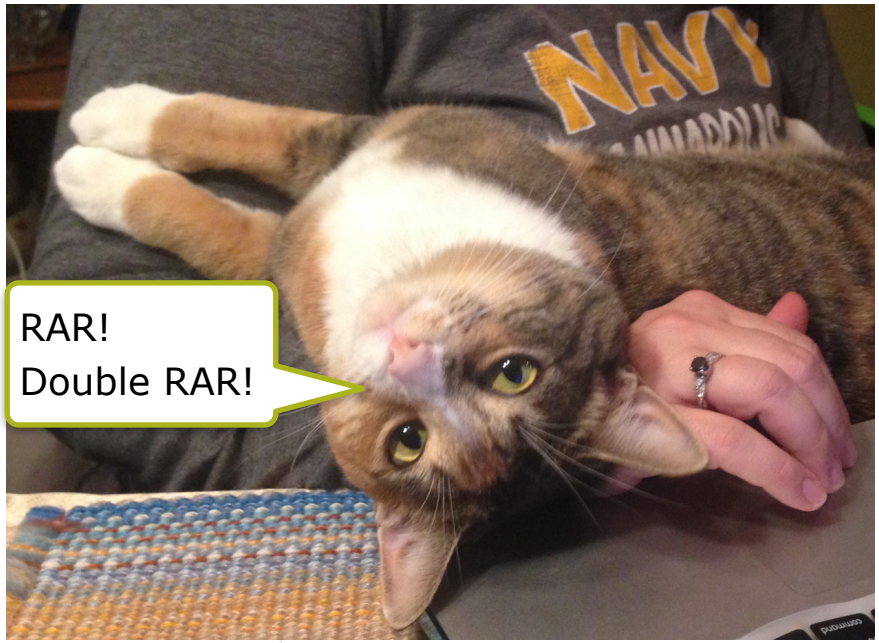
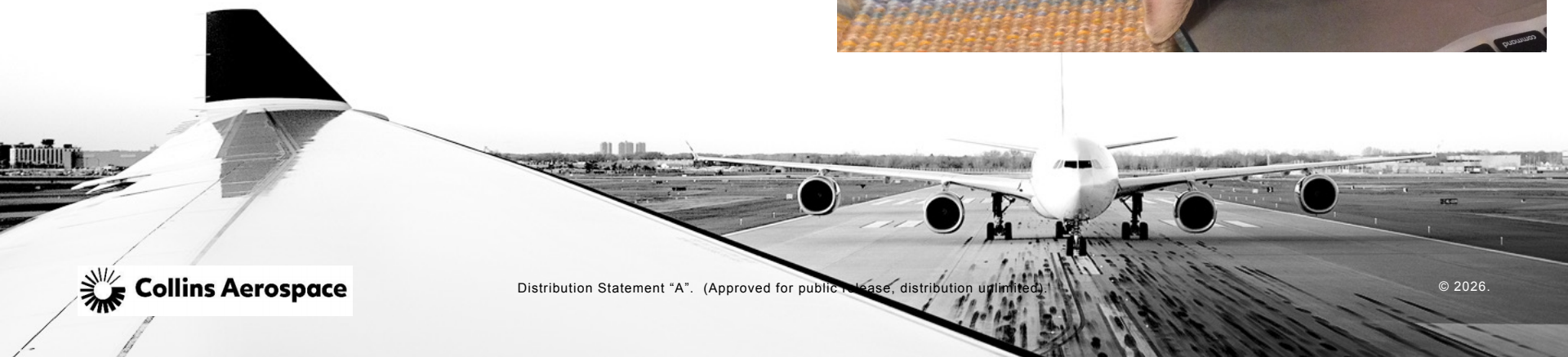
ARRAYSET REVISITED

David S. Hardin

Applied Research and Technology
Collins Aerospace





Collins Aerospace

Distribution Statement “A”. (Approved for public release, distribution unlimited).

© 2026.

DISCLAIMER

The views, opinions and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

No Generative AI systems were harmed or exploited in the creation of this presentation.

Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

DARPA PROVERS

PIPELINED REASONING OF VERIFIERS ENABLING ROBUST SYSTEMS

- Develop automated, scalable formal methods tools that are integrated into traditional development pipelines using “proof engineering” techniques
- Enable product engineers to produce and maintain high-assurance national security systems
- The Collins-led INSPECTA team is making formal verification across the entire software development stack accessible to non-formal methods experts through automated analysis, DevOps integration, and improved user feedback
    - Key Results: SysML v2 System Modeling Language support with assume/guarantee contracts, as well as code generation for modern memory-safe languages, specifically Rust, with Verus annotations
    - We are also researching the use of LLMs to assist in the generation of architecture models, as well as memory-safe language source code, from natural language requirements, employing assume/guarantee contracts and source code annotations that can be automatically verified using SMT solvers
        - Neuro-symbolic approach thwarts hallucinations

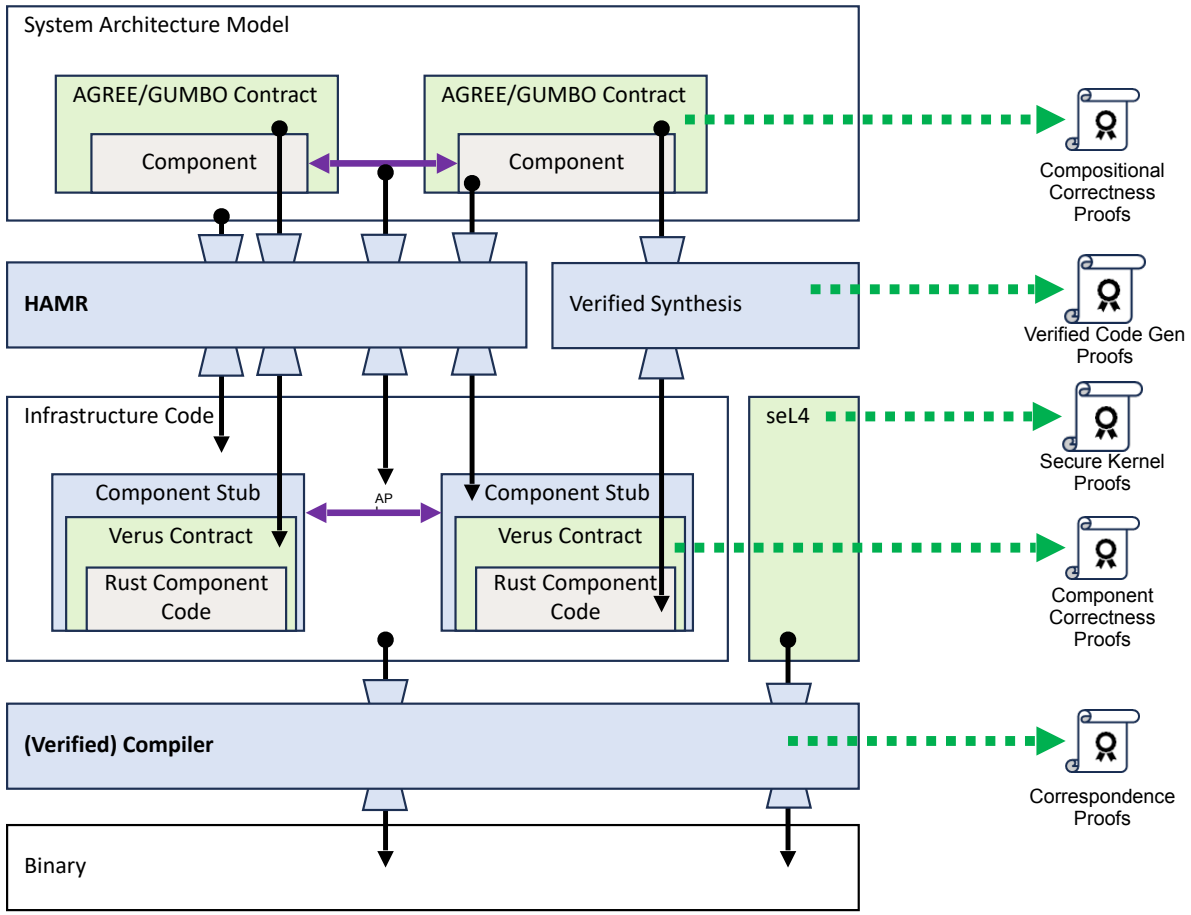
Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

DARPA PROVERS: INSPECTA TOOLCHAIN

INDUSTRIAL-SCALE PROOF ENGINEERING FOR CRITICAL TRUSTWORTHY APPLICATIONS

With INSPECTA, engineers are able to generate comprehensive formal assurance across the entire development stack without requiring deep formal methods expertise.

https://loonwerks.com/projects/inspecta.html



Distribution Statement "A". (Approved for public release, distribution unlimited)..

