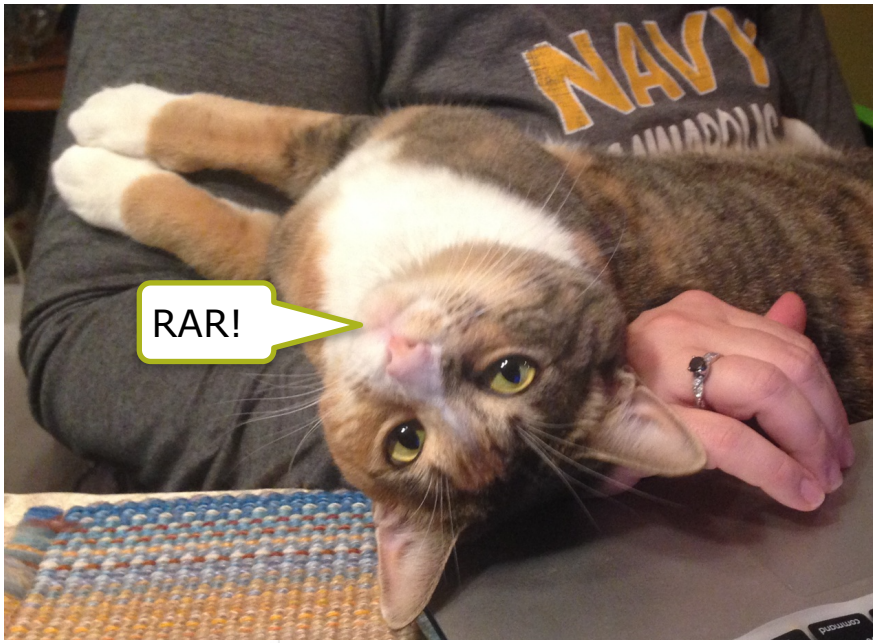
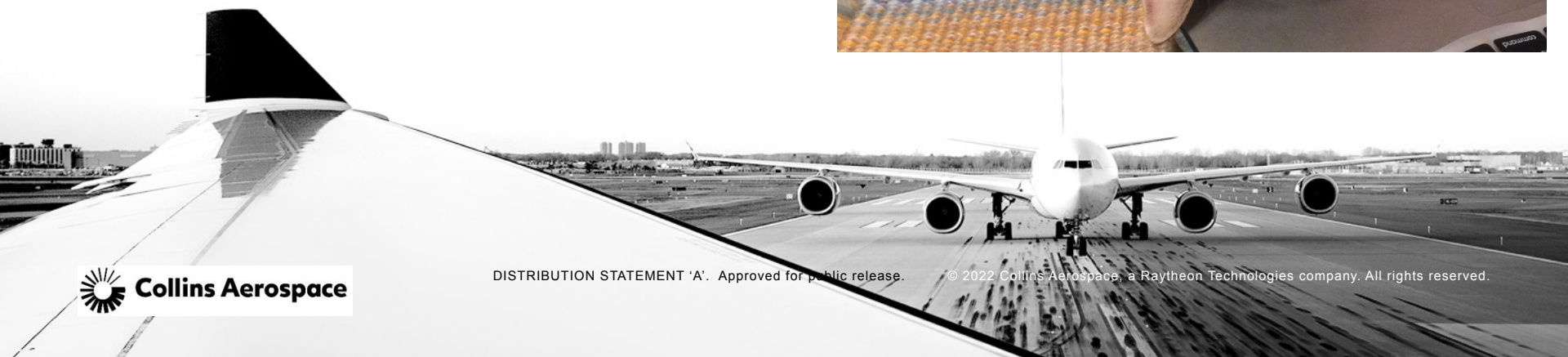
HARDWARE/SOFTWARE CO-ASSURANCE USING THE RUST PROGRAMMING LANGUAGE AND ACL2

