Arrayset Revisited

David Hardin
Collins Aerospace
Cedar Rapids, IA USA
david.hardin@collins.com

Rust has gained significant popularity as a modern, type-safe, memory-safe, and formally analyzable programming language. We also see potential for Rust as a High-Level Synthesis (HLS) language, providing hardware/software co-assurance for critical systems, and have crafted a Rust subset for this purpose inspired by Russinoff's Restricted Algorithmic C (RAC), a language we have imaginatively named Restricted Algorithmic Rust (RAR). In this paper, we revisit an array-backed set (technically, a multiset) data structure, suitable for hardware implementation and previously implemented using a number of formal and semi-formal tools. We describe a revised implementation of the RAR arrayset, which has allowed us to prove full functional correctness for arbitrary-sized backing arrays in ACL2. We compare and contrast the ACL2 formalization to that accomplished in Verus, an SMT-based verification tool employing requires/ensures annotations. Verus is used in various development efforts at Amazon Web Services, and has been integrated into the INSPECTA Model-Based Systems Engineering toolchain for the DARPA PROVERS program. On the PROVERS effort, we have adopted a hybrid approach, in which Verus establishes basic correctness properties, and an interactive theorem prover such as ACL2 is employed as needed to discharge more difficult theorems. We also report on progress toward hardware synthesis capability for the INSPECTA toolchain, allowing engineers to allocate verified system functionality to either hardware or software, and to revisit those allocations as differing product variants are developed from the same basic system model.

1 Introduction

The Rust programming language has garnered significant adoption as a modern, type-safe, memory-safe, and formally analyzable programming language. Amazon Web Services [6], Google [31] and Microsoft [5] are major Rust adopters, and Rust is now an official language for Linux kernel development [17]. All this comes none too soon, as use of C/C++ continues to spawn a seemingly never-ending parade of security vulnerabilities, which continue to manifest at a high rate [22] despite the emergence and use of sophisticated C/C++ analysis tools.

Our interest in Rust stems from its potential as a hardware/software co-assurance language. This interest is motivated in part by emerging application areas, such as autonomous and semi-autonomous platforms for land, sea, air, and space, that require sophisticated algorithms and data structures, are subject to stringent accreditation/certification, and encourage hardware/software co-design approaches. (For an unmanned aerial vehicle use case illustrating a formal methods-based systems engineering environment, please consult [21].) In this paper, we explore the use of Rust as a High-Level Synthesis (HLS) language [24]. Most incumbent HLS languages are a subset of C, e.g. Mentor Graphics' Algorithmic C [19], or Vitis/Vivado HLS by Xilinx [1]. (NB: Xilinx is now part of AMD.) A Rust-based HLS would bring a single modern, type-safe, and memory-safe expression language for both hardware and software realizations, with very high assurance.

As formal methods researchers, another keen interest is in being able to reason about application-level logic written in the imperative style favored by industry. Much progress has been made to this

Submitted to:
ACL2 2026

© D.S. Hardin
This work is licensed under the
Creative Commons Attribution License.

end in recent years, to the point that developers can verify the correctness of common algorithm and data structure code that utilizes common idioms such as records, loops, modular integers, and the like, and verified compilers can guarantee that such code is compiled correctly to binary [15]. Particular progress has been made in the area of hardware/software co-design algorithms, where array-backed data structures are common [8,9]. (NB: This style of programming also addresses one of the shortcomings of Rust, namely its lack of support for cyclic data structures.)

We maintain that the strong typing, memory safety, and popularity of Rust make it a suitable candidate for an HLS. To evaluate this claim, as well as to enable hardware/software co-design using Rust, we have crafted a Rust subset, inspired by Russinoff's Restricted Algorithmic C (RAC) [29], which we have imaginatively named Restricted Algorithmic Rust, or RAR, first described in [10] – indeed, this paper may be considered to be a follow-on to that work. In our first implementation of a RAR toolchain, we merely "transpile" (perform a source-to-source translation of) the RAR source into RAC. By so doing, we leverage a number of existing hardware/software co-assurance tools with a minimum investment of time and effort. By transpiling RAR to RAC, we gain access to existing HLS compilers (with the help of some simple C preprocessor directives, we are able to generate code for either the Algorithmic C or Vitis/Vivado HLS toolchains). But most importantly for our research, we leverage the RAC-to-ACL2 translator that Russinoff and colleagues at Arm have successfully utilized in industrial-strength floating point hardware verification.

We have implemented several representative algorithms and data structures in RAR, including:

- a suite of array-backed algebraic data types, previously implemented in RAC (as reported in [8]);
- a significant subset of the Monocypher [32] modern cryptography suite, including XChacha20 and Poly1305 (RFC 8439) encryption/decryption, Blake2b hashing, and X25519 public key cryptography; and
- a DFA-based JSON lexer, coupled with an LL(1) JSON parser. The JSON parser has also been implemented using Greibach Normal Form (previously implemented in RAC, as described in [12]).

The RAR examples created to date are similar to their RAC counterparts in terms of expressiveness, and we deem the RAR versions somewhat superior in terms of readability (granted, this is a very subjective evaluation).

In this paper, we will describe the development and formal verification of an array-based set data structure in RAR. Along the way, we will introduce the RAR subset of Rust, the RAR toolchain, the array-based set example, and detail the ACL2-based verification techniques, as well as the ACL2 books that we brought to bear on this example. It is hoped that this explication will convince the reader of the practicality of RAR as a high-assurance hardware/software co-design language, as well as the feasibility of performing full functional correctness proofs of RAR code. We will then conclude with related and future work.

