2013 46th Hawaii International Conference on System Sciences

The Specification, Verification, and Implementation of a High-Assurance Data Structure: An ACL2 Approach

David S. Hardin
Trusted Systems Group
Rockwell Collins Advanced Technology Center
Email: dshardin@rockwellcollins.com

***Abstract*—We present a complete specification and formal verification of a high-assurance data structure, namely an array-based set (or alternatively, a multiset), of arbitrary size, using the ACL2 theorem prover. This particular data structure is a sanitized version of one that was used in a high-assurance development at Rockwell Collins. We additionally demonstrate how this high-assurance specification can be readily translated to an implementation in a traditional, imperative programming language. The specification is accomplished via the ACL2 single-threaded object (stobj) formulation, as the stobj yields the most straightforward translation to traditional programming language implementations, as well as high-performance, scalable executable specifications within the ACL2 system itself. Use of the stobj is known to present certain difficulties in proof development, but we describe a simple means to access the underlying representation of the stobj, which makes reasoning much more straightforward. We discuss characteristics of ACL2 that aided the proofs, as well as ones that presented challenges. We also compare the ACL2 proof results with other verification efforts previously attempted on this data structure using model checking and symbolic execution.**

I. Introduction

Security-critical applications are commonly certified according to the Common Criteria at the highest Evaluation Assurance Levels (EALs) [5]. At the highest assurance level, the application must be formally specified, and must be proven to meet its specification [16], [15]. This can be a very costly and time-consuming process. In order to evaluate formal methods techniques to improve secure system evaluation – measured in terms of completeness, human effort required, time, and cost – Rockwell Collins researchers have occasionally formulated challenge problems, which are sanitized versions of functionality found in our own high-assurance designs.

Not surprisingly, classical data structures such as stacks, queues, double-ended queues (deques), and sets find broad application in security-critical applications. 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 [13], [14] are quite subtle. Unsurprisingly, subtle implementations are generally frowned upon by the high-confidence software community. Thus, in our challenge problem formulations, we eschew unbounded, purely functional datatypes in favor of something more realistic.

II. 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 (as per [6]):

- 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.

III. The ACL2 Theorem Prover

We utilize the ACL2 theorem proving system [10] for much of our high-assurance verification work, as it best presents a single model for formal analysis and simulation (N.B.: high-speed simulation is very useful in the validation of computer system models). ACL2 provides a highly automated theorem proving environment for machine-checked formal analysis, and its logic is an applicative subset of Common Lisp [11]. The fact that ACL2 reasons about a real programming language

1530-1605/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2013.541

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 functional data structures, as will be detailed in the following sections.

A. ACL2 Single-Threaded Objects

ACL2 enforces restrictions on the declaration and use of specially-declared structures called single-threaded objects, or stobjs [3]. From the perspective of the ACL2 logic, a stobj is just an ordinary ACL2 object, and can be reasoned about in the usual way. Ordinary ACL2 functions are employed to "access" and "update" stobj fields (defined in terms of the list operators `nth` and `update-nth`). However, ACL2 enforces strict syntactic rules on stobjs to ensure that "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.

IV. An Array-Based Set/Multiset Implementation using an ACL2 Single-Threaded Object

In this section, we describe an array-based set/multiset implementation using an ACL2 single-threaded object, similar to the approach taken in [6]. This implementation is based on a challenge problem formulated in SPARK, which was in turn a sanitization of source code produced from a Simulink model created during a Rockwell Collins high-assurance development [7]. Thus, we chose to implement the set using the stobj in as similar a manner to the SPARK code as possible; for example, we did not allow the set to be resizable, even though the stobj framework allows this. Finally, we iterated on the details of the design in order to achieve a specification that could be reasoned about without undo pain; this led us initially to slightly de-optimize the `get-element` function, changing it to linearly scan through the `elements` array rather than utilizing the `next` array to proceed to the next used element. However, no changes were made that jeopardized the overall fidelity to the original challenge problem – indeed, the `get-element` function was subsequently re-optimized, using the `find-via-next-list` function described later in this paper.