David S. Hardin

Applied Research and Technology
Collins Aerospace





Collins Aerospace

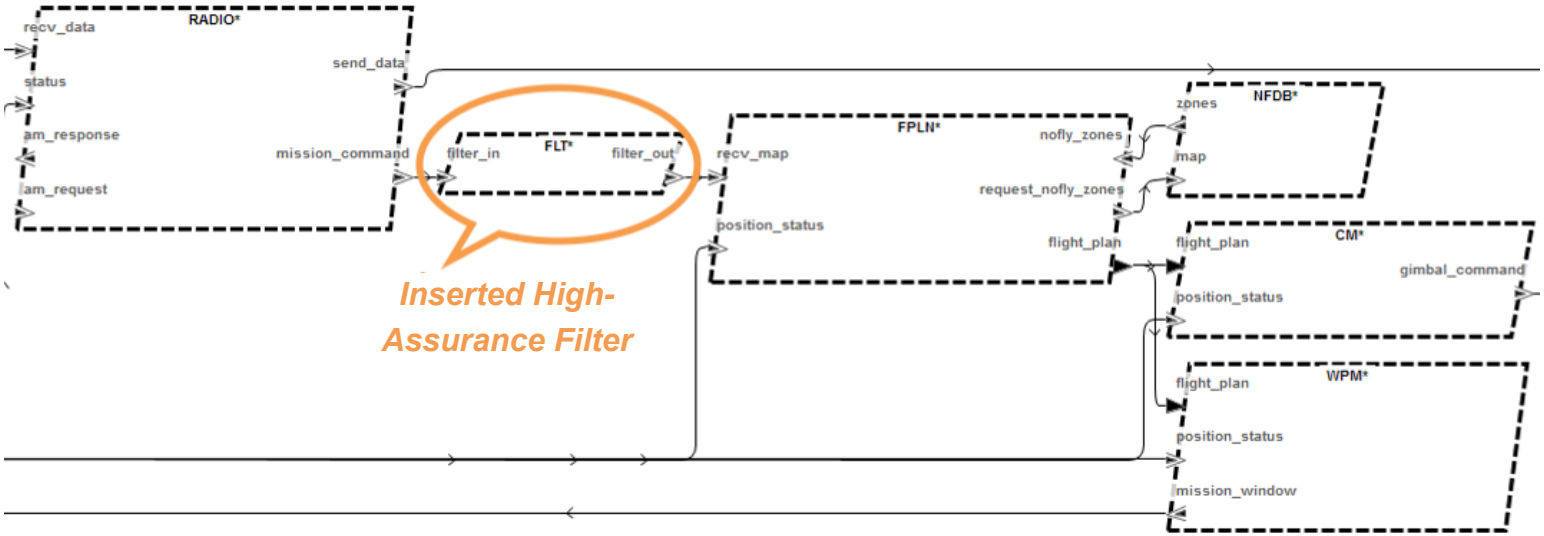
DISCLAIMER

The views expressed are those of the authors and do not reflect the official policy or position of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

DARPA CASE

- The goal of the DARPA Cyber-Assured Systems Engineering (CASE) program is to “develop the necessary design, analysis and verification tools to allow system engineers to design-in cyber resiliency”
- Architecture models in the DARPA CASE program are expressed in the SAE standard Architectural Analysis and Design Language (AADL)
- The CASE Cyber Requirements tools examine the AADL model for the system, identifying potential cyber vulnerabilities
- The CASE user then identifies *security-enhancing architectural transformations* to be applied to the model to address the vulnerabilities
- Let’s say the need for an input well-formedness filter was identified:
  - The CASE user adds the filter to the model, and specifies the high-level filter behavior, e.g. using a regular expression
  - The CASE tools then automatically synthesize the filter and produce a proof of filter correctness all the way down to the binary level
  - This filter is hosted on a high-integrity operating system, e.g. seL4