© 2026.

HARDWARE/SOFTWARE CO-DESIGN AND CO-ASSURANCE

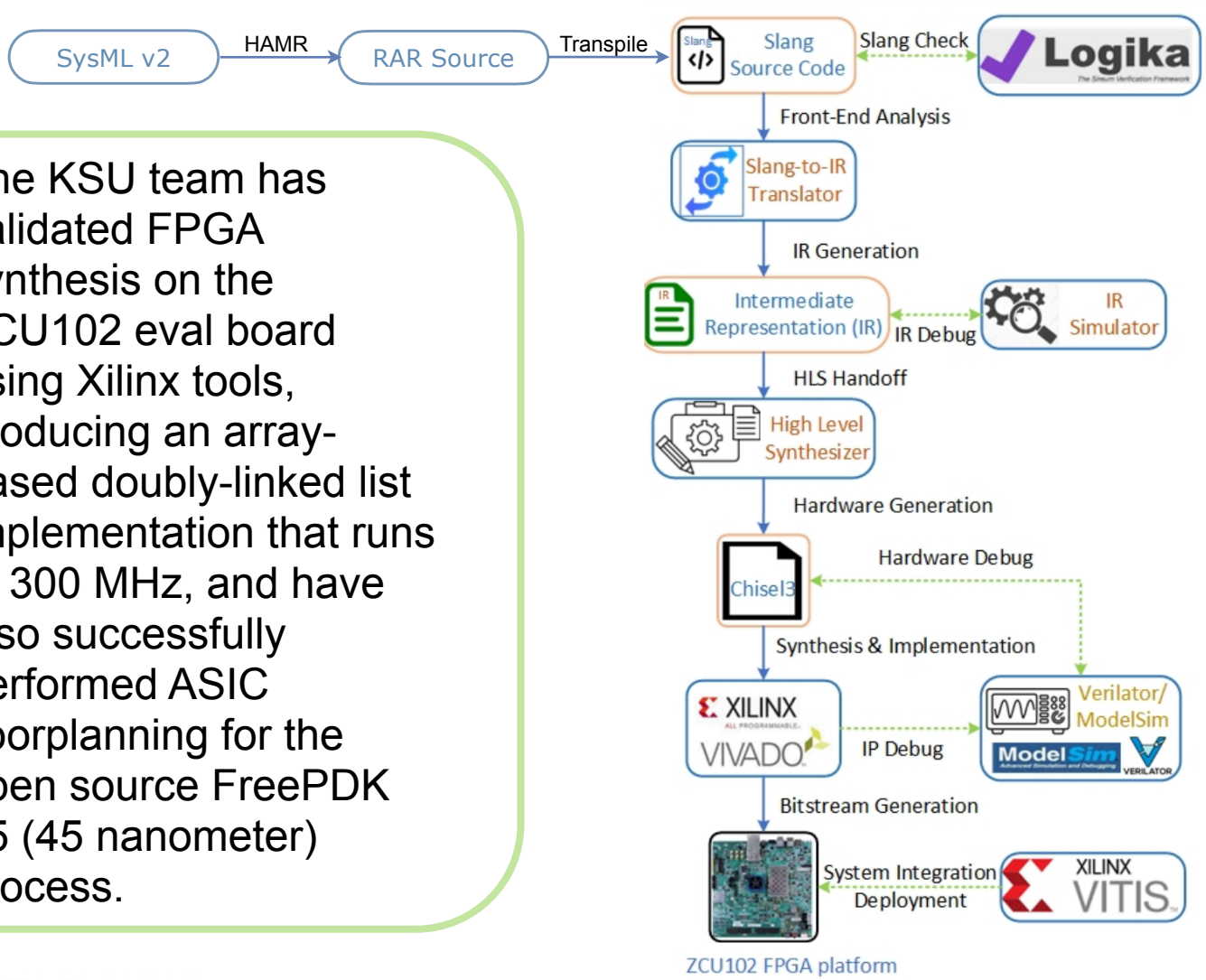
- We desire to create high-assurance architectural components using hardware/software co-design/co-assurance techniques
  - The INSPECTA high-level Architectural Modeling approach supports both hardware- and software-based realizations
  - Being able to defer and/or change the allocation of functionality to hardware or software is highly desirable
  - Hardware provides greater tamper resistance, as well as higher performance