2 INSPECTA: A High-Assurance Systems Engineering Toolchain

We aspire to create an integrated hardware/software co-assurance toolchain that enables developers to input various familiar high-level specifications, and produce verified implementations for those specifications, in software, hardware, or a combination of the two. We are realizing elements of the aspirational vision on the DARPA PROVERS (Pipelined Reasoning Of Verifiers Enabling Robust Systems) program by way of the INSPECTA (Industrial-Scale Proof Engineering for Critical Trustworthy Applications) toolset [7]. INSPECTA integrates design, code generation, and formal verification activities, enabling

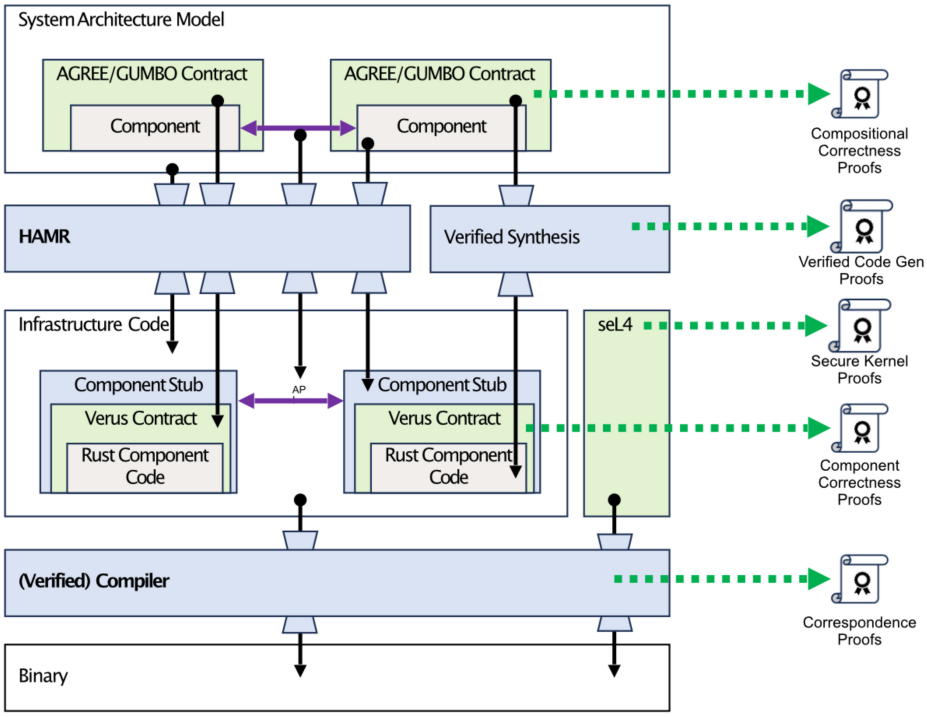

Figure 1: INSPECTA: High-Assurance System Development Layers, Tools, and Proof Artifacts.

systems engineers to design-in resiliency for today's complex avionics systems using the SysML v2 systems modeling language. The INSPECTA tools automatically generate skeletal implementations for SysML v2 models in a memory-safe language (MSL), specifically Rust, allowing developers to state and prove formal properties about the system model that can be automatically refined and reproved against the generated design in Rust. This capability to automatically generate verification-enhanced memory-safe language code from architecture models featuring assume/guarantee contracts is novel, providing high assurance while saving engineering time and effort. The relationships amongst the various layers of the system design, the tools that allow one to proceed from the more abstract layers to the more concrete layers, as well as the proof artifacts that can be generated at each layer are depicted in Fig. 1.

A fundamental aspect of the INSPECTA approach is the use of system architecture models to provide a framework for analyzing system behavior and organizing the assurance evidence produced. Architecture modeling allows engineers to describe the important elements of distributed, real-time, embedded systems (processors, memory, buses, processes, threads, and data interconnections) with sufficiently rigorous semantics that can support formal reasoning. SysML v2 is the latest general-purpose modeling language developed to facilitate Model-Based Systems Engineering (MBSE). SysML can be used to represent system requirements, structure, and behavior, as well as specify properties that can be used to analyze and verify the system.

Researchers at Kansas State University have developed a unique multi-platform, multi-language skeletal code generation framework from system models, called HAMR [3]. HAMR can be used to automatically generate code for deployment for the Java Virtual Machine (primarily for host-based simulation/testing), Linux, or the verified seL4 microkernel. seL4 is formally verified from its high-

level security properties down to its binary implementation [14]. Notably, seL4 features mathematically proven separation between threads, thus providing a very high assurance foundation upon which to produce application-level property proofs. Recent HAMR innovations include targeting code generation for MSLs, including precondition/postcondition annotations necessary for formal verification. These annotations are refined from the system-level contracts defined in the SysML v2 model. MSLs avoid many common vulnerabilities that are the bane of C/C++ development, and are increasingly required for high-assurance development, in the wake of recommendations by cybersecurity agencies of the U.S. Government [25, 33]. The ability to automatically generate MSL code from a system model helps to produce complete systems, even when MSL developers are scarce.

Recent HAMR development has targeted the Rust programming language. Rust is a modern, high-level programming language designed to combine the code generation efficiency of C/C++ with drastically improved type safety and memory management features. Additionally, HAMR supports a precondition/postcondition specification and verification tool for Rust called Verus [16], under development by a team of researchers led by Carnegie Mellon University.

