offset used by a particular data structure is a crucial part of its specification. This is in stark contrast to the first kind of linked lists in which the linking offset is guaranteed to be zero.

The crucial insight underlying our work is that for analyzing low-level software, *the reachability predicate must be a relation on pointers rather than objects.* A pointer is a pair comprising an object and an integer offset into the object, and the program memory is a map from pointers to pointers. We introduce an integer-indexed set of binary reachability predicates: for each integer n, the predicate $\mathtt{R}_n$ is a binary relation on the set of pointers. Suppose n is an integer and p and q are *pointers.* Then $\mathtt{R}_n(p, q)$ holds if and only if either $p = q$, or recursively $\mathtt{R}_n(*(p + n), q)$ holds, where $*(p + n)$ is the pointer stored in memory at the address obtained by incrementing p by n.

Our reachability predicate captures the insight that in low-level programs a list of pointers is constructed by performing an alternating sequence of pointer arithmetic (with respect to a constant offset) and memory lookup operations. For example, let p be the address of the `Flink` field of an object in the linked list at the bottom of Figure 1. Then, the forward-going list is captured by the

[1] In Linux, the `CONTAINING_RECORD` macro corresponds to the `list_entry` macro.

```
typedef struct { int data; LIST_ENTRY link; } A;

struct { LIST_ENTRY a; } g;

requires BS(&g.a) && B(&g.a, 0) == &g.a
requires forall(x, list(g.a.Flink, 0), x == &g.a || Off(x) == 4)
requires forall(x, list(g.a.Flink, 0), x == &g.a || Obj(x) != Obj(&g.a))
modifies decr(list(g.a.Flink, 0), 4)
ensures forall(x, list(g.a.Flink, 0), x == &g.a || deref(x-4) == 42)

void list_iterate() {
  LIST_ENTRY *iter = g.a.Flink;
  while (iter != &(g.a)) {
    A *elem = CONTAINING_RECORD(iter, A, link);
    elem->data = 42;
    iter = iter->Flink;
  }
}
```

Fig. 2. Example

pointer sequence $p, *(p + 0), *(*(p + 0) + 0), \ldots$. Similarly, assuming that the size of a pointer is 4, the backward-going list is captured by the pointer sequence $p, *(p + 4), *(*(p + 4) + 4), \ldots$.

The new reachability predicate is a generalization of the existing reachability predicate and can just as well describe the linked list at the top of Figure 1. Suppose the offset of the Flink field in the linked objects is k and q is the address of the start of some object in the list. Then, the forward-going list is captured by $q, *(q+k), *(*(q+k)+k), \ldots$ and the backward-going list is captured by $q, *(q + k + 4), *(*(q + k + 4) + k + 4), \ldots$.

2.1 Example

We illustrate the use of our reachability predicate in program verification with the example in Figure 2. The example has a type A and a global structure g with a field a. The field a in g and the field link in the type A have the type LIST_ENTRY, which was defined earlier. These fields are used to link together in a circular doubly-linked list the object g and a set of objects of type A. The field a in g is the dummy head of this list. The procedure list_iterate iterates over this list setting the data field of each list element to 42.

In addition to verifying the safety of each memory access in list_iterate, we would like to verify two additional properties. First, the only parts of the caller-visible state modified by list_iterate are the data fields of the list elements. Second, the data field of each list element is 42 when list_iterate terminates.

To prove these properties on list_iterate, it is crucial to have a precondition stating that the list of objects linked by the Flink field of LIST_ENTRY is circular.

To specify this property, we extend the notion of well-founded lists, first described in an earlier paper [17], to our new reachability predicate. The predicate $\mathtt{R}_n$ is well-founded with respect to a set BS of *blocking pointers* if for all pointers p, the sequence $*(p+n), *(*(p+n)+n), \ldots$ contains a pointer in BS. This member of BS is called the *block* of p with respect to the offset n and is denoted by $\mathtt{B}_n[p]$. Typical members of BS include pointer values that indicate the end of linked lists, e.g., the null pointer or the head `&g.a` of the circular lists in our example.