- Thus, we have been investigating the use of High-Level Synthesis (HLS) hardware/software co-design languages that also support formal verification

Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

HARDWARE/SOFTWARE CO-SYNTHESIS FROM SYSML V2 MODELS (KANSAS STATE)

The KSU team has validated FPGA synthesis on the ZCU102 eval board using Xilinx tools, producing an array-based doubly-linked list implementation that runs at 300 MHz, and have also successfully performed ASIC floorplanning for the open source FreePDK 45 (45 nanometer) process.

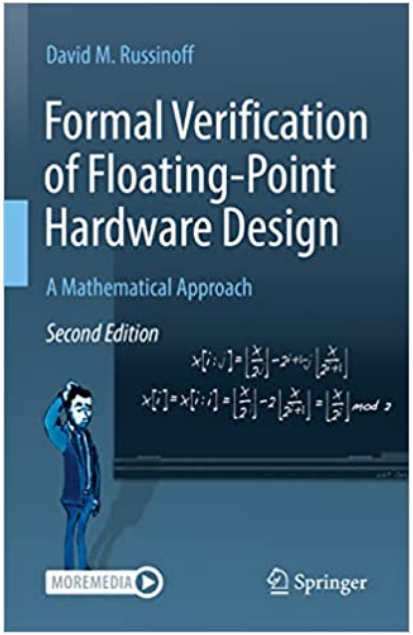


Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

THE RAC APPROACH TO HARDWARE/ SOFTWARE VERIFICATION

- The hardware/software verification approach we employ was developed by David Russinoff and John O'Leary, while both were at Intel
  - The approach was initially based on SystemC, and was called MASC
  - Russinoff changed the source language from SystemC to Algorithmic C after he moved to Arm, made several enhancements, and renamed the system RAC (Restricted Algorithmic C)

- RAC is extensively documented in Russinoff's book, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*, wherein RAC is applied to the verification of realistic Arm floating-point designs
  - RAC, and the verifications described in the book, are all available in the standard ACL2 theorem prover distribution



Distribution Statement "A". (Approved for public release, distribution unlimited)..
© 2026.

ALGORITHMIC C

- The Algorithmic C datatypes “provide a basis for writing bit-accurate algorithms to be synthesized into hardware”
- Example use:
  - `typedef ac_int<112,false> ui112;`

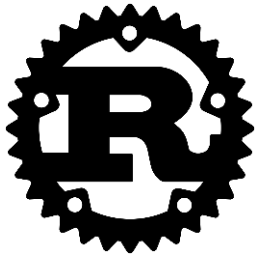
  declares an unsigned 112-bit type used in floating-point hardware datapaths
- Supported by Mentor hardware synthesis tools, e.g. Catapult; for details, see **`https://hlslibs.org`**
- Restricted Algorithmic C (RAC) further restricts Algorithmic C to facilitate proof; see Chapter 15 of Russinoff’s book for details
- NB: We use cpp macros to support either Algorithmic C or Xilinx Vivado HLS in hardware synthesis

Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

REALIZING THE H/W-S/W CO-ASSURANCE VISION USING RUST

- INSPECTA focuses on Rust as the primary application development language.
- Thus, we are exploring the hardware/software co-design co-assurance toolchain vision by supporting a Rust subset called Restricted Algorithmic Rust, or RAR

- Rust has several assurance advantages over C/C++, including:
  - Improved type safety
  - Vastly improved memory safety
    - A “single-owner” rule for memory references (similar to stobjs)
  - No arbitrary pointer arithmetic
  - ***…in short, the sources of 80% of C/C++ security flaws are eliminated outright!***



- Basic Rust syntax is familiar to C/C++ developers, easing the transition
- The Rust compiler produces efficient, and importantly, energy-efficient code, which makes Rust a favorite for embedded and sustainable computing

- Rust has recently broken into the Top 10 of the TIOBE Index of the world’s most popular programming languages

Distribution Statement “A”. (Approved for public release, distribution unlimited)..

