Verification of a Rust Implementation of Knuth's Dancing Links using ACL2

David S. Hardin

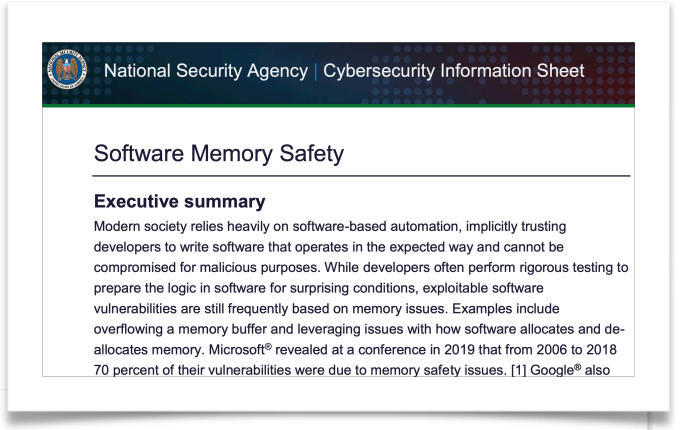
2023 ACL2 Workshop
13 November 2023

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.

Motivation

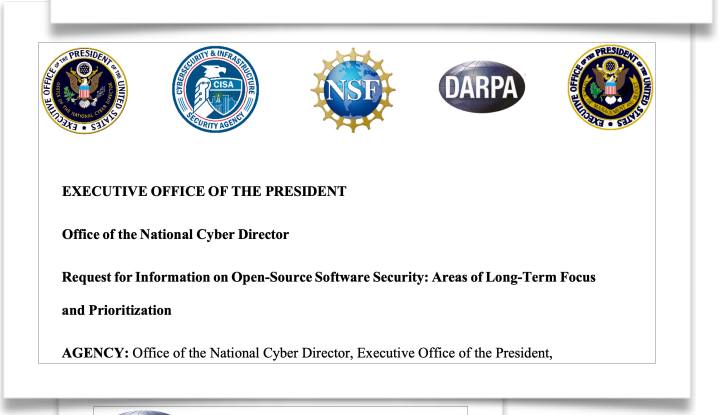
- An emerging consensus amongst computer science thought leaders is that memory-safe programming language technology needs to be adopted more broadly:
  - “NSA recommends using a memory safe language when possible.” (11/2022)
  - “The [White House] has established an Open-Source Software Security Initiative (OS3I) to champion the adoption of memory safe programming languages and open-source software security.” (08/2023)
  - Microsoft, Google, and Amazon have all announced significant Rust initiatives.
  - Rust verification technology features prominently in the Collins Aerospace winning proposal for the DARPA PROVERS program. (09/2023)


National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

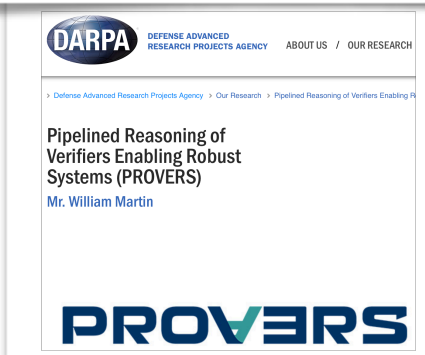
Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also


EXECUTIVE OFFICE OF THE PRESIDENT

Office of the National Cyber Director

Request for Information on Open-Source Software Security: Areas of Long-Term Focus and Prioritization

AGENCY: Office of the National Cyber Director, Executive Office of the President,


DARPA DEFENSE ADVANCED RESEARCH PROJECTS AGENCY ABOUT US / OUR RESEARCH

Defense Advanced Research Projects Agency › Our Research › Pipelined Reasoning of Verifiers Enabling R

Pipelined Reasoning of Verifiers Enabling Robust Systems (PROVERS)