DARPA CASE: SIMPLE UAV USE CASE

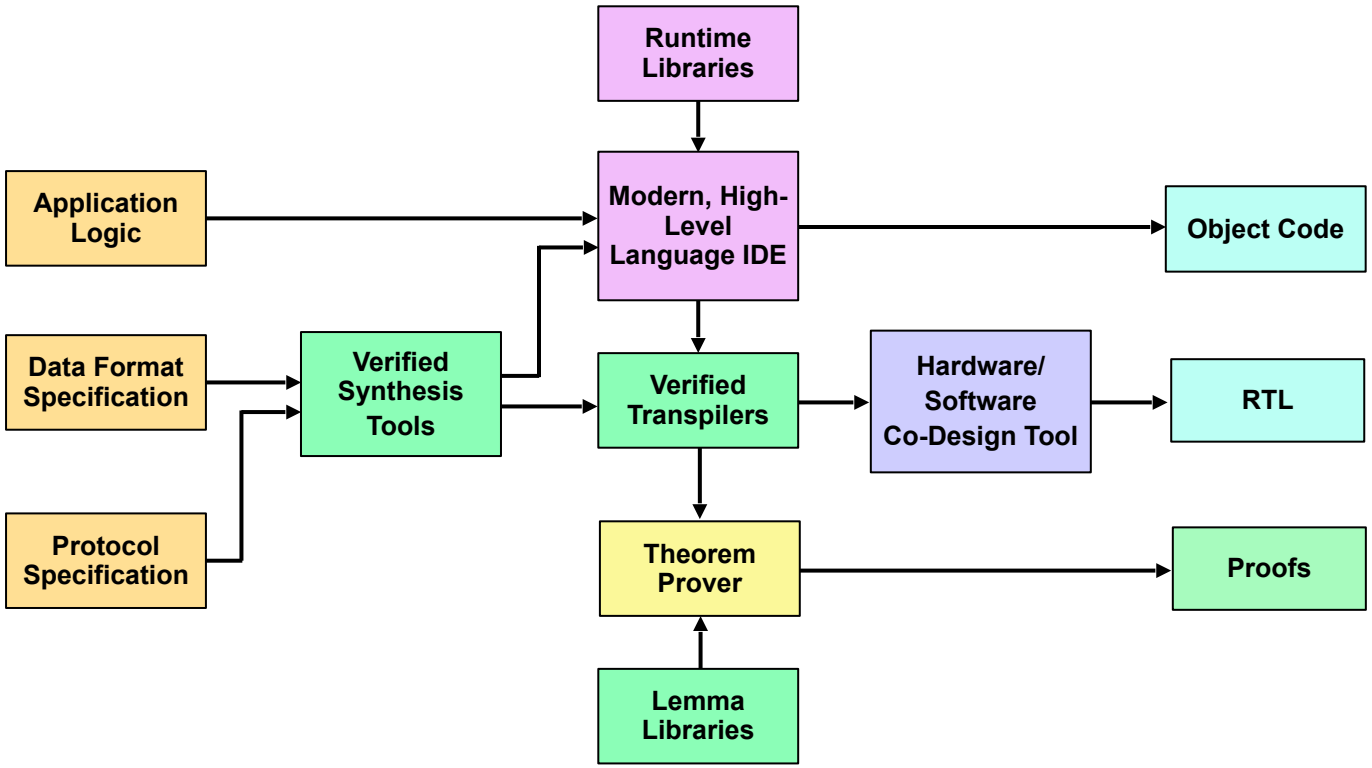


HARDWARE/SOFTWARE CO-DESIGN AND CO-ASSURANCE FOR DARPA CASE

- We desire to create CASE-style high-assurance architectural components using hardware/software co-design/co-assurance techniques
  - The CASE 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

