Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Low-Level Memory Model and an Accompanying Reachability Predicate

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

[1] University of California, Berkeley, e-mail: shaunakc@cs.berkeley.edu
[2] Microsoft Research, e-mail: shuvendu,qadeer@microsoft.com
[3] University of British Columbia, e-mail: zrakamar@cs.ubc.ca

Received: date / Revised version: date

Abstract. Reasoning about program heap, especially if it involves handling unbounded, dynamically heap-allocated data structures such as linked lists and arrays, is challenging. Furthermore, sound analysis that precisely models heap becomes significantly more challenging in the presence of low-level pointer manipulation that is prevalent in systems software.

The *reachability* predicate has already proved to be useful for reasoning about the heap in type-safe languages where memory is manipulated by dereferencing object fields. In this paper, we present a memory model suitable for reasoning about low-level pointer operations that is accompanied by a 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 our 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 [3, 22, 13, 37, 12, 1]. These tools can scale to large software systems; however, this scalability is achieved at the price of precision. Heap and 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. This is a significant drawback since studies have shown that many of the systems software code bugs and reported failures with high impact on availability are still related to memory management [11, 28, 36].

Because of the vast size of available memory in todays computer systems, faithfully representing each memory allocation and access in a static verifier is not going to scale. Therefore, verification tools rely on abstract memory models that trade precision for scalability, and in turn, they define the operational semantics of their programming language with respect to the chosen memory model. We present in this paper a memory model that is precise enough to capture most of the low-level pointer operations, and yet abstracts enough details to enable verification to scale. We also give the respective operational semantics of the C language.

The *reachability predicate* [30] 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# [34, 26, 2, 23, 8]. This predicate is inadequate for reasoning about low-level software, which com-

monly 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 3.

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, in addition to the memory model, we introduce in this paper the accompanying reachability predicate suitable for verifying low-level programs written in C. We describe how to automatically compute the precise update for the described predicate and a method for reasoning about it using automated first-order theorem provers. With recent noticable performance improvements of theorem provers, we believe that the introduced reachablity predicate could be used for suitably extending theorem-prover-based static checkers for low-level programs, making them therefore more precise.

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 fOr C programs). We report on our preliminary encouraging experience with Havoc on a set of illustrative C programs.

2 Related work

Havoc is a static assertion checker for annotated C programs in the same style that ESC/Java [20] is a static checker for annotated Java programs, and Boogie [4] is a static checker for Spec# [6] programs.[1] 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 SATABS [12], SATURN [37], Calysto [1], and VerifiedC [35]. Traditionally, such tools either overapproximate unbounded data structures, which in turn leads to a lot of false warnings, or simply treat them unsoundly as bounded data structures (often with only one element), which causes real bugs to be missed. The work of McPeak and Necula [29] 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 [27] is a verification tool based on abstract interpretation using 3-valued logic [34]. It provides a general specification logic combining first-order logic with reachability. Recently, an axiomatization of reachability in first-order logic was also added to the system [26]. However, TVLA has mostly been applied to Java programs and, to our knowledge, cannot handle the interaction of reachability with pointer arithmetic.

Caduceus [19] 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 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.

There are many approaches to checking of heap-manipulating programs that employ the idea of local reasoning based on the frame rule of separation logic [33, 32]. The frame rule in this context shows that it is sound to look at only a fragment of the input heap when analyzing each program instruction. This important property has been used to speed up and increase scalability of many analyses [38, 7, 21, 18]. While many of these approaches infer loop invariants of list-manipulating programs automatically, they don't handle low-level features of C programs that are prevalent in systems code, such as pointer arithmetic, at all. Havoc, on the other hand, supports reasoning about linked data structures in the presence of low-level pointer manipulations, while loop invariants currently have to be provided manually.

Calcagno et al. have used separation logic to reason about memory safety and absence of memory leaks in low-level code [9]. 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.

3 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

[1] Spec# is an extension of C# that provides specification primitives for writing method pre- and postconditions and object invariants.