Mr. William Martin

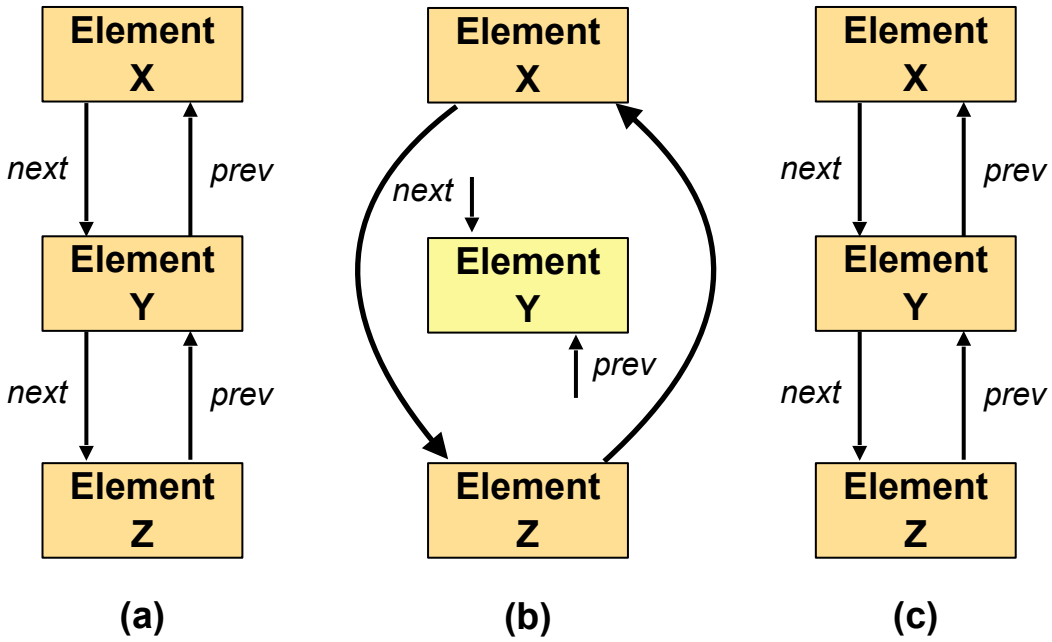
PROVERS

Example: Dancing Links for Exact Cover Problems

- An exact cover problem attempts to find, for an n x m matrix with binary elements, all of the subsets of the rows of the matrix such that all the column sums are exactly one.
  - This basic notion naturally extends to matrix elements that are in some numerical range; Sudoku is an extended exact cover problem for a 9 x 9 matrix with element values in the range of 1 to 9, inclusive.

- The exact cover problem is NP-complete, but recursive, nondeterministic backtracking algorithms to find exact covers have been devised.
  - One such procedure is Knuth's Algorithm X. Elements of the matrix are connected via circular doubly-linked lists, and individual elements are removed, or restored, as the algorithm proceeds.
  - Removals and restorations out of/into the list are quite common, so one should make these operations efficient. Knuth's "Dancing Links" is one such optimization.
  - The Dancing Links technique is documented in Knuth's *TAOCP*, vol. 4B (2022).

Dancing Links, Illustrated



(a) Doubly-linked circular list portion prior to remove operation.

(b) After remove of element Y.

