Verified Hardware/Software Co-Assurance: Enhancing Safety and Security for Critical Systems

David S. Hardin
Trusted Systems Group
Collins Aerospace
Email: david.hardin@collins.com

Abstract*—Experienced developers of safety-critical and security-critical systems have long emphasized the importance of applying the highest degree of scrutiny to a system's I/O boundaries. From a safety perspective, input validation is a traditional "best practice." For security-critical architecture and design, identification of the attack surface has emerged as a primary analysis technique. One of our current research focus areas concerns the identification of and mitigation against attacks along that surface, using mathematically-based tools. We are motivated in these efforts by emerging application areas, such as assured autonomy, that feature a high degree of network connectivity, require sophisticated algorithms and data structures, are subject to stringent accreditation/certification, and encourage hardware/software co-design approaches. 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, to determine its suitability for the creation of safety-critical/security-critical input filters. We focus first on software implementation, but extending to hardware as well as hardware/software co-designs. We have implemented a high-assurance filter for JSON-formatted data used in an Unmanned Aerial Vehicle (UAV) application. Our JSON filter is built using a table-driven lexer/parser, supported by mathematically-proven lexer and parser table generation technology, as well as verified data structures. Filter behavior is expressed in a subset of Algorithmic C, which defines a set of C++ header files providing support for hardware design, including the peculiar bit widths utilized in that discipline, and enables compilation to both hardware and software platforms. The Russinoff-O'Leary Restricted Algorithmic C (RAC) toolchain translates Algorithmic C source to the Common Lisp subset supported by the ACL2 theorem prover; once in ACL2, filter behavior can be mathematically verified. We describe how we utilize RAC to translate our JSON filter to ACL2, present proofs of correctness for its associated data types, and describe validation and performance results obtained through the use of concrete test vectors.

I. Introduction

Experienced safety-critical and security-critical system architects have long emphasized the importance of applying the highest degree of scrutiny to a system's I/O boundaries.[1] From a safety perspective, input validation has long been a "best practice." For security-critical architecture and design, identification of the attack surface has emerged as an important analysis technique. One of our current research focus areas on the DARPA Cyber-Assured Systems Engineering (CASE) program[2] concerns the identification of and mitigation against attacks along that surface, using the highest-assurance techniques and tools available.

The need for verification techniques for sophisticated engineering artifacts developed using hardware/software co-design methods is increasing, notably for autonomous and cyber-resilient systems employing complex data structures. This begs the question: Can these algorithms and data structures be specified in a language that admits implementation in both software and hardware, while also supporting formal verification? In this paper, we explore a particular toolchain, initially developed for verified floating-point hardware design, to determine whether it can also be utilized for verified hardware/software co-design in other application areas.

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, high-assurance design rules, such as mandated by RTCA DO-178C Level A [1] for flight-critical systems, frown on dynamic memory allocation, preferring simple array-based data structure implementations. This discipline is also a benefit to 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, *e.g.*, use-after-free.

In order to provide efficient implementations of high-level data structures and complex algorithms used in modern cyber-physical systems with the high assurance needed for accreditation, we are developing a design-with-formal-verification method that involves verified *lowering* of high-level design specifications to array-based implementations [2].

II. Application: High-Assurance Filtering of JSON-Formatted Data

By way of a motivating example, consider the model of a mission control system for an unmanned aerial vehicle (UAV) shown in Fig. 1. This model, a simplification of the AADL model used on CASE, is based on the UxAS UAV developed at the U.S. Air Force Research Laboratory [3], and uses legacy

[1]DISTRIBUTION STATEMENT A. Approved for public release.

[2]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.

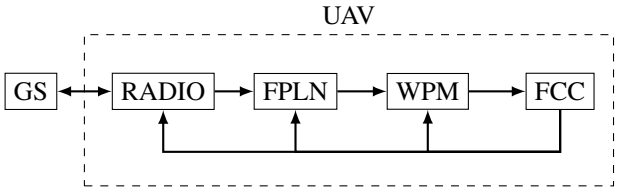

Fig. 1. Simplified model of a UAV mission controller.

components from that system. The two main components of the system are a ground station (GS) and the UAV, which has as subcomponents a radio (RADIO), a flight planner (FPLN), a waypoint manager (WPM), and a flight control computer (FCC). General mission parameters flow from GS to FPLN, by way of the radio link. The flight planner generates the full flight plan—a sequence of waypoints to follow—and sends it to the waypoint manager. The WPM processes a fixed-size window of the next few waypoints to be dealt with by the flight control computer. The FCC is a separate computer which is in charge of actually flying the vehicle, interpreting the waypoints and incoming sensor data and sending directives to adjust control surfaces, etc. As the FCC makes progress, it tells the WPM to advance the window, and also sends back various sensor data to the FPLN, WPM, and potentially the GS via the RADIO.

