Verifying Low-Level Implementations of High-Level Datatypes

Christopher L. Conway and Clark Barrett

New York University, Dept. of Computer Science
{cconway,barrett}@cs.nyu.edu

Abstract. For efficiency and portability, network packet processing code is typically written in low-level languages and makes use of bit-level operations to compactly represent data. Although packet data is highly structured, low-level implementation details make it difficult to verify that the behavior of the code is consistent with high-level data invariants. We introduce a new approach to the verification problem, using a high-level definition of packet types as part of a specification rather than an implementation. The types are not used to check the code directly; rather, the types introduce functions and predicates that can be used to assert the consistency of code with programmer-defined data assertions. We describe an encoding of these types and functions using the theories of inductive datatypes, bit vectors, and arrays in the CVC3 SMT solver. We present a case study in which the method is applied to open-source networking code and verified within the CASCADE verification platform.

1 Introduction

Packet-level networking code is critical to communications infrastructure and vulnerable to malicious attacks. This code is typically written in low-level languages like C or C++. Packet fields are "parsed" using pointer arithmetic and bit-wise operators to select individual bytes and sequences of bits within a larger untyped buffer (e.g., a `char` array). This approach yields high-performance, portable code, but can lead to subtle errors.

An alternative is to write packet-processing code in special-purpose high-level languages, e.g., binpac [17], Melange [13], Morpheus [1], or Prolac [9]. These languages typically provide a facility for describing network packets as a set of nested, and possibly recursive, datatypes. The language compilers then produce low-level packet-processing code which aims to match or exceed the performance of the equivalent hand-coded C/C++. This requires an expensive commitment to rewriting existing code.

We propose a new approach, one which fuses the power of higher-level datatypes with the convenience and efficiency of legacy code. The key idea is to use a high-level description of "packet types" as the basis for a *specification*, not an *implementation*. Instead of using a compiler to try to reproduce a performant implementation, we can annotate the existing implementation to indicate the intended high-level semantics, then verify that the implementation is consistent with those semantics.

We make use of the theories of inductive datatypes, bit vectors, and arrays in CVC3 to encode the relationship between the high-level and low-level semantics. Using this encoding, it is possible to verify that the low-level code represents, in essence, an implementation of a well-typed high-level specification.
In this paper we will present our proposed notation for defining packet datatypes and stating datatype invariants in C code. We describe the translation of the datatype definition and code assertions into verification conditions in the CVC3 SMT solver. The encoding relies crucially on automatically generated separation invariants, which allow CVC3 to efficiently reason about recursive data structures without producing false assertion failures due to spurious aliasing relationships. Finally, we present a case study applying our approach to real code from the BIND DNS server. We are able to verify high-level data invariants of the code with reasonable efficiency. To our knowledge, no other verification tool is capable of automatically proving such datatype invariants on existing C code.

2 A Motivating Example

Figure 1(a) illustrates the definition of a simple, high-level list datatype in a notation similar to that of languages like ML and Haskell. The type has two constructors: `cons`, which creates a list node with an associated `data` array and a `cdr` field representing the remainder of the list, and `nil`, which represents an empty list. Figure 1(b) gives the high-level pseudo-code for a function that computes the length of a list, defined as the number of `cons` values encountered via `cdr` "links" before a `nil`. The code simply checks whether `lst` is a `cons` value using the "tester" function `isCons`. If it is, it increments the length and updates `lst` using the `cdr` field. If it is not, it returns the computed length.
In a high-level language, the compiler is given the freedom to implement datatypes like `List` as it chooses, typically using linked heap structures to represent individual datatype values. The programmer concentrates on the high-level semantics of the algorithm, allowing the compiler to encode and decode the data. By contrast, in packet processing code, the datatype is defined in terms of an explicit data layout. The data is "packed" into a contiguously allocated block of memory. The high-level algorithm and the encoding and decoding of data are intertwined.
The `List` type in Fig. 1(c) illustrates a simple "packed" linked list implementation. Like the definition in Fig. 1(a), `List` is a union type with two variants. However, instead of simply declaring a set of data fields, each variant explicitly defines its own representation. The representation of a `cons` value is: a 1-bit `tag` field (the highest-order bit of the first byte), a 7-bit `count` field (the lower-order bits of the first byte), a `data` field of exactly `count` bytes, and another `List` value `cdr`, which follows immediately in memory. The value of `tag` is constrained by the constant bit vector value `0b1`. The constraint requires the `tag` bit of a `cons` value always to be `1`. The representation of a `nil` value has a similar constraint: a `nil` value consists of a single 8-bit `tag` field, which must be `0x00`. The

```
type List =
  cons {
    count: Nat,
    data: Int array,
    cdr: List
  }
| nil
```
(a)

```
Nat list_length(List lst) {
    Nat count = 0;
    while( isCons(lst) ) {
        count++;
        lst = cdr(lst);
    }
    return count;
}
```
(b)

```
type List =
  cons {
    tag:1 = 0b1,
    count: 7,
    data: u_char[count],
    cdr: List
  }
| nil {
    tag:8 = 0x00
  }
```
(c)

```
u_int list_length(const u_char *p) {
    u_int n, count = 0;
    while( (n = *p++) & 0x80 ) {
        { isCons(prev(p)) }
        count++;
        p += n & 0x7f;
        { toList(p) = cdr(prev(p)) }
    }
    if( n != 0 ) // malformed list
        return (-1);
    { isNil(p) }
    return count;
}
```
(d)