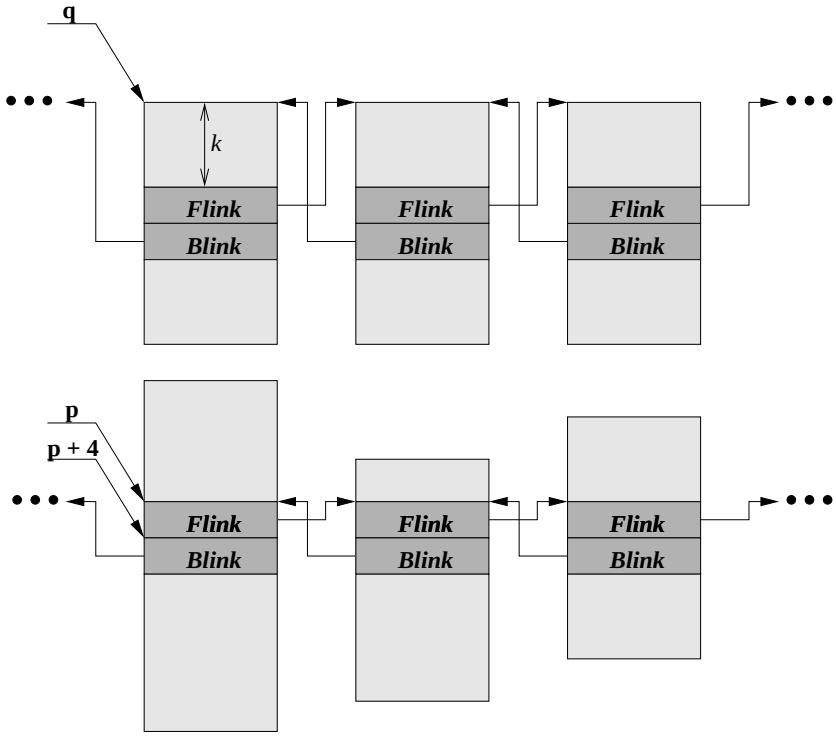

Fig. 1. Doubly-linked lists in Java and C.

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 *object references*. For example, $\mathtt{R}_{\mathtt{Flink}}(a, b)$ holds for object references 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[2].

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 object references*. Therefore, we introduce an integer-indexed set of binary reachability predicates: for each integer n, the

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

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 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

$$p, *(p + 4), *(*(p + 4) + 4), \ldots .$$

Our 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

$$q, *(q + k + 4), *(*(q + k + 4) + k + 4), \ldots .$$

3.1 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. For an example of such a heap structure see Figure 3. The procedure `list_iterate` iterates over this list, setting the `data` field of each list element to 42.

Except for verifying the safety of each memory access in `list_iterate`, we would also 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 [23], 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 *blocker* 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

$$\mathtt{g.a.Flink}, *(\mathtt{g.a.Flink} + 0), \ldots$$

upto but excluding the pointer $\mathtt{B}_0(\mathtt{g.a.Flink})$. In Havoc, we represent a pointer as a pair comprised of an object reference and an integer offset into the object, and the program memory is a map from pointers to pointers. Therefore, 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 reference (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.

```
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. Motivating example.

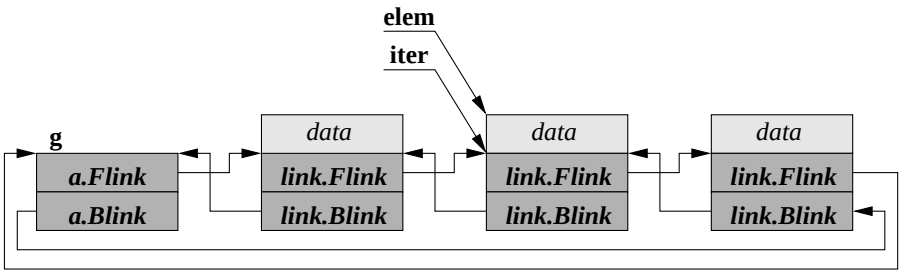

Fig. 3. Example of an input structure for our motivating example.

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.

4 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` × `int` of pointers. In Havoc, each variable from 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` × `int` → `ptr` constructs a pointer value from its components. The selector functions `Obj` : `ptr` → `ref` and `Off` : `ptr` → `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 Figure 4.

Currently, Havoc's memory model understands only word-aligned memory accesses. For example, writing a 4-byte integer value to the memory location `Ptr`$(o, 20)$ is not going to affect the value stored at the memory location `Ptr`$(o, 21)$. Similarly, reading a byte from the memory location `Ptr`$(o, 21)$ is not going to return the

```
procedure malloc(n: ptr) returns new:ptr
requires Obj(n) == null && 0 < Off(n)
modifies Alloc
ensures old(Alloc)[Obj(new)] == UNALLOCATED
ensures Alloc[Obj(new)] == ALLOCATED
ensures Off(new) == 0
ensures Off(n) <= Size(Obj(new))
ensures (forall o:ref :: o == Obj(new) ||
                         old(Alloc)[o] == Alloc[o])
ensures (forall i: int ::
           Obj(Mem[Obj(new),i] ) == null)
ensures (forall i:int :: BS[Obj(new),i])


procedure free(p: ptr)
requires Alloc(Obj(p)) == ALLOCATED && Off(p) == 0
modifies Alloc
ensures alloc[Obj(p)] != UNALLOCATED
ensures alloc[Obj(p)] != ALLOCATED
ensures (forall o:ref :: o == Obj(p) ||
                         old(Alloc)[o] == Alloc[o])
```

Fig. 4. Specification of procedures `malloc` and `free` that are used to model memory allocation.

