VERIFIED HARDWARE/SOFTWARE CO-ASSURANCE: ENHANCING SAFETY AND SECURITY FOR CRITICAL SYSTEMS

NOTE: CORE OF THIS WORK PREVIOUSLY PUBLISHED IN IEEE SYSCON 2020

David S. Hardin

Trusted Systems Group
Collins Aerospace

with additional contributions by

Matthew Weis and Robby
Kansas State University

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

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.

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

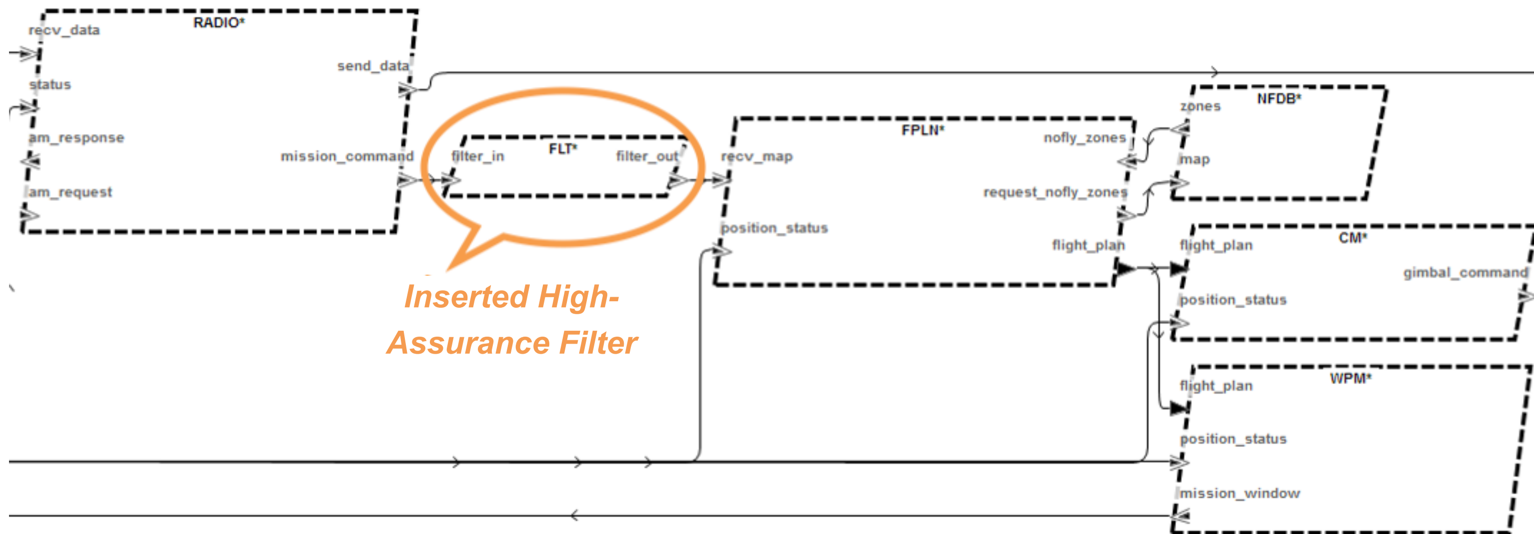
DARPA CASE: SIMPLE UAV USE CASE

- Architecture models in our CASE effort are expressed in the SAE standard Architectural Analysis and Design Language (AADL)
- The CASE Cyber Requirements tools examine the AADL model for the system, in this case a UAV, identifying potential cyber vulnerabilities
- The CASE user then identifies architectural transformations to be applied to the model to address the vulnerabilities
- Let's say the need for an input validation 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

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

DARPA CASE: SIMPLE UAV USE CASE (CONT'D.)



© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

EXAMPLE: JSON MESSAGE FORMAT

- In our use case, a UAV air-ground communications system employs JSON to encode certain messages sent between the ground-based control station and the UAV

- JSON (JavaScript Object Notation) is a popular 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 (grouped within `{}`)
  - an ordered list of values (grouped within `[]`)

- For example, a UAV coordinate could be encoded in JSON as:


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


Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

VERIFIED FILTERING OF JSON MESSAGES

- 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 that 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, to ensure that it is legal JSON

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

LEXICAL ANALYSIS

- We describe the lexical tokens, or *lexemes*, of JSON by way of regular expressions
- JSON lexemes include:
  - true, false, null, string, float, and integer values
  - `{`, `}`, `[`, `]`, `:`, `,`
- A lexical analyzer generator accepts these regex token specifications, and produces a lexer that takes an input candidate JSON message, and produces a list of JSON tokens

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

FAST REGULAR EXPRESSION PATTERN MATCHING

- Brzozowski (1964) presents a method for compiling a regular expression to a Deterministic Finite-state Automaton (DFA), which is subsequently run on candidate strings
  - Thus, regular expression matching becomes quite fast
- We have performed a correctness proof for regular expression compilation to DFAs, and contributed it to the HOL4 theorem prover source distribution
  - In **`examples/formal-languages/regular`**
- Thus, we can utilize a verified compiler toolchain to create a verified regular expression pattern matcher down to the binary level
  - The matcher is, in essence, a simple DFA state traverser function
- We then combine the individual regular expressions for our JSON tokens to create an overall DFA table for the lexer

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

SYNTACTIC ANALYSIS

- We utilize a verified parser generator to create a verified parser table for JSON
  - In particular, we employ the Vermillion verified LL(1) parser generator (Lasser *et al.* 2019)
    - We present a set of grammar rules for legal JSON messages to Vermillion;
    - extract the verified parser table from Vermillion;
    - then transfer the verified parser table to our target system

- By generating both the lexer table and parser table in formally verified fashion, we enable *verified data* to be transferred from host-based provers to the target system

- The parser function is a bit more complex than the lexer, in that it requires a rule stack; otherwise, it is also a fairly straightforward table traverser

