Efficient, Formally Verifiable Data Structures using ACL2 Single-Threaded Objects for High-Assurance Systems

David S. Hardin
Advanced Technology Center
Rockwell Collins, Inc.
Cedar Rapids, IA, USA
dshardin@rockwellcollins.com

Samuel S. Hardin
Department of Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
sshardin@iastate.edu

ABSTRACT

Classical data structures such as stacks, queues, and double-ended queues (deques) find broad use in security-critical applications. At the highest Evaluation Assurance Level (EAL) of the Common Criteria, such data structures must be formally specified, and proven to meet their specifications. Formal verification systems can readily reason about unbounded, functional data structures. However, such data structures are in the main not appropriate for direct implementation in high-confidence software systems, both because of their unbounded nature, and also due to the complexity of the functional forms (e.g., the use of two lists, one reversed, to implement a deque). We will show how a formally verified data structure specified using the ACL2 single-threaded object facility can be much more readily translated into high-assurance implementations expressed in conventional programming languages. Finally, we show how this translated data structure code can be compiled into efficient machine code for a common embedded microprocessor using a verified compiler, and executed on an EAL6+ verified operating system.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods, reliability*

General Terms

Reliability, Security, Verification

Keywords

ACL2, certification, data structures, deque, high assurance, theorem proving

1. INTRODUCTION

Security-critical applications are commonly certified according to the Common Criteria at the highest Evaluation Assurance Levels (EALs) [5]. At the highest EAL, EAL7, the application must be formally specified, and must be proven to meet its specification. This can be a very costly and time-consuming process. One of the continuing research goals at Rockwell Collins is to improve secure system evaluation – measured in terms of completeness, human effort required, time, and cost – through the use of highly automated formal methods. In support of this goal, we have developed practical techniques for formal computer system platform modeling and analysis utilizing the ACL2 theorem prover (e.g., [9], [16]). This work has led to an EAL7 level MILS certification for the Rockwell Collins AAMP7G microprocessor [17], as well as an EAL6+ certification for the Green Hills INTEGRITY-178B Real-Time Operating System (RTOS) kernel, executing on a PowerPC-based single-board computer [8].

Once such a formal model of the computing platform has been developed, one can begin to reason about applications that execute on that platform. Not surprisingly, classical data structures such as stacks, queues, and double-ended queues (deques) find broad application in security-critical applications. In our experience, we have found that deques are particularly common, as they are employed in device drivers, audit queueing, etc. We have also observed that deque implementation can be error-prone. Thus, we have investigated techniques for the formal verification of classical data structure implementations, focusing on queueing structures.

Formal verification systems can readily reason about unbounded, functional data structures. However, such data structures are in the main not appropriate for high-confidence software systems. Purely functional data structure implementations produce a new copy of the data structure for each mutator operation. This tends to generate garbage, necessitating some form of garbage collection, with the performance impact, increased memory usage, and additional runtime support that this entails. Moreover, the garbage collector itself would have to be formally verified. Moreover, purely functional implementations of common data structures (queues, deques, etc.) can be difficult for engineers and evaluators to understand; even simplified approaches [14, 15] are quite subtle. Unsurprisingly, subtle implementations are generally frowned upon by the high-confidence software community.

2. HIGH-ASSURANCE SOFTWARE DEVELOPMENT VS. FORMAL VERIFICATION

High-Assurance software development methodologies (e.g., SPARK [1]) vary somewhat in their details, but can be generally characterized by the following:

- Static programming language subset.
- Fixed-size data structures.
- No recursion.
- Straightforward implementation.
- Freedom from exceptions.

By contrast, formal verification environments largely support the following methodologies:

- Dynamic programming language subset.
- Functional data structures.
- Recursion.
- Subtle implementation.

One goal of our research program, then, is to find a way to bridge the gap between the formal verification environment and the high-assurance implementation environment, allowing us to implement verifiable data structures, as well as to verify implementable data structures.

3. A VERIFICATION ENVIRONMENT: THE ACL2 THEOREM PROVER

We utilize the ACL2 theorem proving system [11] for much of our high-assurance verification work, as it best presents a single model for formal analysis and simulation. ACL2 also provides a highly automated theorem proving environment for machine-checked formal analysis, and its logic is an applicative subset of Common Lisp. The fact that ACL2 reasons about a real programming language suggests that ACL2 could be an appropriate choice for data structure verification work. An additional feature of ACL2, single-threaded objects, adds to its strength as a vehicle for reasoning about data structures, as will be detailed in the following sections.