```
function PLUS(ptr, ptr) returns ptr;
axiom (forall x,y:ptr ::
  (Obj(x) == null ==>
   PLUS(x,y) == Ptr(Obj(y), Off(x)+Off(y))) &&
  (Obj(y) == null ==>
   PLUS(x,y) == Ptr(Obj(x), Off(x)+Off(y))) &&
  (Obj(x) != null && Obj(y) != null ==>
   Obj(PLUS(x,y)) == null))

function MINUS(ptr, ptr) returns ptr;
axiom (forall x,y:ptr ::
  (Obj(y) == null ==>
   MINUS(x,y) == Ptr(Obj(x), Off(x)-Off(y))) &&
  (Obj(y) != null && Obj(x) == Obj(y) ==>
   MINUS(x,y) == Ptr(null, Off(x)-Off(y))) &&
  (Obj(y) != null && Obj(x) != Obj(y) ==>
   Obj(MINUS(x,y)) == null))

function LT(ptr, ptr) returns bool;
axiom (forall x,y:ptr :: LT(x,y) <==>
                         Off(MINUS(y,x)) > 0)
```

Fig. 5. Specification of functions `PLUS`, `MINUS`, and `LT` that are used to model arithmetic and comparison operations on the type `ptr`.

```
C1   typedef struct { int x; int y[10]; } DATA;

C2   DATA *create() {
C3     int a;
C4
C5     DATA *d =
C6       (DATA *) malloc(sizeof(DATA));
C7     init(d->y, 10, &a);
C8
C9     d->x = a;
C10
C11    return d;
C12  }

C13  void init(int *in, int size,
C14            int *out) {
C15    int i;
C16    i = 0;
C17    while (i < size) {
C18      in[i] = i;
C19      *out = *out + i;
C20      i++;
C21    }
C22  }
```

```
B1   procedure create() returns d:ptr {
B2     var a:ptr;
B3     call a := malloc(Ptr(null,4));
B4     call d := malloc(Ptr(null,44));
B5
B6     call init(PLUS(d, Ptr(null,4)),
B7               Ptr(null,10), a);
B8     Mem[PLUS(d, Ptr(null,0))] := Mem[a];
B9     call free(a);
B10
B11  }