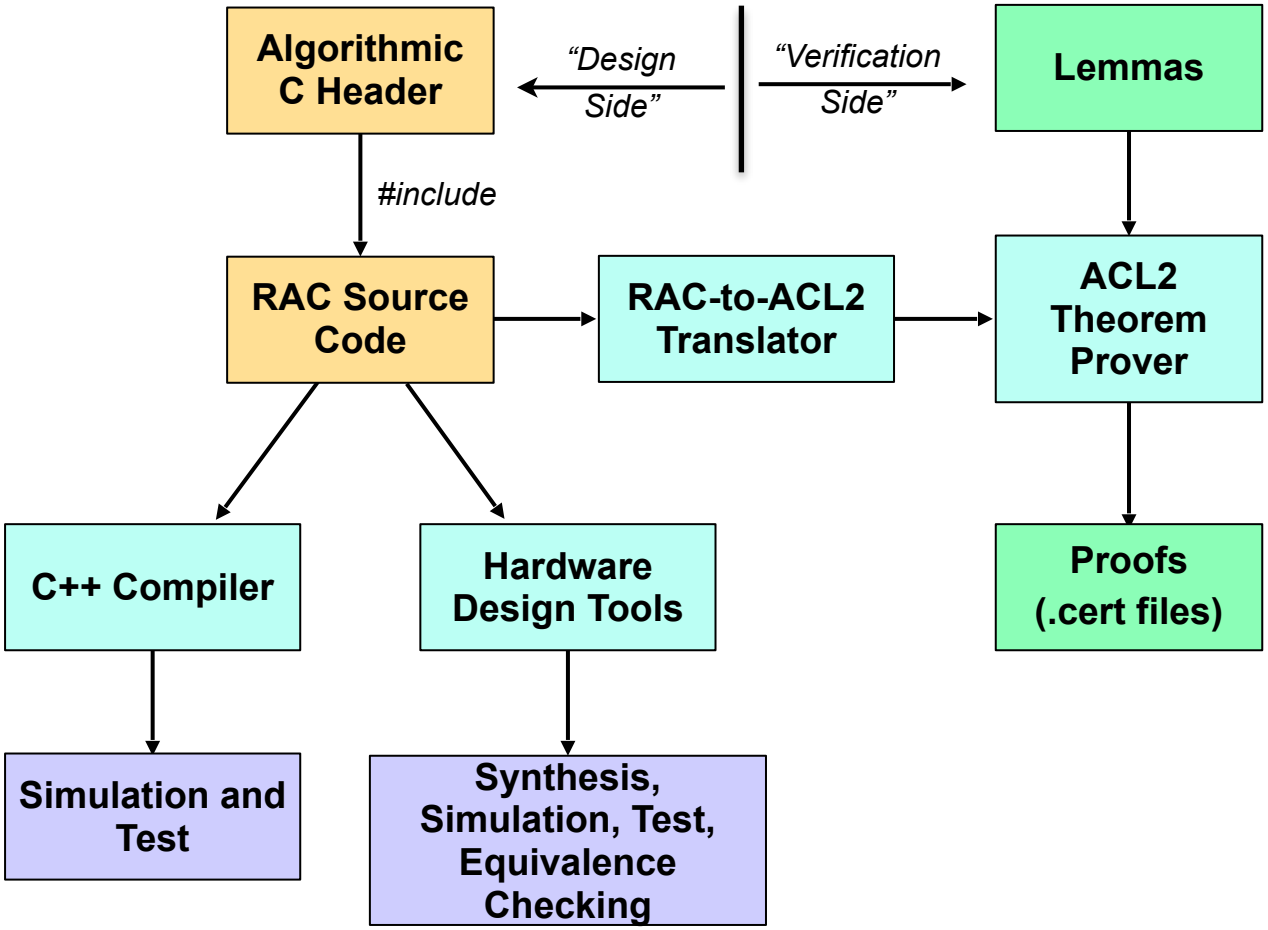
© 2026.