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

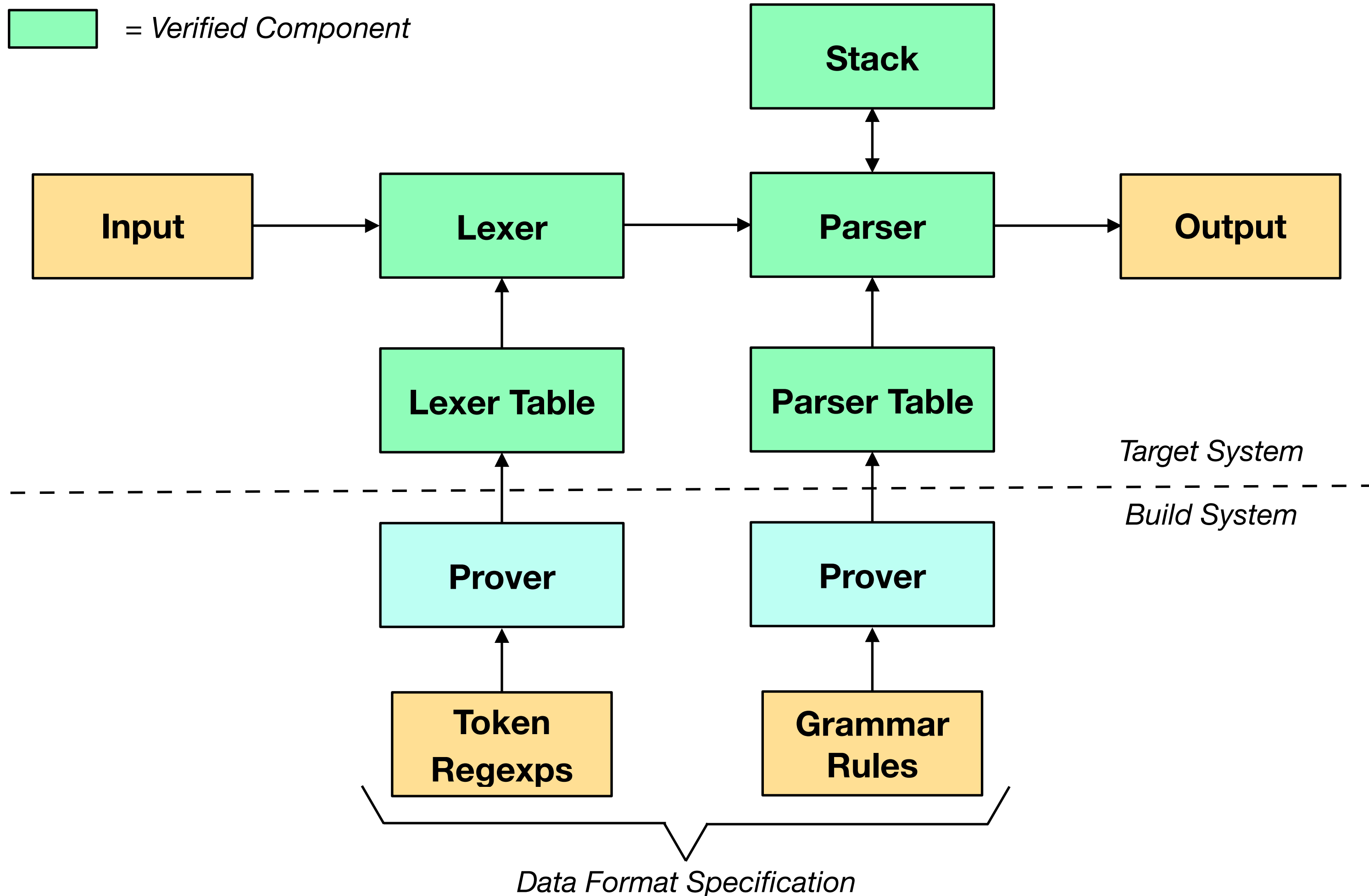
HARDWARE/SOFTWARE CO-DESIGN AND CO-ASSURANCE FOR THE JSON FILTER

- We wish to create a lexical/syntactic filter for JSON using hardware/ software co-design and co-assurance techniques
- We utilize the verified lexer generation and verified parser generation capabilities described earlier to create the lexer table and parser table for JSON (“verified data”)
- The table traversal code is written by hand in the hardware/software co-design language
  - In the future, this code will be automatically generated
- For the parser rule stack, we utilize a previously verified fixed-size stack component written in the hardware/software co-design language

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

JSON FILTER BUILT FROM HARDWARE/ SOFTWARE CO-ASSURANCE COMPONENTS



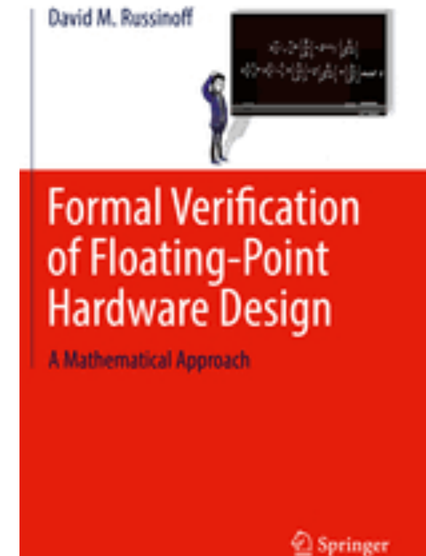
Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

THE RUSSINOFF-O'LEARY 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 (ACL2 Workshop 2014)
  - 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 2018 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 as part of the standard ACL2 distribution



© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

ALGORITHMIC C

- The Algorithmic C datatypes “provide a basis for writing bit-accurate algorithms to be synthesized into hardware”
- The Algorithmic C datatypes are defined via an open source C++ header file that users can #include in their designs
  - No runtime library required
- 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
- Further information is available at **`https://hlslibs.org`**

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

RESTRICTED ALGORITHMIC C (RAC)

- Restricted Algorithmic C defines a C++ subset that promotes proof, hardware synthesis, and simulation

- Use case: A hardware developer expresses hardware functionality in RAC, which is then translated into a theorem prover language used by the verification expert

- RAC encompasses many of the restrictions common in “high-assurance” C, such as no function pointers, disallowing recursion, etc.
  - RAC disallows all pointers, as well as function side-effects
  - Certain control constructs (e.g., breaking out of a **`for`** loop) are also disallowed