B12  procedure init(in:ptr, size:ptr,
B13                 out:ptr) {
B14    var i:ptr;
B15    i := Ptr(null,0);
B16    while (LT(i, size)) {
B17      Mem[PLUS(in, Ptr(null,Off(i)*4))] := i;
B18      Mem[out] := PLUS(Mem[out], i);
B19      i := PLUS(i, Ptr(null,1));
B20    }
B21  }
```

Fig. 6. Translation of a simple C example on the left into the slightly simplified (to make it more readable) BoogiePL code on the right.

second byte of the integer that was written to $\texttt{Ptr}(o, 20)$, but rather an unconstrained value. We are planning to address this deficiency in the future.

Havoc takes an annotated C program and translates it into a BoogiePL [15] 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 6 shows two procedures, `create` and `init`, on the left and their translations on the right. The example uses the C struct type `DATA` defined on line C1.

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` on line C7 shows that we treat field accesses and pointer arithmetic uniformly. Since the array field `y` is at an offset 4 in `DATA`, we treat `d->y` as `d+4` on line B6. Array accesses are also translated using pointer arithmetic. For instance, array access on line C18 is translated as `in+i*4` on line B17 since the size of each array element is 4. The translation uses the function `PLUS` to model pointer arithmetic and the function `LT` on line B16 to model arithmetic comparison operations on the type `ptr`. The definitions of these functions are given in Figure 5.

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 on line B3. Then, the C expression `&a` on line C7 is translated as `a` on line B7, while the expression `a` on line C9 is translated as the heap access `Mem[a]` on line B8. Note that our translator allocates a static variable on the heap only if the program takes the address of that variable or if the type of the variable is a structure or union. We allocate all structures on the heap. 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. Therefore, the translation freed the local variable `a` on line B9.

5 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 [23], we define the reachability predicate on well-founded heaps. Let the heap be represented by the function $\texttt{Mem} : \texttt{ptr} \rightarrow \texttt{ptr}$ and let $\texttt{BS} \subseteq \texttt{ptr}$ be a set of pointers. We define a sequence of functions $f^i : \texttt{int} \times \texttt{ptr} \rightarrow \texttt{ptr}$ for $i \geq 0$ as follows: for all $n \in \texttt{int}$ and $u \in \texttt{ptr}$, we have $f^0(n, u) = u$ and $f^{i+1}(n, u) = \texttt{Mem}[f^i(n, u) + n]$ for all $i > 0$. Then `Mem` is *well-founded* with respect to the set of *blocking pointers* `BS` and *offset* n if for all $u \in \texttt{ptr}$, there is $i > 0$ such that $f^i(n, u) \in \texttt{BS}$. 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 $f^i(n, u) \in \texttt{BS}$. Using these concepts, we now define for each $n \in \texttt{int}$, a predicate $\texttt{R}_n \subseteq \texttt{ptr} \times \texttt{ptr}$ and a function $\texttt{B}_n : \texttt{ptr} \rightarrow \texttt{ptr}$.

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

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

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

The assertion enforces that the heap stays well-founded with respect to the blocking set `BS` and the offset n. It is inserted for each offset n of interest in the program. The value of $\texttt{B}_n[\texttt{u}]$ is updated only if $\texttt{x} - n$ is reachable from `u` and otherwise remains unchanged. Similarly, the value of $\texttt{R}_n[\texttt{u}, \texttt{v}]$ is updated only if $\texttt{x} - n$ is reachable from `u` and otherwise remains unchanged. These updates are generalizations of the updates provided in our earlier paper [23] 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.

6 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 7.

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

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

Fig. 7. Core annotation language. Syntactic sugar, that allows writing many common idioms succinctly and more conviniently, is added on top of it.