A first step towards securing such a system is *isolation*: when an untrusted legacy component is compromised or otherwise malfunctions, it must not interfere with the correct execution of other components, or read privileged data from other components through unintended channels. To achieve this, the connections between the components that are explicitly present in design diagrams such as Fig. 1 must be the only communication channels present in the running system. While such isolation can be achieved by running each component on its own physically isolated hardware, we enable miniaturization, reduce cost, and gain flexibility by using the same general-purpose hardware to run several distinct components. In order to recover isolation in this setting, we host our components on seL4 [4], a verified capability-based microkernel accompanied by formal proof of isolation properties down to the binary level.

While isolation defends against attacks through *unintended* channels, it does nothing to guard against attacks through the *intended* channels. For example, a compromised ground station or radio driver could be used to feed malformed messages that the legacy flight planner is not equipped to handle. This could be exploited to induce crashes, privilege escalation through buffer overflow, etc.

Our goal on the CASE program is that the system designer, having identified the potential for such a vulnerability (our CASE tools provide automated analysis to aid in the detection of such vulnerabilities), should have at her disposal a toolbox of *security-enhancing architectural transformations* for inserting countermeasures into the architecture. These countermeasures should be non-intrusive, reusable and highly trustworthy. They should not modify the (potentially brittle) legacy components of the design — instead, they place security enhancements *around* them. They should also be easy to configure and deploy for the needs of different systems. When we introduce more components into a system, they too potentially become part of the attack surface. Thus, each component inserted by a transformation is accompanied by a formal proof connecting the component's intended behavior with the behavior of its implementation down to the binary level.

The concrete transformation we consider in this paper is *verified filter insertion* [5], which is a pattern to prevent attacks that rely on malformed messages. This transformation is illustrated on the UAV mission control example in Fig. 2. The filter controls the flow of messages on the communication channel, ensuring that only well-formed messages get forwarded from the radio driver to the flight controller, and also that every well-formed message gets forwarded. The precise definition of "well-formed" will vary from design to design; hence, it is a parameter that the system designer can instantiate. In this paper, we consider filters where well-formedness can be decided by adherence to the JSON standard [6].

JSON (JavaScript Object Notation) is a popular lightweight interchange format for structured data. JSON is text-based, and is relatively simple to generate and parse. JSON data payloads are built from two basic primitives: a collection of name-value pairs, and an ordered list of values. In our use case, a UAV air-ground communications system employs JSON to encode certain messages sent between the UAV and its ground-based control station. For example, a UAV coordinate could be encoded in JSON as:

```
{"lat":42.008, "long":-91.644, "alt":5000}
```

To aid in thwarting cyber attacks, we need to construct a filter component that checks whether a given air-ground message is legal JSON, and rejects any malformed messages. However, as this added filter component could itself contain vulnerabilities (thus *increasing* rather than *decreasing* the attack surface), we need to design said filter in the highest assurance manner possible.

In order to create such a JSON filter, we need to perform both lexical and syntactic level analysis on any candidate JSON message received[3].

A. Lexical Analysis

For the lexical level analysis, we have constructed verified tools that generate a verified lexer based on Deterministic Finite-state Automata (DFAs), where the individual lexeme specifications are given as regular expressions [7].

In previous work [5], we generated regular-expression-based filter functions in Higher Order Logic that were then compiled into verified machine code via the facilities of the verified CakeML compiler [8]. This work benefited from a method,

[3]Note that we assume that there is some system-defined maximum message size, such that the message to be checked can be held all at once in memory.

