Hardware/Software Co-Assurance using the Rust Programming Language and ACL2

David Hardin

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

The Rust programming language has garnered significant interest and use as a modern, type-safe, memory-safe, and potentially formally analyzable programming language. Our interest in Rust stems from its potential as a hardware/software co-assurance language, with application to critical systems such as autonomous vehicles. We report on the first known use of Rust as a High-Level Synthesis (HLS) language. Most incumbent HLS languages are a subset of C. A Rust-based HLS brings a single modern, type-safe, and memory-safe expression language for both hardware and software realizations with high assurance. As a study of the suitability of Rust as an HLS, we have crafted a Rust subset, inspired by Russinoff's Restricted Algorithmic C (RAC), which we have imaginatively named Restricted Algorithmic Rust, or RAR. In our first implementation of a RAR toolchain, we simply transpile 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. In this paper, we describe the RAR Rust subset, detail our prototype RAR toolchain, and describe the implementation and verification of several representative algorithms and data structures written in RAR, with proofs of correctness conducted using the ACL2 theorem prover.

1 Introduction

The Rust programming language has garnered significant interest and use as a modern, type-safe, memory-safe, and potentially formally analyzable programming language. Google [29] and Amazon [20] are major Rust adopters, and Linus Torvalds has commented positively on the near-term ability of the Rust toolchain to be used in Linux kernel development [1]. The latter capability 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 [19] despite the emergence and use of sophisticated C/C++ analysis tools. Moreover, the extremely aggressive optimizations in modern C/C++ compilers have lead some researchers to declare that C is no longer suitable for system-level programming [32], arguably C's major *rasion d'etre*.

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 [18].) In this paper, we explore the use of Rust as a High-Level Synthesis (HLS) language [23]. Most incumbent HLS languages are a subset of C, e.g. Mentor Graphics' Algorithmic C [16], or Vivado HLS by Xilinx [31]. 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

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 [11]. Particular progress has been made in the area of hardware/software co-design algorithms, where array-backed data structures are common [6, 7]. (NB: This style of programming also addresses one of the shortcomings of Rust, namely its lack of support for cyclic data structures.)

As a study of the suitability of Rust as an HLS, we have crafted a Rust subset, inspired by Russinoff's Restricted Algorithmic C (RAC) [27], which we have imaginatively named Restricted Algorithmic Rust, or RAR. In fact, 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 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 [6]);

- a significant subset of the Monocypher [30] 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 [9]).

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 the performing full functional correctness proofs of RAR code. We will then conclude with related and future work.

2 An Aspirational Integrated Toolchain

In order to place our research goals in context, let us consider an aspirational integrated hardware/software co-assurance toolchain, as shown in Fig. 1.

In this approach, developers can input various familiar high-level specifications, and produce verified implementations for those specifications, in software, hardware, or a combination of the two. Let us now consider each element of Fig. 1 in turn, proceeding left-to-right and top-to-bottom.

2.1 Inputs

Many different common input specification types are anticipated, from AADL models to algorithms in type-safe programming languages to ABNF protocol specifications to lexer/parser specifications; only a
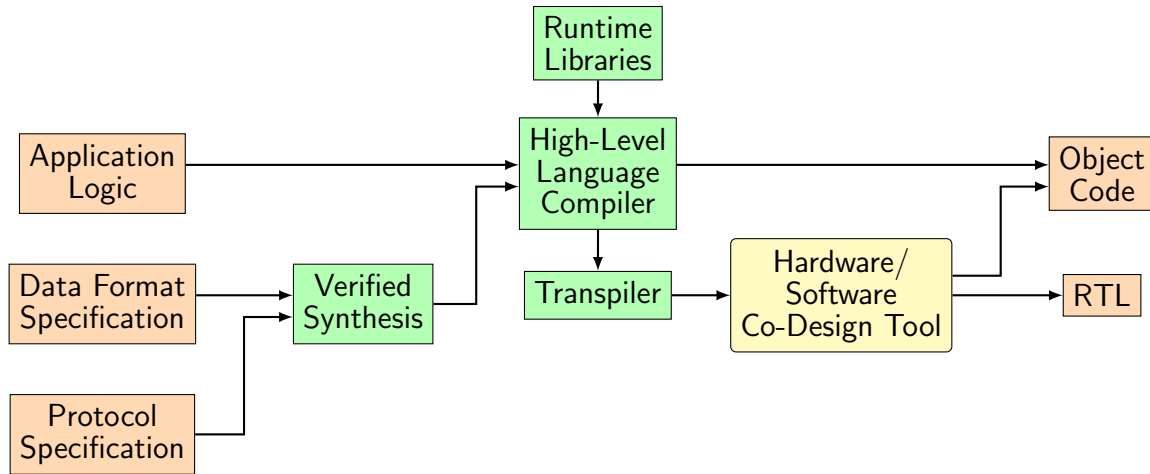
Figure 1: Aspirational integrated hardware/software co-design, co-assurance toolchain.