2.1 The SysML v2 Systems Modeling Language

SysML v2 is the second major version of the Systems Modeling Language, a standard language for modeling systems and their behavior [26] promulgated by the Object Management Group. It builds upon SysML v1, offering enhanced features such as a textual representation, standard API, and improved expressiveness for modeling complex systems, as well as a formal semantics. SysML v1 has been adopted by industry, but mostly has been used for requirements elicitation and documentation support. However, the increased expressiveness of SysML v2, coupled with the ability to interchange models with third party tools via its textual representation, enables a much deeper integration of architectural modeling with detailed development. The INSPECTA research team has also pioneered the use of formally verified assume/guarantee contracts for MBSE.

3 RAC: Hardware/Software Co-Assurance at Scale

In order to begin to realize our aspirational vision for hardware/software co-assurance at scale, we have conducted several experiments employing a state-of-the-art toolchain, due to Russinoff and O'Leary, and originally designed for use in floating-point hardware verification [29], to determine its suitability for the creation of safety-critical/security-critical applications in various domains. Note that this toolchain has already demonstrated the capability to scale to industrial designs in the floating-point hardware design and verification domain, as it has been used in design verifications for CPU products at both Intel and Arm.

Algorithmic C [19] is a High-Level Synthesis (HLS) language, and is supported by hardware/software co-design environments from Mentor Graphics, *e.g.*, Catapult [20]. Algorithmic C defines C++ header files that enable compilation to both hardware and software platforms, including support for the peculiar bit widths employed, for example, in floating-point hardware design.

The Russinoff-O'Leary Restricted Algorithmic C (RAC) toolchain, depicted in Fig. 2, translates a subset of Algorithmic C source to the Common Lisp subset supported by the ACL2 theorem prover, as augmented by Russinoff's Register Transfer Logic (RTL) books.

The ACL2 translator converts imperative RAC code to functional ACL2 code. Loops are translated into tail-recursive functions, with automatic generation of measure functions to guarantee admission into

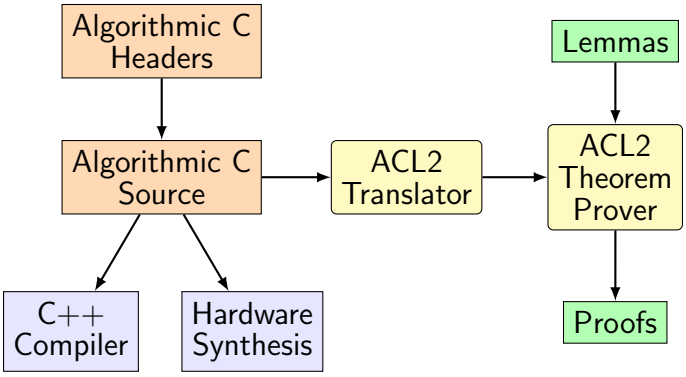

Figure 2: Restricted Algorithmic C (RAC) toolchain.

the logic of ACL2 (RAC subsetting rules ensure that loop measures can be automatically determined). Structs and arrays are converted into functional ACL2 records. The combination of modular arithmetic and bit-vector operations of typical RAC source code is faithfully translated to functions supported by Russinoff's RTL books. ACL2 is able to reason about non-linear arithmetic functions, so the usual concern about formal reasoning about non-linear arithmetic functions does not apply. Finally, the RTL books are quite capable of reasoning about a combination of arithmetic and bit-vector operations, which is a very difficult feat for most automated solvers.

4 Rust and RAR

The Rust programming language [13] is a modern, high-level programming language designed to combine the code generation efficiency of C/C++ with drastically improved type safety and memory management features. A distinguishing feature of Rust is that a non-scalar object may only have one owner. For example, one cannot assign a reference to an object in a local variable, and then pass that reference to a function. This restriction is similar to those imposed on ACL2 single-threaded objects (stobjs) [4], with the additional complexities of enforcing such "single-owner" restrictions in the context of a general-purpose, imperative programming language. The Rust runtime performs array bounds checking, as well as arithmetic overflow checking (the latter can be disabled by a build environment setting).

In most other ways, Rust is a fairly conventional modern programming language, with interfaces (called traits), lambdas (termed closures), and pattern matching, as well as a macro capability. Also in keeping with other modern programming language ecosystems, Rust features a language-specific build and package management system, named cargo.

4.1 Restricted Algorithmic Rust

As we wish to utilize the RAC toolchain as a backend in our initial work, Restricted Algorithmic Rust is semantically equivalent to RAC. Thus, we adopt the same semantic restrictions as described in Russinoff's book. Additionally, in order to enable translation to RAC, as well as to ease the transition from C/C++, RAR supports a commonly used macro that provides a C-like *for* loop in Rust. Note that, despite the restrictions, RAR code is proper Rust; it compiles to binary using the standard Rust compiler.