3.1 ACL2 Single-Threaded Objects

ACL2 enforces restrictions on the declaration and use of specially-declared structures called single-threaded objects, or stobjs [4]. ACL2 enforces strict syntactic rules on stobjs to ensure that they are not copied; thus, "old" states of a stobj are guaranteed not to exist. This property means that ACL2 can provide destructive implementation for stobjs, allowing stobj operations to execute quickly. In short, an ACL2 single-threaded object combines a functional semantics about which we can readily reason, utilizing ACL2's powerful heuristics, with a relatively high-speed imperative implementation that more closely follows "normal" design rules for high assurance.

4. A DEQUE IMPLEMENTATION USING AN ACL2 SINGLE-THREADED OBJECT

In this section, we describe a deque implementation using an ACL2 single-threaded object. This is not intended to be the ultimate deque implementation; indeed, many alternative designs are possible. We chose to implement the deque in a way that would be typical for a high-assurance developer; thus, we did not allow the deque to be resizable, even though the stobj framework allows this. We also did not attempt to achieve optimal time bounds for all operations, but rather implemented the deque in such a way that fast block data move instructions could be used. Finally, we iterated on the design a couple of times to achieve an implementation that could be processed by ACL2 without any involved induction hints, etc., or appealing to anything more than the most common arithmetic reasoning facilities. In particular, we did not use modular arithmetic to maintain the head and tail indexes, for example, but rather used ACL2 to establish that the arithmetic that we performed would never cause an overflow.

Also note that we have implemented stack and single-ended queue data structures using a similar technique, and the reader can well imagine how this technique could extend to other common data structures.

4.1 Deque Definitions

First, we present the basic single-threaded object declaration for a deque:

```
;; Fixed-size deque.

(defstobj dqst
  (arr :type (array t (2048)) :initially (empty))
  (hd  :type (unsigned-byte 11) :initially 0)
  (tl  :type (unsigned-byte 11) :initially 0))
```

In this declaration, `hd` is the index of the front of the deque, `tl` is the index of the back of the deque plus one, and `arr` is the array of deque data. In this example, both `hd` and `tl` are declared to be unsigned 11-bit values, but this type information is used only to optimize code generation, and is ignored by the logic. Since `tl` indexes one beyond the back of the queue, the maximum capacity of the deque is the declared size of the array minus one. So, in the example above, the maximum capacity of the deque is 2047. This maximum capacity value was chosen somewhat arbitrarily, but is representative of the sorts of queue sizes that one encounters in high-assurance systems. One of the advantages of the ACL2 stobj approach is that this maximum capacity can be increased to a value much larger than what is seen in practice without noticeably affecting proof times or execution times.

`defstobj` defines a number of predicate, accessor and updater functions for the elements of the stobj. For example, `(hd dqst)` returns the index of the head; `(arri i dqst)` returns the ith element of the deque array; `(update-tl x dqst)` updates the value of the tail index to `x`; `(update-arri i val dqst)` updates the deque at array element `i` to `val`; and `(dqstp dqst)` indicates whether its argument is a deque.

Next we present some basic predicates and accessors on the deque. Not all operators that have been implemented are shown for the sake of brevity. All functions that accept the deque stobj as a parameter must have a `declare` form that so indicates.

Note that all operators are defined using ACL2's `defund`, which means "define the function and then disable the definition". Disabling the definition prevents the theorem prover from performing unnecessary expansions, limiting its search, and often making the difference between a successful and an unsuccessful proof attempt. In the theorems that follow, we will often see hints that enable definitions for the functions whose definitions are truly needed (e.g., it is not enough just to know that the function accepts a deque and returns a natural).

```
(defund size-of (dqst)
  (declare (xargs :stobjs dqst))
  (if (> (tl dqst) (hd dqst))
      (- (tl dqst) (hd dqst))
      0))

(defund is-empty (dqst)
  (declare (xargs :stobjs dqst))
  (equal (hd dqst) (tl dqst)))

(defund get-first (dqst)
  (declare (xargs :stobjs dqst
                  :guard-hints (("Goal" :in-theory
                                 (enable is-empty)))))
  (if (is-empty dqst)
      (empty)
      (arri (hd dqst) dqst)))
```