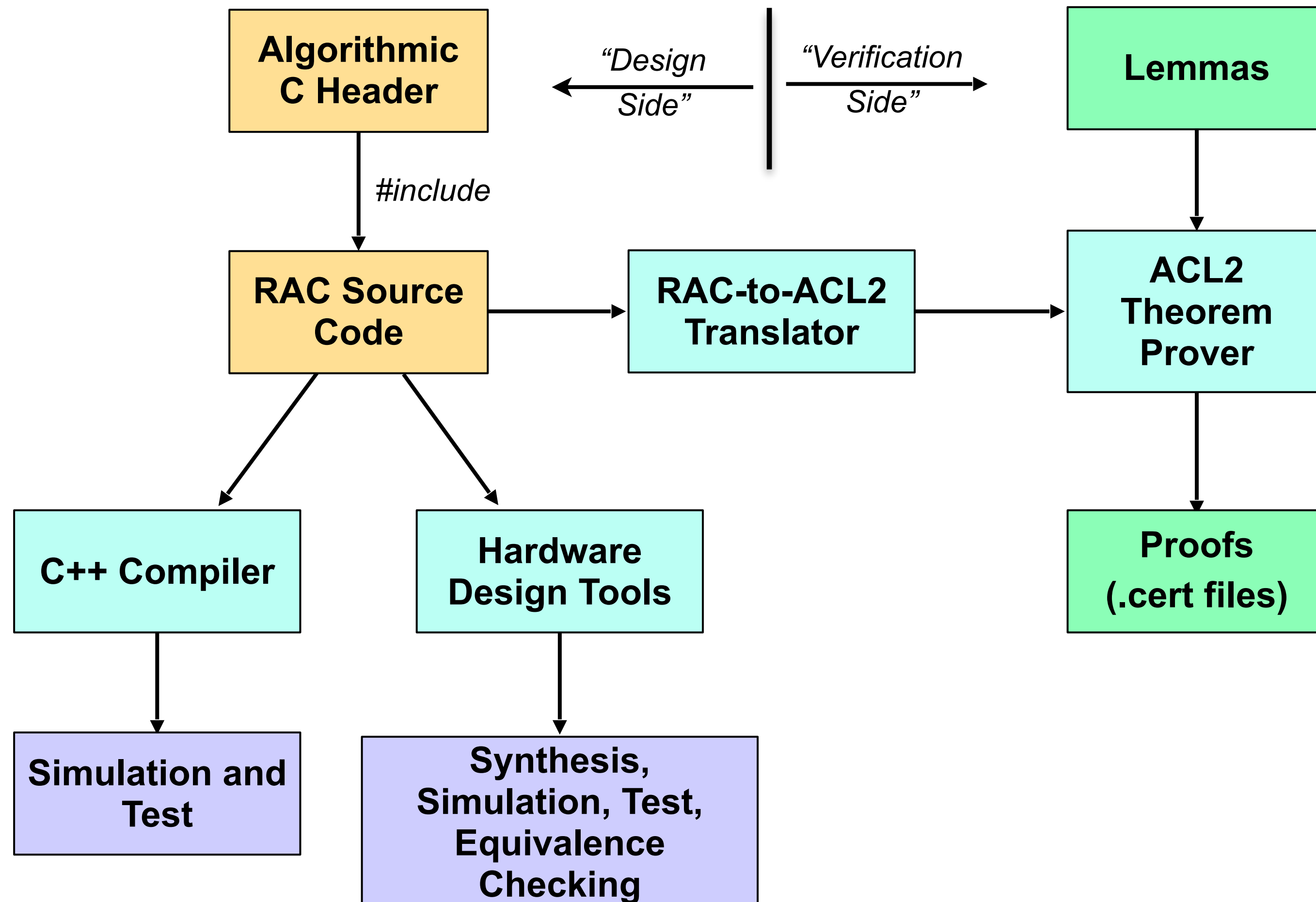
- RAC supports bit slices, C++ arrays, and C++ tuples for multiple-value return

- For more information on RAC, please consult Chapter 15 of Russinoff’s book

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

RESTRICTED ALGORITHMIC C TOOLCHAIN



© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

ACL2

- ACL2 is “A Computational Logic for Applicative Common Lisp”, developed by Matt Kaufmann and J Moore
    - Recipients of the 2005 ACM Software Systems Award
- ACL2 developers model their system as Common Lisp functions, then state and prove theorems about their model using ACL2’s highly automated proof heuristics
    - These functions and theorems are gathered into libraries, called books, which are proved once, then utilized many times
- ACL2 has been used in many large academic and industrial verification efforts:
    - Floating-point unit verification (AMD, ARM, Centaur, Intel, Oracle)
    - AAMP7 separation kernel microcode and Green Hills INTEGRITY-178B kernel information flow verification (Collins Aerospace)
    - Used to certify the correctness of the “world’s largest math proofs” (Heule)
        - Proofs are discovered by massively parallel SAT solving