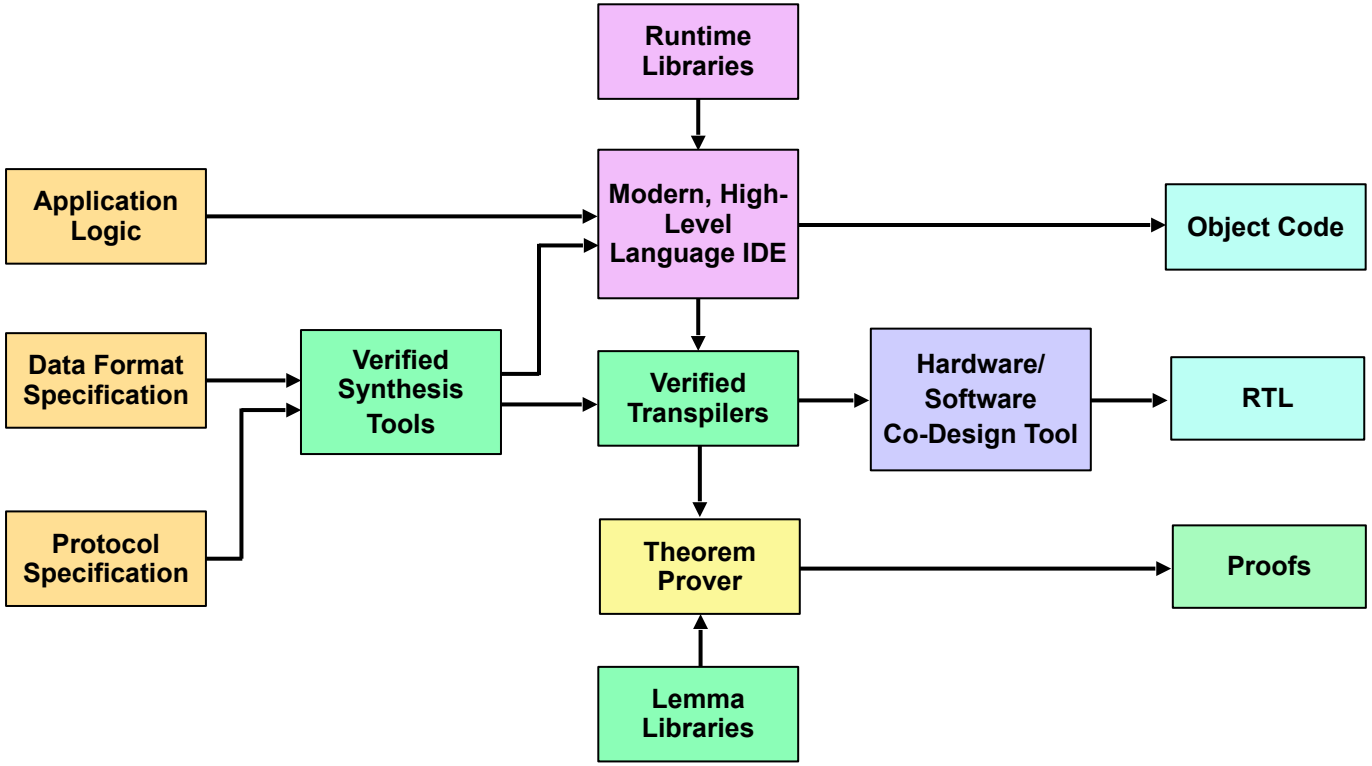
(c) After restore of element Y.

Hardware/Software Co-Design and Co-Assurance

- We desire to create high-assurance components using hardware/ software co-design/co-assurance techniques
  - The high-level Architectural Modeling approach in DARPA CASE, and now PROVERS supports both hardware- and software-based realizations
  - The ability to defer and/or change the allocation of functionality to hardware or software provides development flexibility
  - 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 for components that also support formal verification

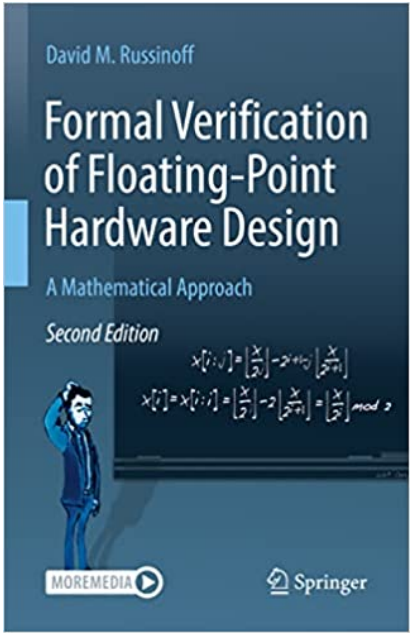
Hardware/Software Co-Design/ Co-Assurance Toolchain