RAR is transpiled to RAC via a source-to-source translator, as depicted in Fig. 3. Our transpiler is based on the `plex` parser and lexer generator [30] source code. We thus call our transpiler *Plexi*, a

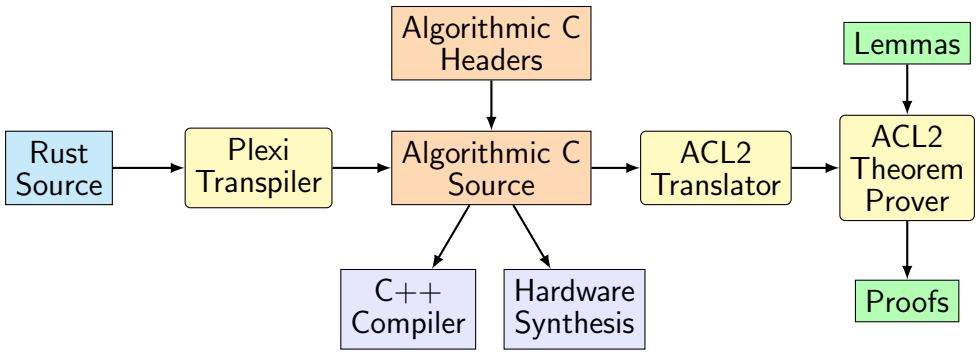

Figure 3: Restricted Algorithmic Rust (RAR) toolchain.

nickname given to a famous (and now highly sought-after) line of Marshall guitar amplifiers of the mid-1960s. Plexi performs lexical and syntactic transformations that convert RAR code to RAC code. This RAC code can then be compiled using a C/C++ compiler, fed to an HLS-based FPGA compiler, as well as translated to ACL2 via the RAC ACL2 translator, as illustrated in Fig. 3.

Recently, we have investigated the synthesis of Field-Programmable Gate Array (FPGA) hardware directly from high-level architecture models, in collaboration with colleagues at Kansas State University. The goal of this work is to enable the generation of high-assurance hardware and/or software from high-level architectural specifications expressed in SysML V2 [26], with proofs of correctness in ACL2. Thus far, we have demonstrated the integration of a transpiler from Slang (the Scala-like implementation language of HAMR [28]) to Xilinx Vitis/Vivado HLS, and have demonstrated the creation of a high-level doubly-linked list data structure hosted on the FPGA fabric of a Xilinx Zynq Ultrascale+ ZCU102 FPGA Evaluation board, as well as successful execution of test cases for the doubly-linked list on that hardware [27]. The last remaining step is to provide a RAR-to-Slang translation pass in order to complete the Rust-to-FPGA path.

5 Example: Array-Based Set

In this section, we present a revised version of an array-based set implementation using RAR, first reported in [10]. The arrayset 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 Collins high-assurance development [11].

5.1 Arrayset Definitions

First, we present the basic RAR declaration for the Arrayset.

```
const ARR_SZ: uint = 1024;    // Can be sized up to billions

#[derive(Copy, Clone)]
struct Arrayset {
  anext: [usize; ARR_SZ],
  avals: [i64; ARR_SZ],
  free_lst: usize,
```

```
  used_lst: usize,
  err: usize,
}

fn free_head(aset: Arrayset) -> usize {
  let mut retval: usize = ARR_SZ;
  if (aset.free_lst < ARR_SZ) {
    retval = aset.anext[aset.free_lst];
  }
  return retval;
}

fn used_head(aset: Arrayset) -> usize {
  let mut retval: usize = ARR_SZ;
  if (aset.used_lst < ARR_SZ) {
    retval = aset.anext[aset.used_lst];
  }
  return retval;
}
```

In this declaration, the array `avals` holds the set elements, the `anext` array contains indices indicating the next element in either the free or used list, `free_lst` is the index of the last element of the free list, and `used_lst` is the index of the last element of the used list. Note that indices in Rust are normally declared to be of the `usize` type.

The ingenuity of this particular array-backed set implementation is the use of a single `anext` array to hold both the free list and the used list. Each element of the `anext` array is in one of the two lists, but not both. In a change from the previous version of the arrayset described in [10], the free list and used list are circular. We made this change in order to be able to state and prove a simple, but powerful, invariant: that subsequent to initialization, the contents of the `anext` array is some permutation of the values 0 through `ARR_SZ - 1`, inclusive. This means that every element in the `anext` array can be reached in only one way, greatly simplifying the task of proving that a given Arrayset operation maintains the integrity of that array.

One of the jobs of the mutators `aset_init()`, `aset_add()`, `aset_del_head()`, and `aset_del()` is to maintain the integrity of the free list and used list contained within the single `anext` array, and it is a primary obligation of the verification effort to show that this is the case.

Fig. 4(a) shows the state of an Arrayset of size 5 after some number of `aset_add` and `aset_del` operations have been performed. First, note that all arrays are zero-based, but the zeroth elements are not used, as RTL uses 0 for its default (or "none") value. The `used_lst` index into the `anext` array is 1, indicating the "tail" of the used list, while the `free_lst` index is 5, designating the "tail" of the free list. If `used_lst` or `free_lst` is equal to `ARR_SZ`, that indicates a null used list or free list. Reading the `anext` array at the `used_lst` index yields the index of the next element in the used list, the `used_head` of 2. If we then read the `avals` array at index 2, we get the value at the head of the used list, specifically 33. Reading the value in the `anext` array yields 1, the value of `used_lst`; thus we are at the tail of the circular used list, and the corresponding elements in the `avals` array are {33, 22}, the contents of the used list. (Note that other components of `avals` have values, but as they are free elements, those values are not shown in the Figure.)