few representative input types are shown in Fig. 1. The input specification formats for Verified Synthesis will be discussed further in Section 2.2. As for Application Logic, our approach anticipates inputs in modern, type-safe language(s) supported by verified compilation, as described in Section 2.4.

These input forms may be subject to certain subsetting rules, as not all input specification utterances are appropriate for formal synthesis/analysis. Additionally, input specifications may be augmented with pragmas (often introduced as structured comments) to help guide the execution of the formal tools described in the sections that follow.

2.2   Verified Synthesis

Verified Synthesis constitutes a number of different verified program synthesis tools for declarative specification input forms commonly utilized by engineers and computer scientists, including state machine specifications, protocol specifications (*e.g.*, ABNF), regular expressions, grammars, and the like. Our goal is for the Verified Synthesis tools to generate source code in the language(s) of the High-Level Language Compiler(s) of Section 2.4.

2.3   Runtime Libraries

Runtime Libraries constitute a set of runtime services that a verified application may require. These include the usual runtime services, as well as a number of common algebraic data types. Our goal is for the runtime libraries to be formally verified, inspired by the verified runtime libraries for CakeML [11].

Note that high-assurance design rules in particular domains may require specializations of common runtime library services. For example, cyber-physical system designers generally limit the space and time allocations for any given function, and require that algorithms deliver results within a finite time, or suffer a watchdog timeout. Furthermore, domain-specific high-assurance design rules, such as mandated by RTCA DO-178C Level A [26] for flight-critical systems, frown on dynamic memory allocation, preferring simple array-based data structure implementations. This discipline also benefits hardware/software co-design, as array-based implementations are much easier to realize in hardware than dynamic data structures, with their requirements for *malloc* and *free* operations — not to mention the attendant programming errors that can result from dynamic memory management, eg, use-after-free.

Thus, an algebraic data type may include a high-level, functional implementation for which correctness proofs are easier to obtain, as well as a fixed-memory version, accompanied by a proof stating that the latter specializes the former. These specialization proofs could possibly be performed with the aid of machine-learning-based automated proof refinement capabilities in the future.

2.4 High-Level Language Compiler

We strongly advocate type-safe and memory safe modern languages, preferably with verified compilers. A few successful verified compilers have been created in recent years, notably CompCert C [12] and CakeML [11]. However, C is hardly an appealing language for future high-assurance system development; a verified compiler does little to address the many shortcomings of C from an assurance perspective, including unrestricted pointer arithmetic, a seemingly never-ending parade of buffer overflow and other memory integrity/memory management vulnerabilities, unrestricted function pointers, *etc*. On the other hand, Standard ML has a rather small developer community, and its functional orientation makes it not particularly well-suited for embedded systems programming. The F* dialect of ML, however, is the basis for some very interesting work on high-assurance network protocols at Microsoft Research [24] that shares many of our protocol verification objectives.

We aspire to provide verified compilation support for high-assurance subsets of popular, modern functional/imperative hybrid languages, *e.g.*, Scala, Rust, or Swift. These languages exhibit type safety, restrict pointer operations, and are capable of producing efficient, product-quality code. An exemplary approach would include the creation of a *verified compiler toolkit*, which would be used to develop a set of verified compilers for semantically similar, but perhaps syntactically quite different, languages. Additionally, it is hoped that the verified synthesis tools of Section 2.2 can be employed to build verified compilers using this toolkit approach.