The RAC Approach to Hardware/ Software Verification

- The hardware/software verification approach we leverage was developed by David Russinoff and John O'Leary, while both were at Intel, and later refined by Russinoff and colleagues at Arm

- Russinoff's method is called Restricted Algorithmic C (RAC), as it is based on Mentor's HLS Algorithmic C
  - RAC is extensively documented in Russinoff's book, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*
  - In Russinoff's text, 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

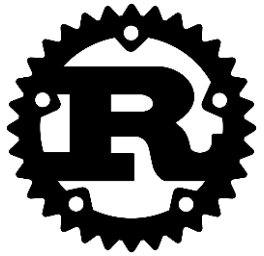
- Supported by Mentor hardware synthesis tools, e.g. Catapult; for details, see **`https://hlslibs.org`**

- 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

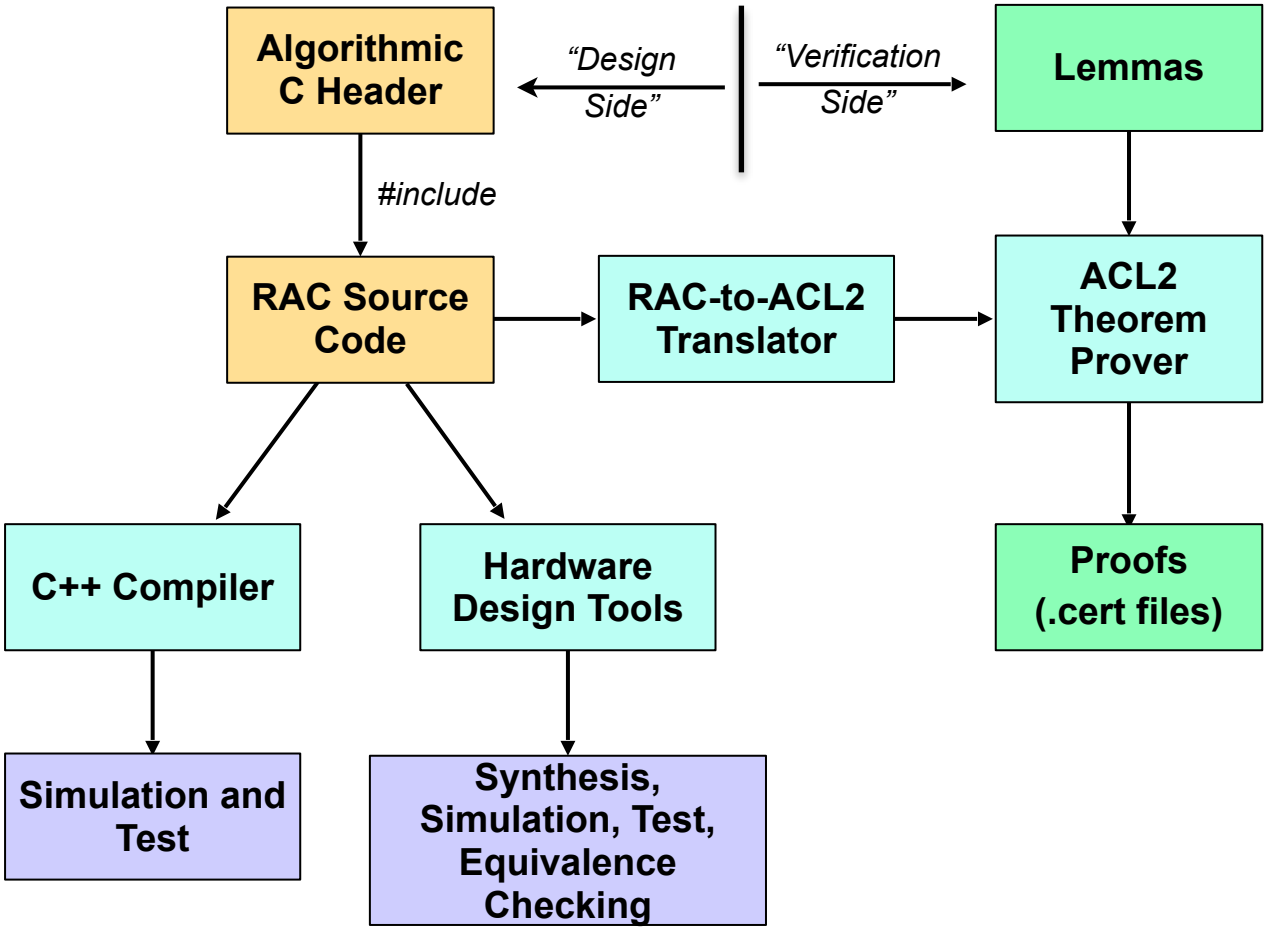
Hardware/Software Co-Assurance using Rust?