HARDWARE/SOFTWARE CO-DESIGN/ CO-ASSURANCE TOOLCHAIN

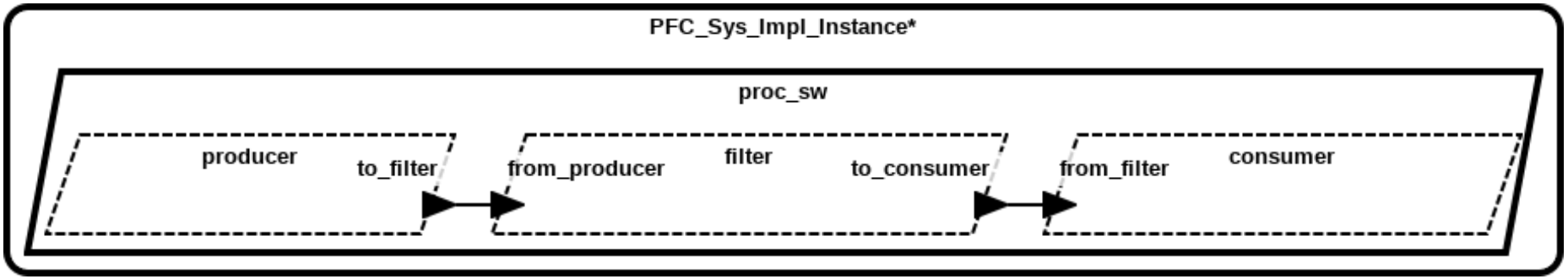
(ASPIRATIONAL)



HARDWARE/SOFTWARE CO-SYNTHESIS FROM AADL MODELS (KANSAS STATE UNIVERSITY)

Demo: Synthesize Hardware for CASE-generated filter

AADL Model:



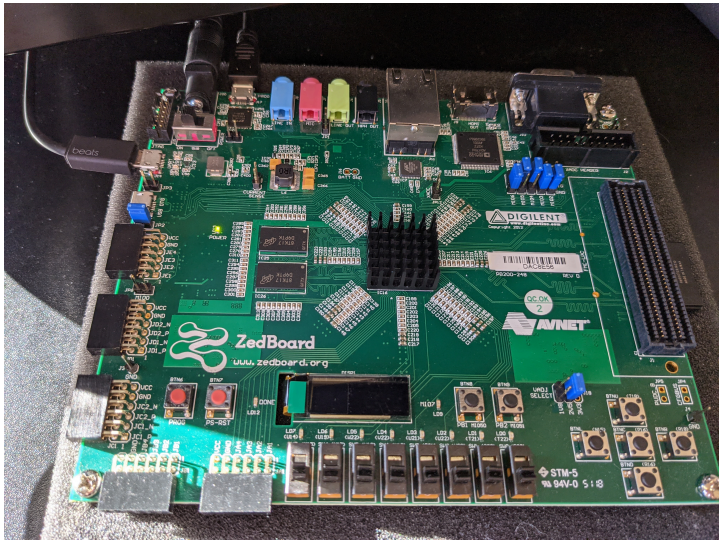
…mapped to
Linux software process

…mapped to
Linux process with FPGA hardware driver
to access hardware-based filter implementation

…mapped to
Linux software process

Note: The KSU team currently uses the Xilinx Vivado HLS tools to perform hardware synthesis