$$
\begin{array}{rl}
[\![n]\!] & = \texttt{Ptr(null,}n\texttt{)} \\
[\![\texttt{x}]\!] & = \begin{cases} \texttt{Mem[bpl\_x]}, & \text{if address of x is taken} \\ \texttt{bpl\_x}, & \text{otherwise} \end{cases} \\
[\![\texttt{addr(x)}]\!] & = \texttt{bpl\_x} \\
[\![e_1 \texttt{ + } e_2]\!] & = \texttt{Ptr(Obj(}[\![e_1]\!]\texttt{),Off(}[\![e_1]\!]\texttt{) + Off(}[\![e_2]\!]\texttt{))} \\
[\![e_1 \texttt{ - } e_2]\!] & = \texttt{Ptr(null,Off(}[\![e_1]\!]\texttt{) - Off(}[\![e_2]\!]\texttt{))} \\
[\![\texttt{deref}(e)]\!] & = \texttt{Mem[}[\![e]\!]\texttt{]} \\
[\![\texttt{block}(e,n)]\!] & = \texttt{B}_n\texttt{[}[\![e]\!]\texttt{]} \\
[\![\texttt{old(x)}]\!] & = \texttt{old(}[\![\texttt{x}]\!]\texttt{)} \\
[\![\texttt{old\_deref}(e)]\!] & = \texttt{old(Mem)[}[\![e]\!]\texttt{]} \\
[\![\texttt{old\_block}(e,n)]\!] & = \texttt{old(B}_n\texttt{)[}[\![e]\!]\texttt{]} \\
 & \\
[\![\texttt{alloc}(e)]\!] & = \texttt{Alloc[Obj(}[\![e]\!]\texttt{)] == ALLOCATED} \\
[\![\texttt{old\_alloc}(e)]\!] & = \texttt{old(Alloc)[Obj(}[\![e]\!]\texttt{)] == ALLOCATED} \\
[\![\texttt{Obj}(e_1) \texttt{ == Obj}(e_2)]\!] & = \texttt{Obj(}[\![e_1]\!]\texttt{) == Obj(}[\![e_2]\!]\texttt{)} \\
[\![\texttt{Off}(e_1) \texttt{ < Off}(e_2)]\!] & = \texttt{Off(}[\![e_1]\!]\texttt{) < Off(}[\![e_2]\!]\texttt{)} \\
[\![\texttt{! } \phi]\!] & = \texttt{! } [\![\phi]\!] \\
[\![\phi_1 \texttt{ \&\& } \phi_2]\!] & = [\![\phi_1]\!] \texttt{ \&\& } [\![\phi_2]\!] \\
[\![\texttt{forall}(\texttt{x},S,\phi)]\!] & = \texttt{(forall x : ptr :: ! } [\![\texttt{in(x,}S\texttt{)}]\!] \texttt{ || } [\![\phi]\!]\texttt{)} \\
 & \\
[\![\texttt{in}(e,\{e'\})]\!] & = [\![e]\!] \texttt{ == } [\![e']\!] \\
[\![\texttt{in}(e,\texttt{BS})]\!] & = \texttt{BS}[\![e]\!]\texttt{]} \\
[\![\texttt{in}(e,\texttt{list}(e',n))]\!] & = \texttt{R}_n\texttt{[}[\![e']\!]\texttt{,}[\![e]\!]\texttt{]} \\
[\![\texttt{in}(e,\texttt{old\_list}(e',n))]\!] & = \texttt{old(R}_n\texttt{)[}[\![e']\!]\texttt{,}[\![e]\!]\texttt{]} \\
[\![\texttt{in}(e,\texttt{array}(e_1,n,e_2))]\!] & = \texttt{(exists i : int :: 0 <= i \&\& i < Off(}[\![e_2]\!]\texttt{) \&\& } [\![e]\!] \texttt{ == } [\![e_1]\!] \texttt{ + } n\texttt{*i)} \\
[\![\texttt{in}(e,\texttt{incr}(C,n))]\!] & = [\![\texttt{in}(e \texttt{ - } n, C)]\!] \\
[\![\texttt{in}(e,\texttt{decr}(C,n))]\!] & = [\![\texttt{in}(e \texttt{ + } n, C)]\!] \\
[\![\texttt{in}(e,\texttt{deref}(C))]\!] & = \texttt{(exists x : ptr :: } [\![\texttt{in(x,}C\texttt{)}]\!] \texttt{ \&\& Mem[x] == } [\![e]\!]\texttt{)} \\
[\![\texttt{in}(e,\texttt{old\_deref}(C))]\!] & = \texttt{(exists x : ptr :: } [\![\texttt{in(x,}C\texttt{)}]\!] \texttt{ \&\& old(Mem)[x] == } [\![e]\!]\texttt{)} \\
[\![\texttt{in}(e,\texttt{union}(C_1,C_2))]\!] & = [\![\texttt{in}(e,C_1)]\!] \texttt{ || } [\![\texttt{in}(e,C_2)]\!] \\
[\![\texttt{in}(e,\texttt{intersection}(C_1,C_2))]\!] & = [\![\texttt{in}(e,C_1)]\!] \texttt{ \&\& } [\![\texttt{in}(e,C_2)]\!] \\
[\![\texttt{in}(e,\texttt{difference}(C_1,C_2))]\!] & = [\![\texttt{in}(e,C_1)]\!] \texttt{ \&\! ! } [\![\texttt{in}(e,C_2)]\!]
\end{array}
$$

Fig. 8. Translation of expressions from our annotation language into BoogiePL. In this figure, `bpl_x` refers to the BoogiePL variable corresponding to the C variable `x`.

post-state and $\texttt{old(x)}$ refers to the value of `x` in the pre-state of the procedure. The expressions $\texttt{deref}(e)$ and $\texttt{old\_deref}(e)$ refer to the value stored in memory at the address e in the post-state and pre-state, respectively.[3] The expressions $\texttt{block}(e,n)$ and $\texttt{old\_block}(e,n)$ represent $\texttt{B}_n[e]$ in the post-state and pre-state of the procedure, respectively. To be able to reason about sets of pointers, we use the expression $\texttt{forall}(\texttt{x},S,\phi)$ which says that for all elements `x` of some set of pointers S formula ϕ has to hold.