Our checker Havoc enforces a programming discipline associated with well-founded lists. Havoc provides an *auxiliary* variable BS whose value is a set of pointers and allows program statements to add or remove pointers from BS. Further, each heap update in the program is required to preserve the well-foundedness of $\mathtt{R}_n$ with respect to each offset n of interest.

The first precondition of `list_iterate` uses the notion of well-foundedness to express that `&g.a` is the head of a circular list. In this precondition, `B(&g.a,0)` refers to $\mathtt{B}_0$`[&g.a]`. We use $\mathtt{B}_0$ to specify that the circular list is formed by the `Flink` field, which is at offset 0 within `LIST_ENTRY`. The second precondition illustrates how facts about an entire collection of pointers are expressed in our specification language. In this precondition, the expression `list(g.a.Flink,0)` refers to the finite and non-empty set of pointers in the sequence `g.a.Flink`,$*($`g.a.Flink` $+\ 0), \ldots$ upto but excluding the pointer $\mathtt{B}_0$(`g.a.Flink`). Also, the function `Off` retrieves the offset (or the second component) from a pointer. This precondition states that the offset of each pointer in `list(g.a.Flink,0)`, excluding the dummy head, is equal to 4, the offset of the field sequence `link.Flink` in the type `A`. The third precondition uses the function `Obj`, which retrieves the object (or the first component) from a pointer. This precondition says that the object of each pointer, excluding the dummy head, in `list(g.a.Flink,0)` is different from the object of the dummy head.

The `modifies` clause illustrates yet another constructor of a set of pointers provided by our language. If S is a set of pointers, then `decr`(S, n) is the set of pointers obtained by decrementing each pointer in S by n. The modifies clause captures the update of the `data` field at relative offset -4 from the members of `list(g.a.Flink,0)`.

The postcondition of the procedure introduces the operator `deref`, which returns the content of the memory at a pointer address. This postcondition says that the value of the `data` field of each object in the list, excluding the dummy head, is 42.

Using loop invariants provided by us (not shown in the figure), Havoc is able to verify that the implementation of this procedure satisfies its specification. Note that in the presence of potentially unsafe pointer arithmetic and casts, it is nontrivial to verify that the heap update operation `elem->data := 42` does not change the linking structure of the list. Since Havoc cannot rely on the static type of the variable `elem`, it must prove that the offset of `elem` before the operation is 0 and therefore the operation cannot modify either linking field.

```
typedef struct { int x; int y[10]; } DATA;

DATA *create() {                    procedure create() returns d:ptr {
  int a;                              var a:ptr;
                                      a := call malloc(4);
  DATA *d =                           d := call malloc(44);
    (DATA *) malloc(sizeof(DATA));
  init(d->y, 10, &a);                 call init(PLUS(d, Ptr(null,4)),
                                                Ptr(null,10), a);
  d->x = a;                           Mem[PLUS(d, Ptr(null,0))] := Mem[a];
                                      call free(a);
  return d;
}                                   }

void init(int *in, int size,         procedure init(in:ptr, size:ptr,
          int *out) {                               out:ptr) {
  int i;                               var i:ptr;
  i = 0;                               i := Ptr(null,0);
  while (i < size) {                   while (LT(i, size)) {
    in[i] = i;                           Mem[PLUS(in, i)] := i;
    *out = *out + i;                     Mem[out] := PLUS(Mem[out], i);
    i++;                                 i := PLUS(i, Ptr(null,1));
  }                                    }
}                                    }
```

Fig. 3. Translation of C programs

3 Operational Semantics of C

Our semantics for C programs depends on three fundamental types, the uninterpreted type `ref` of object references, the type `int` of integers, and the type `ptr` $=$ `ref` $\times$ `int` of pointers. Each variable in a C program, regardless of its static type, contains a pointer value. A *pointer* is a pair containing an object reference and an integer offset. An integer value is encoded as a pointer value whose first component is the special constant `null` of type `ref`. The constructor function `Ptr` : `ref` $\times$ `int` $\rightarrow$ `ptr` constructs a pointer value from its components. The selector functions `Obj` : `ptr` $\rightarrow$ `ref` and `Off` : `ptr` $\rightarrow$ `int` retrieve the first and second component of a pointer value, respectively.