2.5 Transpiler

A Transpiler in the context of Fig. 1 is a verified source-to-source translator that takes the development source language(s) to a language that can be processed by the Hardware/Software Co-Design Tool. A Transpiler in our context translates a higher-level language, *e.g.*, Rust or Scala, to a lower-level one, *e.g.*, Algorithmic C; an unverified transpiler of this sort has been created by Robby of Kansas State University, and is part of the DARPA CASE tools [25]. It is hoped that the verified compiler toolkit described in Section 2.4 can be used to build any needed transpilers with high assurance.

2.6 Hardware/Software Co-Design Tool

A Hardware/Software Co-Design tool, as the name implies, allows a developer to allocate her designs to hardware, software, or a combination of the two, producing software code and RTL as output. Sophisticated co-design environments also allow the user to perform "what if" analyses, with the aid of simulation capability, adjusting the allocations to hardware vs. software in order to optimize for desired properties. Example tools of this sort include Simulink [15], as well as the development environments for "system" languages such as SystemC, Algorithmic C, *etc*. The Hardware/Software Co-Design Tool is not likely to be a verified program itself (at least not initially), but should at least provide the capability to export a design model that can be analyzed using formal verification tools.

We have been evaluating a particular hardware/software design approach employed by floating-point hardware verification researchers, detailed further in Section 3.

Formal Verification "Comfort Zone"	*Real-World Development*
Functional programming	Imperative programming
Total, terminating functions	Partial, potentially non-terminating functions
Non-tail-recursive functions	Loops
Okasaki-style pure functional algebraic data types	Structs, Arrays
Infinite-precision Integers, Reals	Modular Integers, IEEE 754 floating point
Linear Arithmetic	Linear and Non-linear Arithmetic
Arithmetic or Bit Vectors	Arithmetic *and* Bit Vectors

Table 1: Formal verification vs. real-world development attributes.

2.7   Object Code

"Object Code" is a broad term for the output of compilation/assembly, from machine-independent virtual machines (*e.g.*, JVM [13], LLVM [14]) to the machine code for a given CPU (*e.g.*, ARM, x86, RISC-V, PowerPC). An object code file is typically encoded in a binary format. We are specifically interested in compilation toolchains, such as CakeML, that provide verified compilation down to the object code level; as well as verified "lifters" that can take object code, and abstract it to functions in a given logic, *e.g.*, Myreen's Decompilation into Logic work [22].

2.8   RTL (Register Transfer Logic)

Register Transfer Logic (RTL) describes a hardware design at a functional level, but at a sufficient level of fidelity that allows for automated synthesis of a gate-level netlist, as well as automated analysis and simulation. This netlist can then be placed and routed to produce FPGA or ASIC implementations. RTL is expressed using Hardware Description Languages (HDLs), such as VHDL, Verilog, SystemVerilog, or Bluespec.

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 [27], 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 [16] is a High-Level Synthesis (HLS) language, and is supported by hardware/software co-design environments from Mentor Graphics, *e.g.*, Catapult [17]. 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.
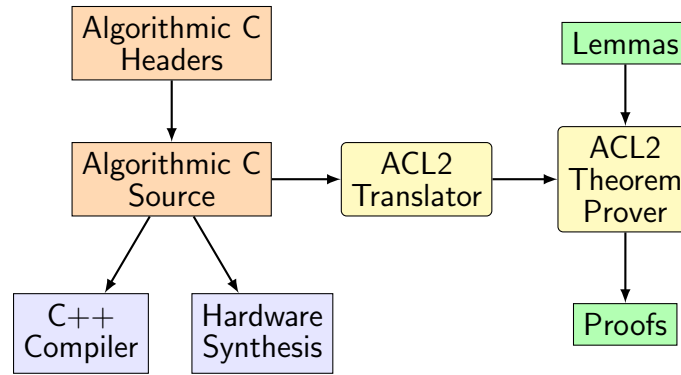
Figure 2: Restricted Algorithmic C (RAC) toolchain.

The ACL2 Translator component of Fig. 2 provides a case study in the bridging of Formal Modeling and Real-World Development concerns, as summarized in Table 1. 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 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.

Recently, we have investigated the synthesis of Field-Programmable GateArray (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 the Architecture Analysis and Design Language (AADL) [4], with proofs of correctness in ACL2.