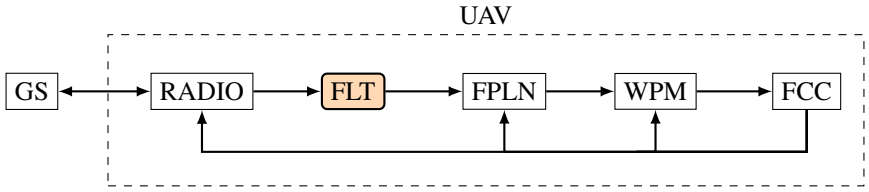

Fig. 2. Simplified model of a UAV mission controller with inserted filter.

developed for the CASE program, to prove liveness properties for non-terminating CakeML programs [9]. In the present work, we wish to experiment with hardware/software co-design utilizing the tools described in Section III, so we have used the same verified DFA tables as generated previously, but have written a simple DFA traverser by hand in a language better-suited for hardware/software co-design; in future work, this traverser code will be automatically generated.

1) Formalization of Regular Expression Matching: The semantics of regular expressions, $\mathcal{L}(-)$, maps from a regular expression to a formal language (set of strings). Strings are represented as lists of characters. Regular expression operations are described below:

$$
\begin{array}{lcll}
\mathcal{L}(\mathsf{Epsilon}) & = & \{\varepsilon\} & ;\text{empty string} \\
\mathcal{L}(\mathsf{Symbs}\ P) & = & \{[x] \mid x \in P\} & ;\text{char from set} \\
\mathcal{L}(\mathsf{Or}\ r_1\ r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) & ;\text{disjunction} \\
\mathcal{L}(\mathsf{Cat}\ r_1\ r_2) & = & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) & ;\text{concatenation} \\
\mathcal{L}(\mathsf{Star}\ r) & = & (\mathcal{L}(r))^* & ;\text{Kleene star} \\
\mathcal{L}(\mathsf{Not}\ r) & = & \overline{\mathcal{L}(r)} & ;\text{complement} \\
\mathcal{L}(\mathsf{And}\ r_1\ r_2) & = & \mathcal{L}(r_1) \cap \mathcal{L}(r_2) & ;\text{intersection}
\end{array}
$$

The concrete syntax of regular expressions used to specify lexemes in our system is that used by various programming languages. As an example, consider a simple regular expression for a hexadecimal number, where both `0x` and `0X` hexadecimal prefix indicators are accepted:

```
0(x|X)[0-9a-fA-F]+
```

Here `|` indicates disjunction, `+` means "one or more," and `[]` encloses a set of possible characters. The above regular expression can thus be read as "`0` followed by either `x` or `X`, followed by one or more characters in the ranges `0` through `9`, `a` through `f`, or `A` through `F`."

2) Regular Expression Compilation: A variety of means exist to translate a regular expression to a corresponding DFA. We chose one from Brzozowski [10], who proposed an algorithm that compiles regular expressions directly to DFAs, avoiding low-level automata constructions and treating non-standard—but useful—boolean operations such as negation and intersection uniformly. The core of the algorithm is an elegant quotient construction identifying regular expressions with DFA states. Recent work [11] has shown that his method often generates minimal DFAs and can be extended to large character sets and character classes.

We have formalized and proved a version of Brzozowski's algorithm in HOL4. The complete HOL4 proof that the array-based DFA code generated by the Brzozowski method implements the semantics of regular expressions is provided in the HOL4 distribution[4].

B. Syntactic Analysis

For syntactic-level analysis, we employ the Vermillion verified LL(1) parser generator due to Lasser, *et. al*. [12], coupled with a parse table traverser hand-written using the toolchain described in Section III. The parse table traverser is a bit more complex than that for the lexer, in that it requires a rule stack; for this, we employ a verified stack component, discussed in Section IV. We also record the rules used as we proceed in a list, allowing us to reconstruct the parse tree later on.

Our LL(1) grammar rules for JSON are presented below. An initial capital letter indicates a nonterminal symbol; the rest are terminals. As before, ε designates the empty string. The terminal symbols fls (false), flt (float), int (integer), nul (null), str (string), tru (true), open brace, close brace, open bracket, close bracket, colon, and comma constitute the JSON lexemes produced by the lexer.