Fig. 1. Defining and using a simple linked list datatype.

fact that the `tag` bit of a `cons` value must be `1` while the bits of a `nil` value must all be `0` ensures that we can unambiguously decode `cons` and `nil` values. (A full grammar for "packed" datatype definitions is given in Section 3.1.)

Figure 2 illustrates the interpretation of a sequence of bytes as a `List` value. The first byte (`0x82`) has its high bit set; thus, it is a `cons` value. The low-order bits tell us that `count` is `2`; thus, `data` has two elements: `0x01` and `0x02`. The `cdr` field is another `List` value, encoded starting at the next byte. This byte (`0x81`) is also a `cons` value, since it also has its high bit set. Its `count` field is `1`, its `data` field the single element `0x03`. Its `cdr` is the `List` value at the next byte (`0x00`), a `nil` value.

Figure 1(d) gives a low-level implementation of the length function, which operates over the implicit `List` value pointed to by the input `p`. (The bracketed, italicized portions of the code are verification annotations, which are described in Section 3.3.) Note that the structure of the function is very similar to the code in Fig. 1(b), but that high-level operations have been replaced by their low-level equivalents—pointer arithmetic and bit-masking operations are used to detect constructors and select fields. A notable addition is the `if` statement that appears after the `while` loop. In the high-level code, we could assume that the data was well-formed, i.e., that every list is either a `cons` or a `nil` value. In the low-level implementation, we may encounter byte sequences which are not assigned a meaning by the datatype definition—in this case a non-zero byte in which the high bit is not set, which satisfies the data constraints of neither `cons` nor `nil`. The function handles this erroneous case by returning an error code.

The challenge, in essence, is to prove that the low-level code in Fig. 1(d) is a refinement of the high-level code in Fig. 1(b). To this end, we need

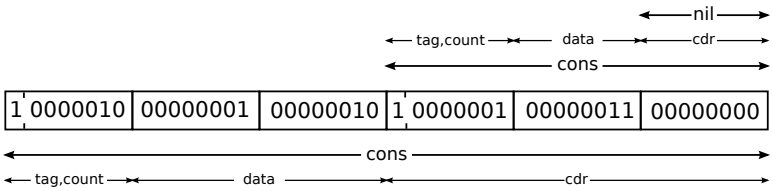

Fig. 2. The layout of a `List` value.

to build a bridge between the high-level semantics of the datatype and the low-level implementation.

3 Our Approach

The verification process proceeds in four steps:

1. The programmer provides a datatype declaration, as in Fig. 1(c), defining the high-level structure and layout of the data.
2. Using the datatype declaration, we generate a set of CVC3 declarations and axioms encoding the relationship between the high-level type and its implementation.
3. The programmer adds code annotations specifying the expected behavior of the low-level code, in terms of functions derived from the datatype definition.
4. We use the CASCADE verification platform to translate the code and annotations into a set of verification conditions to be checked by CVC3. If all of the verification conditions are valid, then the code satisfies the specification.

3.1 Datatype definition

Figure 3 gives the full grammar for datatype definitions. The notation for datatype definitions is similar to that of disjoint union types in higher-level languages like ML and Haskell. There is an important distinction: unlike datatype implementations generated by compilers, it is up to the user to ensure that the encoding of values is unambiguous and consistent. The declaration should provide all of the information needed both to encode a datatype value as a sequence of bytes and to decode a well-formed sequence of bytes as a high-level datatype value.