RAR EXAMPLES DEVELOPED TO DATE

- A suite of array-backed algebraic data types, previously implemented in RAC
  - Stack, Singly-linked list, Doubly-linked list, Circular Queue, Deque, etc.
- A significant subset of the Monocypher modern cryptography suite, including XChacha20 and Poly1305 (RFC 8439) encryption/ decryption, Blake2b hashing, and X25519 public key cryptography
- A DFA-based JSON lexer, coupled with an LL(1) JSON parser
  - The JSON parser has also been implemented using Greibach Normal Form

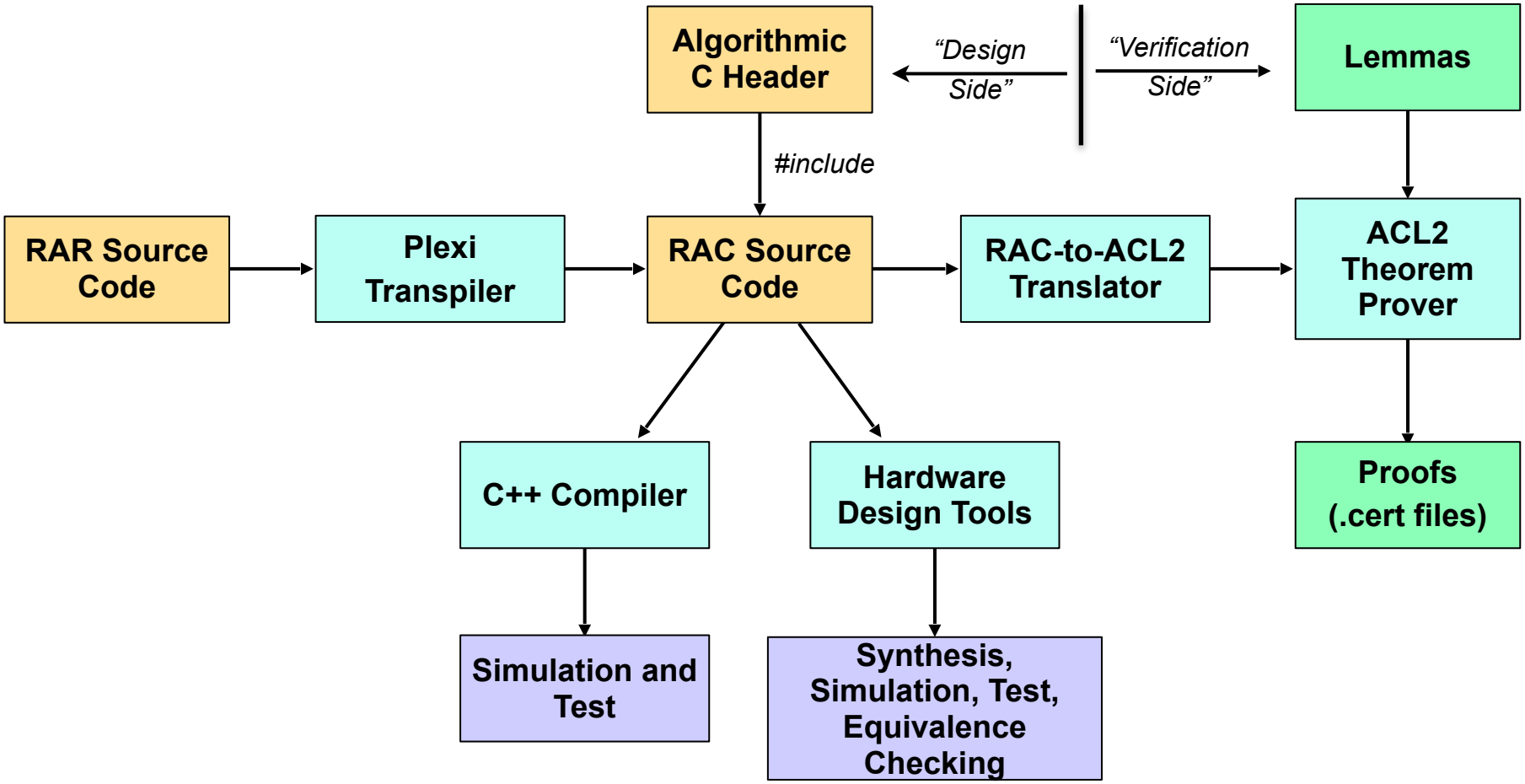
Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

RESTRICTED ALGORITHMIC C TOOLCHAIN



Distribution Statement "A". (Approved for public release, distribution unlimited)..
© 2026.