TESTING ON FPGA DEVELOPMENT BOARD



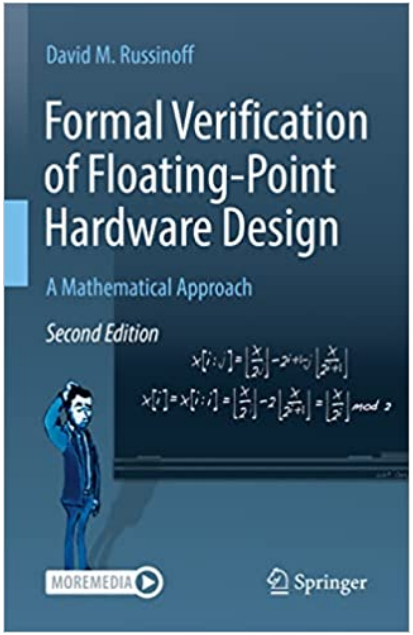

```
root@os:~# Consumer_proc_sw_consumer_App &
[1] 818
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_producer (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_producer_to_filter (data out)
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_filter (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_filter_from_producer (data in)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_filter_to_consumer (data out)
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_consumer (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_consumer_from_filter (data in)
Art: Connected ports: PFC_Sys_Impl_Instance_proc_sw_producer_to_filter -> PFC_Sys_Impl_Instance_proc_sw_filter_from_producer
Art: Connected ports: PFC_Sys_Impl_Instance_proc_sw_filter_to_consumer -> PFC_Sys_Impl_Instance_proc_sw_consumer_from_filter
root@os:~# Producer_proc_sw_producer_App &
[2] 819
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_producer (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_producer_to_filter (data out)
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_filter (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_filter_from_producer (data in)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_filter_to_consumer (data out)
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_consumer (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_consumer_from_filter (data in)
Art: Connected ports: PFC_Sys_Impl_Instance_proc_sw_producer_to_filter -> PFC_Sys_Impl_Instance_proc_sw_filter_from_producer
Art: Connected ports: PFC_Sys_Impl_Instance_proc_sw_filter_to_consumer -> PFC_Sys_Impl_Instance_proc_sw_consumer_from_filter
root@os:~# Filter_proc_sw_filter_App &
[3] 820
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_producer (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_producer_to_filter (data out)
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_filter (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_filter_from_producer (data in)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_filter_to_consumer (data out)
Art: Registered component: PFC_Sys_Impl_Instance_proc_sw_consumer (periodic: 1000)
Art: - Registered port: PFC_Sys_Impl_Instance_proc_sw_consumer_from_filter (data in)
Art: Connected ports: PFC_Sys_Impl_Instance_proc_sw_producer_to_filter -> PFC_Sys_Impl_Instance_proc_sw_filter_from_producer
Art: Connected ports: PFC_Sys_Impl_Instance_proc_sw_filter_to_consumer -> PFC_Sys_Impl_Instance_proc_sw_consumer_from_filter
root@os:~# Main
root@os:~# Main
root@os:~# Consumer_proc_sw_consumer_App starting ...
```

```
…
Consumer_proc_sw_consumer_App starting ...
Producer_proc_sw_producer_App starting ...
PFC_Sys_Impl_Instance_proc_sw_producer: Sending [00, 00, 00, 00, 00, 00, 00, 00, 00, 3A,
00, 00]
Filter_proc_sw_filter_App starting ...
PFC_Sys_Impl_Instance_proc_sw_filter: Payload approved - MissionData([00, 00, 00, 00, 00,
00, 00, 00, 00, 3A, 00, 00])
PFC_Sys_Impl_Instance_proc_sw_consumer: Received MissionData([00, 00, 00, 00, 00, 00, 00,
00, 00, 3A, 00, 00])
PFC_Sys_Impl_Instance_proc_sw_producer: Sending [00, 6F, 6F, 6F, 00, 00, 00, 00, 00, 3A,
00, 00]
PFC_Sys_Impl_Instance_proc_sw_filter: Payload rejected - MissionData([00, 6F, 6F, 6F, 00,
00, 00, 00, 00, 3A, 00, 00])
```

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



ALGORITHMIC C

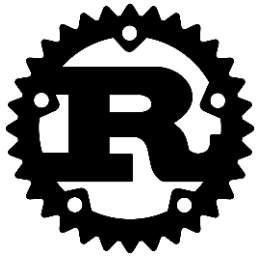
- The Algorithmic C datatypes “provide a basis for writing bit-accurate algorithms to be synthesized into hardware”
- Example use:
  - `typedef ac_int<112,false> ui112;`

  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

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

- Recently, we have begun work to realize the hardware/software co-design co-assurance toolchain vision by supporting a Rust language 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!***