A type consists of a set of constructors. Each constructor has a set of fields. A field type is one of four kinds: a bit vector of constant integer size, a plain C scalar type, an array of C type elements, or another datatype. (The syntax of C type declarators is that of ANSI/ISO C [2].) Bit vectors and C types may have value constraints. Bit vector constants are preceded by `0b` (for binary constants) or `0x` (for hexadecimal constants). Arrays have a length: either a constant integer or the value of a prior field—the declaration language supports a limited form of dependent types.

$$
\begin{aligned}
\mathit{Type} &::= \texttt{type}\ \mathit{Id}\ \texttt{=}\ \mathit{Cons}\ (\texttt{|}\ \ \mathit{Cons})* \\
\mathit{Cons} &::= \mathit{Id}\ \{\ \ \mathit{Field}\ (\texttt{,}\ \mathit{Field})*\ \} \\
\mathit{Field} &::= \mathit{Id}\ \texttt{:}\ \mathit{FieldType} \\
\mathit{FieldType} &::= \mathit{BvType} \mid \mathit{CType} \mid \mathit{ArrType} \mid \mathit{TypeId} \\
\mathit{BvType} &::= \mathit{IntConst}\ (\texttt{=}\ \mathit{BvConst})? \\
\mathit{BvConst} &::= \texttt{0b[01]+} \mid \texttt{0x[0-9a-fA-F]+} \\
\mathit{CType} &::= \mathit{CScalarType}(\texttt{=}\ \mathit{CConst})? \\
\mathit{ArrType} &::= \mathit{CType}\ \texttt{[}\mathit{ArrLength}\texttt{]} \\
\mathit{ArrLength} &::= \mathit{IntConst} \mid \mathit{Id} \\
\mathit{TypeId} &::= \mathit{Id}
\end{aligned}
$$

Fig. 3. Grammar for datatype definitions.

$$
\begin{array}{l}
\textbf{datatype}\ \ \mathit{List} = \ \mathit{cons}\ \{\ \mathit{count} : \mathcal{BV}_7,\ \ \mathit{data} : (\mathcal{BV}_N, \mathcal{BV}_8)\ \mathsf{array},\ \ \mathit{cdr} : \mathit{List}\ \} \\
\qquad\qquad\qquad\quad \mid\ \mathit{nil} \\
\qquad\qquad\qquad\quad \mid\ \mathit{undefined}
\end{array}
$$

$$
\begin{array}{rlrl}
\mathit{toList} : (\mathcal{BV}_N, \mathcal{BV}_8)\ \mathsf{array} \times \mathcal{BV}_N \to \mathit{List} & & m : (\mathcal{BV}_N, \mathcal{BV}_8)\ \mathsf{array} \\
\mathit{sizeOfList} : \mathit{List} \to \mathbb{N} & & \ell : \mathcal{BV}_N
\end{array}
$$

$$
\begin{array}{rcll}
\multicolumn{4}{l}{\textbf{let}\ x = \mathit{toList}(m, \ell)\ \textbf{in}} \\
\mathit{isCons}(x) & \iff & m[\ell][7] & (\textsc{ConsTest}) \\
\mathit{isNil}(x) & \iff & m[\ell] = 0 & (\textsc{NilTest}) \\
\mathit{isCons}(x) & \implies & \mathit{count}(x) = m[\ell][6{:}0] & \\
 & & \quad \wedge\ (\forall 0 \le i < \mathit{count}(x).\ \ \mathit{data}(x)[i] = m[\ell + i + 1]) & \\
 & & \quad \wedge\ \ \mathit{cdr}(x) = \mathit{toList}(m, \ell + \mathit{count}(x) + 1) & (\textsc{ConsSel})
\end{array}
$$

$$
\begin{array}{rl}
\mathit{sizeOfList}(\mathit{cons}(\mathit{count}, \_, \mathit{cdr})) = 1 + \mathit{count} + \mathit{sizeOfList}(\mathit{cdr}) & (\textsc{ConsSize}) \\
\mathit{sizeOfList}(\mathit{nil}) = 1 & (\textsc{NilSize}) \\
\mathit{sizeOfList}(\mathit{undefined}) = 0 & (\textsc{UndefSize})
\end{array}
$$

Fig. 4. Datatype definition and axioms for the type `List`

3.2 Translation to Cvc3

It is straightforward to translate the datatype definition into an inductive datatype in the input language of CVC3. The translation for the `List` datatype is given in Fig. 4. We use $\mathbb{N}$ to denote the type of natural numbers; $\mathcal{BV}_k$ to denote the type of bit vectors of size k (i.e., k-tuples of booleans); and (α, β) array to denote the type of arrays with indices of type α and elements of type β. We use N to denote the (platform-dependent) size of a pointer (i.e., the type of pointers is $\mathcal{BV}_N$). For an array a, $a[i]$ denotes the element of a at index i; similarly, for a bit vector b, $b[i]$ denotes the ith bit of b and $b[j{:}i]$ denotes the *extraction* of bits i through j (the result is a bit vector of size $j-i+1$). The size of the result of arithmetic operations on bit vectors is the size of the larger operand; the smaller operand is implicitly zero-extended. When used in an integer context, bit vectors are interpreted as unsigned.

The translation produces a CVC3 datatype definition reflecting the data layout of the declaration augmented with an explicit *undefined* value. Note that the `tag` fields are omitted from the definition—since they are constrained by constants, they are only needed to decode the high-level data value.