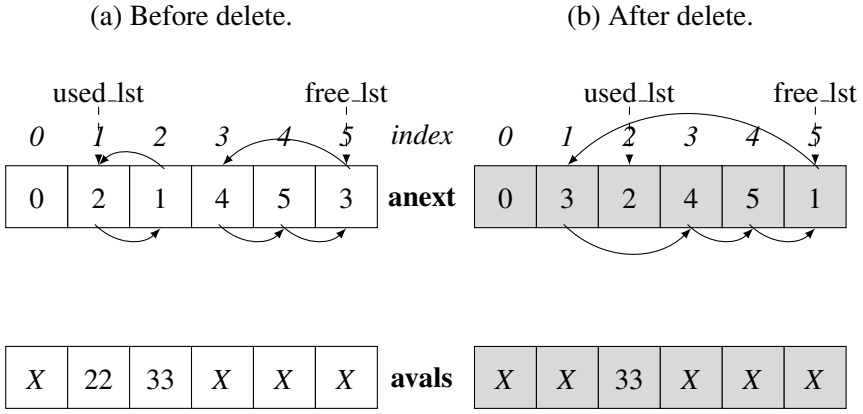

Figure 4: (a) An Arrayset data structure with contents {33, 22}, size = 5.
(b) The Arrayset after deleting element 22.

```
fn aset_add(val: i64, mut aset: Arrayset) -> Arrayset {

  aset = aset_transfer(false, aset);  // false => free head to used head

  if (aset.err == 1) {      // no error
    aset.avals[aset.anext[aset.used_lst]] = val;
  }

  return aset;
}
```

Figure 5: `aset_add()` function in RAR.

On the other hand, to traverse the free list, we start with the `free_lst` index, which is 5. Reading element 5 of `anext`, we get 1. Reading index 1 of the `anext` list, we obtain 3; reading element 3 yields 4, and reading element 4 results in 5, delivering us back to the tail of the circular free list.

If we then execute `aset_del(22, aset)`, we obtain the Arrayset shown in Figure 4(b). 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 `anext` array.

The `aset_add()` function is coded in RAR as shown in Fig. 5. The operation of `aset_add()` is as follows. If there is no room (i.e., the free list is empty), no change is made to the Arrayset. Otherwise, `aset_transfer` (not shown) moves an element from the free list to the used list. If that function succeeds, it returns an index which `aset_add` then uses to store `val` in the `avals` array. Note that no check is made in this particular implementation of Arrayset to determine whether the element `val` to be added is already in the `avals` array; thus this Arrayset is technically an ArrayMultiset.

`aset_del_head` is basically a wrapper for the `aset_transfer` function, which transfers the head of the used list to the head of the free list. `aset_del()` proceeds similarly to `aset_add`, although it

```
fn aset_del_head(mut aset: Arrayset) -> Arrayset {

  aset = aset_transfer(true, aset);  // true => used head to free head

  return aset;
}


fn aset_del(val: i64, mut aset: Arrayset) -> Arrayset {
  let mut prev_index: usize = ARR_SZ;

  if (aset.used_lst < ARR_SZ) {
    prev_index = aset_find_val_from(aset.used_lst, val, aset);

    if ((prev_index < ARR_SZ) && (prev_index != aset.free_lst)) {

      // Rotate used pointer
      aset.used_lst = prev_index;

      aset = aset_del_head(aset);
    } else {
      aset.err = ARR_SZ;
    }
  } else {
    aset.err = ARR_SZ;
  }
  return aset;
}
```

Figure 6: `aset_del()` function in RAR.

is admittedly a bit more complicated (see Fig. 6). `aset_del()`, in turn, depends on the `aset_find_val_from()` function (not shown), which searches for an element with a given value `val` by traversing the `anext` array, and examining values using corresponding indices into the `avals` array. This function actually returns an index `previ` such that `aset.avals[aset.anext[previ]] == val`, as that index is needed for `anext` array bookkeeping.

In order to delete an element at some index, it is easiest to "rotate" the used list pointer to the index before the one to be deleted, then call the `del_head` function to do most of the work. Since we are dealing with a set, order is unimportant, so we do not need to restore the `used_lst` index at the end of the `aset_del` function.

```
(DEFUND ASET_DEL (VAL ASET)
 (LET ((PREV_INDEX (ARR_SZ)))
   (MV-LET
      (PREV_INDEX ASET)
      (IF1 (LOG< (AG 'USED_LST ASET) (ARR_SZ))
           (LET ((PREV_INDEX (ASET_FIND_VAL_FROM
                              (BITS (AG 'USED_LST ASET) 63 0)
                              (BITS (SI VAL 64) 63 0)
                              ASET)))
             (MV PREV_INDEX
                 (IF1 (LOGAND1 (LOG< PREV_INDEX (ARR_SZ))
                               (LOG<> PREV_INDEX (AG 'FREE_LST ASET)))
                      (LET ((ASET (AS 'USED_LST PREV_INDEX ASET)))
                        (ASET_DEL_HEAD ASET))
                      (AS 'ERR (ARR_SZ) ASET))))
           (MV PREV_INDEX (AS 'ERR (ARR_SZ) ASET)))
     ASET)))
```

Figure 7: aset_del() function translated to ACL2 using the RAC tools.

5.2 Translation to ACL2

We use `Plexi` to transpile the RAR source to RAC (not shown), then use the RAC translator to convert the resulting RAC source to ACL2. The translation of `aset_del()` appears in Fig. 7.

The first thing to note about Fig. 7 is that struct and array 'get' and 'set' operations become untyped record operators, `AG` and `AS`, respectively — these are slight RAC-specific customizations of the usual ACL2 untyped record operators. Further, `IF1` is a RAC-specific macro, and `LOG>`, `LOGAND1`, and `LOG<` are all RTL functions. Thus, much of the proof effort involved with RAR code is reasoning about untyped records and RTL — although not a lot of RTL-specific knowledge is needed, at least in our experience. An additional observation to make here is that, even though we are two translation steps away from the original RAR source, the translated function is nonetheless quite readable, which is a rare thing for machine-generated code.