- Recently, we began work to realize the hardware/software co-assurance toolchain vision, inspired by RAC, for a Rust language subset called Restricted Algorithmic Rust, or RAR

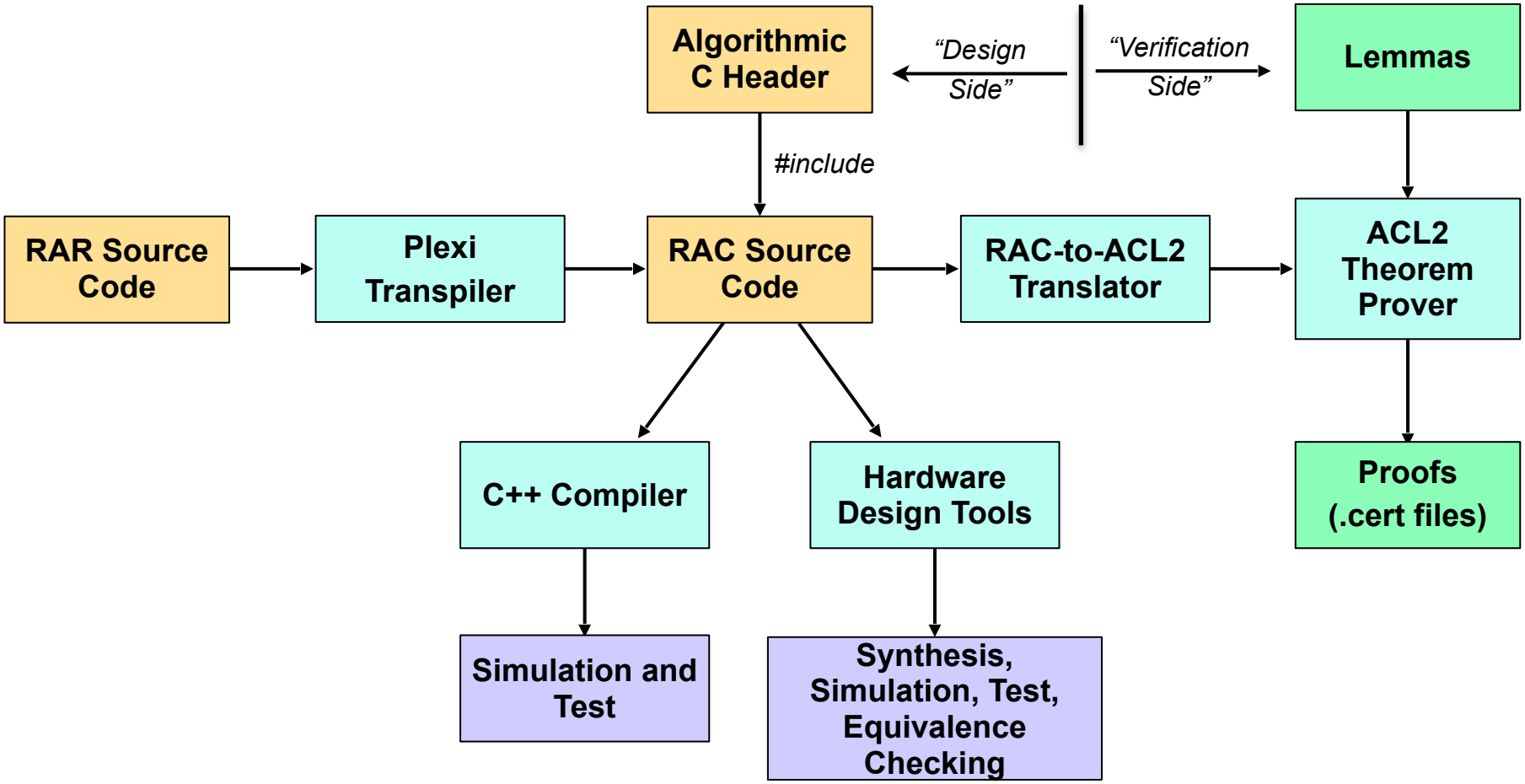


- Rust has several assurance advantages over C/C++, including:
  - Improved type safety
  - Vastly improved memory safety
  - No arbitrary pointer arithmetic
  - …in short, the sources of 80% of C/C++ security flaws are eliminated outright!
- Basic Rust syntax is familiar to C/C++ developers, easing the transition
- The Rust compiler produces efficient and energy-efficient code, which makes Rust a favorite for sustainable computing

Restricted Algorithmic C Toolchain



Restricted Algorithmic Rust Toolchain



RAR Toolchain Details

- The Plexi RAR-to-RAC translator:
  - Is based on the open source plex parser and lexer generator tool, written in Rust
  - Rapid prototyping principles used to produce a tool that works “well enough”
    - Translates RAR code to RAC code one line at a time
  - Future work will investigate replacing this tool with a fully-verified transpiler

- A number of examples have been processed using the RAR toolchain, including:
  - Array-Backed Verified Algebraic Data Types: Stack, Singly-linked list, Doubly-linked list, Circular Queue, Deque, etc.
  - A DFA-based JSON lexer, coupled with an LL(1) JSON parser
  - A significant subset of the Monocypher modern cryptography suite

Dancing Links in RAR

- A circular doubly-linked list (CDLL) is specified in RAR as follows:

```
const CDLL_MAX_NODE1: usize = 8191;