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. Recently, as a further study of the basic "stobj formulation and replacement" method described herein, we formalized the efficient sparse set data structure described by Briggs and Torczon [4], and were able to complete the proofs of basic correctness theorems in just two days (granted, the array-based set described in [4] is somewhat simpler than the one described here, although it is similar in requiring two equal-length arrays, as well as an unsigned scalar, which holds the element count). Along the way, we identified needed predicates for the `member()`, `add_member()`, and `delete_member()` functions that the C code depicted in [4] failed to include.

A. ArraySet Definitions

First, we present the basic single-threaded object declaration for the ArraySet.

```
;; Set to t for multiset operation
(defconst *MULTISETP* nil)

(defconst *max-elements* 16)

(defmacro terminator () *max-elements*)

(defmacro null-value () -1)

(defstobj arrayset
  (elements
     :type (array (integer -1 *)
                  (*max-elements*))
     :initially -1)
  (next
     :type (array (integer 0 16)
                  (*max-elements*))
     :initially 0)
  (free-head
     :type (integer 0 16)
     :initially 0)
  (used-head
     :type (integer 0 16)
     :initially 16))
```

First note that the entire specification is parameterized by the `*MULTISETP*` boolean (throughout, we utilize the Lisp convention of adding the 'p' suffix to names to indicate a boolean-valued predicate); if true ('t' in Lisp parlance), the specification allows multiple elements of the same value to be stored in the data structure, yielding a multiset, or bag ; if false ('nil' in Lisp), multiple values are not allowed (a traditional set).

In this declaration, `elements` holds the set elements, `next` contains indices indicating the next element in either the free or used list, `free-head` is the index of the first element of the free list, and `used-head` is the index of the first element of the used list.

The ingenious bit about this particular array-backed set implementation is the use of a single `next` array to hold both the free list and the used list. Each element of the `next` array is in one of the two lists, but not both. The free list and used list are both terminated by elements in the next array with the value `terminator`. Any value outside the range of valid element indices would do for the terminators; we have used `max-elements`. One of the jobs of the mutators `init`, `add-element`, and `delete-element` is to maintain the

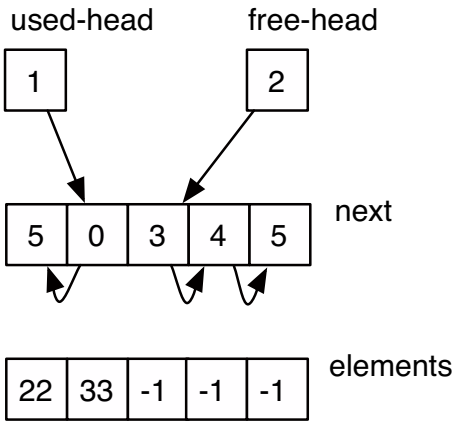

Fig. 1. An ArraySet data structure with contents {33, 22}, size = 5

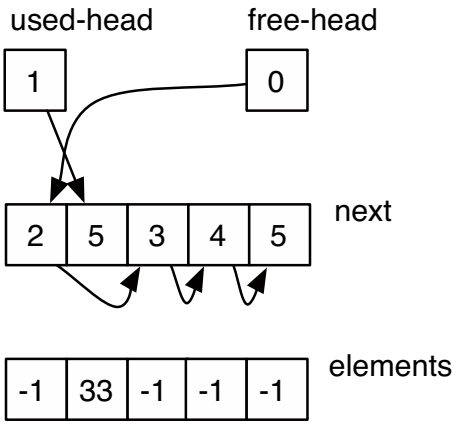

Fig. 2. The ArraySet after deleting element 22

integrity of the free list and used list contained within the single `next` array, and it is a primary obligation of the ACL2 proofs to show that this is the case.

Figure 1 shows the state of an ArraySet of size 5 after some number of `add-element` and `delete-element` operations have been performed. The `used-head` is 1, which is an index into the `next` array (note that all arrays are zero-based). The `free-head` is 2. If we read the `next` array at index `used-head`, we get the next element in the used list, namely 0. If we then read the `next` array at index 0, we get 5, which is the terminator; thus, we have arrived at the end of the used list. The corresponding elements in the `elements` array are 33 and 22; thus, the ArraySet content is {33, 22} (note that all other components of `elements` have the value -1, indicating free elements).

On the other hand, if we traverse the `next` array starting at the `free-head` index, we get 3. Following the free list, we read the third element of the `next` list, which is 4. Reading index 4 of the `next` list, we get 5, which is the terminator. Thus, we have reached the end of the free list.

