# Put Me on the RAC

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

We have a keen research interest in Hardware/Software Co-Assurance. 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. As part of our research, 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 [7], to determine its suitability for the creation of various safety-critical/security-critical applications in numerous domains.

Algorithmic C [5] 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 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. In this extended abstract, we summarize several recent experiments on the use of the RAC toolchain in various safety-critical/security-critical domains, focusing first on software implementations, but extending to hardware design/verification, as well as hardware/software co-assurance (in future work).

In a first experiment, we created a number of algebraic data types with fixed maximum size, suitable for both hardware and software implementation, in Restricted Algorithmic C, including lists, stacks, queues, deques, and binary trees. This approach was inspired by our previous work on fixed-size formally verified algebraic data types [4], and produced a set of reusable data structures that could be employed in further work. We used the RAC toolchain to translate these algebraic data types to ACL2, performed proofs of correctness in ACL2, and validated the data types by testing. This testing was conducted using a C++ compilation toolchain, as well as within ACL2. This dual validation was not redundant effort, as the RAC translator is not itself formally verified. Our success with this effort gave us confidence that the RAC toolchain could be productively used in domains outside of floating-point hardware design and verification.

As a simplified example, consider the development of a basic stack data type, implemented using a fixed-size array. The Algorithmic C header file declaration for this type is given below. We arbitrarily set the maximum number of stack elements to 16383 for this example.

```
#define STK_MAX_SZ 16383
#define STK_OK 0
#define STK_OCCUPANCY_ERR 255

struct STKObj {
  ui14 nodeTop;
  array<i64, STK_MAX_SZ> nodeArr;
};
```

Note the use of specific integer widths, such as the *ui14* declaration for the 14-bit unsigned integer indices, that are not readily available in "vanilla" C++. One might deem these exacting type declarations to be bothersome, but, in our experience, the benefits of such strong typing in areas

G. Passmore, R. Gamboa (Eds.): ACL2 Workshop 2020
EPTCS ??, 2020, pp. 1–3, doi:10.4204/EPTCS.??.??

such as early error identification outweigh the costs. Also note that the RAC toolchain translates this struct-of-array into an ACL2 record, with the usual ACL2 record *AG* (get) and *AS* (set) operators.

The body of the stack code consists of a number of basic C++ functions implementing various stack operations. Three such stack operators are the *push*, *pop*, and *top* operators, depicted below.

```
STYP STK_push (i64 v, STKObj amp(SObj)) {
  if (SObj.nodeTop > 0) {
    // Note: Stack grows down
    SObj.nodeTop--;
    SObj.nodeArr[SObj.nodeTop] = v;
  }
  return SVAL;
}

STYP STK_pop (STKObj amp(SObj)) {
  if (SObj.nodeTop < STK_MAX_SZ) {
    SObj.nodeTop++;
  }
  // Note: Pop of empty stack is a nop
  return SVAL;
}


tuple<ui8, i64> STK_top (STKObj amp(SObj)) {
  if (SObj.nodeTop < STK_MAX_SZ) {
    return tuple<ui8, i64>(STK_OK, SObj.nodeArr[SObj.nodeTop]);
  } else {
    return tuple<ui8, i64>(STK_OCCUPANCY_ERR, 0);
  }
}
```

Note that the stack "grows" down, towards smaller array indices, and that the *top* function returns a two-tuple consisting of an error code as well as the top-of-stack value.

Once the stack data type code has been translated into ACL2 by the RAC toolchain, we can begin to reason about the translated functions in the ACL2 environment, using the RTL books, as well as other ACL2 books. One functional correctness property to prove of our stack representation is that the top-of-stack resulting from a *push* followed by a *pop* is the same as the original top-of-stack, given that space exists for the *push*. This is expressed in ACL2 as

```
(defthm STK_top-of-STK_pop-of-STK_push-val--thm
  (implies
   (and
    (stkp Obj)
    (spacep Obj)
    (acl2::signed-byte-p 64 n))
   (equal (nth 1 (STK_top (STK_pop (STK_push n Obj))))
          (nth 1 (STK_top Obj)))))
```

where *stkp* is a well-formedness predicate for the stack object, and *spacep* is true if there is space for more elements on the stack. ACL2 readily proves this theorem after a few basic lemmas are introduced.

In a second experiment, we created an Instruction Set Architecture (ISA) simulator for a representative 64-bit RISC ISA in the RAC C++ subset [2]. It is common practice to create such a

simulator early in the microprocessor development process, usually written in C/C++, in order to allow for early compiler toolchain development, to serve as a platform to perform tradeoffs for various candidate ISA enhancements, as well as to act as an "oracle" for subsequent detailed microprocessor development. We used the RAC tool to translate the simulator code to ACL2, produced small binary programs for the ISA used to validate the simulator, and then utilized the ACL2 Codewalker decompilation-into-logic facility to prove those test programs correct.

In a third effort, we implemented a high-assurance filter for JSON-formatted data used in an Unmanned Air Vehicle (UAV) application [3] using the RAC toolchain. Our JSON filter was built using a table-driven lexer/parser, supported by mathematically-proven lexer and parser table generation technology, as well as a verified stack data structure (reused from our earlier RAC data structure work). As in previous experiments, the JSON filter implementation was validated via tests generated via C++ compilation, as well as in ACL2, with the outputs of the two test environments compared for equality. We also gathered performance data, indicating that our RAC-generated lexer/parser for JSON was very competitive in speed (up to 20% faster) when compared to JSON parsers generated by non-verified parser generators.

The aforementioned efforts have focused on software implementations of RAC applications. Recently, we have begun work on targeting Field-Programmable Gate Array (FPGA) hardware using the Mentor Graphics Catapult [6] toolsuite, in collaboration with colleagues at Kansas State University. A goal of this collaboration 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) [1]. Future experiments will include the generation of high-assurance hardware/software co-designs with proofs of correctness in ACL2.

# References

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

[2] David S. Hardin (2019): *Instruction Set Architecture Specification, Verification, and Validation using Algorithmic C and ACL2*. In: *Workshop on Instruction Set Architecture Specification (SpISA19)*. Available at `https://www.cl.cam.ac.uk/~jrh13/spisa19/paper_09.pdf`.

[3] David S. Hardin (2020): *Verified Hardware/Software Co-Assurance: Enhancing Safety and Security for Critical Systems*. In: *Proceedings of the 2020 IEEE Systems Conference (to appear)*.

[4] David S. Hardin & Konrad L. Slind (2018): *Using ACL2 in the Design of Efficient, Verifiable Data Structures for High-Assurance Systems*. In Shilpi Goel & Matt Kaufmann, editors: *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and its Applications*, *EPTCS* 280, pp. 61–76, doi:10.4204/EPTCS.280.5.

[5] Mentor Graphics Corporation (2019): *Algorithmic C (AC) Datatypes*. Available at `https://www.mentor.com/hls-lp/downloads/ac-datatypes`.

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

[7] David M. Russinoff (2018): *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, doi:10.1007/978-3-319-95513-1.