RESTRICTED ALGORITHMIC RUST TOOLCHAIN



Distribution Statement "A". (Approved for public release, distribution unlimited)..

© 2026.

THE PLEXI RAR-TO-RAC TRANSPILER

- Based on the open source *plex* parser and lexer generator tool, written in Rust
- Translates RAR code to RAC code one line at a time
- Rapid prototyping principles used to produce a tool that works “well enough”
  - Future work will investigate replacing plexi with ACL2-targeted Aeneas (presented at this Workshop)

Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

EXAMPLE: ARRAY-BASED SET

- Technically a multiset, since the add function does not check whether a given value is already stored

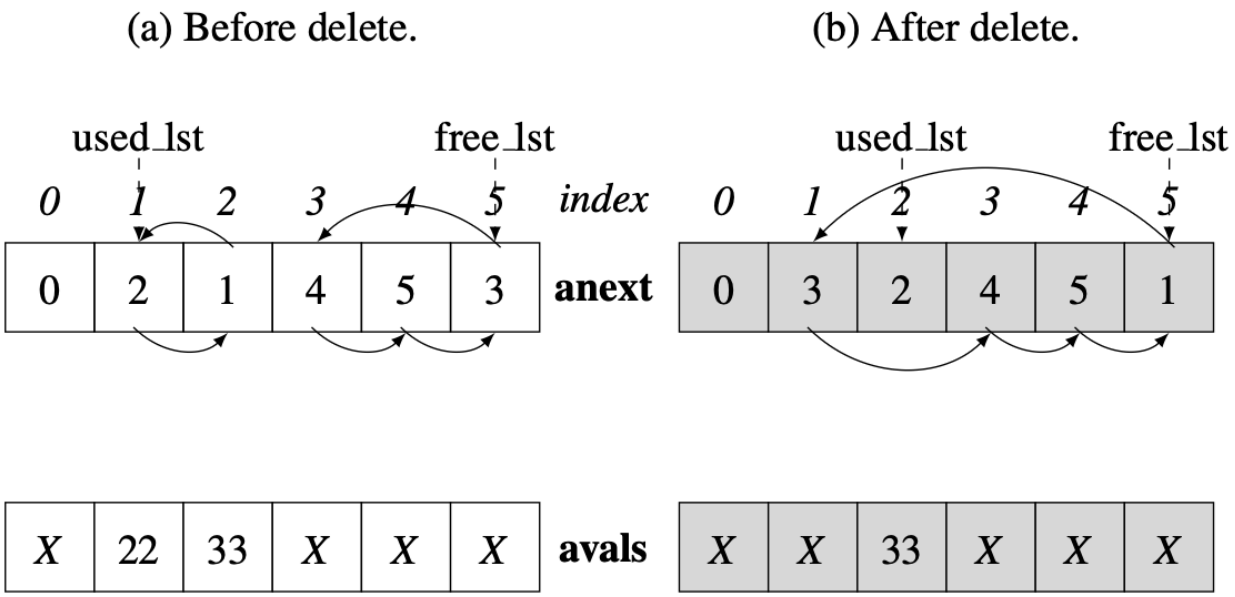
- Array backing means the arrayset has a fixed maximum size — but this size can be quite large (billions of entries)
- Supports free and used lists using a single “anext” array

- Changes since initial implementation (circa 2022)
  - Used and Free lists are now circular; enables an easily-stated invariant — no anext duplicates
  - used_lst and free_lst indices now point to the tail of the used and free lists, respectively, not the head
  - Added an error code in the Arrayset data structure

Distribution Statement “A”. (Approved for public release, distribution unlimited)..

© 2026.