If we then execute `(delete-element 22 arrayset)`, we obtain the ArraySet shown in Figure 2. As one can readily observe, the used list is shortened by one, and the free list is lengthened by one, all using elements from the single `next` array.

`defstobj` defines a number of predicate, accessor and updater functions for the elements of the stobj. For example, `create-arrayset` creates a new arrayset; `elementsp` is a recognizer for the elements array; `(elementsi i arrayset)` returns the ith element of the elements array; `(update-elementsi i val arrayset)` updates the value of the elements array at index `i` to `val`; and `(arraysetp arrayset)` indicates whether its argument is an `arrayset`.

Next we present some basic predicates and accessors on the ArraySet. Not all operators that have been implemented are shown for the sake of brevity. But before we do so, we must admit to one essential transformative "trick" that we play on our stobj formulation. Anyone who has attempted to reason using stobjs knows that it is difficult at best; one likely will end up with expressions that are a mish-mash of `car`, `caar`, `cdr`, `cadr`, etc., `nth`, and `update-nth`, as well as objects whose definitions are, well, opaque (evaluate `arrayset` at the ACL2 command line, and one will get `<arrayset>`). Fortunately, lead ACL2 developer Matt Kaufmann provided a key insight: use the ACL2 `:pcb! :max` command to print out the stobj-related definitions right after introducing the `defstobj` form, grab the printed output and modify the function definitions slightly to change the size from some fixed integer value to an arbitrary `max`, and the full power of ACL2 is once again available.

For example, consider the `create-arrayset` function that the `defstobj` synthesizes:

```
(DEFUN CREATE-ARRAYSET NIL
  (DECLARE (XARGS :GUARD T
                  :VERIFY-GUARDS T))
  (LIST (MAKE-LIST 16
                   :INITIAL-ELEMENT '-1)
        (MAKE-LIST 16
                   :INITIAL-ELEMENT '0)
        '0 16))
```

This can be readily edited to produce a `CREATE-ARRAYSET` that works for arbitrary set size:

```
(DEFUN CREATE-ARRAYSET (max)
  (DECLARE (XARGS :GUARD (natp max)
                  :VERIFY-GUARDS T))
  (LIST (MAKE-LIST max
                   :INITIAL-ELEMENT '-1)
        (MAKE-LIST max
                   :INITIAL-ELEMENT '0)
        '0 max))
```

So, with this bit of trickery, we dispense with the `defstobj` altogether, and from now on, work with the new, arbitrary-sized sets. For example, Figure 3 details the "post-

```
(defun add-element (val arrayset)
  (declare (xargs :guard (and (arraysetp arrayset) (natp val))))
  (cond ((not (natp val)) arrayset)
        ((not (arraysetp arrayset)) arrayset)
        ((= (elements-length arrayset)
            (free-head arrayset))
         arrayset)
        ((and (not *MULTISETP*) (elementp val arrayset))
         arrayset)
        (t
         (let ((curr-index (free-head arrayset)))
           (seq arrayset
                (update-free-head (nexti (free-head arrayset) arrayset) arrayset)
                (update-elementsi curr-index val arrayset)
                (update-nexti curr-index (used-head arrayset) arrayset)
                (update-used-head curr-index arrayset))))))
```

Fig. 3. add-element function, after stobj elimination

stobj" `add-element` function.

The operation of `add-element` is as follows. Following some recapitulation of the guard conditions, if the free head has the value of the terminator, then there is no room, and no change is made to the ArraySet. Also, if the element `val` to be added is already in the set (and the multiset option is not set), then the original ArraySet is returned. Otherwise, we adjust the free head to the next element in the free list, insert the new `val` at the element index indicated by the old free head, copy the old free head value to the used head position in the next array, and finally set the used head to the old free head. Aside from the obvious syntactic differences, this function is a fairly straightforward translation of the SPARK version, even though the SPARK version is imperative, and the ACL2 version is functional.