The heap of a C program is modeled using two map variables, `Mem` and `Alloc`, and a map constant `Size`. The variable `Mem` maps pointers to pointers and intuitively represents the contents of the memory at a pointer location. The variable `Alloc` maps object references to the set {`UNALLOCATED, ALLOCATED, FREED`} and is used to model memory allocation. The constant `Size` maps object references to positive integers and represents the size of the object. The procedure call `malloc(n)` for allocating a memory buffer of size `n` returns a pointer `Ptr`$(o, 0)$ where o is an object such that `Alloc`$[o]$ = `UNALLOCATED` before the call and `Size`$[o] \geq n$. The procedure modifies `Alloc`$[o]$ to be `ALLOCATED`. The procedure call `free(p)` for freeing a memory buffer whose address is contained in `p` requires that `Alloc[Obj(p)] == ALLOCATED` and `Off(p) == 0` and updates `Alloc[Obj(p)]` to `FREED`. The full specification of `malloc` and `free` is given in a detailed report [8].

HAVOC takes an annotated C program and translates it into a BoogiePL [11] program. BoogiePL has been designed to be an intermediate language for program verification tools that use automated theorem provers. This language is simple and has well-defined semantics. The operational semantics of C, as interpreted by HAVOC, is best understood by comparing a C program with its BoogiePL translation. Figure 3 shows two procedures, `create` and `init`, on the left and their translations on the right. The example uses the C struct type `DATA`.

Note that variables of both static type `int` and `int*` in C are translated uniformly as variables of type `ptr`. The translation of the first argument `d->y` of the call to `init` shows that we treat field accesses and pointer arithmetic uniformly. Since the field `y` is at an offset 4 in `DATA`, we treat `d->y` as `d+4`. The translation uses the function `PLUS` to model pointer arithmetic and the function `LT` to model arithmetic comparison operations on the type `ptr`. The definitions of these functions are also given in the detailed report [8].

The example also shows how we handle the `&` operator. In the procedure `create`, the address of the local variable `a` is passed as an out-parameter to the procedure `init`. Our translation handles this case by allocating `a` on the heap. Note that our translator allocates a static variable on the heap only if the program takes the address of that variable. For example, there is no heap allocation for the local variable `i` in the procedure `init`. To prevent access to the heap-allocated object corresponding to a local variable of a procedure, it is freed at the end of the procedure.

4 Reachability and Pointer Arithmetic

We now give the formal definition of our new reachability predicate in terms of the operational semantics of C as interpreted by HAVOC. As in our previous work [17], we define the reachability predicate on well-founded heaps. Let the heap be represented by the function $\mathtt{Mem} : \mathtt{ptr} \rightarrow \mathtt{ptr}$ and let $\mathtt{BS} \subseteq \mathtt{ptr}$ be a set of pointers. We define a sequence of functions $f^i : \mathtt{int} \times \mathtt{ptr} \rightarrow \mathtt{ptr}$ for $i \geq 0$ as follows: for all $n \in \mathtt{int}$ and $u \in \mathtt{ptr}$, we have $f^0(n,u) = u$ and $f^{i+1}(n,u) = \mathtt{Mem}[f^i(n,u) + n]$ for all $i > 0$. Then `Mem` is *well-founded* with respect to the set of *blocking pointers* `BS` and *offset* n if for all $u \in \mathtt{ptr}$, there is $i > 0$ such that $f^i(n,u) \in \mathtt{BS}$. If a heap is well-founded with respect to `BS` and n, then the function idx_n maps a pointer u to the least $i > 0$ such that $f^i(n,u) \in \mathtt{BS}$. Using these concepts, we now define for each $n \in \mathtt{int}$, a predicate $\mathtt{R}_n \subseteq \mathtt{ptr} \times \mathtt{ptr}$ and a function $\mathtt{B}_n : \mathtt{ptr} \rightarrow \mathtt{ptr}$.