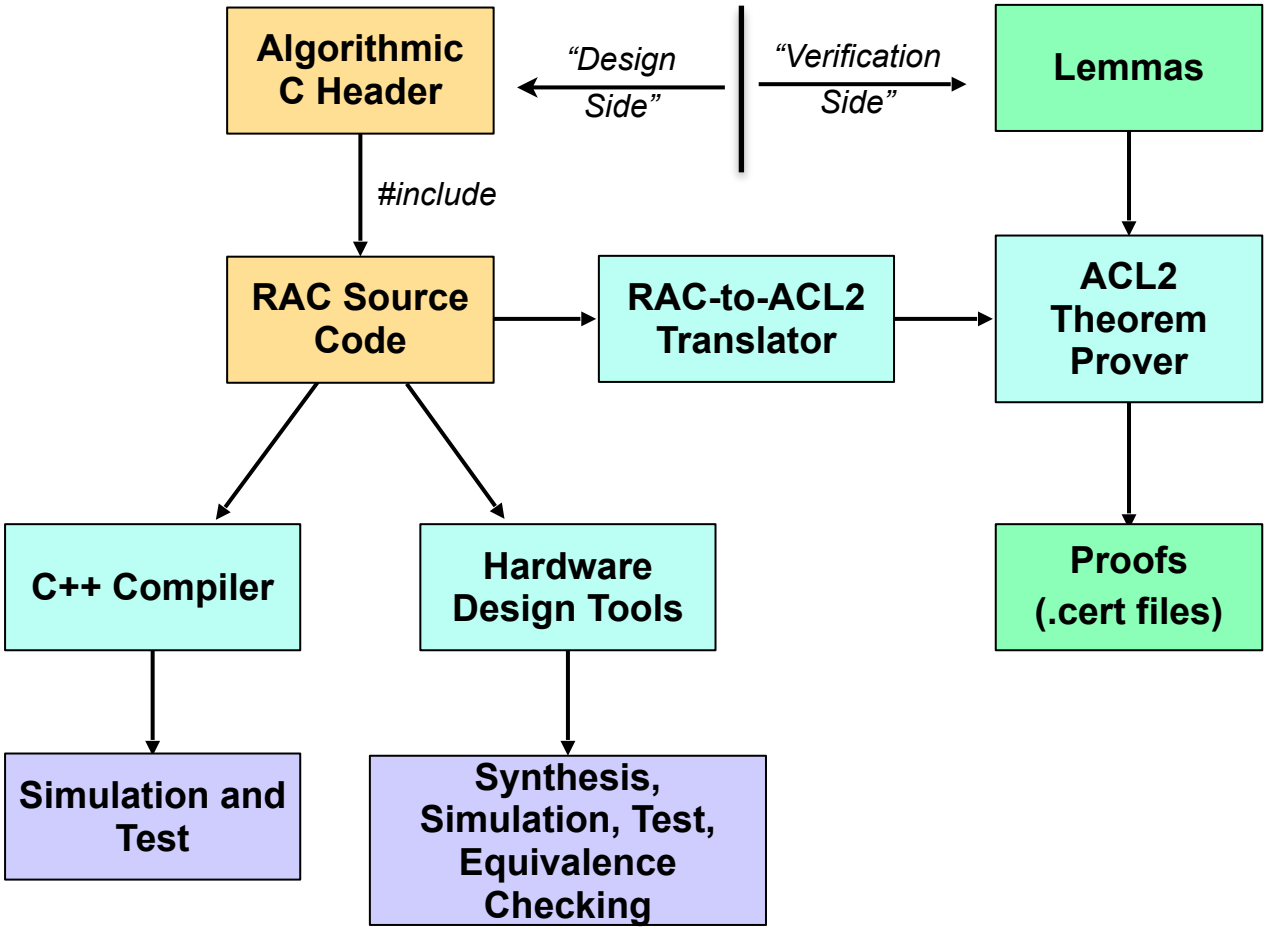
- Basic Rust syntax is familiar to C/C++ developers, easing the transition

- The Rust compiler produces efficient, and importantly, energy-efficient code, which makes Rust a favorite for sustainable computing