`delete-element` proceeds similarly, although it is admittedly a bit more complicated (see Figure 4). `delete-element`, in turn, depends on the `find-via-next-list` function (see Figure 5), which searches for an element with a given value by traversing the `next` array, and examining values using corresponding indices into the `elements` array. Note that `find-via-next-list`, while a recursive function, is written in tail-recursive style so that it can be compiled to an efficient iterative form. Otherwise, this function is very much as one would write it in an imperative programming language (N.B.: `cond` is a multi-way conditional in Lisp), albeit with a few more type-check predicates (and these are optimized away for execution using the information in the `declare` form).

One aspect of `find-via-next-list` that *is* unusual is the `countdown` parameter. Any function that is to be admitted into the ACL2 environment must first be proved to terminate. Much of the time, ACL2 does a very good job of proving termination without any help from the user. In the case of `find-via-next-list`, however, the function may actually fail to terminate without the addition of the `countdown` parameter. Consider, for example, if the `next` array contains a cycle that comprises several elements of that array. Without the `countdown` parameter, `find-via-next-list` might very well traverse that cycle forever. Callers of `find-via-next-list` typically set the value of the `countdown` parameter to the length of the `next` array; `countdown` is then decremented on successive recursive calls to `find-via-next-list`, and is tested against zero (using the ACL2 `zp` predicate) each time. If that test succeeds, the function immediately returns a failure condition (any return value less than 0 is a failure condition). The requirement to prove termination is one of the most vexing aspects of AC2 for the novice user, but as this case demonstrates, it arguably results in higher assurance functions.

Both `add-element` and `delete-element` make use of the `seq` macro, which automatically provides the let-bindings that are needed to have one 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 for use with stobjs, but it works equally well in our new "post-stobj" environment.

B. ArraySet Theorems

Once we have defined the ArraySet single-threaded object; applied the `:pcb! :max` trick; defined a number of predicates, accessors, and updaters for the "post-stobj" ArraySet;

```
(defun delete-element (val arrayset)
  (declare (xargs :guard (and (arraysetp arrayset) (natp val))))
  (cond
    ((not (natp val)) arrayset)
    ((not (arraysetp arrayset)) arrayset)
    ((= (used-head arrayset) (elements-length arrayset)) arrayset)
    ((= (elementsi (used-head arrayset) arrayset) val)
      (let ((curr (used-head arrayset)))
        (seq arrayset
          (update-elementsi curr (null-value) arrayset)
          (update-used-head (nexti curr arrayset) arrayset)
          (update-nexti curr (free-head arrayset) arrayset)
          (update-free-head curr arrayset))))
    (t
       (let ((prev-index (find-via-next-list
                              val
                              (elements-length arrayset)
                              (used-head arrayset)
                              (elements-length arrayset)
                              nil
                              arrayset)))
         (cond
            ((< prev-index 0) arrayset)
            ((>= prev-index (elements-length arrayset)) arrayset)
            (t
               (let ((curr-index (nexti prev-index arrayset)))
                 (cond
                   ((< curr-index 0) arrayset)
                   ((>= curr-index (elements-length arrayset)) arrayset)
                   (t
                      (seq arrayset
                        (update-elementsi curr-index (null-value) arrayset)
                        (update-nexti prev-index (nexti curr-index arrayset) arrayset)
                        (update-nexti curr-index (free-head arrayset) arrayset)
                        (update-free-head curr-index arrayset)))))))))))
```

Fig. 4. delete-element function, after stobj elimination

and admitted these operations into the logic, we can now begin to prove theorems about the data structure implementation.

We begin by introducing an important relation between the free head and the used head that we expect all operations to maintain:

```
(defun free-head-used-head-relation
         (arrayset)
  (and (arraysetp arrayset)
       (not (= (free-head arrayset)
               (used-head arrayset)))))
```