EXAMPLE: ARRAY-BASED SET



(a) Arrayset with contents {33, 22}, size = 5

(b) After delete of element 22

Distribution Statement "A". (Approved for public release, distribution unlimited)..
© 2026.

RAR EXAMPLE: ARRAY-BASED SET

```
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,
}
```


Distribution Statement “A”. (Approved for public release, distribution unlimited)..

© 2026.

RAR EXAMPLE: ARRAY-BASED SET (CONT'D.)

```
fn aset_add(val: i64, mut aset: Arrayset) -> Arrayset {
  let mut idx: usize = ARR_SZ;

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

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


fn aset_del_head(mut aset: Arrayset) -> Arrayset {
  let idx: usize;

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

  return aset;
}
```


Distribution Statement "A". (Approved for public release, distribution unlimited)..
© 2026.

RAR EXAMPLE: ARRAY-BASED SET (CONT'D.)

```
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_lst pointer
      aset.used_lst = prev_index;
      aset = aset_del_head(aset);
    } else {
      aset.err = ARR_SZ;
    }
  } else {
    aset.err = ARR_SZ;
  }
  return aset;
}
```


Distribution Statement "A". (Approved for public release, distribution unlimited)..

© 2026.

TRANSLATION TO ACL2

```
(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)))
```


Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

EXAMPLE RAR CORRECTNESS THEOREMS

```
(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))))
```


Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

COMPARISON TO OTHER VERIFICATION TOOLS

- This basic array-based set has been the subject of several verification attempts
  - Early attempts based on model-checking were unable to verify any but the smallest-sized arraysets (8 entries or fewer)
  - The modern SMT-based Verus Rust verifier scales successfully to large maximum arrayset sizes
    - Currently unable to verify all properties proved in ACL2 using Verus, due to ACL2's superiority in list-based reasoning

- When automated verification tools (e.g., SMT solvers) do not suffice, we need to translate our code into the language of an automated theorem prover — in our case, ACL2
  - Other PROVERS work is researching translation to Lean
  - Work by Robby at Kansas State is exploring another alternative, namely adding induction support to SMT-based code reasoning (Logika)

Distribution Statement "A". (Approved for public release, distribution unlimited)..
© 2026.

VERUS

- Verus is an open source Rust code verification environment under development by an international team led by Carnegie Mellon University
- Industry users include AWS, Microsoft

- Verus has been utilized in a number of operating system, concurrent data structure, and distributed algorithm verification efforts

- Verus utilizes Rust syntax to express precondition and postcondition annotations, loop invariants, etc.

- Verus employs an SMT solver backend to prove “ensures” postconditions, given “requires” preconditions

Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

VERUS SPEC FOR ASET_DEL FUNCTION

```
#[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.anext@.no_duplicates(),
    ret.avals.len() == ARR_SZ,
    ret.used_lst != ret.free_lst,
    (ret.err == 1) ==> (aset.avals[ret.anext[ret.free_lst as int] as int] == val),
)]
```


Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

CONCLUSION AND FUTURE WORK

- The INSPECTA toolchain developed under DARPA PROVERS can be used by non-specialist developers to create verified system architectures specified in SysML v2, with automatically synthesized and verified implementations
  - The INSPECTA is team is developing a hardware/software co-design/co-assurance approach, focusing on the Rust programming language

- When automated verification tools (e.g., SMT solvers) do not suffice, we need to translate Rust into the language of an automated theorem prover.
  - This work focuses on translation of Rust code to ACL2
  - The efforts reported here stand on the broad shoulders of the great Restricted Algorithmic C work
    - Thus producing Restricted Algorithmic Rust (RAR)

Distribution Statement “A”. (Approved for public release, distribution unlimited)..
© 2026.

CONCLUSION AND FUTURE WORK (CONT'D.)

- We have used RAR to implement and verify an array-based set, revisiting an earlier reported verification effort from 2022.

- We improve the earlier work by providing full functional correctness proofs, as well as generalized invariants that (hopefully) makes verification easier for all provers/solvers.

- In future work, we will continue to enhance our tools, focusing on:
  - Use of ACL2-targeted Aeneas framework
  - Enhanced integration with KSU hardware synthesis effort
  - Continued development of Verus proof infrastructure for comparison

Distribution Statement "A". (Approved for public release, distribution unlimited)..
© 2026.