The set expressions are divided into the basic set expressions in *Set* and the compound set expressions in *CmpdSet*. The expression $\texttt{array}(e_1,n,e_2)$ refers to the set of pointers $\{e_1, e_1+n, e_1+2*n, \ldots, e_1+\texttt{Off}(e_2)*n\}$. The expressions $\texttt{list}(e,n)$ and $\texttt{old\_list}(e,n)$ represent the list of pointers described by the reachability predicate $\texttt{R}_n$ in the post-state and pre-state, respectively. The compound set expressions include $\texttt{incr}(C,n)$ and $\texttt{decr}(C,n)$ which respectively increment and decrement each element of C by n, and

[3] Note that the translation of $\texttt{old\_deref}(e)$ wraps only the implicit map `Mem` with `old`, which is sometimes necessary. Therefore, using $\texttt{old(deref}(e)\texttt{)}$ that would wrap `old` around the whole translation of expression $\texttt{deref}(e)$ wouldn't be sufficient.

`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.

The translation function $\llbracket \circ \rrbracket$ that recursively translates each expression from our annotation language into the corresponding BoogiePL expression is formally defined in Figure 8. The function translates integer value n as a pointer whose first component is the constant `null`. Based on whether its address has been taken or not, a variable `x` is translated into a memory reference `Mem`[`bpl_x`] or a BoogiePL variable `bpl_x`, respectively. Because variables whose address has been taken are allocated on the heap, the expression `addr`(`x`) is simply translated as a BoogiePL variable. Arithmetic expressions perform corresponding operations on the integer components of pointers, while pointer dereference accesses the map `Mem` that represents contents of the memory. The `block`(e, n) expression is translated into application of the BoogiePL function $\mathtt{B}_n$. The expressions `old`(`x`), `old_deref`(e), and `old_block`(e, n) simply wrap `x`, `Mem`, and $\mathtt{B}_n$ with `old`(), respectively.

Our annotation language contains the `forall`$(\mathtt{x}, S, \phi)$ expression whose translation uses the *element of set* $\llbracket$`in`$(\mathtt{x}, S)\rrbracket$ expression to check whether pointer `x` is an element of S. Two important, basic `in` checks that we support are `in`$(e,$ `array`$(e_1, n, e_2))$ and `in`$(e,$ `list`$(e', n))$ for arrays and lists of pointers, respectively. The element of array check `in`$(e,$ `array`$(e_1, n, e_2))$ is translated into a BoogiePL expression that looks for an index `i` such that `i` is at least 0 and less than the size of the array, and furthermore that the element is at the appropriate offset. The expression `in`$(e,$ `list`$(e', n))$ is translated into the $\mathtt{R}_n$ predicate to check whether $\llbracket e \rrbracket$ is reachable from $\llbracket e' \rrbracket$.

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 9. The translation refers to the translation function $\llbracket \circ \rrbracket$ defined in Figure 8.

We also allow each loop to be annotated with a formula representing its invariant.

In Figure 9, the translation of `requires` ϕ and `ensures` ψ is obtained in a straightforward fashion by applying the translation function $\llbracket \circ \rrbracket$ 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 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 9 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 `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 `decr`(C, n), then $\mathtt{B}_n$[`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.

7 Implementation

We have developed Havoc, a prototype tool for verifying C programs annotated with specifications in our annotation language. We use the ESP [13] 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 4 and Section 6. The Boogie verifier [4] 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 [5] of the standard *weakest precondition* transformer [17]. The resulting VC is checked for validity using the Z3 theorem prover [14]. [4]

7.1 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 [31].

[4] In the early versions of this work, we used the older Simplify theorem prover [16], which also supports all of the required theories, instead of Z3. Currently, Z3 is the best choice performance-wise, although any other theorem prover that accepts Boogie output format and has the required theories could be used instead of it.