We also define additional good-state functions, of the sort shown below. This particular function, `next-array-free-head-used-head-good`, states that for a good state, the `arrayset` input satisfies the `arraysetp` predicate; there are no self-loops in the `next` array (i.e., the value of the `next` array at index i does not equal i); and if the `free-head` and `used-head` are not equal to the terminator value (which is the length of the `elements` and `next` arrays), then the `free-head` and `used-head` values are not to be found in the `next` array (N.B.: `member-equal` is an ACL2 variant of the Common Lisp `member` function).

```
(defun next-array-free-head-used-head-good
         (arrayset)
  (and (arraysetp arrayset)
       (no-self-loopsp (next arrayset))
       (if (not (= (free-head arrayset)
                   (elements-length
                     arrayset)))
           (not (member-equal
                  (free-head arrayset)
```

```
(defun find-via-next-list (val countdown index term f-u arrayset)
  (declare (xargs :guard (and (integerp val) (natp countdown) (integerp index)
                              (booleanp f-u) (integerp term) (arraysetp arrayset))))
  (cond
     ((not (arraysetp arrayset)) -1)
     ((= (elements-length arrayset) 0) -2)
     ((not (integerp countdown)) -3)
     ((< countdown 0) -4)
     ((not (integerp index)) -5)
     ((not (integerp term)) -6)
     ((not (integerp val)) -7)
     ((equal val (null-value)) -8)
     ((> countdown (elements-length arrayset)) -9)
     ((< index 0) -10)
     ((equal index term) -11)
     ((>= index (elements-length arrayset)) -12)
     ((zp countdown) -13) ;; indicates a cycle
     ((xor f-u (equal (elementsi index arrayset) (null-value))) -14)
     ((< (nexti index arrayset) 0) -15)
     ((>= (nexti index arrayset) (elements-length arrayset)) -16)
     ((equal (nexti (nexti index arrayset) arrayset) index) -17) ;; also a cycle
     ((equal (elementsi (nexti index arrayset) arrayset) val) index)
     (t (find-via-next-list val (1- countdown) (nexti index arrayset)
                            term f-u arrayset))))
```

Fig. 5. find-via-next-list function

```
                (next arrayset)))
         t)
      (if (not (= (used-head arrayset)
                  (elements-length
                    arrayset)))
          (not (member-equal
                 (used-head arrayset)
                 (next arrayset)))
          t)))
```

Another good-state-related function is `size-free-plus-size-used-correct`, which totals the number of elements on the free list plus the number of elements on the used list, and checks to see if that number equals the total number of elements in the ArraySet, as follows (`size-via-next-list` is not presented here, but is implemented in a manner similar to `find-via-next-list`):

```
(defun
  size-free-plus-size-used-correct
    (arrayset)
       (= (+ (size-via-next-list
                (elements-length arrayset)
                (free-head arrayset)
                (elements-length arrayset)
                t
                (elements arrayset)
                (next arrayset))
             (size-via-next-list
               (elements-length arrayset)
               (used-head arrayset)
               (elements-length arrayset)
               nil
               (elements arrayset)
               (next arrayset)))
          (elements-length arrayset)))
```

The `good-state` function is then defined as follows. It determines whether the `arrayset` is well-formed; that the `free-head` is not equal to the `used-head`; that the two functions described above return t (true); and that if one combines the `free-head`, `used-head`, and `next` array into a single list, that all natural numbers up to the size of the `next` list minus 1 appear once in that list, and that the natural number corresponding to the maximum size of the arrayset appears twice (the two free list and used list terminators).

```
(defun good-state (arrayset)
  (and
    (arraysetp arrayset)
    (free-head-used-head-relation
      arrayset)
    (next-array-free-head-used-head-good
      arrayset)
    (size-free-plus-size-used-correct
```

```
    arrayset)
  (next-free-used-count-good
    (len (next arrayset))
    (list* (free-head arrayset)
           (used-head arrayset)
           (next arrayset)))))
```

Given this definition of a good ArraySet state, we can prove functional correctness theorems for the ArraySet operations, of the sort stated below (theorems in ACL2 are stated using the `defthm` form):

```
(defthm add-element-works--thm
  (implies
    (and (good-state arrayset)
         (not
           (= (elements-length arrayset)
              (free-head arrayset)))
         (not (elementp val arrayset))
         (natp val))
    (elementp
       val
       (add-element val arrayset))))
```

This theorem is a logical implication stating that if certain preconditions are met (e.g., the arrayset is not already full), then the arrayset resulting from an invocation of `(add-element val arrayset)` contains the value `val`. Functional correctness proofs of `add-element` proceeded fairly smoothly; however correctness proofs for `delete-element` required much more effort, as they required many theorems about `find-via-next-list`, and related functions.