$$\begin{aligned} \mathtt{R}_n[u,v] &\equiv \exists i.\ 0 \leq i < idx_n(u) \wedge v = f^i(n,u) \\ \mathtt{B}_n[u] &\equiv f^{idx_n(u)}(n,u) \end{aligned}$$

Suppose a program performs the operation `Mem[x] := y` to update the heap. Then HAVOC performs the *most precise* update to the predicate $\mathtt{R}_n$ and the function $\mathtt{B}_n$ by automatically inserting the following code just before the operation.

$$
\begin{array}{rrl}
n \in & \mathit{int} & \\
e \in & \mathit{Expr} ::= & n \mid \mathtt{x} \mid \mathtt{addr(x)} \mid e \;\mathtt{+}\; e \mid e \;\mathtt{-}\; e \mid \mathtt{deref}(e) \mid \mathtt{block}(e, n) \mid \\
 & & \mathtt{old(x)} \mid \mathtt{old\_deref}(e) \mid \mathtt{old\_block}(e, n) \\
S \in & \mathit{Set} ::= & \{e\} \mid \mathtt{BS} \mid \mathtt{list}(e, n) \mid \mathtt{old\_list}(e, n) \mid \mathtt{array}(e, n, e) \\
\phi \in & \mathit{Formula} ::= & \mathtt{alloc}(e) \mid \mathtt{old\_alloc}(e) \mid \mathtt{Obj}(e) \;\mathtt{==}\; \mathtt{Obj}(e) \mid \mathtt{Off}(e) \;\mathtt{<}\; \mathtt{Off}(e) \mid \\
 & & \mathtt{in}(e, S) \mid \mathtt{!}\; \phi \mid \phi \;\mathtt{\&\&}\; \phi \mid \mathtt{forall}(\mathtt{x}, S, \phi) \\
C \in & \mathit{CmpdSet} ::= & S \mid \mathtt{incr}(C, n) \mid \mathtt{decr}(C, n) \mid \mathtt{deref}(C) \mid \mathtt{old\_deref}(C) \\
 & & \mathtt{union}(C, C) \mid \mathtt{intersection}(C, C) \mid \mathtt{difference}(C, C)
\end{array}
$$

Fig. 4. Annotation language

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{assert}(\mathtt{R}_n[\mathtt{y}, \mathtt{x} - n] \Rightarrow \mathtt{BS}[\mathtt{y}])} \\
\mathtt{B}_n & \mathtt{:=} & \lambda\ \mathtt{u} : \mathtt{ptr}.\ \ \mathtt{R}_n[\mathtt{u}, \mathtt{x} - n]?\ (\mathtt{BS}[\mathtt{y}]\ ?\ \mathtt{y}\ :\ \mathtt{B}_n[\mathtt{y}]) :\ \mathtt{B}_n[\mathtt{u}] \\
\mathtt{R}_n & \mathtt{:=} & \lambda\ \mathtt{u}, \mathtt{v} : \mathtt{ptr}. \\
 & & \mathtt{R}_n[\mathtt{u}, \mathtt{x} - n] \\
 & & ?\ (\mathtt{R}_n[\mathtt{u}, \mathtt{v}] \wedge \neg \mathtt{R}_n[\mathtt{x} - n, \mathtt{v}]) \ \ \vee\ \ \mathtt{v} = \mathtt{x} - n \ \ \vee\ \ (\neg \mathtt{BS}[\mathtt{y}] \wedge \mathtt{R}_n[\mathtt{y}, \mathtt{v}]) \\
 & & :\ \mathtt{R}_n[\mathtt{u}, \mathtt{v}]
\end{array}
$$

The assertion enforces that the heap stays well-founded with respect to the blocking set BS and the offset n. The value of $\mathtt{B}_n[\mathtt{u}]$ is updated only if $\mathtt{x} - n$ is reachable from u and otherwise remains unchanged. Similarly, the value of $\mathtt{R}_n[\mathtt{u}, \mathtt{v}]$ is updated only if $\mathtt{x} - n$ is reachable from u and otherwise remains unchanged. These updates are generalizations of the updates provided in our earlier paper [17] to account for pointer arithmetic.