CVC3 automatically generates a set of datatype testers and field selectors. The testers *isCons*, *isNil*, and *isUndefined* are predicates that hold for a *List* value x iff x is, respectively, a *cons*, *nil*, or *undefined* value. The selectors *count*, *data*, and *cdr* are functions that map a *List* value to the value of corresponding field.

Note that the definition of *List* itself does not include any data constraints on field values. These constraints are introduced by the function *toList*, which maps a pointer-indexed array of bytes m and a location ℓ to the *List* value represented by the sequence of bytes starting at ℓ in m. The axioms CONSTEST and NILTEST enforce the data constraints on the `tag` fields of `cons` and `nil`, respectively. The axiom CONSSEL represents the encoding of the remaining fields of `cons`. Note that there is no explicit rule for the value *undefined*: if the data constraints given in CONSTEST and NILTEST do not apply, then the only remaining value that *toList* can return is *undefined*.

The function *sizeOfList* maps a *List* value to the size of its encoding in bytes. By convention, the size of *undefined* is 0.

3.3 Code assertions

The functions generated by the CVC3 translation are exposed in the assertion language as functions that take a single pointer argument. In the case of the function `toList`, the additional array argument, representing the configuration of memory, is introduced in the verification condition translation. The pointer argument of the other functions is implicitly converted to a `List` value using `toList`. The assertion language also provides auxiliary functions `init` and `prev`, mapping variables to their initial values in, respectively, the current function and loop iteration.

Returning to the code in Fig. 1(d), the bracketed, italicized assertions state the expected high-level semantics of the implementation. Specifically, they assert:

- The loop test succeeds only for `cons` values.
- The body of the loop sets `p` to the `cdr` of its initial value in each loop iteration.
- If the value is well-formed, then `p` points to a `nil` value when the function returns.

The functions representing testers rely on the data constraints of the type, e.g., `p` points to a `cons` value iff the byte sequence pointed to by `p` satisfies the data constraints of `cons` (i.e., the high bit of `*p` is set). The functions representing testers rely on the structure of the type, e.g., `toList(q)==cdr(p)` iff `p` points to a `cons` value and `q==p+count(p)+1`.

Loops can be annotated with invariants: we can separately prove initialization and preservation of the invariant, and that each assertion in the body of the loop is valid when the invariant is assumed on entry.

3.4 Verification condition generation

The final verification step is to use the CASCADE verification platform to translate the code and assertions into formulas that can be validated by CVC3. Verification is driven by a *control file*, which defines a set of paths to check and allows annotations and assertions to be injected at arbitrary points along a path. Each code assertion is transformed into a verification condition, which is passed to CVC3 and checked for validity. For each condition, CVC3 will return "valid" (the condition is always true), "invalid" (the condition is not always true), or "unknown" (due to incompleteness, CVC3 could not prove invalidity). CASCADE returns "valid" for a path iff CVC3 returns "valid" for every assertion on the path. If CVC3 returns "invalid" or "unknown" for any assertion, CASCADE returns "invalid", along with a counterexample.

Note 1. Since the background axioms that define datatypes are universally quantified, deciding validity of the generated verification conditions is undecidable in general. CVC3 will never return "invalid" for any verification condition that it cannot prove valid; instead, it will return "unknown" when a pre-determined instantiation limit is reached. There are fragments of first-order logic that are decidable with instantiation-based algorithms [6]. Encoding the datatype assertions in a decidable fragment of first-order logic is a subject for future work.

CASCADE supports a number of encodings for C expressions and program semantics. For datatype verification, we make use of a bit vector encoding, which is parameterized by the platform-specific size of a pointer and of a memory word.
An additional consideration is the memory model used in the verification condition. The memory model specifies the interpretation of pointer values and the effect of memory accesses (both reads and writes) on the program state. A memory model may abstract away details of the program's concrete semantics (e.g., by discarding information about the precise layout of structures in memory) or it may refine the concrete semantics (e.g., by choosing a deterministic allocation strategy). We discuss the memory model in detail in the next section.