$$
\begin{array}{lcl@{\qquad}lcl}
\mathit{Value} & \rightarrow & \{\ \mathit{Pairs}\ \} & \mathit{Pairs} & \rightarrow & \mathit{Pair}\ \mathit{PairsTl} \\
\mathit{Value} & \rightarrow & [\ \mathit{Elts}\ ] & \mathit{PairsTl} & \rightarrow & \varepsilon \\
\mathit{Value} & \rightarrow & \text{str} & \mathit{PairsTl} & \rightarrow & ,\ \mathit{Pair}\ \mathit{PairsTl} \\
\mathit{Value} & \rightarrow & \text{int} & \mathit{Pair} & \rightarrow & \text{str}\ :\ \mathit{Value} \\
\mathit{Value} & \rightarrow & \text{flt} & \mathit{Elts} & \rightarrow & \varepsilon \\
\mathit{Value} & \rightarrow & \text{tru} & \mathit{Elts} & \rightarrow & \mathit{Value}\ \mathit{EltsTl} \\
\mathit{Value} & \rightarrow & \text{fls} & \mathit{EltsTl} & \rightarrow & \varepsilon \\
\mathit{Value} & \rightarrow & \text{nul} & \mathit{EltsTl} & \rightarrow & ,\ \mathit{Value}\ \mathit{EltsTl} \\
\mathit{Pairs} & \rightarrow & \varepsilon & & &
\end{array}
$$

The structure of the overall filter for JSON, including both lexical and syntactic analysis toolchains, is displayed in Fig. 3.

III. Algorithmic C and Hardware/Software Co-Assurance

We take our primary inspiration for hardware/software co-assurance from the hardware verification domain. To achieve the highest level of assurance, we need a development language that allows us to specialize designs to hardware or software, and that can be reasoned about using automated formal verification tools. In his *tour de force* book on hardware floating-point formal verification [13], David Russinoff details a 20-year quest to provide mathematical proofs for complex arithmetic hardware utilizing an automated theorem prover,

[4]In `examples/formal-languages/regular`.

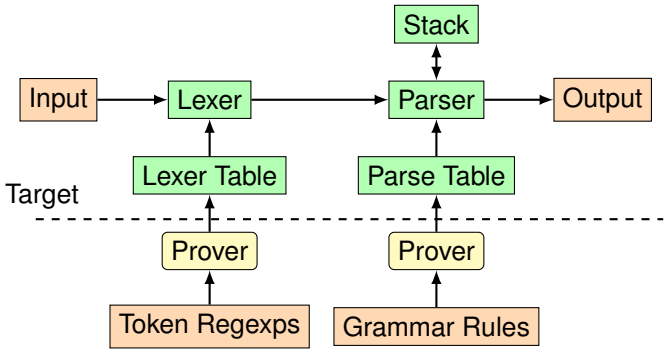

Fig. 3. JSON filter built from hardware/software co-assurance components.

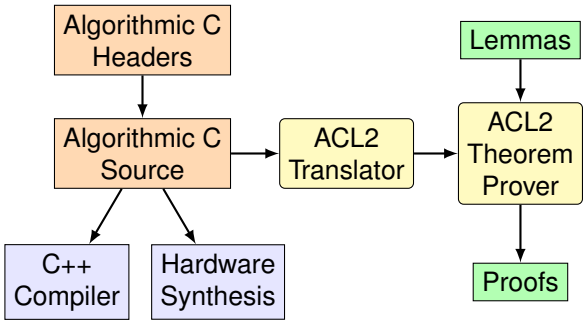

Fig. 4. Restricted Algorithmic C (RAC) toolchain.

```
#define STK_MAX_NODE1 16383

struct STKNode {
  i64 val;
  ui1 used;
  ui14 next;
};

struct STKObj {
  ui14 nodeHd;
  ui14 nodeTl;
  ui14 nodeCount;
  array<STKNode, STK_MAX_NODE1> nodeArr;
};
```

Fig. 5. Stack datatype declaration in Algorithmic C.

namely ACL2. Russinoff begins by presenting a formalization of modular arithmetic in standard mathematical notation, but backed by ACL2 collections of ACL2 lemmas, called *books* in ACL2 parlance. These *RTL* books are named after the "Register Transfer Logic" artifacts that hardware designers produce using Hardware Description Languages (HDLs); the RTL books are designed to assist in reasoning about the formalization of designs expressed at the level of RTL. After developing progressively more complex arithmetic circuits, Russinoff concludes his text with an exposition of the complete mathematical verification of representative Arm floating-point RTL for addition, multiplication, fused multiply/add, division, and square root. All of the sources and tools described in Russinoff's book are available as part of the ACL2 distribution, so the curious reader can reproduce his proofs.