We note that the ability to provide such updates as described above guarantees that if a program's assertions —preconditions, postconditions, and loop invariants— are quantifier-free, then its verification condition is quantifier-free as well. This property is valuable because the handling of quantifiers is typically the least complete and efficient aspect of all theorem provers that combine first-order reasoning with arithmetic.

5 Annotation Language

Our annotation language has three components: basic expressions that evaluate to pointers, set expressions that evaluate to sets of pointers, and formulas that evaluate to boolean values. The syntax for these expressions is given in Figure 4.

The set of basic expressions is captured by *Expr*. The expression addr(x) represents the address of the variable x. The expression x represents the value of x in the post-state and old(x) refers to the value of x in the pre-state of the procedure. The expressions $\mathtt{deref}(e)$ and $\mathtt{old\_deref}(e)$ refer to the value stored in memory at the address e in the post-state and pre-state, respectively. The expressions $\mathtt{block}(e, n)$ and $\mathtt{old\_block}(e, n)$ represent $\mathtt{B}_n[e]$ in the post-state and pre-state of the procedure, respectively.

The set expressions are divided into the basic set expressions in *Set* and the compound set expressions in *CmpdSet*. The expression $\mathtt{array}(e_1, n, e_2)$ refers to the set of pointers $\{e_1, e_1 + n, e_1 + 2 * n, \ldots, e_1 + \mathtt{Off}(e_2) * n\}$. The expressions

```
// translation of requires φ
requires ⟦φ⟧

// translation of ensures ψ
ensures ⟦ψ⟧

// translation of modifies C
modifies Mem
ensures forall x:ptr::      old(Alloc)[Obj(x)] == UNALLOCATED ||
                            old(⟦in(x, C)⟧) ||
                            old(Mem)[x] == Mem[x]
modifies R_n
ensures forall x:ptr::      old(Alloc)[Obj(x)] == UNALLOCATED ||
                            exists y:ptr:: old(R_n)[x,y] && old(⟦in(y + n, C)⟧) ||
                            forall z:ptr:: old(R_n)[x,z] == R_n[x,z]
modifies B_n
ensures forall x:ptr::      old(Alloc)[Obj(x)] == UNALLOCATED ||
                            exists y:ptr:: old(R_n)[x,y] && old(⟦in(y + n, C)⟧) ||
                            old(B_n)[x] == B_n[x]

// translation of frees D
modifies Alloc
ensures forall o:ref::      old(Alloc)[o] == UNALLOCATED ||
                            (old(⟦in(Ptr(o, 0), D)⟧) &&
                            Alloc[Obj(x)] != UNALLOCATED) ||
                            Alloc[x] == old(Alloc)[x]
```

Fig. 5. Translation of `requires` ϕ, `ensures` ψ, `modifies` C, and `frees` D

`list`(e, n) and `old_list`(e, n) represent the list of pointers described by the reachability predicate $\mathtt{R}_n$ in the post-state and pre-state, respectively. The compound set expressions include `incr`(C, n) and `decr`(C, n) which respectively increment and decrement each element of C by n, and `deref`(C) and `old_deref`(C) which read the contents of memory at the members of C in the post-state and pre-state, respectively. The expressions `union`(C, C), `intersection`(C, C), and `difference`(C, C) provide the basic set-theoretic operations.

HAVOC is designed to be a modular verifier. Consequently, we allow each procedure to be annotated by four possible specifications, `requires` ϕ, `ensures` ψ, `modifies` C, and `frees` D, where $\phi, \psi \in$ *Formula* and $C, D \in$ *CmpdSet*. The default value for ϕ and ψ is *true*, and for C and D is $\emptyset$. The translation of these specifications is given in Figure 5. The translation refers to the translation function $[\![\circ]\!]$, which is defined in the Appendix of the detailed report [8]. We also allow each loop to be annotated with a formula representing its invariant.