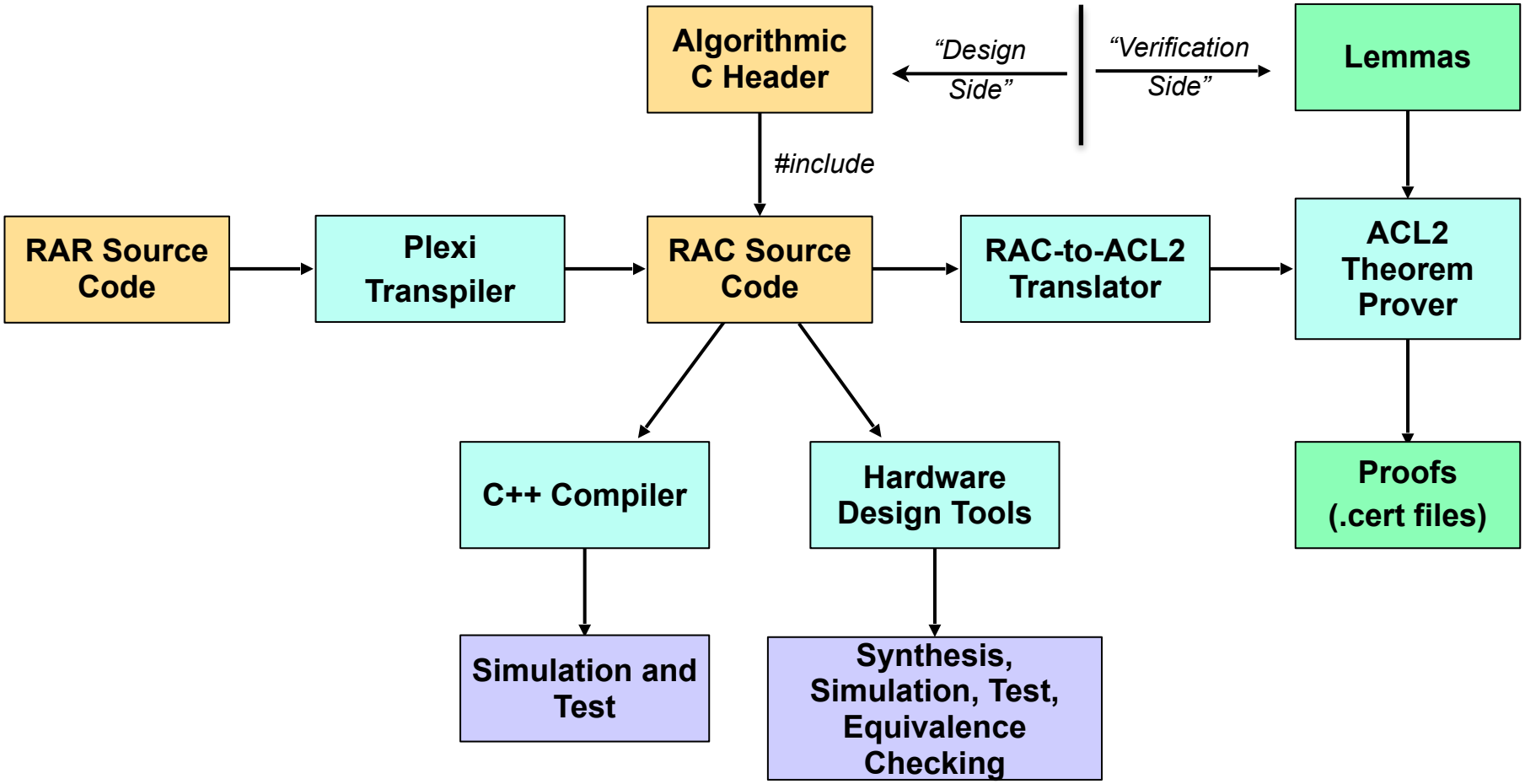
RAR EXAMPLES DEVELOPED TO DATE

- 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

RESTRICTED ALGORITHMIC C TOOLCHAIN



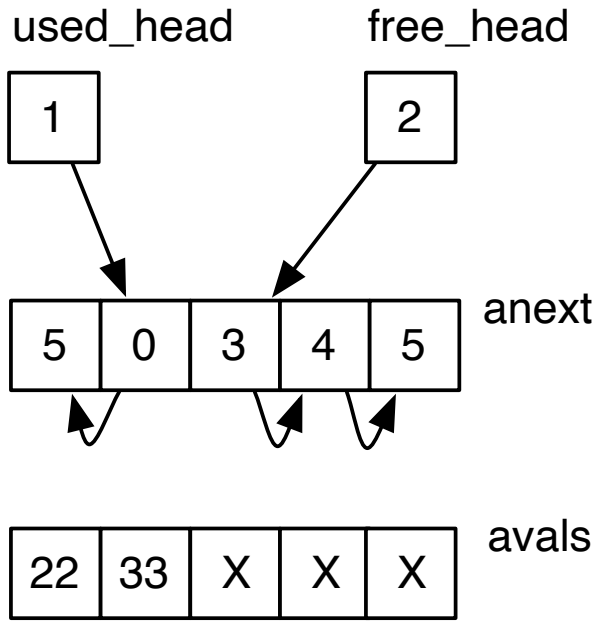
RESTRICTED ALGORITHMIC RUST TOOLCHAIN



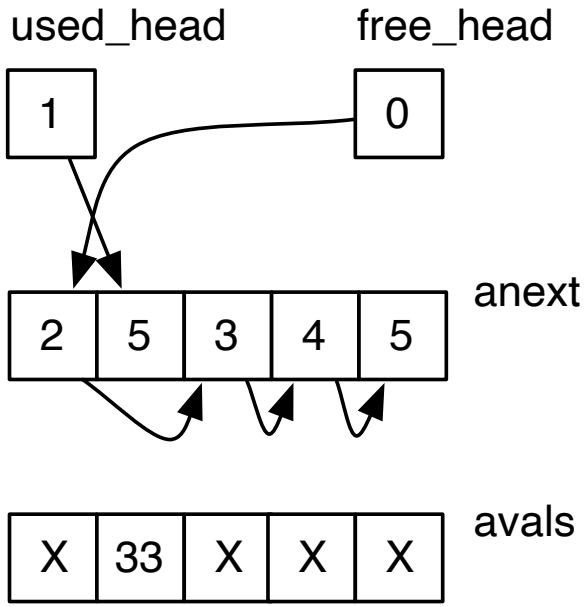
THE PLEXI RAR-TO-RAC TRANSPILER

- 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 this tool with a fully-verified transpiler

EXAMPLE: ARRAY-BASED SET



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



(b) After delete of element 22

RAR EXAMPLE: ARRAY-BASED SET

```
const ARR_SZ: uint = 256;

#[derive(Copy, Clone)]

struct Arrayset {
  anext: [usize; ARR_SZ],
  avals: [i64; ARR_SZ],
  free_head: usize,
  used_head: usize,
}
```

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

```
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;
    }
  }
}
```

TRANSLATION TO ACL2

```
(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))))))
```

EXAMPLE RAR CORRECTNESS THEOREMS

```
(defthm as-anext-preseves-arraysetp
  (implies
   (and (arraysetp aset)
        (array-of-u64p v)
   (arraysetp (as 'anext v aset))))


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

CONCLUSION AND FUTURE WORK

- We have detailed a method and toolchain for the creation of formally verified critical system components developed for the DARPA CASE program
  - We have demonstrated how this toolchain can be used to implement security-enhancing transformations on system architectures specified in AADL, with automatically synthesized and verified implementations
- We have also described methods and tools for enhancing the safety and security of critical systems using a hardware/software co-design/co-assurance approach using the Rust programming language
  - Our efforts stand on the broad shoulders of the great Restricted Algorithmic C work
- In future work, we will continue to enhance our verified synthesis tools for Restricted Algorithmic Rust, focusing on:
  - Enhanced proof automation via improved RAR/RAC ACL2 books
  - Enhanced integration with KSU hardware synthesis effort
  - Improvements to RAR-to-RAC transpiler