`contains` is a more complex predicate that computes whether its input element `e` is contained within the deque. The `contains` predicate performs the search for `e` using a "helper" function `index-of-from-back`, starting at index `i`, usually set to the size of the deque, and decrementing toward the front of the deque. `contains` is interesting in that we must first prove a theorem about the return type of the helper function before `contains` can be admitted into the logic.

```
(defund index-of--from-back (i e dqst)
  (declare (xargs :stobjs dqst :guard (natp i)
                  :guard-hints (("Goal" :in-theory
                                 (enable size-of)))))
  (if (or (zp i) (> i (size-of dqst)))
      0 ;; Nothing at the 0th index from the tail --
        ;; failure indication
      (if (equal e (arri (+ (hd dqst) (- i 1)) dqst))
          (- (size-of dqst) (- i 1))
          (index-of--from-back (- i 1) e dqst))))

(defthm iofb-nat--thm
  (implies (and (natp i) (dqstp dqst))
           (natp (index-of--from-back i e dqst)))
  :hints (("Goal" :in-theory
           (enable index-of--from-back size-of))))
```

```
(defund contains (e dqst)
  (declare (xargs :stobjs dqst
                  :guard-hints
                  (("Goal" :in-theory
                           (enable index-of--from-back
                                   size-of)))))
  (/= 0 (index-of--from-back (size-of dqst) e dqst)))
```

Finally, we come to the add and remove mutators, two of which, `add-first` and `remove-last`, are described herein. These operations are difficult to implement in a traditional purely functional environment, as the deque is typically implemented as two lists, one of which must be reversed [14, 15]. The `shift-up-to` helper function assists `add-first` in the case where the deque is not full, but the head equals 0, requiring that room be made at the head. Even though `shift-up-to` is a non-trivial recursive function, ACL2 readily performs the termination analysis necessary to admit this function into the logic without any user intervention.

Both `add-first` and `remove-last` make use of the `seq` macro, which automatically provides the let-bindings that are needed to have one stobj operation "follow" another, without cluttering the source code. `seq` is defined as follows:

```
(defmacro seq (stobj &rest rst)
  (cond ((endp rst) stobj)
        ((endp (cdr rst)) (car rst))
        (t `(let ((,stobj ,(car rst)))
              (seq ,stobj ,@(cdr rst))))))
```

The `seq` macro was originally authored by J Moore, and we thank him for it.

The definitions of `add-first` and `remove-last`, along with the helper function `shift-up-to`, follow. `shift-up-to` is used to make room at the head of the deque so that the new element `e` can be added.

```
;; Shift up to the ith offset from the head
(defund shift-up-to (i dqst)
  (declare (xargs :stobjs dqst :guard (natp i)))
  (if (or (zp i) (>= (+ (hd dqst) i) (tl dqst)))
      dqst
      (seq dqst
        (update-arri (+ (hd dqst) i)
                     (arri (+ (hd dqst) (- i 1)) dqst)
                     dqst)
        (shift-up-to (- i 1) dqst))))

(defund add-first (e dqst)
  (declare (xargs :stobjs dqst
                  :guard-hints
                  (("Goal" :in-theory
                           (enable size-of is-full
                                   is-full-front)))))
  (if (or (equal e (empty))
          (is-full dqst)
          (< (tl dqst) (hd dqst)))
      dqst
```

```
      (if (not (is-full-front dqst))
          (seq dqst
            (update-hd (- (hd dqst) 1) dqst)
            (update-arri (hd dqst) e dqst))
          (seq dqst
            (update-tl (+ (tl dqst) 1) dqst)
            (shift-up-to (- (size-of dqst) 1) dqst)
            (update-arri 0 e dqst)))))

(defund remove-last (dqst)
  (declare (xargs :stobjs dqst
                  :guard-hints
                  (("Goal" :in-theory
                           (enable size-of)))))
  (if (= (size-of dqst) 0)
      dqst
      (seq dqst
        (update-tl (- (tl dqst) 1) dqst)
        (update-arri (tl dqst) (empty) dqst))))
```

4.2 Deque Theorems

Once we have defined the deque single-threaded object; defined a number of predicates, accessors, and updaters for the deque; and admitted these operations into the logic, we can now begin to prove theorems about the data structure.

We begin by introducing an important relation between `tl` and `hd` that we expect all deque operations to maintain:

```
(defun tail-head-relation (dqst)
         (declare (xargs :stobjs dqst))
         (and (natp (hd dqst))
              (natp (tl dqst))
              (<= (hd dqst) (- (maxnode) 1))
              (<= (tl dqst) (- (maxnode) 1))
              (<= (hd dqst) (tl dqst))))
```

We then proceed to prove that this relation in fact holds for all deque operations. A sample theorem of this sort is given below:

```
(defthmd add-first--tl-hd--thm
  (implies (tail-head-relation dqst)
           (tail-head-relation (add-first e dqst)))
  :hints (("Goal" :in-theory
           (e/d (add-first size-of is-full
                 is-full-front)
                (update-arri)))))
```

This theorem states that if the `tail-head-relation` defined above holds for the deque before the execution of `add-first`, it will hold afterward. `e/d` is a hint to enable the items in the first list following the `e/d` verb, while disabling the items in the second list.