Another type of theorem that one may wish to prove is an "inversion" theorem, namely that adding and element then deleting it (when the element wasn't present already) yields a data structure where that element is not present, i.e.

```
(defthm
  delete-element-of-add-element--thm
   (implies
     (and (good-state arrayset)
          (not (elementp val arrayset))
          (natp val))
     (not
       (elementp val
         (delete-element val
           (add-element val arrayset))))))
```

Of course, in addition to the theorem above, correctness of `add-element` and `delete-element` also depend on the preservation of key data structure invariants, namely that the `next` array free list and used list contain no cycles; that the length of the free list plus the length of the used list is the total length of the `next` array; that the free elements are all set to the `null-value`; etc. This is captured in the `good-state` invariant, as described earlier. Thus, the overall correctness theorem for each operation is given as follows:

```
(defthm
  add-element-preserves-good-state--thm
   (implies
     (good-state arrayset))
     (good-state
       (add-element val arrayset)))

(defthm
  delete-element-preserves-good-state--thm
   (implies
     (good-state arrayset))
     (good-state
       (delete-element val arrayset)))
```

V. Translation to Traditional Programming Languages

One of the advantages of the stobj formulation is that it permits straightforward translation of applicative Common Lisp specifications to traditional, imperative programming language implementations, which can then be compiled and executed on the target system. A semi-automated translation of the ArraySet stobj and add-element functions to C are shown in Figure 6. As one can see, the translation from ACL2 to C is basically line-for-line.

VI. Some ACL2 Issues

The verification of the ArraySet data structure in ACL2 was more challenging than anticipated. The first major issue, reasoning about stobjs, was dealt with early on, as described above. As the proofs progressed, it became obvious that a number of distributed ACL2 books would be useful (distributed books are files that collect theorems related to a given subject area, and that are included in the ACL2 source distribution); however, it was not always clear how to use them together. Books could not, for example, be included in just any order. In particular, including books in the "wrong" order and/or in the wrong place in the source file could produce disastrous results. By purely empirical means, it was found that including `arithmetic/top-with-meta`, followed later on by `coi/bags/top`, (containing theorems on bag, or multiset, reasoning) followed still later on by `coi/lists/nth-and-update-nth` (theorems on the Lisp `nth` and `update-nth` functions), and finally, almost at the end by `data-structures/list-defthms` (theorems about Lisp lists) worked fairly well, but it took a while to find this combination. Note in particular that the most modern arithmetic books, the `arithmetic-5` books, could not be successfully employed (the reason for this is yet to be investigated, as the older arithmetic book has been sufficient for our purposes).

Beyond book incompatibility and ordering problems, we found that several theorems needed to be disabled in order to make speedy progress on the last five percent of the proofs. (When a theorem is "disabled" in ACL2 parlance, it continues to be valid, but is not used in attempts to prove subsequent theorems, unless that theorem is later "enabled"

```
#define MULTISETP 0

#define MAXELEMENTS 16

typedef struct {
  int elements[MAXELEMENTS];
  int next[MAXELEMENTS];
  int free_head;
  int used_head;
} ARRAYSET;

ARRAYSET *add_element(int val, ARRAYSET *as) {
  int curr_index;

  if (val < 0) return as;
  if (!arraysetp(as)) return as;
  if (as->free_head == MAXELEMENTS) return as;
  if (!MULTISETP && elementp(val, as)) return as;

  curr_index = as->free_head;
  as->free_head = as->next[as->free_head];
  as->elements[curr_index] = val;
  as->next[curr_index] = as->used_head;
  as->used_head = curr_index;

  return as;
}
```

Fig. 6. Semi-automated translation of ArraySet stobj and add-element mutator to C

once again. A theorem is typically disabled when it is determined that the theorem is ineffective in proving subsequent theorems.) This was determined empirically by use of ACL2's `accumulated-persistence` mechanism, which provides the ACL2 user with detailed information on successful and failed proof attempts.

VII. Comparison to other Automated Verification Tools

The ArraySet example has so far been subjected to automated formal verification by two other approaches: model-checking [7] and symbolic execution [2]. In the former case, the ArraySet example, as well as its correctness properties, were formulated as Simulink/Stateflow [9] models (indeed, this was the original form), and processed by the Rockwell Collins Gryphon toolset [12]. In the particular instantiation of the Gryphon toolset in use at the time, the backend model checker employed was Prover [8]. Prover was able to automatically establish correctness properties for the ArraySet example, but only up to an array size of 3; beyond that, state space explosion occurred [7].