The development language that Russinoff uses in his text is a subset of Algorithmic C [14]. Algorithmic C entails a set of freely available C++ header files providing support for hardware development, including the peculiar bit widths utilized in floating-point design, and enables compilation to both hardware and software platforms. John O'Leary and David Russinoff have created a toolchain for a subset of Algorithmic C, called Restricted Algorithmic C, or *RAC*. (RAC started as a similar toolchain for SystemC, called MASC [15].) The RAC toolchain accepts RAC source, parses it to an S-expression-based intermediate form, then translates this intermediate to S-expression forms acceptable to ACL2. A simplified view of the RAC toolchain is shown in Fig. 4; for a complete depiction, see Fig. V.1 of [13].

RAC imposes several restrictions on the Algorithmic C developer. The most significant of these is that all RAC function arguments must be pass-by-value, and all functions must be side-effect-free. Additional restrictions apply to loops, *etc*., in order to ease the translation of loops into ACL2 recursion, and so on; see Chapter 15 of [13] for details. C++ loops are translated into ACL2 tail-recursive functions[5]. The translator automatically generates ACL2 *measures* that are used in the proofs of termination that must be performed before ACL2 functions can be admitted into the logic; with the appropriate measures, ACL2 termination proofs proceed automatically without user input.

The challenge at hand, then, is to determine the RAC toolchain's effectiveness for the development of high-assurance hardware/software algorithms and data structures in varied domains, particularly our JSON filter.

IV. A Verified Stack Datatype for the JSON filter

As an example, consider the development of a basic stack datatype, implemented using a fixed-size array. The Algorithmic C header file declaration for this type is given in Fig. 5. We arbitrarily set the maximum number of stack elements to 16383 for this example.

Note the use of specific integer widths, such as the *ui14* declaration for the 14-bit unsigned integer edge 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 such as early error identification outweigh the costs. Also note that the struct-of-arrays of Fig. 5 is translated into ACL2 records, 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. One such stack operator is the *push* operator, depicted in Fig. 6.

A. Verification by ACL2 Proof

Once the stack datatype 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

[5]Tail-recursive functions can be efficiently compiled to loops, avoiding potential call stack overflow issues, but are somewhat more difficult to reason about than non-tail-recursive functions.

```
STYP STK_push (i64 v, STKObj amp(SObj)) {
  if ((SObj.nodeCount == MAX_NODE) || (v == BAD_VAL)) {
    return SVAL;
  } else {
    ui14 prevHd = SObj.nodeHd;
    ui14 index = STK_find_free_node(SObj);

    if (index > STK_MAX_NODE) {
      return SVAL;
    } else {
      SObj.nodeHd = index;
      if (SObj.nodeCount == 0) {
        SObj.nodeTl = index;
      }
      SASN STK_add_node_at_index(index, v, SObj);
      SObj.nodeArr[index].next = prevHd;
      return SVAL;
    }
  }
}
```

Fig. 6. Stack push function in Algorithmic C.