4 Memory Model

In order to accurately reflect the datatype representation, we require a memory model that is bit-precise. At the same time, to avoid a blow-up in verification complexity and overly conservative results, we would like a relatively high-level model that preserves the separation invariants of the implementation. To this end, we define a memory model based on separation analysis [7] that we call a *partitioned heap*.
The flat model. First, we will define for comparison a simple model which is self-evidently sound. A *flat memory model* interprets every pointer expression as a bit vector of size N. Every allocated object in the program is associated with a region of memory (i.e., a contiguous block of locations) distinct from all previously allocated regions. The

state of memory is modeled by a single pointer-indexed array m. The value stored at location ℓ is thus $m[\ell]$.
Using the flat memory model, we can translate the first assertion in Fig. 1(d) into the verification condition

$$m_1 = m_0[\texttt{\&p} \mapsto m_0[\texttt{\&p}] + 1] \wedge m_2 = m_1[\texttt{\&n} \mapsto m_0[m_0[\texttt{\&p}]]] \wedge m_2[\texttt{\&n}][7] \\ \implies \mathit{isCons}(\mathit{toList}(m_2, m_0[\texttt{\&p}]))$$

where we use `&x` to denote the location in memory of the variable `x` (i.e., its *lvalue*) and $a[i \mapsto e]$ to denote the update of array a with element e at index i. Assuming `&p`, `&n`, and m[`&p`] are distinct, the validity of the formula is a direct consequence of the axiom ConsTest.
The flat model accurately represents unsafe operations like casts between incompatible types and bit-level operations on pointers. However, it is a very weak model—its lack of guaranteed separation between objects makes it difficult to prove strong properties of data-manipulating programs.

Example 1. Consider the Hoare triple

$$\{\ \texttt{toList(q)==cdr(p)}\ \}\ \texttt{i++}\ \{\ \texttt{toList(q)==cdr(p)}\ \}$$

where `p` and `q` are known to not alias `i`. In a flat memory model, this is interpreted as

$$\mathit{toList}(m_0, m_0[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(m_0, m_0[\texttt{\&p}])) \\ \wedge\ m_1 = m_0[\texttt{\&i} \mapsto m_0[\texttt{\&i}] + 1] \\ \implies \mathit{toList}(m_1, m_1[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(m_1, m_1[\texttt{\&p}]))$$

Since *toList* is defined axiomatically using recursion (see Fig. 4), it is not immediately obvious that the necessary lemma

$$\mathit{toList}(m_0, m_0[\texttt{\&p}]) = \mathit{toList}(m_1, m_1[\texttt{\&p}])$$

is implied (similarly for `q`). Even if `p` and `q` can never point to `i`, we cannot rule out the possibility that the `List` values pointed to by `p` and `q` depend in some way on the value of `i`. Now, suppose we add the assumption

`allocated(p,p+sizeOfList(p))`,

where `allocated(x,y)` means that pointer `x` is the base of a region of memory, disjoint from all other allocated regions, bounded by pointer `y`. Even then, the proof of the assertion relies on the following theorem, which is beyond the capability of automated theorem provers like Cvc3 to prove:

$$(\forall y : x \le y \le x + \mathit{sizeOfList}(\mathit{toList}(m_0, x)) : m_0[y] = m_1[y]) \\ \implies \mathit{toList}(m_0, x) = \mathit{toList}(m_1, x)$$

□

What we require is a separation invariant allowing us to apply the "frame rule" of separation logic [19, 15]:

$\{$ `toList(q)==cdr(p)*i==`v $\}$ `i++` $\{$ `toList(q)==cdr(p)*i==`$v+1$ $\}$

where `*` denotes *separating conjunction*: $A{*}B$ holds iff memory can be partitioned into two disjoint regions R and R' where A and B hold, respectively.

The partitioned model. The separation invariants we need can be obtained using separation analysis [7]. The analysis can be understood as the inverse of *may-alias analysis* [10, 11]: if pointers `p` and `q` can never alias, then the objects they point to must be separated (i.e., they occupy disjoint regions of memory).

The output of the separation analysis is a *partition* $P = \{P_1, \ldots, P_k\}$, where each P_i represents a disjoint region of memory, and a map from pointer expressions to regions—if expression E is mapped to partition P_i, then E can only point to objects allocated in region P_i. If the separation analysis maps pointers expressions E and E' to different partitions, then E and E' cannot be aliased in any well-defined execution of the program.

A *P-partitioned memory model* for partition $P = \{P_1, \ldots, P_k\}$ interprets every pointer expression as a pair $(\ell, i) \in \mathcal{BV}_N \times \mathbb{N}$, where ℓ is a location and i is a partition index. The state of memory is modeled by a collection of pointer-indexed arrays $\langle m_1, \ldots, m_k \rangle$. The location pointed to by pointer expression (ℓ, i) is the array element $m_i[\ell]$.

Example 2. The program in Fig. 1(d) can be divided into two partitions. The first partition contains the parameter `p` and local variables `n` and `size`. The second partition contains the object pointed-to by `p`. We represent the two partitions by two memory arrays, s and h, respectively. Thus, the value of the variable `n` is represented by the array element s[`&n`]; the value of the expression `*p` is represented by the array element $h[s$[`&p`]].