const CDLL_MAX_NODE: usize = CDLL_MAX_NODE1 - 1;

#[derive(Copy, Clone)]
struct CDLLNode {
  alloc: u2,
  val: i64,
  prev: usize,
  next: usize,
}

#[derive(Copy, Clone)]
struct CDLL {
  nodeHd: usize,
  nodeCount: usize,
  nodeArr: [CDLLNode; CDLL_MAX_NODE1],
}
```

Dancing Links remove() function

```
fn CDLL_remove(n: usize, mut CDObj: CDLL) -> CDLL {
  if (n > CDLL_MAX_NODE) {
    return CDObj;
  } else {
    if (n == CDObj.nodeHd) {  // Can't remove head
      return CDObj;
    } else {
      if (CDObj.nodeCount < 3) {  // Need three elements
        return CDObj;
      } else {
        let nextNode: usize = CDObj.nodeArr[n].next;
        let prevNode: usize = CDObj.nodeArr[n].prev;

        CDObj.nodeArr[prevNode].next = nextNode;
        CDObj.nodeArr[nextNode].prev = prevNode;

        CDObj.nodeCount = CDObj.nodeCount - 1;

        return CDObj; } } } }
```

Dancing Links restore() function

```
fn CDLL_restore(n: usize, mut CDObj: CDLL) -> CDLL {
  if (n > CDLL_MAX_NODE) {
    return CDObj;
  } else {
    if (n == CDObj.nodeHd) {  // Can't restore head
      return CDObj;
    } else {
      if ((CDObj.nodeCount < 2) ||          // Need two elements
          (CDObj.nodeCount == CDLL_MAX_NODE1))  {  // full list
        return CDObj;
      } else {
        let prevNode: usize = CDObj.nodeArr[n].prev;
        let nextNode: usize = CDObj.nodeArr[n].next;

        CDObj.nodeArr[prevNode].next = n;
        CDObj.nodeArr[nextNode].prev = n;
        CDObj.nodeCount = CDObj.nodeCount + 1;
        return CDObj;
      } } } }
```