Later, John Hatcliff's team at Kansas State University utilized the ArraySet SPARK code, automatically produced from the earlier Simulink model by another part of the Gryphon toolchain [7], as a challenge problem for their newly-developed Bakar Kiasan symbolic execution-based formal analysis tool. Bakar Kiasan processes an extended SPARK contract annotation language that includes, among other enhancements, user-defined correctness predicates expressed using SPARK syntax. Bakar Kiasan was able to prove functional correctness for the ArraySet example up to an array size of 8 before combinatorics began to overwhelm its symbolic execution engine [2].

VIII. Conclusion

We have demonstrated the ability to establish the correctness of practical data structures commonly found in high-assurance systems through automated formal verification. In the particular case of an array-backed set/multiset, we were able to compare the scalability of several techniques, including model-checking, symbolic execution, and the ACL2 theorem prover. The model checking and symbolic execution approaches were able to prove correctness for small array sizes (up to 3, and up to 8, respectively), but state space explosion did not allow them to go beyond that. The initial formulation of the data structure using ACL2's single-threaded object allowed us to closely follow the structure of the imperative SPARK code of the ArraySet example; we were then able to directly access

the underlying logical definitions introduced by `defstobj` to eventually prove correctness for any array size. On the negative side, the ACL2 effort required a fair amount of human labor in order to achieve these proofs; some of this work could have been avoided if existing books were more interoperable. But, now that the effort has been made, and much of the infrastructure needed for reasoning about array-backed data structures has been created, additional proof efforts involving similar data structures will proceed more quickly (as our recent work on the Briggs-Torczon sparse set data structure demonstrates).

IX. Acknowledgments

Many thanks to Mike Whalen of the University of Minnesota for formulating the original version of the ArraySet example while a Rockwell Collins employee, and for all of his continuing efforts on the Gryphon toolset; to John Hatcliff of Kansas State University for taking on the ArraySet example as a challenge problem for his symbolic execution framework; to Matt Kaufmann at the University of Texas at Austin for his initial help in getting these proofs started by way of the `:pcb!` `:max` trick; and to Dave Greve at Rockwell Collins for his sage advice, especially regarding forward-chaining rules and dealing with book incompatibilities. Thanks also go to the anonymous reviewers for their helpful comments.

References

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

[2] Jason Belt, John Hatcliff, Robby, Patrice Chalin, David Hardin, and Xianghua Deng. Bakar kiasan: Flexible contract checking for critical systems using symbolic execution. In *Proceedings of the Third NASA Formal Methods Symposium (NFM 2011)*, pages 58 – 72, 2011.

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

[4] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. In *ACM Letters on Programming Languages and Systems*, volume 2, pages 59–69, 1993.

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

[6] David S. Hardin and Samuel S. Hardin. Efficient, formally verifiable data structures using acl2 single-threaded objects for high-assurance systems. In S. Ray and D. Russinoff, editors, *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, pages 100 – 105. ACM, 2009.

[7] David S. Hardin, T. Douglas Hiratzka, D. Randolph Johnson, Lucas G. Wagner, and Michael W. Whalen. Development of security software: A high assurance methodology. In K. Breitman and A. Cavalcanti, editors, *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM'09)*, pages 266 – 285. Springer, 2009.

[8] Prover Technologies Inc. Prover sl/de plug-in product description. `http://www.prover.com/products/prover_plugin`.

[9] The Mathworks Inc. Simulink product description. `http://-www.mathworks.com/Simulink`.

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

[11] LispWorks Ltd. Common lisp hyperspec. `http://-www.lispworks.com/documentation/HyperSpec/-Front/index.htm`.

[12] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. In *Communications of the ACM*, volume 53(2). ACM, February 2010.

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

[14] 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.

[15] Raymond J. Richards. Modeling and security analysis of a commercial real-time operating system kernel. In D. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 301 – 322. Springer, 2010.

[16] Matthew M. Wilding, David A. Greve, Raymond J. Richards, and David S. Hardin. Formal verification of partition management for the aamp7g microprocessor. In D. Hardin, editor, *Design and Verification of Microprocessor Systems for High-Assurance Applications*, pages 175 – 192. Springer, 2010.