A partitioned memory model solves the problem of Example 1 by isolating the *List* value in its own partition:

$$\begin{aligned} \mathit{toList}(h_0, s_0[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(h_0, s_0[\texttt{\&p}])) \wedge s_1 = s_0[\texttt{\&i} \mapsto s_0[\texttt{\&i}] + 1] \\ \implies \mathit{toList}(h_0, s_1[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(h_0, s_1[\texttt{\&p}])) \end{aligned}$$

Given that `&p`, `&q` and `&i` are distinct, the formula is trivially valid. □

We say a program is *memory safe* if all reads and writes through pointers occur only within allocated objects. Like pointer analysis, the soundness of the separation analysis is conditional on memory safety. Thus, the soundness of verification using a partitioned memory model will likewise be conditional on memory safety.

It may seem questionable to attempt to verify a program using information which depends for its correctness on prior verification of the same program. In previous work, we showed that a sound combination is possible, as long as the verification procedure ensures that no memory safety errors occur along the path under consideration [5]. It is thus essential that the verification conditions include assertions that establish the memory safety of the statements along each path in the program.

In our experience, a partitioned memory model can make an order-of-magnitude difference in verification time compared to a flat memory model—indeed, properties are provable by CVC3 using a partitioned model that cannot be proved using a flat model (see Section 5.1).

5 Case Study: Compressed Domain Names

To demonstrate the utility of our approach, we will describe a more complex application, taken from real code. We will show the definition of a real-world datatype, the annotations for a function operating on that datatype, and the results of using CASCADE to verify the function.

A definition for the datatype `Dn`, representing an RFC 1035 *compressed domain name* [14], is given in Fig. 5. `Dn` is a union type with three variants: `label`, `indirect`, and `nullt`. The representation of a `label` value is: a 2-bit `tag` field (which must be zeroes), a 6-bit `len` field (which must *not* be all zeroes), a `label` field of exactly `len` bytes, and another `Dn` value `rest`, which follows immediately in memory. An `indirect` value has a 2-bit `tag` (which must be `0b11`) and a 14-bit offset. A `nullt` value has only an 8-bit `tag`, which must be zero. The constraints on the `tag` fields of `label`, `indirect`, and `nullt` allow us to distinguish between values.

```
type Dn =
    label {
      tag:2 = 0b00,
      len:6 != 0b000000,
      name:u_char[len],
      rest:Dn
    }
  | indirect {
      tag:2 = 0b11,
      offset:14
    }
  | nullt {
      tag:8 = 0x00
    }
```

Fig. 5. Definition of the `Dn` datatype.

Consider the function `ns_name_skip` in Fig. 6. The low-level pointer and bit-masking operations represent the traversal of the high-level `Dn` data structure. The correctness of the implementation is properly expressed in terms of that data structure.

In terms of the type `Dn`, the code in Fig. 6 is straightforward. The pointer `cp`, the value pointed to by the parameter `ptrptr`, points to a `Dn` value. The loop test (Line 12) assigns the first byte of the value to the variable `n` and advances `cp` by one byte. If `n` is `0`, then `cp` pointed to a `nullt` value and the loop exits. Otherwise (Line 14), the `switch` statement checks the two most significant bits of `n`—the `tag` field of a `label` or `indirect`

```
1  #define NS_CMPRSFLAGS (0xc0)
2
3  int
4  ns_name_skip(const u_char **ptrptr, const u_char *eom) {
5      { allocated(*ptrptr, eom) }
6      const u_char *cp;
7      u_int n;
8
9      cp = *ptrptr;
10     { @invariant: cp ≤ eom ⟹
11         cp + sizeOfDn(cp) = init(cp) + sizeOfDn(init(cp)) }
12     while (cp < eom && (n = *cp++) != 0) {
13         /* Check for indirection. */
14         switch (n & NS_CMPRSFLGS) {
15           case 0:               /* normal case, n == len */
16             { isLabel(prev(cp)) }
17             cp += n;
18             { rest(prev(cp)) = toDn(cp) }
19             continue;
20         case NS_CMPRSFLGS:      /* indirection */
21             { isIndirect(prev(cp)) }
22             cp++;
23             break;
24         default:                /* illegal type */
25             __set_errno (EMSGSIZE);
26             return (-1);
27         }
28         break;
29     }
30     if (cp > eom) {
31         __set_errno (EMSGSIZE);
32         return (-1);
33     }
34     { cp = eom ∨ cp = init(cp) + sizeOfDn(init(cp)) }
35     *ptrptr = cp;
36     return (0);
37 }
```

Fig. 6. The function `ns_name_skip` from BIND

value. If the `tag` field contains zeroes (Line 15), `cp` is advanced past the `label` field to point to the `Dn` value of the `rest` field. If the `tag` field contains ones (Line 20), `cp` is advanced past the `offset` field and breaks the loop. The `default` case of the `switch` statement returns an error code—the `tag` field was malformed. At the end of the loop, if `cp` has not exceeded the bound `eom`, the value of `cp` is one greater than the address of the last byte of the `Dn` value that `cp` pointed to initially. This is the contract of the function: given a reference to a pointer to a valid `Dn` value, it advances the pointer past the `Dn` value or to the bound `eom`, whichever comes first, and returns `0`; if the `Dn` value is invalid, it returns `-1`.