In Figure 5, the translation of `requires` ϕ and `ensures` ψ is obtained in a straightforward fashion by applying the translation function $[\![\circ]\!]$ to ϕ and ψ respectively. Then, there are four pairs of modifies and ensures clauses. The translation of `modifies` C is captured by the first three pairs and the translation of `frees` D is captured by the fourth pair. Our novel use of set expressions in these specifications results in a significant reduction in the annotation overhead at the C level.

The first pair of `modifies` and `ensures` clauses in Figure 5 states that the contents of `Mem` remains unchanged at each pointer that is allocated and not a member of C in the pre-state of the procedure. The second pair is parameterized by an integer offset n and specifies the update of $\mathtt{R}_n$. Similarly, the third pair specifies the update of $\mathtt{B}_n$. Based on the set C provided by the programmer in the `modifies` clause, one such pair is automatically generated for each offset n of interest. The postcondition corresponding to $\mathtt{R}_n$ says that if the set of pointers reachable from any pointer `x` is disjoint from the set $\mathtt{decr}(C, n)$, then that set remains unchanged by the execution of the procedure. The postcondition corresponding to $\mathtt{B}_n$ says that if the set of pointers reachable from any pointer `x` is disjoint from the set $\mathtt{decr}(C, n)$, then $\mathtt{B}_n[\mathtt{x}]$ remains unchanged by the execution of the procedure. These two postconditions are guaranteed by our semantics of reachability and the semantics of the `modifies` clause. Consequently, Havoc only uses these postconditions at call sites and does not attempt to verify them. The set D in the annotation `frees` D is expected to contain only pointers with offset 0. Then, the fourth pair states that the contents of `Alloc` remain unchanged at each object that is allocated and is such that a pointer to the beginning of that object is not a member of D in the pre-state of the procedure.

6 Implementation

We have developed Havoc, a prototype tool for verifying C programs annotated with specifications in our annotation language. We use the ESP [10] infrastructure to construct the control flow graph and parse the annotations. Havoc translates an annotated C program into an annotated BoogiePL program as described in Section 3. The Boogie verifier generates a verification condition (VC) from the BoogiePL description, which implies the partial correctness of the BoogiePL program. The VC generation in Boogie is performed using a variation [3] of the standard *weakest precondition* transformer [13]. The resulting VC is checked for validity using the Simplify theorem prover [12].

6.1 Proving Verification Conditions

The verification condition generated is a formula in first-order logic with equality, augmented with the following theories:

1. The theory of integer linear arithmetic with symbols $+, \leq$ and constants $\ldots, -1, 0, 1, 2, \ldots$.
2. The theory of arrays with the `select` and `update` symbols [21].
3. The theory of *pairs*, consisting of the symbols for the pair constructor `Ptr`, and the selector functions `Obj` and `Off`.
4. The theory of the new reachability predicate, consisting of the symbols $\mathtt{R}_n$, $\mathtt{B}_n$, `BS` and `Mem`.

$$\begin{aligned}\forall u:\texttt{ptr}.\;\; & u = \texttt{Ptr}(\texttt{Obj}(u), \texttt{Off}(u))\\ \forall x:\texttt{ref}, i:\texttt{int}.\;\; & x = \texttt{Obj}(\texttt{Ptr}(x,i))\\ \forall x:\texttt{ref}, i:\texttt{int}.\;\; & i = \texttt{Off}(\texttt{Ptr}(x,i))\end{aligned}$$

Fig. 6. Axioms for the theory of pairs

To verify the verification conditions, the Simplify theorem prover requires axioms about the theory of pairs and the theory of reachability. The axioms for the theory of pairs are fairly intuitive and are given in Figure 6. The axioms for the theory of reachability are given in Figure 7. Note that the symbol $+$ in Figure 7 is the addition operation on pointers. We have overloaded $+$ for ease of exposition. The first axiom defines that $\texttt{R}_n[u,v]$ is true if and only if either $v = u$, or the pointer $\texttt{Mem}[u+n]$ is not a blocking pointer in $\texttt{BS}$ and $\texttt{R}_n[\texttt{Mem}[u+n], v]$ is true. The second axiom similarly defines $\texttt{B}_n[u]$. We call these two axioms the base axioms of reachability because they attempt to capture the recursive definitions of $\texttt{R}_n$ and $\texttt{B}_n$.