```
// 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[o] != UNALLOCATED) ||
                               Alloc[o] == old(Alloc)[o])
```

Fig. 9. Translation of `requires` ϕ, `ensures` ψ, `modifies` C, and `frees` D.

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

Fig. 10. Axioms for the theory of pairs.

$$\begin{array}{ll} [\textsc{reflexivity}] & \forall x :\ \texttt{R}_n[x, x] \\ [\textsc{antisymmetry}] & \forall x, y :\ \texttt{R}_n[x, y] \wedge \texttt{R}_n[y, x] \Rightarrow x = y \\ [\textsc{transitivity}] & \forall x, y, z :\ \texttt{R}_n[x, y] \wedge \texttt{R}_n[y, z] \Rightarrow \texttt{R}_n[x, z] \\ [\textsc{ordering}] & \forall x, y, z :\ \texttt{R}_n[x, y] \wedge \texttt{R}_n[x, z] \Rightarrow \texttt{R}_n[y, z] \vee \texttt{R}_n[z, y] \\ [\textsc{reach1}] & \forall x :\ \texttt{BS}[\texttt{Mem}[x + n]] \vee \texttt{R}_n[x, \texttt{Mem}[x + n]] \\ [\textsc{reach2}] & \forall x, z :\ \texttt{R}_n[x, z] \Rightarrow x = z \vee (\neg\texttt{BS}[\texttt{Mem}[x + n]] \wedge \texttt{R}_n[\texttt{Mem}[x + n], z]) \\ [\textsc{well-founded1}] & \forall x :\ \texttt{R}_n[\texttt{Mem}[x + n], x] \Rightarrow \texttt{BS}[\texttt{Mem}[x + n]] \\ [\textsc{well-founded2}] & \forall x :\ \texttt{R}_n[x, y] \wedge \texttt{BS}[y] \Rightarrow x = y \\ [\textsc{block1}] & \forall x :\ \texttt{BS}[\texttt{Mem}[x + n]] \Rightarrow \texttt{B}_n[x] = \texttt{Mem}[x + n] \\ [\textsc{block2}] & \forall x :\ \texttt{BS}[\texttt{B}_n[x]] \\ [\textsc{block3}] & \forall x, y :\ \texttt{R}_n[x, y] \Rightarrow \texttt{B}_n[x] = \texttt{B}_n[y] \end{array}$$

Fig. 11. Reachability axioms. Note that the symbol + is the addition operation on pointers. We have overloaded + for ease of exposition.

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 low-level reachability predicate, consisting of the symbols $\texttt{R}_n$, $\texttt{B}_n$, `BS` and `Mem`.

To discharge the verification conditions, a theorem prover requires axioms about the theory of pairs, and a way of handling the theory of the new low-level reachability predicate. The axioms for the theory of pairs are fairly intuitive and are given in Figure 10. In our previous paper [10], we described how we initially supported the theory of the new low-level reachability predicate. The support for the theory was built on a sound but incomplete axiomatization of the theory of well-founded lists without pointers as in Java and C# [23]. To account for low-level C operations such as pointer arithmetic and

internal pointers that the new reachability predicate supports, we suitably generalized the described incomplete axiomatization for Java/C#. In this work, however, we turn to a similarly extended decision procedure for well-founded reachability over objects that has proven to be much more effective [24,25].

The decision procedure for well-founded reachability is based on a sound, complete, and terminating set of rewrite rules. The prototype implementation of the decision procedures emulates the rewrite rules in an SMT solver by encoding them using universally-quantified first-order axioms with appropriate matching *triggers*. Triggers are subterms of the quantified formula that are used by the underlying theorem prover in deciding how to instantiate universal quantifiers [16]. Figure 11 presents the set of axioms. To support the low-level reachability predicate described in this paper, these axioms are subtle generalizations of the previously published ones [24] to account for low-level C operations such as pointer arithmetic and internal pointers. The main difference is that fields are modeled as offsets within an object and the reachability relation is then defined with respect to these offsets.

8 Evaluation

In this section, we describe our experience applying Havoc to a set of small to medium size C examples. Figure 12 lists the examples considered in this paper. The examples manipulate singly- and doubly-linked lists. In addition to performing operations on linked data structures, the examples also use pointer arithmetic, internal pointers into objects, and cast operations. The examples range from 10 to 150 lines of C code. For all these examples, we check a set of partial correctness properties including (but not limited to) the implicit memory-safety requirements. Next, we'll give a brief description of each example and additional properties we check, when applicable:

`iterate` – initializes the data elements of a cyclic list. This is the example from Figure 2 in Section 3.

`iterate_acyclic` – similar to `iterate`, just initializes the data elements of an acyclic list.

`slist_add` – adds a node to an acyclic singly-linked list.

`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.

`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.

`dlist_add`, `dlist_remove` – 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.

`allocator` – low-level storage allocator that closely resembles the `malloc_firstfit_acyclic` example described by Calcagno et al. [9]. The allocator maintains a list of free blocks within a single large object; each call to the allocate routine returns either (i) a block within the object larger than or equal to the requested block size, or (ii) null, otherwise. The acyclic linked list threads through the free blocks, and 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.

`list_appl` – simple application with multiple doubly-linked lists, parent pointers, and usage of the primitive doubly-linked list operations; we verify that the disjoint lists satisfy certain data invariants.

`muh_free` – simplified version of the `muh` example described in the recent paper by Lahiri and Qadeer [25]; we check the absence of the double-free property.

Example	Time(s)
`iterate`	1.50
`iterate_acyclic`	1.43
`slist_add`	1.36
`reverse_acyclic`	1.37
`slist_sorted_insert`	4.85
`dlist_add`	1.75
`dlist_remove`	1.65
`allocator`	2.00
`list_appl`	30.22
`muh_free`	8.20

Fig. 12. Results of assertion checking using Havoc. Z3 was used as the theorem prover. The experiments were conducted on a 3.6GHz, 2GB machine running Windows XP.

Figure 12 gives the running times taken by Havoc to check the assertions in each example. It takes only a small fraction of the presented time to generate verification conditions. Therefore, reported times are largely dominated by the time it takes Z3 to discharge the generated VCs. The results are very encouraging since Z3 proved most of our examples in just a couple of seconds. Only `list_appl` is taking a little bit more time due to the use of complex invariants that connect the forward-going and backward-going links in a doubly-linked list. In addition, we also believe that the presented results could

be further improved. Currently, the decision procedure for our low-level reachability predicate is just a prototype implementation using universally quantified axioms. Actually implementing the presented rewrite-rule-based decision procedure in an SMT solver will provide additional performance boost and is an area of future work.

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 that casts a pointer variable `cursor` to an `unsigned int` and then performs an integer addition

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

or similarly