5.3 Arrayset Theorems

Once we have translated the Arrayset functions into ACL2, we can begin to prove theorems about the data structure implementation. We begin by defining some common preconditions, such as `used_lst` should not equal `free_lst`, as well as specific preconditions related to used-to-free or free-to-used transfers. A particularly useful precondition is that the `anext` array contains no duplicates. Given these sorts of preconditions, we can then prove functional correctness theorems for the Arrayset operations, of the sort stated in Fig. 8.

ACL2 performs the correctness proof for `aset_add` automatically. The correctness proof for the `aset_del` operation is considerably more complex, as `aset_del` makes numerous modifications to the Arrayset data structure. Thus, user assistance is currently required in order to perform the correctness proof.

```
(defthm aset_add-val-at-used_head--thm
  (implies
   (and (asetp aset)
        (aset-free-to-used-preconditionp aset)
        (unsigned-byte-p 64 val))
   (= (si (ag (used_head (aset_add val aset))
              (ag 'avals (aset_add val aset))) 64)
      (si val 64))))

(defthm aset_del-val-at-free_head--thm
  (implies
   (and (asetp aset)
        (aset-used-to-free-preconditionp aset)
        (no-dups-anext aset)
        (unsigned-byte-p 64 val)
        (< (aset_find_val_from (ag 'used_lst aset) val aset) (ARR_SZ)))
   (= (si (ag (free_head (aset_del val aset))
              (ag 'avals (aset_del val aset))) 64)
      (si val 64)))
```

Figure 8: Sample ACL2 correctness theorems for `aset_add()` and `aset_del()`.

In an extension to the proofs performed in [10], we define `aset-used-list` and `aset-free-list` functions in ACL2 (not shown) which traverse the `anext` array from the `used_head` and `free_head` indices, respectively, and produce lists of used and free `anext` entries, respectively. (Note that we do not define these functions in Rust, as they are not needed for the operation of the arrayset, but rather are for verification use only. We then prove that the aforementioned add and delete functions produce the expected modifications to the used and free lists. For example, we prove that the used list after `aset_del_head` is the `cdr` of the used list before that operation; and that the free list after `aset_del_head` is the `cons` of the `used_head` onto the front of the free list before the `aset_del_head` operation, as depicted in Fig. 9.

It should not be surprising to note that the proofs of these extended theorems made significant use of the ACL2 proof builder.

6 Comparison with other Automated Verification Tools

The Arrayset example has so far been subjected to automated formal verification by two other approaches: model-checking [11] and symbolic execution [2]. In the former case, the Arrayset example, as well as its correctness properties, were formulated as Simulink/Stateflow [18] models (indeed, this was the original form), and processed by the Collins Gryphon toolset [23]. In the particular instantiation of the Gryphon toolset in use at the time, the backend model checker 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 [11].

Later, John Hatcliff's team at Kansas State University utilized the Arrayset SPARK code, automat-

```
(defthm aset_del_head-used-list--thm
  (implies
   (and (asetp aset)
        (aset-used-to-free-preconditionp aset)
        (no-dups-anext aset)
        (= (+ (len (aset-used-list aset)) (len (aset-free-list aset)))
           (1- (ARR_SZ))))
   (= (aset-used-list (aset_del_head aset))
      (cdr (aset-used-list aset)))))

(defthm aset_del_head-free-list--thm
  (implies
   (and (asetp aset)
        (aset-used-to-free-preconditionp aset)
        (no-dups-anext aset)
        (= (+ (len (aset-used-list aset)) (len (aset-free-list aset)))
           (1- (ARR_SZ))))
   (= (aset-free-list (aset_del_head aset))
      (cons (used_head aset) (aset-free-list aset)))))
```

Figure 9: Sample ACL2 used and free list correctness theorems.

ically produced from the earlier Simulink model by another part of the Gryphon toolchain [11], 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].

6.1 Rust/Verus Comparison

These previous automated verification attempts were made several years ago; we now wish to compare the verification capabilities of RAR with modern Rust verification tools that utilize Satisfiability Modulo Theories (SMT) solvers on the backend. To perform this analysis, we chose the Rust verification tool Verus. With Verus, programmers express proofs and specifications using extended Rust syntax (introduced using Rust's macro facility), allowing proofs to take advantage of Rust's high-assurance ownership restrictions. This extended verification syntax is "erased" for code compilation, allowing Verus-enhanced code to be compiled using the standard Rust compiler. Verus is utilized in numerous software development organizations, notably Amazon Web Services.

Like many verification-enhanced programming environments, Verus provides means to express formal preconditions for a given function, which provide the needed input constraints to achieve a successful verification for that function (expressed as `requires` blocks), as well as postconditions that provide the output conditions (expressed as `ensures` blocks) that can be proved to arise from the preconditions. Verus also provides means to express loop invariants (via `invariant` blocks within the loop bodies).

```
#[verus_spec(ret =>
  requires
    aset.free_lst > 0, aset.free_lst <= ARR_SZ,
    aset.used_lst > 0, aset.used_lst <= ARR_SZ,
    aset.anext[0] == 0,
    forall|i: int| 0 < i < aset.anext.len() ==>
                  (0 < #[trigger] aset.anext[i] < ARR_SZ),
    aset.anext@.no_duplicates(),
    aset.used_lst != aset.free_lst,
    aset.err == 1,
  ensures
    ret.free_lst > 0, ret.free_lst <= ARR_SZ,
    ret.used_lst > 0, ret.used_lst <= ARR_SZ,
    ret.anext[0] == 0,
    ret.anext.len() == ARR_SZ,
    forall|i: int| 0 < i < ret.anext.len() ==>
                  (0 < #[trigger] ret.anext[i] < ARR_SZ),
    ret.avals.len() == ARR_SZ,
    ret.anext@.no_duplicates(),
    ret.used_lst != ret.free_lst,
    (ret.err == 1) ==>
     (aset.avals[ret.anext[ret.free_lst as int] as int] == val),
)]
```

Figure 10: aset_del() Verus annotations.