$$\texttt{R}_n[u,v] \Leftrightarrow (v = u \vee (\neg\texttt{BS}[\texttt{Mem}[u+n]] \wedge \texttt{R}_n[\texttt{Mem}[u+n], v]))$$
$$v = \texttt{B}_n[u] \Leftrightarrow (\texttt{BS}[\texttt{Mem}[u+n]]\ ?\ v = \texttt{Mem}[u+n] : v = \texttt{B}_n[\texttt{Mem}[u+n]])$$

$$\begin{aligned}\texttt{R}_n[u,v] \wedge \texttt{R}_n[v,w] &\Rightarrow \texttt{R}_n[u,w]\\ \texttt{BS}[u] \wedge \texttt{R}_n[v,u] &\Rightarrow u = v\\ \texttt{R}_n[u,v] &\Rightarrow \texttt{B}_n[u] = \texttt{B}_n[v]\\ u = \texttt{Mem}[u+n] &\Rightarrow \texttt{BS}[u]\\ \neg\texttt{BS}[\texttt{Mem}[u+n]] &\Rightarrow \texttt{R}_n[\texttt{Mem}[u+n]] = \texttt{R}_n[u] \setminus \{u\}\end{aligned}$$

Fig. 7. Derived axioms for the theory of reachability predicate. The variables u, v and w are implicitly universally quantified.

It is well known that the reachability predicate (ours as well as the classic one) cannot be expressed in first-order logic [6]. Hence, similar to our previous work [17], we provide a sound but (necessarily) incomplete axiomatization of the theory by providing a set of derived axioms following the base axioms in Figure 7. Since the definitions of $\texttt{R}_n$ and $\texttt{B}_n$ are well-founded, these derived axioms can be proved from the base axioms using well-founded induction. The derived axioms are subtle generalizations of similar axioms presented for well-founded lists without pointers [17]. They have sufficed for all the examples in this paper.

7 Evaluation

In this section, we describe our experience applying Havoc to a set of examples. These examples illustrate the use of pointer arithmetic, internal pointers, arrays, and linked lists in C programs. For each of these examples, we prove a variety of partial correctness properties, including the absence of null dereferences.

Figure 8 lists the examples considered in this paper. `iterate` is the example from Figure 2 in Section 2. `iterate_acyclic` and `array_iterate` are versions

Example	Time(s)
`iterate`	1.8
`iterate_acyclic`	1.7
`array_iterate`	1.4
`slist_add`	1.5
`reverse_acyclic`	2.0

Example	Time(s)
`array_free`	2.5
`slist_sorted_insert`	16.43
`dlist_add`	38.9
`dlist_remove`	45.4
`allocator`	901.8

Fig. 8. Results of assertion checking. Simplify was used as the theorem prover. The experiments were conducted on a 3.2GHz, 2GB machine running Windows XP.

of `iterate` for an acyclic list and an array, respectively. `reverse_acyclic` performs in-place reversal of an acyclic singly-linked list; we verify that the output list is acyclic and contains the same set of pointers as the input list. The example `slist_add` adds a node to an acyclic singly-linked list. `dlist_add` and `dlist_remove` are the insertion and deletion routines for cyclic doubly-linked lists used in the Windows kernel. The examples using doubly-linked lists require the use of $\mathtt{R}_0$ and $\mathtt{R}_4$ to specify the lists reachable through the `Flink` and `Blink` fields of the `LIST_ENTRY` structure. The example `slist_sorted_insert` inserts a node into a sorted (by the data field) linked list; we verify that the output list is sorted. This example illustrates the use of arithmetic reasoning (using $\leq$) on the data fields. The example `array_free` takes as input an array `a` of pointers, and iterates over the array to free the pointers that are not `null`. We check that an object is freed at most once. To verify this property, we needed to express the invariant that if i is distinct from j, then the pointers $\mathtt{a}[i]$ and $\mathtt{a}[j]$ are aliased only if they both point to `null`.