```
return (unsigned int) (cursor + 1);
```

we had written the following incorrect code

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

Note that the two statements return different values because the size of `RegionHeader`, which is the static type of the structure the pointer `cursor` is pointing to (i.e. the type of `cursor` is `RegionHeader*`), is not 1. 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.

9 Conclusions and future work

In this work, we introduced a memory model and the accompanying 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 based on the classical reachability predicate used in existing verification tools. In addition, we described a decision procedure for the predicate based on a set of rewrite rules. 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. Furthermore, we tested Havoc on a set of illustrative C programs that perform operations on linked data structures as well as use low-level C pointer manipulations.

Based on the presented results, we believe that Havoc is a good foundation for building a powerful safety checker for systems software based on automated first-order theorem proving. We are currently working to extend Havoc with techniques for inference and abstraction to reduce the manual annotation requirement by automatically inferring many annotations. That would make Havoc much easier to adopt and enable its use on realistic code bases inside Windows.

The reachability predicate we introduced is mainly suitable for describing linked lists, while more complex recursive data structures, such as trees, are beyond its reach. Lists are the most commonly used recursive data structures in systems software, and therefore were the natural place to start. The obvious next step, which we are planning to address in the future, is to extend the theory of reachability to be able to handle more complex data structures.

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 Boogie. Finally, we thank Tom Ball and Rustan Leino for providing valuable feedback on the paper.

References

1. D. Babić and A. J. Hu. Calysto: Scalable and precise extended static checking. In *International Conference on Software Engineering (ICSE '08)*, pages 211–220, 2008.
2. I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *International Conference on Verification, Model checking, and Abstract Interpretation (VMCAI '05)*, pages 164–180, 2005.
3. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.
4. M. Barnett, B.-Y. E. Chang, R. DeLine, B. J. 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Objects and Components (FMCO '05)*, pages 364–387, 2005.
5. 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.
6. 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 (CASSIS '05)*, pages 49–69, 2005.
7. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *International Conference on Computer Aided Verification (CAV '07)*, pages 178–192, 2007.
8. J. Bingham and Z. Rakamarić. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '06)*, pages 207–221, 2006.
9. 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)*, pages 182–203, 2006.

10. S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A reachability predicate for analyzing low-level software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07)*, pages 19–33, 2007.
11. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, 2001.
12. 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, 2004.
13. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, pages 57–68, 2002.
14. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, pages 337–340, 2008.
15. 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.
16. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.
17. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
18. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, pages 287–302, 2006.
19. J. Filliâtre and C. Marché. Multi-prover verification of C programs. In *International Conference on Formal Engineering Methods (ICFEM '04)*, pages 15–29, 2004.
20. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*, pages 234–245, 2002.
21. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium (SAS)*, pages 240–260, 2006.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*, pages 58–70, 2002.
23. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*, pages 115–126, 2006.
24. S. K. Lahiri and S. Qadeer. A decision procedure for well-founded reachability. Technical Report MSR-TR-2007-43, Microsoft Research, 2007.
25. S. K. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*, pages 171–182, 2008.
26. 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 *International Conference on Automated Deduction (CADE '05)*, pages 99–115, 2005.
27. T. Lev-Ami and S. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symposium (SAS '00)*, pages 280–301, 2000.
28. Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *1st Workshop on Architectural and System Support for Improving Software Dependability (ASID '06)*, pages 25–33, 2006.
29. S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *International Conference on Computer Aided Verification (CAV '05)*, pages 476–490, 2005.
30. G. Nelson. Verifying reachability invariants of linked structures. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '83)*, pages 38–47, 1983.
31. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.
32. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic (CSL '01)*, pages 1–19, 2001.
33. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Annual IEEE Symposium on Logic in Computer Science (LICS '02)*, pages 55–74, 2002.
34. 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.
35. W. Schulte, S. Xia, J. Smans, and F. Piessens. A glimpse of a verifying C compiler (extended abstract). In *C/C++ Verification Workshop*, 2007.
36. M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. In *International Symposium on Fault-Tolerant Computing (FTCS '91)*, pages 2–9, 1991.
37. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 351–363, 2005.
38. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer Aided Verification (CAV '08)*, pages 385–398, 2008.