Additionally, we can prove functional correctness theorems for compositions of operations on the deque, of the sort stated below:

```
(defthm add-first--get-first--thm
  (implies (and (not (is-full dqst))
                (not (= e (empty)))
                (not (< (tl dqst) (hd dqst)))
                (dqstp dqst))
           (equal (get-first (add-first e dqst)) e))
  :hints (("Goal"
           :in-theory (e/d (size-of add-first
                            get-first is-full
                            is-full-front
                            shift-up-to
                            is-empty)
                           (update-arri)))))
```

This theorem states that if certain preconditions are met, `(get-first (add-first e dqst))` equals the element `e` that was added.

Several correctness theorems of the form above were developed for the various deque operations; in all, over 50 theorems were stated and proved over the course of a two-week period. All theorems are proved without human assistance within seconds on a modern CPU.

5. TRANSLATION TO CONVENTIONAL PROGRAMMING LANGUAGES

In order to utilize the data structures we have developed in a typical high-assurance development environment, we must first translate the source code from ACL2 to a more conventional programming language. Note that it is possible to compile the ACL2 source to machine code (indeed, this is done routinely as part of ACL2's book certification process), and utilize that machine code directly on the target system. However, utilizing ACL2 as a source language would be difficult for most developers and evaluators, and the code generated, while effective in many ways (e.g., tail recursion is compiled into iteration), would be difficult to utilize as standalone object modules.

We have hand-translated our ACL2 stobj-based data structures into SPARK Ada [1]; a static subset of Java [7]; and C [10]. In fact, due to the static nature of the Java translation, the C and Java versions are in fact very similar. Most of the translation work could have been performed automatically; the biggest challenge was converting recursion to iteration. (Note that converting recursion to iteration is not needed when the compiler used supports tail recursion elimination.) The latter effort was aided in the case of Java by the fact that the recursive implementations of `shift-up-to` and `shift-down-to` could be replaced by calls to `java.lang.System.arraycopy()` (in C, `memmove()` could be used to the same effect).

Even with a highly automated translation, a "code-to-spec review" of the sort typically conducted by government evaluators (as discussed in [9]) would still need to be held in order to ensure that the specification that is reasoned about (the ACL2 stobj specification) corresponds to the code that is generated.

The example that follows shows a portion of the C translation. The C version was chosen because C is familiar to

most readers, and it affords us an opportunity to discuss formally verified machine code generation for a formally verified operating system (see Section 6). Note that in the stobj formulation, the elements of the underlying array used to implement the deque have an unrestricted type specification (`:type (array t ...)`), but in this translation we have chosen to restrict the array elements to type `int *`. Other types (e.g., `int`) would have done just as well (indeed, in practice, one may well create multiple translations from the ACL2 specification with different types for the array elements as needed), but we wanted to exercise the basic pointer-handling capability of a new C subset compiler, discussed further in Section 6.

```
#define MAXNODE  2048

int *arr[MAXNODE];
int hd = 0;
int tl = 0;

int sizeOf() {
    if (tl > hd) {
        return tl - hd;
    } else {
        return 0;
    }
}

int isEmpty() {
    return hd == tl;
}

int *getFirst() {
    if (isEmpty()) {
        return empty();
    } else {
        return arr[hd];
    }
}
```

6. MACHINE CODE TRANSLATION, AND A VERIFIED "STACK"

Once we have the data structure translated into a "conventional" source language, the translated implementation can then be compiled into the machine code of choice for execution on the preferred operating system and CPU. Fortunately, within the past couple of years, a verified compiler for a significant subset of C has been developed by Xavier Leroy at INRIA as part of the CompCert project ([3], [12]). And as it happens, the CompCert compiler generates code for the PowerPC architecture, which was also the target CPU architecture for the Green Hills INTEGRITY-178B RTOS verification effort [8], in addition to being a common CPU type for embedded high-assurance systems. These developments present a new opportunity for a formally verified "stack" (inspired by the "CLInc Stack" [2]) from the application layer all the way down to the CPU.