The Verus annotations for the `aset_del` function appear in Fig. 10. The process of producing these annotations was greatly aided by the fact that we had previously performed the ACL2 proofs, so we knew what sorts of preconditions (*requires* annotations) and postconditions (*ensures* annotations) would be needed. Note that in Verus, the return value from a function is referred to as `ret`; in the case of `aset_del`, ret is the returned `aset`.

We were pleasantly surprised that Verus proofs could scale to arbitrarily large array sizes; we tried `ARR_SZ` values up to a billion with no problems. The second source of surprise was that Verus could readily prove not only the equivalent of the correctness proofs of Fig. 8, but also that the `aset_add` and `aset_del` operations resulted in an `anext` array with no duplicates. We are currently in the process of proving the full suite of correctness results in Verus that we have proved in ACL2. Challenges include the relative lack of support for reasoning about permutations in Verus, lack of support for concrete counterexamples, as well as some awkwardness in moving amongst array/sequence/multiset/etc. representations. It is also not yet clear how easy it will be to state and prove the various used list and free list theorems in Verus. It is likely that we will need to craft versions of these functions that operate on a Rust `Seq` sequence type, which is analogous to a Lisp list. Verus includes significant support for reasoning about `Seq` types, but it cannot match the verification support that ACL2 provides for Lisp lists.

7 Conclusion

We have developed a prototype toolchain to allow the Rust programming language to be used as a hardware/software co-design and co-assurance language for critical systems, standing on the shoulders of Russinoff's team at Arm, and all the great work they have done on Restricted Algorithmic C. We have demonstrated the ability to establish the correctness of several practical data structures commonly found in high-assurance systems (*e.g.*, array-backed singly-linked lists, doubly-linked lists, stacks, and deques) through automated formal verification, enabled by automated source-to-source translation from Rust to RAC to ACL2, and we detailed the specification and verification of one such data structure, an array-backed set. We have also successfully applied our toolchain to cryptography and data format filtering examples typical of the sorts of algorithms that one encounters in critical systems development.

We have revisited the crafting and verification of an array-backed set using the RAR toolchain, and have now created a complete set of correctness theorems for that data structure using the ACL2 theorem prover. The basic Arrayset approach was developed many years ago, and verification attempted using various techniques over the years. We have thus been able to compare the scalability and usability of several verification techniques, including model-checking, symbolic execution, as well as ACL2. Historically, 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. While the ACL2 proof efforts required a fair amount of human labor in order to achieve the Arrayset formalization and proofs, now that this effort has been made, additional proof efforts involving similar data structures should proceed more quickly.

Recently, we have begun to evaluate the Verus Rust verification tool, which utilizes an SMT solver on its backend, on the Arrayset verification problem. Thus far, Verus demonstrates significant improvement in both scalability and usability, and we will continue this evaluation in future work. In other future work, we will work with our colleagues at Kansas State University on the direct synthesis and verification of FPGA hardware from RAR source code.

8 Acknowledgments

This work was funded by DARPA contract HR001118900001[1]. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

Many thanks to Mike Whalen of Amazon for formulating the original version of the Arrayset example while a Collins employee; 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, then at the University of Texas at Austin, for his help on the earlier stobj-based version of the Arrayset code; to David Russinoff of Arm for answering questions about the RAC toolchain; to Bryan Parno at Carnegie Mellon University for his help with the Verus toolchain; and to Robby of Kansas State University for his ongoing work to further our hardware/software co-assurance efforts. Thanks also go to the anonymous reviewers for their insightful comments.

[1] Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] AMD Inc. (2026): *Vitis High-Level Synthesis User Guide (UG1399)*. Available at `https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction`.

[2] Jason Belt, John Hatcliff, Robby, Patrice Chalin, David S. Hardin & Xianghua Deng (2011): *Bakar Kiasan: Flexible Contract Checking for Critical Systems using Symbolic Execution*. In: *Proceedings of the Third NASA Formal Methods Symposium (NFM 2011)*, pp. 58–72, doi:10.1007/978-3-642-20398-5_6.

[3] Jason Belt, John Hatcliff, Robby, John Shackleton, Jim Carciofini, Todd Carpenter, Eric Mercer, Isaac Amundson, Junaid Babar, Darren Cofer, David Hardin, Karl Hoech, Konrad Slind, Ihor Kuz & Kent McLeod (2023): *Model-driven development for the seL4 microkernel using the HAMR framework*. *Journal of Systems Architecture* 134, doi:10.1016/j.sysarc.2022.102789. Available at `https://www.sciencedirect.com/science/article/pii/S1383762122002740`.

[4] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. In: *Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Portland, OR, USA, January 19-20, 2002, Proceedings*, *LNCS* 2257, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[5] Thomas Claburn (2023): *Microsoft is busy rewriting core Windows code in memory-safe Rust*. Available at `https://www.theregister.com/2023/04/27/microsoft_windows_rust/`.