4 Rust and RAR

The Rust Programming Language [10] 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 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) [3], 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 sytem, named cargo.
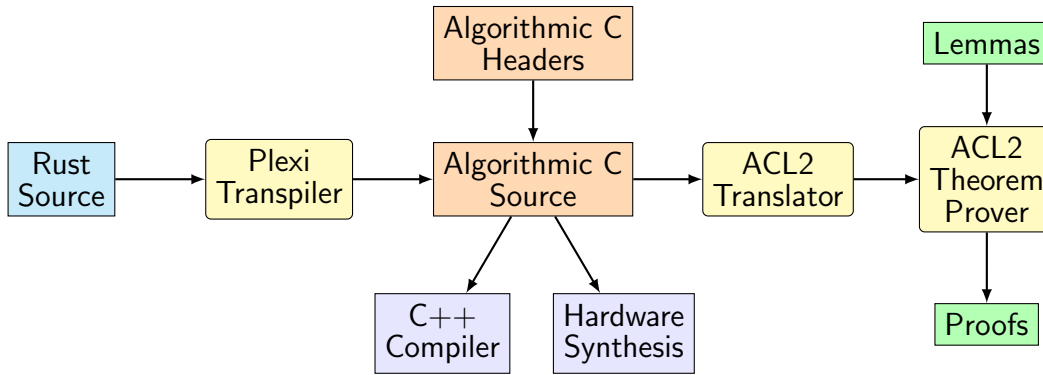
Figure 3: Restricted Algorithmic Rust (RAR) toolchain.

4.1 Restricted Algorithmic Rust

As we wish to utilize the RAC toolchain as a backend in our initial work, the 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 [28] source code. We thus call our transpiler *Plexi*, a 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.

5   Example: Array-Based Set

In this section, we describe an array-based set implementation using RAR. 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 Collins high-assurance development [8].

5.1   Arrayset Definitions

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

```
const ARR_SZ: uint = 256;

#[derive(Copy, Clone)]
struct Arrayset {
  anext: [usize; ARR_SZ],
  avals: [i64; ARR_SZ],
  free_head: usize,
  used_head: usize,
}
```
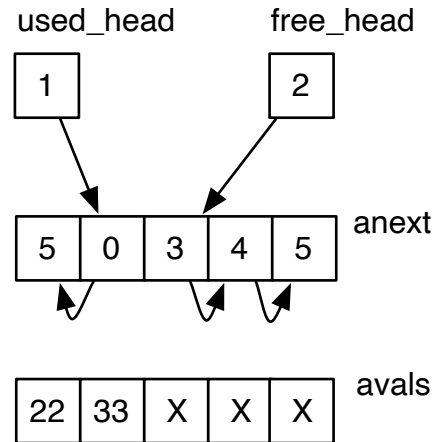
Figure 4: An Arrayset data structure with contents {33, 22}, size = 5.

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_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. Note that indices in Rust are normally declared to be of the `usize` type.

The ingenious bit about 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. The free list and used list are both terminated by elements in the next array with the value `ARR_SZ`. Any value outside the range of valid element indices would do for the terminators. One of the jobs of the mutators `aset_init()`, `aset_add()`, 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 ACL2 proofs to show that this is the case.

Fig. 4 shows the state of an Arrayset of size 5 after some number of `aset_add` and `aset_del` operations have been performed. The `used_head` is 1, which is an index into the `anext` array (note that all arrays are zero-based). The `free_head` is 2. If we read the `anext` array at index `used_head`, we get the next element in the used list, namely 0. If we then read the `anext` 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 `avals` array are 33 and 22; thus, the Arrayset content is {33, 22}. (Note that all other components of `avals` have values, but since they are free elements, those values do not matter.)