Annotating the source code. The datatype definition is translated into an inductive datatype with supporting functions and axioms, as in Section 3.2. The translation generates testers *isLabel*, *isIndirect*, and *isNullt*; selectors *len*, *name*, *rest*, etc.; and the encoding functions *toDn* and *sizeOfDn*. Each of these functions is now available for use in source code assertions, as in the bracketed, italicized portions in Fig. 6.

The annotations in Fig. 6 also make reference to some auxiliary functions: `init(x)` represents the initial value of a variable `x` in the function; `prev(x)` represents the previous value of a variable `x` in a loop (i.e., the value at the beginning of an iteration).

On entry to the function (Line 5), we assume that the region pointed to by `*ptrptr` and bounded by `eom` is properly allocated. To each `switch` case (Lines 15 and 20), we add an assertion stating that the observed `tag` value (i.e, `n & NS_CMPRSFLGS`) is consistent with a particular datatype constructor (i.e., `label` or `indirect`). (Note that `prev(cp)` refers to the value of `cp` *before* the loop test, which has side effects). The loop invariant (Lines 10-11) states that `cp` advances through the `Dn` data structure pointed to by `init(cp)`—in each iteration of the loop, if `cp` has not exceeded the bound `eom`, it points to a `Dn` structure (perhaps the "tail" of a larger, inductive value) that is co-terminal with the structure pointed to by `init(cp)`. On termination, the loop invariant implies the desired post-condition: if no error condition has occurred, `*ptrptr` will point to the byte immediately following the `Dn` value pointed to by `init(cp)`—the pointer will have "skipped" the value. Note that we do not require an assertion stating that `cp` is reachable from `init(cp)` via `rest` "pointers" to prove the desired property—the property is provable using purely inductive reasoning.

Using the code annotations, CASCADE can verify the function by generating a set of verification conditions representing non-looping static paths through the function. Fig. 7 gives an example of such a verification condition. It represents the path from the head of the loop through the `0` case of the `switch` statement (Line 15), ending with the `continue` statement and asserting the preservation of the loop invariant. (Note that we assume here that pointers are 8 bits. Larger pointer values are easily handled, but the formulas are more complicated.) As in Section 4, the verification condition uses a partitioned memory model with two memory arrays, s and h: the values of local variables and parameters are stored in s while the `Dn` value pointed to by `cp` is stored in h. Proposition (1) asserts the loop invariant on entry. Propositions (2)–(5) represent the evaluation, including effects, of the loop test. Proposition (6) represents

the matching of the `switch` case. Propositions (7)–(9) capture the body of the case block. Finally, Proposition (10) (the proposition we would like to prove, given the previous assumptions) asserts the preservation of the loop invariant.

$$
\begin{aligned}
& s_0[\texttt{\&cp}] \geq s_0[\texttt{\&eom}] \\
& \vee\ s_0[\texttt{\&cp}] + sizeOfDn(toDn(h_0, s_0[\texttt{\&cp}])) \\
& = init(\texttt{\&cp}) + sizeOfDn(toDn(h_0, init(\texttt{\&cp}))) && (1) \\
& s_0[\texttt{\&cp}] < s_0[\texttt{\&eom}] && (2) \\
& s_1 = s_0[\texttt{\&cp} \mapsto s_0[\texttt{\&cp}] + 1] && (3) \\
& s_2 = s_1[\texttt{\&n} \mapsto h_0[s_0[\texttt{\&cp}]]] && (4) \\
& s_2[\texttt{\&n}] \neq 0 && (5) \\
& s_2[\texttt{\&n}][7:6] = 0 && (6) \\
& is\_label(toDn(h_0, s_0[\texttt{\&cp}])) && (7) \\
& s_3 = s_2[\texttt{\&cp} \mapsto s_2[\texttt{\&cp}] + s_2[\texttt{\&n}]] && (8) \\
& rest(toDn(h_0, s_0[\texttt{\&cp}])) = toDn(h_0, s_3[\texttt{\&cp}]) && (9) \\
\hline
& s_4[\texttt{\&cp}] \geq s_4[\texttt{\&eom}] \\
& \vee s_4[\texttt{\&cp}] + sizeOfDn(toDn(h_0, s_4[\texttt{\&cp}])) \\
& = init(\texttt{\&cp}) + sizeOfDn(toDn(h_0, init(\texttt{\&cp}))) && (10)
\end{aligned}
$$

Fig. 7. Verification conditions for `ns_name_skip`.