[6] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak & Andrew M. Wells (2024): *Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization*. In: *Proceedings of the ACM on Programming Languages (PACMPL)*, 8, Association for Computing Machinery, New York, NY, USA, doi:10.1145/3649835.

[7] David Hardin, Isaac Amundson, Junaid Babar, Darren Cofer, Saqib Hasan, Karl Hoech, Jason Belt, John Hatcliff, Robby & Stefan Hallerstede (2025): *Automated SysML v2 System Model to Memory-Safe Language Code Generation for Avionics Applications*. In: *AIAA DATC/IEEE 44th Digital Avionics Systems Conference (DASC)*, pp. 1–9, doi:10.1109/DASC66011.2025.11257170.

[8] David S. Hardin (2020): *Put Me on the RAC*. In: *Proceedings of the Sixteenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-20)*, pp. 142–145, doi:10.4204/eptcs.327.13.

[9] David S. Hardin (2020): *Verified Hardware/Software Co-Assurance: Enhancing Safety and Security for Critical Systems*. In: *Proceedings of the 2020 IEEE Systems Conference*, doi:10.1109/SysCon47679.2020.9381831.

[10] David S. Hardin (2022): *Hardware/Software Co-Assurance using the Rust Programming Language and ACL2*. In: *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2-22)*, pp. 202–216, doi:10.4204/EPTCS.359.16.

[11] David S. Hardin, T. Douglas Hiratzka, D. Randolph Johnson, Lucas G. Wagner & Michael W. Whalen (2009): *Development of Security Software: A High Assurance Methodology*. In K. Breitman & A. Cavalcanti, editors: *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM'09)*, Springer, pp. 266–285, doi:10.1007/978-3-642-10373-5_14.

[12] David S. Hardin & Konrad L. Slind (2021): *Formal Synthesis of Filter Components for Use in Security-Enhancing Architectural Transformations*. In: *Proceedings of the Seventh Workshop on Language-Theoretic Security, 42nd IEEE Symposium and Workshops on Security and Privacy (LangSec 2021)*, doi:10.1109/SPW53761.2021.00024.

[13] Steve Klabnik, Carol Nichols & Chris Krycho (2026): *The Rust Programming Language, 3rd Edition*. No Starch Press.

[14] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2010): *seL4: formal verification of an operating-system kernel*. *Communications of the ACM* 53(6), pp. 107–115, doi:10.1145/1743546.1743574.

[15] Ramana Kumar, Magnus O. Myreen, Michael Norrish & Scott Owens (2014): *CakeML: a verified implementation of ML*. In Suresh Jagannathan & Peter Sewell, editors: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, ACM, pp. 179–192, doi:10.1145/2535838.2535841.

[16] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno & Chris Hawblitzel (2023): *Verus: Verifying Rust Programs Using Linear Ghost Types*. *Proc. ACM Program. Lang.* 7(OOPSLA1), doi:10.1145/3586037.

[17] Godfrey Lebo (2025): *Rust goes mainstream in the Linux kernel*. Available at `https://medium.com/@emorilebo/rust-goes-mainstream-in-the-linux-kernel-7898d1ff2280`.

[18] The MathWorks, Inc. (2020): *Simulink*. Available at `https://www.mathworks.com/products/simulink.html`.

[19] Mentor Graphics Corporation (2016): *Algorithmic C (AC) Datatypes*. Available at `https://www.mentor.com/hls-lp/downloads/ac-datatypes`.

[20] Mentor Graphics Corporation (2020): *Catapult High-Level Synthesis*. Available at `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/`.

[21] Eric Mercer, Konrad Slind, Isaac Amundson, Darren Cofer, Junaid Babar & David Hardin (2023): *Synthesizing Verified Components for Cyber Assured Systems Engineering*. In: *Software and Systems Modeling*, 22, pp. 1451–1471, doi:10.1007/s10270-023-01096-3.

[22] Matt Miller (2019): *A proactive approach to more secure code*. Available at `https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/`.

[23] Steven Miller, Michael Whalen & Darren Cofer (2010): *Software model checking takes off*. *Communications of the ACM* 53, pp. 58–64, doi:10.1145/1646353.1646372.

[24] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson & Koen Bertels (2016): *A Survey and Evaluation of FPGA High-Level Synthesis Tools*. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35(10), pp. 1591–1604, doi:10.1109/TCAD.2015.2513673.

[25] National Security Agency (2022): *Software Memory Safety*. Available at `https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF`.

[26] Object Management Group (2026): *OMG Systems Modeling Language (SysML), Version 2.0*. Available at `https://www.omg.org/spec/SysML/2.0/Language/PDF`.

[27] Robby (2026): *Personal communication*.

[28] Robby & John Hatcliff (2021): *Slang: The Sireum Programming Language*. In: *10th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, *LNCS* 13036, pp. 253–273, doi:10.1007/978-3-030-89159-6_17.

[29] David M. Russinoff (2022): *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*, second edition. Springer, doi:10.1007/978-3-030-87181-9.

[30] Geoffry Song (2020): *plex: a parser and lexer generator as a Rust procedural macro*. Available at `https://github.com/goffrie/plex`.

[31] Jeff Vander Stoep (2025): *Rust in Android: move fast and fix things*. Available at `https://blog.google/security/rust-in-android-move-fast-fix-things/`.

[32] Loup Vaillant (2022): *Monocypher: Boring Crypto that Simply Works*. Available at `https://monocypher.org`.

[33] White House Office of the National Cyber Director (2024): *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. Available at `https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf`.