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
  (implies
   (and
    (good-stkp Obj)
    (spacep Obj)
    (acl2::signed-byte-p 64 n)
    (>= n *STK_MIN_VAL*))
   (= (STK_top (STK_pop (STK_push n Obj)))
      (STK_top Obj))))
```

where *good-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.

B. Validation by Simulation

In addition to formal verification of the stack datatype, we can perform validation of the datatype via simulation, both within ACL2, as well as via the C++ compilation and Hardware Synthesis paths depicted in Fig. 4. We have introduced some "preprocessor magic" to the return types, struct types, and return statements of our code so that larger structs can be used in C++ simulations, with the usual pass-by-reference semantics, while preserving the RAC pass-by-value restrictions for analysis.

V. Results

We were able to successfully realize a JSON lexical analyzer and syntactic analyzer in RAC for a significant subset of JSON (we chose not to deal with the complexities of, *e.g.*, Unicode in this first experiment). In our implementation, an input message (sequence of bytes) is presented to the lexical analyzer, and the tokens generated by the lexer are then directly fed to the syntactic analyzer. If the lexical and syntactic analyses both succeed, an input-bytes-to-token map, as well as a parse tree, are generated, and the message is allowed to be further processed by the UAV software; otherwise the input message is rejected. The code for the JSON filter comprised some 1800 RAC source lines, as well as approximately 600 lines of ACL2 theorems specific to the correctness of the JSON filter. All work was performed using the RAC tools provided as part of ACL2 version 8.2 running on Mac OS X.

We were able to test our JSON filter on concrete input messages by running the executable binary code generated by compiling the RAC code using the clang compiler, as well as by invoking the (automatically translated) JSON filter functions in ACL2 from the ACL2 read/eval/print loop; these two approaches yielded identical test results. We also measured performance of our compiled JSON lexer/parser, using test JSON input from [12]. The RAC-derived executable was some 20% faster than a lexer/parser generated by the (unverified) Menhir parser generator [16], running on the same hardware, and presented with the same input. The verified parser reported in [12] is 2-4 times slower than the Menhir-generated version.

VI. Related Work

Much of the work on system-level formal verification has been performed in the context of higher-level Model-Based System Engineering languages such as AADL [17], or Simulink/Stateflow (*e.g.* [18], [19]). Most of the formal verification work of which we are aware utilizes model checkers to establish Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) properties. While hardware/software co-design is enabled by these system-level architecture and design tools, very little has been done on true hardware/software co-assurance. Notable work has been done on the development of High-Assurance Domain-Specific languages targeting both hardware and software implementation (*e.g.* [20], [21]).

Floating-point hardware verification utilizing theorem proving technology has a notable history (*e.g.* [22], [23], [13]). Many of these efforts have focused on engineering artifacts expressed using traditional Hardware Description Languages, such as Verilog; Russinoff's work using Algorithmic C is a notable exception.

Many tools exist for the verification of C code. Tools that take a similar approach to ours include Appel's Hoare Logic for CompCert C [24], which is derived from the operational semantics of the CompCert verified C compiler in Coq [25]. The AutoCorres tool [26] arose out of the `seL4.verified` operating system verification effort; it translates ASTs from a parser for the restricted C dialect used in seL4 to Schirmer's SIMPL theory in Isabelle/HOL [27].

VII. Conclusion and Future Work

We have explored methods and tools for enhancing the safety and security of critical systems using a hardware/software co-assurance approach. We employed a state-of-the-art toolchain, Restricted Algorithmic (RAC), due to Russinoff and O'Leary, and documented in Russinoff's floating-point hardware verification book, to develop high-assurance architectural transformations that can be realized as hardware,

software, or a combination of the two. We utilized the RAC toolchain to translate Restricted Algorithmic C to the subset of Common Lisp supported by the ACL2 theorem prover. We translated an example JSON filter application featuring algebraic datatypes "lowered" to an array-based representation, verified table-driven lexing, as well as verified table-driven parsing, to ACL2, and then carried out correctness proofs in ACL2. Finally, we described methods for validation of our JSON filter by simulation, both within ACL2, as well as by execution of the compiled RAC source code.

In future work, we will investigate various ways to refine the RAC toolchain, *e.g.*, utilizing ACL2 typed records, and improving error handling. In the interest of "eating one's own dog food," we are keen to implement a verified version of the RAC-to-ACL2 translator using the verified lexer/parser technology used to create the JSON filter; however, we first need to develop a means for high-assurance invocation of "action code". We also see a need for RAC code generation from higher-level functional languages, such as Scala or ML. Finally, we will investigate the composition of hardware and software modules, both written in Algorithmic C, with support for proof development for the resulting system.