The CompCert compiler is verified using the Coq theorem prover [6]. (Thus, the verified "stack" described herein is admittedly not as well-integrated as the CLInc Stack.) Although Coq and ACL2 are very different systems, a Coq specification is similar in spirit to one written in ACL2, in that it comprises an operational semantics written in a functional programming language (in the case of Coq, that language is Gallina, an ML-like language). Unlike ACL2, however, the Coq system does not provide an efficient means to execute Gallina programs directly; rather, the Coq system provides a translation to OCaml for this purpose.

Data given by Leroy [12] as well as our own experiments with compiling stobj-translated C code indicate that the CompCert compiler generates reasonably efficient PowerPC code, as can be seen in the compilation output for the deque `getFirst` function below.

```
    .text
    .align  2
    .globl  _getFirst
_getFirst:
    stwu    r1, -64(r1)
    mflr    r2
    stw     r2, 12(r1)
    bl      _isEmpty
    cmpwi   cr0, r3, 0
    bf      2, L102
    addis   r2, 0, ha16(_hd)
    lwz     r3, lo16(_hd)(r2)
    rlwinm  r3, r3, 2, 0xfffffffc
    addis   r2, r3, ha16(_arr)
    lwz     r3, lo16(_arr)(r2)
L103:
    lwz     r2, 12(r1)
    mtlr    r2
    lwz     r1, 0(r1)
    blr
L102:
    bl      _empty
    b       L103
```

One area that is currently lacking with the CompCert compiler is full tail call optimization for C; this will be addressed in a future version of the compiler [13]. Support for tail call elimination would allow us to enjoy a more direct translation of our data structures from ACL2 stobj form to C.

7. CONCLUSION

We have demonstrated how formally verified data structures, specifically queueing data structures of the sort commonly employed in high-assurance system design, and initially developed in a functional programming/automatic theorem prover environment, can be readily translated into high-assurance implementations expressed in conventional programming languages, through the use of ACL2 single-threaded objects. One could also imagine the reverse process, in which a data structure written in a conventional programming language is translated into ACL2 using the single-threaded object facility, and then analyzed for correctness. In either case, we are able to establish the correctness of practical data structure implementations commonly found in high-assurance systems with a minimum of formal verification effort.

The translation from ACL2 to conventional programming

languages has been done by hand to date; but the mechanization of the process would not be difficult, and could be carried out within ACL2 itself, via a "pretty-printing" process. Future work will address this mechanization, and also expand on the types of data structures to be implemented and analyzed.

Finally, we have shown how this translated data structure code can be compiled into efficient machine code for a common embedded microprocessor using a verified compiler, and executed on an EAL6+ verified operating system, thus providing a new opportunity for a formally verified "stack" from the application layer all the way down to the CPU.

8. ACKNOWLEDGMENTS

Many thanks to J Moore and Matt Kaufmann at the University of Texas at Austin for their continuing development of ACL2 and ACL2 single-threaded objects, and to Xavier Leroy of INRIA for his prompt answers to our questions on the CompCert compiler. Thanks also go to Dave Greve, Matt Wilding, and Ray Richards at Rockwell Collins for their encouragement, and to the anonymous referees for their helpful comments.

9. REFERENCES

[1] John Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[2] William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[3] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, January 2009. http://gallium.inria.fr/~xleroy/publi/Clight.pdf.

[4] Robert S. Boyer and J Strother Moore. Single-threaded objects in ACL2. *PADL 2002*, 2002.

[5] Common Criteria Project. *Common Criteria for Information Technology Security Evaluation*, September 2006. http://www.commoncriteriaportal.org.

[6] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.1*, 2006. http://coq.inria.fr/V8.1/refman/index.html.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, Reading, Massachusetts, 2005.

[8] Green Hills Software, Inc. *Green Hills Software Announces World's First EAL6+ Operating System Security Certification*, November 2008. http://www.ghs.com/news/20081117_integrity_EAL6plus_security.html.

[9] David Greve, Raymond Richards, and Matthew Wilding. A Summary of Intrinsic Partitioning Verification. In *ACL2 Workshop 2004*, November 2004.

[10] Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual*. Prentice Hall, 2002.

[11] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[12] Xavier Leroy. A formally verified compiler back-end. http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf, July 2008.

[13] Xavier Leroy. Personal Communication, January 2009.

[14] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[15] Chris Okasaki. Functional data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, pages 40–1 – 40–17. CRC Press, 2005.

[16] Raymond Richards, David Greve, Matthew Wilding, and W. Mark Vanfleet. The Common Criteria, Formal Methods and ACL2. In *ACL2 Workshop 2004*, November 2004.

[17] Rockwell Collins, Inc. *Rockwell Collins receives MILS certification from NSA on microprocessor*, August 2005. http://www.rockwellcollins.com/news/page6237.html.