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

If we then execute `aset_del(22, aset)`, we obtain the Arrayset shown in Figure 5. 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. 6. The operation of `aset_add()` is as follows. 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, 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
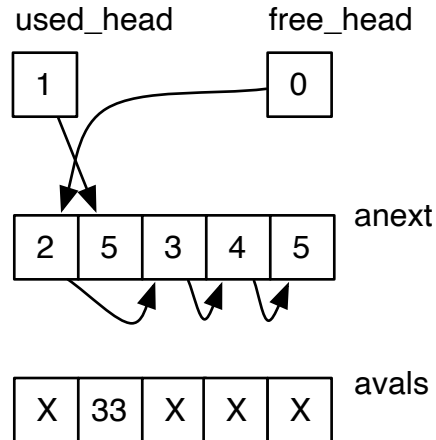
used_head   free_head

Figure 5: The Arrayset after deleting element 22.

```
fn aset_add(val: i64, aset: mut Arrayset) -> Arrayset {
  let curr_index: usize = aset.free_head;

  if (curr_index >= ARR_SZ) {
    return aset;                    // Full
  } else {
    if ((aset.used_head < ARR_SZ) && aset_is_element(val, aset)) {
      return aset;
    } else {

      aset.free_head = aset.anext[aset.free_head];
      aset.avals[curr_index] = val;
      aset.anext[curr_index] = aset.used_head;
      aset.used_head = curr_index;

      return aset;
    }
  }
}
```

Figure 6: aset_add() function in RAR.

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 of [2].

aset_del() proceeds similarly, although it is admittedly a bit more complicated (see Fig. 7). aset_del(), in turn, depends on the aset_element_prev_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 Arrayset bookkeeping.

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_add() appears in Fig. 8.

The first thing to note about Fig. 8 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 start 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 (aset)
  (not (= (ag 'free_head aset) (ag 'used_head aset)))))
```

We then define a "good state" predicate. This function states that for a good state, the aset input satisfies the arraysetp wellformedness predicate; the free_head is not equal to the used_head, there are no duplicates of val in the avals array, and that the length of the free list plus the length of the used list is equal to the total length.

```
(defun good-statep (val aset)
  (and (arraysetp aset)
       (free-head-used-head-relation aset)
       (no-dups val aset)
       (= (+ (aset_len aset) (aset_len_free aset)) (arr_sz))))
```

Given this definition of a good Arrayset state, we can prove functional correctness theorems for the Arrayset operations, of the sort stated below:

```
(defthm aset_add-works--thm
  (implies
   (and (good-statep val aset)
        (integerp val)
        (< (aset_len aset) (arr_sz)))
   (= (aset_is_element val (aset_add val aset)) 1)))
```

```
fn aset_del(val: i64, aset: mut Arrayset) -> Arrayset {
  let mut curr_index: usize = aset.used_head;
  let mut prev_index: usize;

  if (aset.used_head >= ARR_SZ) {
    return aset;                        // Empty
  } else {

    if (aset.avals[curr_index] == val) {
      aset.used_head = aset.anext[curr_index];
      aset.anext[curr_index] = aset.free_head;
      aset.free_head = curr_index;

      return aset;
    } else {
      prev_index = aset_element_prev_from(aset.used_head, val, aset);

      if (prev_index >= ARR_SZ) {
        return aset;
      } else {

        curr_index = aset.anext[prev_index];

        if (curr_index >= ARR_SZ) {
          return aset;
        } else {
          aset.anext[prev_index] = aset.anext[curr_index];
          aset.anext[curr_index] = aset.free_head;
          aset.free_head = curr_index;

          return aset;
        }
      }
    }
  }
}
```

Figure 7: aset_del function in RAR.