VIII. Acknowledgments

Many thanks to Sam Lasser of Tufts University for his assistance with the Vermillion verified LL(1) parser generator code; to David Russinoff of Arm for answering questions about the RAC toolchain; and to Konrad Slind of Collins Aerospace for creating the verified JSON lexer tables. This work was sponsored in part by the Defense Advanced Research Projects Agency (DARPA).

References

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

[2] D. S. Hardin and K. L. Slind, "Using ACL2 in the design of efficient, verifiable data structures for high-assurance systems," in *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, S. Goel and M. Kaufmann, Eds., vol. 280, 2018, pp. 61–76.

[3] S. Rasmussen, D. Kingston, and L. Humphrey, "A brief introduction to unmanned systems autonomy services (UxAS)," in *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, June 2018, pp. 257–268. [Online]. Available: https://doi.org/10.1109/SP.2013.35

[4] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an operating-system kernel," *Comm. ACM*, vol. 53, no. 6, pp. 107–115, 2010. [Online]. Available: http://doi.acm.org/10.1145/1743546.1743574

[5] D. S. Hardin, K. L. Slind, J. Å. Pohjola, and M. Sproul, "Synthesis of verified architectural components for autonomy hosted on a verified microkernel," in *Proceedings of the 53rd Hawaii International Conference on System Sciences*, January 2020, pp. 6365–6374.

[6] *The JSON Data Interchange Syntax Standard (ECMA-404)*, ECMA International, 2017. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[7] D. S. Hardin, K. L. Slind, M. A. Bortz, J. Potts, and S. Owens, "A high-assurance, high-performance hardware-based cross-domain system," in *Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings*, ser. LNCS, vol. 9922. Springer, 2016, pp. 102–113.

[8] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, Jan. 2014, pp. 179–191.

[9] J. Å. Pohjola, H. Rostedt, and M. O. Myreen, "Characteristic formulae for liveness properties of non-terminating CakeML programs," in *2019 International Conference on Interactive Theorem Proving (ITP)*, September 2019, pp. 32:1–32:19. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/lipics-complete/lipics-vol141-itp2019-complete.pdf

[10] J. Brzozowski, "Derivatives of Regular Expressions," *Journal of the ACM*, vol. 11, no. 4, pp. 481–494, October 1964.

[11] S. Owens, J. Reppy, and A. Turon, "Regular-expression derivatives re-examined," *Journal of Functional Programming*, vol. 19, no. 2, pp. 173–190, Mar. 2009.

[12] S. Lasser, C. Casinghino, K. Fisher, and C. Roux, "A Verified LL(1) Parser Generator," in *10th International Conference on Interactive Theorem Proving (ITP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. Harrison, J. O'Leary, and A. Tolmach, Eds., vol. 141, 2019, pp. 24:1–24:18. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/11079

[13] D. M. Russinoff, *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach*. Springer, 2018.

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

[15] J. W. O'Leary and D. M. Russinoff, "Modeling algorithms in SystemC and ACL2," in *Proceedings of the 12th International Workshop on the ACL2 Theorem Prover and its Applications*, vol. 152. EPTCS, 2014, pp. 145–162. [Online]. Available: https://arxiv.org/pdf/1406.1565.pdf

[16] F. Pottier and Y. Regis-Gianas, *Menhir Reference Manual*, INRIA, 2020. [Online]. Available: http://gallium.inria.fr/ fpottier/menhir/manual.pdf

[17] D. Cofer, J. Backes, A. Gacek, D. DaCosta, M. Whalen, I. Kuz, G. Klein, G. Heiser, L. Pike, A. Foltzer, M. Podhradsky, D. Stuart, J. Grahan, and B. Wilson, "Secure mathematically-assured composition of control models," Air Force Research Laboratory Information Directorate, Tech. Rep., September 2017. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/1039782.pdf

[18] M. Whalen, D. Cofer, S. Miller, B. Krogh, and W. Storm, "Integration of formal analysis into a model-based software development process," in *FMICS*, 2007.

[19] D. S. Hardin, T. D. Hiratzka, D. R. Johnson, L. G. Wagner, and M. W. Whalen, "Development of security software: A high assurance methodology," in *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (ICFEM'09)*, K. Breitman and A. Cavalcanti, Eds. Springer, 2009, pp. 266 – 285.

[20] S. Browning and P. Weaver, "Designing tunable, verifiable cryptographic hardware using Cryptol," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed. Springer, 2010, pp. 89–143.

[21] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas Lava," in *Implementation and Application of Functional Languages*, ser. LNCS, vol. 6041. Springer, 2009, pp. 18–35.

[22] J. Harrison, "Floating-point verification using theorem proving," in *Formal Methods for Hardware Verification*, M. Bernardo and A. Cimatti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 211–242.

[23] W. A. Hunt, S. Swords, J. Davis, and A. Slobodova, "Use of formal verification at Centaur Technology," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*, D. S. Hardin, Ed. Springer, 2010, pp. 65–88.

[24] A. W. Appel, *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[25] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[26] D. Greenaway, J. Lim, J. Andronick, and G. Klein, "Don't sweat the small stuff: Formal verification of C code without the pain," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 429–439. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594296

[27] N. Schirmer, "Verification of sequential imperative programs in Isabelle/HOL," Ph.D. dissertation, TU Munich, 2006.