The final example `allocator` is a low-level storage allocator that closely resembles the `malloc_firstfit_acyclic` example described by Calcagno et al. [7]. The allocator maintains a list of free blocks within a single large object; each node in the list maintains a pointer to the next element of the list and the size of the free block in the node. Allocation of a block may result in either removing a node (if the entire free block at the node is returned) from the list, or readjusting the size of the free block (in case only a chunk of the free block is returned). We check two main postconditions: (i) the allocated block (when a non null pointer is returned) is a portion of some free block in the input list, and (ii) the free blocks of the output list do not overlap. This example required the use of $\mathtt{R}_0$ to specify the list of free blocks.

Figure 8 gives the running times taken by Simplify to discharge the verification conditions. The examples involving singly-linked lists and arrays take only a few seconds. The examples involving doubly-linked lists take much longer because they make heavy use of quantifiers to express the invariant that connects the forward-going and backward-going links in a doubly-linked list. The `allocator` example makes heavy use of arithmetic as well as quantifiers, and therefore takes the longest to verify.

Interestingly, Havoc revealed a bug in our implementation of the `allocator`. This bug was caused by an interaction between pointer casting and pointer arithmetic. Instead of the following correct code

```
return ((unsigned int) cursor) + sizeof(RegionHeader);
```

we had written the following incorrect code

```
return (unsigned int) (cursor + sizeof(RegionHeader));
```

Note that the two are different because the size of `RegionHeader`, the static type of `cursor`, is different from the size of `unsigned int`. We believe that such mistakes are common when dealing with low-level C code, and our tool can provide great value in debugging such programs.

8 Conclusions and Future Work

In this work, we introduced a new reachability predicate suitable for reasoning about data structures in low-level systems software. Our reachability predicate is designed to handle internal pointers and pointer arithmetic on object fields. It is a generalization of the classic reachability predicate used in existing verification tools. We have designed an annotation language for C programs that allows concise specification of properties of lists and arrays. We have also developed Havoc, a verifier for C programs annotated with assertions in our specification language.

We believe that Havoc is a good foundation for building powerful safety checkers for systems software based on automated first-order theorem proving. We are currently working to extend Havoc with techniques for inference and abstraction to enable its use on realistic code bases inside Windows.

Acknowledgements. Our formalization of the C memory model has been deeply influenced by discussions with Madan Musuvathi. We are grateful to Stephen Adams, Henning Rohde, Jason Yang and Zhe Yang for their help with the ESP infrastructure. Rustan Leino answered numerous questions about Simplify and Boogie. Finally, we thank Tom Ball and Rustan Leino for providing valuable feedback on the paper.

References

1. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *Verification, Model checking, and Abstract Interpretation (VMCAI '05)*, LNCS 3385, pages 164–180, 2005.
2. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
3. M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.
4. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS 3362, pages 49–69, 2005.

5. J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI '06)*, LNCS 3855, pages 207–221, 2006.
6. E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem.* Springer-Verlag, 1997.
7. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Static Analysis Symposium (SAS '06)*, LNCS 4134, pages 182–203, 2006.
8. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. Technical Report MSR-TR-2006-154, Microsoft Research, 2006.
9. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI–C programs using SAT. *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.
10. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI '02)*, pages 57–68, 2002.
11. R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
12. D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, HPL-2003-148, 2003.
13. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
14. J. Filliâtre and C. Marché. Multi-prover verification of C programs. In *International Conference on Formal Engineering Methods (ICFEM '04)*, LNCS 3308, pages 15–29, 2004.
15. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages (POPL '02)*, pages 58–70, 2002.
17. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Principles of Programming Languages (POPL '06)*, pages 115–126, 2006.
18. T. Lev-Ami, N. Immerman, T. W. Reps, S. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Conference on Automated Deduction (CADE '05)*, LNCS 3632, pages 99–115, 2005.
19. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS '00)*, LNCS 1824, pages 280–301, 2000.
20. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Computer-Aided Verification (CAV '05)*, LNCS 3576, pages 476–490, 2005.
21. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.
22. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, 1998.
23. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Principles of Programming Languages (POPL '05)*, pages 351–363, 2005.