```
(DEFUND ASET_ADD (VAL ASET)
    (LET ((CURR_INDEX (AG 'FREE_HEAD ASET)))
        (IF1 (LOG>= CURR_INDEX (ARR_SZ))
            ASET
            (IF1 (LOGAND1 (LOG< (AG 'USED_HEAD ASET) (ARR_SZ))
                         (ASET_IS_ELEMENT VAL ASET))
                ASET
                (LET* ((ASET (AS 'FREE_HEAD
                                 (AG (AG 'FREE_HEAD ASET)
                                     (AG 'ANEXT ASET))
                                 ASET))
                       (ASET (AS 'AVALS
                                 (AS CURR_INDEX VAL (AG 'AVALS ASET))
                                 ASET))
                       (ASET (AS 'ANEXT
                                 (AS CURR_INDEX (AG 'USED_HEAD ASET)
                                     (AG 'ANEXT ASET))
                                 ASET)))
                    (AS 'USED_HEAD CURR_INDEX ASET))))))
```

Figure 8: aset_add() function translated to ACL2 using the RAC tools.

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.

6   Comparison to other Automated Verification Tools

The Arrayset example has so far been subjected to automated formal verification by two other approaches: model-checking [8] and symbolic execution [2]. In the former case, the Arrayset example, as well as its correctness properties, were formulated as Simulink/Stateflow [15] models (indeed, this was the original form), and processed by the Collins Gryphon toolset [21]. 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 [8].

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 [8], 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].

Our current work also can be compared and contrasted to a previous formalization of the array-

backed set we developed using ACL2's single-threaded object capability [5]. Whereas the use of a stobj made for a much more performant ACL2 implementation, which was useful for validation testing, the stobj-baased formalization had to be constructed by hand. Thus, no direct connection to an imperative implementation, of the sort commonly written by a non-specialist developer, could be made, in contrast to the current approach. Further, reasoning about untyped records is considerably easier than reasoning about stobj's, at least in our experience. We additionally note that in both ACL2 formalizations, all Arrayset operations were written in tail-recursive style, so this was not a factor in the ease of proof.

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 dequeues) 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.

In the particular case of an array-backed set, we were able to compare the scalability of several verification 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. 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.

In future work, we will continue to develop our toolchain, increasing the number of Rust features that we can support in the RAR subset, as well as continuing to improve the ACL2 verification libraries in order to increase the ability to discharge RAR correctness proofs automatically. We will also continue to work with our colleagues at Kansas State University on the direct synthesis and verification of FPGA hardware from RAR source code.

8 Acknowledgments

References

[1] Ron Amadeo (2021): *Google is now writing low-level Android code in Rust*. Available at `https://arstechnica.com/gadgets/2021/04/google-is-now-writing-low-level-android-code-in-rust/`.

[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] 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.

[4] Peter H. Feiler & David P. Gluch (2012): *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st edition. Addison-Wesley Professional.

[5] David S. Hardin (2013): *The Specification, Verification, and Implementation of a High-Assurance Data Structure: An ACL2 Approach*. In: *Proceedings of the 46th Hawaii International Conference on System Sciences*, pp. 5059–5067, doi:10.1109/HICSS.2013.541.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Steve Klabnik & Carol Nichols (2018): *The Rust Programming Language*. No Starch Press.

[11] 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.

[12] Xavier Leroy (2009): *Formal verification of a realistic compiler*. Communications of the ACM 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[13] Tim Lindholm, Frank Yellin, Gilad Bracha & Alex Buckley (2013): *The Java Virtual Machine Specification*. Addison-Wesley.

[14] LLVM Project (2020): *The LLVM Compiler Infrastructure*. Available at `http://llvm.org/`.

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

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

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

[18] Eric Mercer, Konrad Slind, Isaac Amundson, Darren Cofer, Junaid Babar & David Hardin (2021): *Synthesizing Verified Components for Cyber Assured Systems Engineering*. In: *24th*

International Conference on Model-Driven Engineering Languages and Systems (MODELS 2021), doi:10.1109/MODELS50736.2021.00029.

[19] 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/`.

[20] Shane Miller & Carl Lerche (2022): *Sustainability with Rust*. Available at `https://aws.amazon.com/blogs/opensource/sustainability-with-rust/`.

[21] 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.

[22] Magnus Myreen (2009): *Formal verification of machine-code programs*. Ph.D. thesis, University of Cambridge.

[23] 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.

[24] Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi & Jonathan Protzenko (2019): *EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats*. In: *USENIX Security '19*.

[25] 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.

[26] RTCA Committee SC-205 (2015): *DO-178C Software Considerations in Airborne Systems and Equipment Certification*. Available at `https://my.rtca.org/nc__store?search=DO-178C`.

[27] 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.

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

[29] Jeff Vander Stoep & Stephen Hines (2021): *Rust in the Android platform*. Available at `https://security.googleblog.com/2021/04/rust-in-android-platform.html`.

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

[31] Xilinx, Inc. (2018): *Vivado Design Suite User Guide: High-Level Synthesis*. Available at `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf`.

[32] Victor Yodaiken (2021): *How ISO C became unusable for operating systems development*. In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems*, pp. 84–90, doi:10.1145/3477113.3487274.