Example Translation to ACL2

```
(DEFUND CDLL_REMOVE (N CDOBJ)
     (IF1 (LOG> N (CDLL_MAX_NODE))
          CDOBJ
          (IF1 (LOG= N (AG 'NODEHD CDOBJ))
               CDOBJ
               (IF1 (LOG< (AG 'NODECOUNT CDOBJ) 3)
                    CDOBJ
                    (LET* ((NEXTNODE (AG 'NEXT (AG N (AG 'NODEARR CDOBJ))))
                           (PREVNODE (AG 'PREV (AG N (AG 'NODEARR CDOBJ))))
                           (CDOBJ (AS 'NODEARR
                                      (AS PREVNODE
                                          (AS 'NEXT
                                              NEXTNODE
                                              (AG PREVNODE (AG 'NODEARR CDOBJ)))
                                          (AG 'NODEARR CDOBJ))
                                      CDOBJ))
                           (CDOBJ (AS 'NODEARR
                                      (AS NEXTNODE
                                          (AS 'PREV
                                              PREVNODE
                                              (AG NEXTNODE (AG 'NODEARR CDOBJ)))
                                          (AG 'NODEARR CDOBJ))
                                      CDOBJ)))
                          (AS 'NODECOUNT
                              (- (AG 'NODECOUNT CDOBJ) 1)
                              CDOBJ))))))
```

Dancing Links Correctness

```
(defthm restore-of-remove--thm
 (implies
  (and (cdllp Obj)
       (good-nodep n Obj)     ;; various well-formedness predicates
       (not (= n (ag 'nodeHd Obj)))
       (>= (ag 'nodeCount Obj) 3))
  (= (CDLL_restore n (CDLL_remove n Obj))
     Obj)))
```

ACL2 proves 160 circular doubly-linked list functional correctness lemmas and theorems completely automatically.

Related Work

- A number of domain-specific languages targeting both hardware and software realization, and providing support for formal verification, have been created.
  - Cryptol has been employed as a “golden spec” for the evaluation of cryptographic implementations, in which automated tools perform equivalence checking between the Cryptol spec for a given algorithm, and the VHDL implementation

- A number of verification tools have been developed for Rust, including:
  - Cruesot (Inria)
  - Prusti (ETH Zurich)
  - RustHorn (University of Tokyo/Chiba University)
  - Kani (Amazon)
  - Verus (Carnegie-Mellon University)

- With Verus, programmers express proofs and specifications using Rust syntax, allowing proofs to take advantage of Rust’s linear types and borrow checking. Collins Aerospace will be teamed with the Verus team on the DARPA PROVERS program.
  - It will be interesting to attempt the sorts of correctness proofs achievable on our system using these verification tools, and compare the effort required.

Conclusion

- Memory-safe language technology is on the rise, both in leading technology companies, as well as in the view of Government agencies
  - Memory-safe languages have positive impllcations for formal verification
- We have detailed a method and toolchain for the creation of high-assurance components, using a hardware/software co-assurance approach employing the memory-safe Rust programming language
  - Our efforts stand on the broad shoulders of Restricted Algorithmic C
- In future work (e.g., DARPA PROVERS) we will generate Restricted Algorithmic Rust from Architecture Models, as well as from Coq specifications. We will perform Rust code verification, both using the RAR toolchain, as well as SMT-based verification technology. Along the way, we will:
  - Enlarge the Rust subset that can be formally verified
  - Use Rust for hardware synthesis
  - Compare and contrast the expressiveness and effectiveness of the SMT-based approach relative to the RAR toolchain
  - Develop ways to minimize verified Rust code annotation