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

BRIDGING THE DESIGN/VERIFICATION GULF

- A key issue in the formal verification of engineering artifacts is the gulf between the sorts of programs that can be readily specified and verified, and the sorts of programs that "real-world" developers actually write:

Formal Verification "Comfort Zone"	***Real World***
Functional Programming	**Imperative Programming**
Total, terminating functions	**Partial, potentially non-terminating functions**
Non-tail-recursive functions	**Loops**
Okasaki-style pure functional algebraic data types	**Structs and Arrays**
Infinite-precision Integers	**Modular Integers**
Linear Arithmetic	**Linear and non-linear arithmetic**

- The Russinoff-O'Leary toolchain, in combination with the ACL2 theorem prover, does an admirable job of bridging these two worlds

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

RAC-TO-ACL2 TRANSLATOR

- Translates loops into tail-recursive functions
- Generates ACL2 “measures” to aid in function termination proofs
  - All functions to be admitted into ACL2 must be proved to terminate
  - Termination proofs are conducted mostly automatically by ACL2, with hints provided by the measure annotations (if needed)
- Translates fixed-width integer operations into functions defined in Russinoff’s “RTL” (Register Transfer Language) ACL2 books
  - Ensures that translated operations are “wrapped” with an appropriate RTL bit-width coercion operator so as to accurately translate modular integer arithmetic
  - RTL is described in detail in Part I of Russinoff’s book

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

RAC-TO-ACL2 TRANSLATOR (CONT'D.)

- Converts assignments to Lisp let-bindings
- Converts struct/array reads/writes to ACL2 record gets/sets, for which get-over-set, set-over-get, etc. theorems are available
- In addition, ACL2's powerful arithmetic capability allows it to reason about non-linear arithmetic expressions
- ACL2 also features a very capable induction scheme generator
  - ACL2 automatically finds suitable induction schemes for the vast majority of inductive proof attempts, including hybrid schemes
- …allowing us to reason about real-world designs expressed in RAC

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

RESULTS

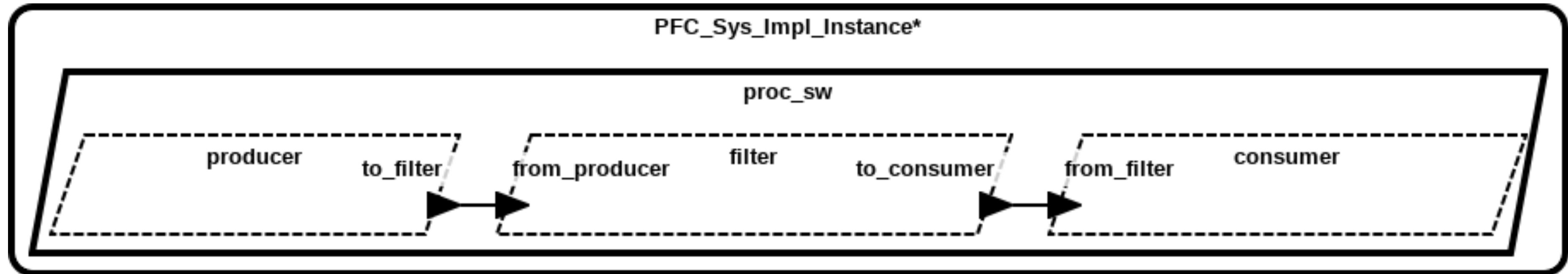
- We successfully realized a JSON lexer/parser-based filter in RAC for a significant subset of JSON
  - We avoided some complexities, e.g. Unicode, for this proof-of-concept effort
- The code for the JSON filter comprised ~1800 RAC source lines, as well as 600 lines of ACL2 theorems dealing with JSON filter correctness
- We tested our JSON filter on concrete input messages by running the executable obtained by compiling the RAC code using a C++ compiler; as well as by executing the translated JSON filter functions in ACL2
  - These tests produced identical results
- We measured the performance of our JSON filter using test data from (Lasser *et al*. 2019) against benchmark code generated by the (unverified) Menhir parser generator; the RAC code was some 20% faster than the Menhir-generated version