5.1 Experiments

Table 1 shows the time taken by CVC3 to prove the verification conditions generated by CASCADE for `ns_name_skip`, using both the flat and partitioned memory models. The times given are for a Intel Dual Core laptop running at 2.2GHz with 4GB RAM. Each VC represents a non-looping, non-erroneous path to an assertion. The two TERM VCs represent the loop exit paths: TERM (1) is the path where the first conjunct is false (`cp >= eom`; TERM (2) is the path where the first conjunct is true (`cp < eom`) and the second is false (`n == 0`). The verification conditions marked with * for the flat memory model timed out after two minutes—we believe that these formulas are not provable in CVC3. All of the verification conditions together can be validated using the partitioned memory model in less than one second.

6 Related Work

Some early work on verification of programs operating on complex datatypes was done by Burstall [4], Laventhal [12], and Oppen and Cook [16]. Their work assumes that data layout is an implementation detail that can be abstracted away. Our work here focuses on network packet processing code, where the linear layout of the data structure is an essential property of the implementation.

Table 1. Running times on `ns_name_skip` VCs.

Name	Lines	Time (seconds) Flat	Part.
INIT	5–12	0.34	0.03
CASE `0` (1)	12-16	13.94	0.05
CASE `0` (2)	12-28	33.42	0.06
CASE `0` (3)	12-19	*	0.12
CASE `0xc0` (1)	12–14, 20–21	6.14	0.04
CASE `0xc0` (2)	12–14, 20–23, 30, 34	*	0.07
TERM (1)	12, 30, 34	0.63	0.06
TERM (2)	12, 30, 34	*	0.05

More recently, O'Hearn, Reynolds, and Yang [15] have approached the problem using *separation logic* [19, 8]. Given assumptions about the structure of the heap, the logic allows for powerful localized reasoning. In this work, we use separation analysis in the style of Hubert and Marché [7] to establish separation invariants, thus "localizing" the verification conditions.

Rakamaric and Hu [18] describe a variation of Burstall's memory model [4, 3] suitable for bit-precise verification of low-level code. However, their approach relies on a compile-time type analysis of the program—since we are trying to verify datatypes that are not explicitly represented in the program code, we must rely on a more primitive model.

7 Conclusions

In this paper, we have presented a novel approach to the verification of low-level packet processing code. Instead of rewriting code in a high-level declarative language, we propose to apply the information derived from a declarative specification to enable checking high-level assertions embedded in the low-level implementation. The approach allows for the continued use of tested, performant code, with the increased assurance of verification. The experimental results are encouraging; we believe our technique can scale to several hundreds or thousands of lines of code. In future work, we intend to extend our technique to a broader class of datatypes, including more typical pointer-linked data structures.

8 Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. 0644299. The authors would like to thank Dejan Jovanović for his help debugging the CVC3 translation and preparing the figures for this paper.

References

1. M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Trans. Netw.*, 1(1):4–19, 1993.
2. American National Standard for Programming Languages - C, Aug. 1989. ANSI/ISO 9899-1990.
3. R. Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.
4. R. Burstall. Some techniques for proving correctness of programs which alter data structures. In *Machine Intelligence*, 1972.
5. C. L. Conway, D. Dams, K. S. Namjoshi, and C. Barrett. Points-to analysis, conditional soundness, and proving the absence of errors. In *Static Analysis Symposium (SAS)*, July 2008.
6. Y. Ge and L. de Moura. Complete instantiation for quantified formulas in Satisfiability Modulo Theories. In *Computer Aided Verification (CAV)*, pages 306–320, 2009.
7. T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV)*, pages 81–93, Mar. 2007.
8. S. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, 2001.
9. E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A readable TCP in the Prolac protocol language. *Computer Communication Review*, 29(4):3–13, 1999.
10. W. Landi and B. G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *Principles of Programming Languages (POPL)*, pages 93–103, Jan. 1991.
11. W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Programming Language Design and Implementation (PLDI)*, pages 235–248, June 1992.
12. M. S. Laventhal. Verifying programs which operate on data structures. In *Reliable Software*, pages 420–426, 1975.
13. A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: creating a "functional" internet. In *European Conf. on Comp. Sys. (EuroSys)*, pages 101–114, 2007.
14. P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987.
15. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic (CSL)*, pages 1–19, 2001.
16. D. C. Oppen and S. A. Cook. Proving assertions about programs that manipulate data structures. In *Symposium on the Theory of Computing (STOC)*, pages 107–116, 1975.
17. R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: a yacc for writing application protocol parsers. In *Internet Measurement Conf. (IMC)*, pages 289–300, 2006.
18. Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 290–304, 2009.
19. J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millenial Perspectives in Computer Science*, 2000.