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

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

Demo: Synthesize Hardware for CASE-generetd filter in C

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

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

EXPERIMENTAL SETUP

- ZedBoard
  - Low-cost ($450) Xilinx Zynq 7000 development board
  - Zynq 7000 includes a Linux-capable 1 GHz Arm Cortex-A9, along with a reprogrammable FPGA fabric
- Hardware/Software Co-Design Tool: Xilinx Vivado HLS
  - Similar to Algorithmic C; a bit less restrictive than RAC
  - We have since created C macros to harmonize Vivado HLS and RAC bit-accurate types and bit slices
- DARPA CASE tools used to synthesize the demo system directly from the high-level AADL model

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

CASE INPUT FILTER IN C (WITH SLIGHT HLS ADAPTATIONS)

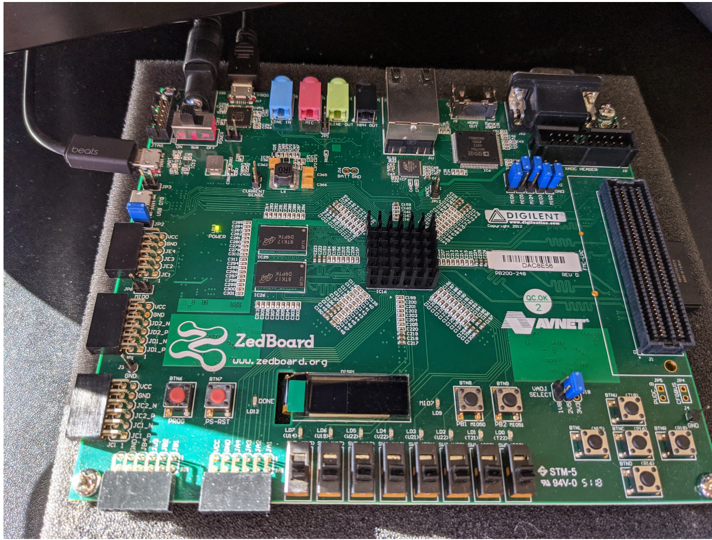
ORIGINAL SPLAT C CODE COURTESY OF KONRAD SLIND

```
/*--------------------------------------------------------------------------
 * DFA CASE_Filter is the compiled form of regexp
 *     ([\166-\255][\255][\255][\255] | [\000-Z][\000][\000][\000])
 *     ([L-\255][\255][\255][\255] | [\000-\180][\000][\000][\000])
 *     ([\000-\152][:][\000][\000] | .[\000][\000][\000] |
 *     .[\001-9][\000][\000])
 *
 * Number of states in DFA: 22
 *--------------------------------------------------------------------------*/

int ACCEPTING_CASE_Filter[22] = { /* elided */ };
unsigned long DELTA_CASE_Filter[22][256] = {  /* elided */  };


#define MAXSIZE 512

int case_filter(unsigned char s[MAXSIZE], int len) {
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE s_axilite port=s
    #pragma HLS INTERFACE s_axilite port=len

    int state, i;
    state = 0;

    #pragma HLS loop_tripcount min=0 max=512
    for (i = 0; i < len; i++) {
        state = DELTA_CASE_Filter[state] [s[i]];
    }
    return ACCEPTING_CASE_Filter[state];
}
```

Collins Aerospace

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

TESTING ON ZEDBOARD




```
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])
```


© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

CONCLUSION

- Safety-critical/security-critical cyber-physical system development is a significant, growing challenge in the age of Internet connectivity
- On the DARPA CASE program, we are developing a method and toolchain for realizing safety- and security-enhancing architectural transformations by means of formally verified component implementations
- We have described a means of realizing verified input filtering components using a hardware/software co-design/co-assurance technique, namely RAC, and presented a JSON filter implemented using RAC, augmented with leading-edge verified lexer table/parser table generation
- Kansas State has developed a prototype hardware/software co-design toolchain, which they have successfully demonstrated using a CASE-derived filter, the CASE tools, and Xilinx Vivado HLS, on an FPGA development board

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.

FUTURE WORK

- We will continue to work to refine the RAC tools, in collaboration with its developers at Arm (second edition of Russinoff's book is in preparation)
- Kansas State is continuing to work on extending their toolchain
  - We will work with KSU to harmonize Vivado HLS and RAC source code
  - We will collaborate on additional hardware/software co-designs, including proofs of correctness
- Currently limited to Linux builds; Drivers, etc. for seL4 are a future goal
- We would like to implement a verified version of the RAC-to-ACL2 translator using the verified lexer/parser technology used to create the JSON filter
  - We first need to develop high-assurance invocation of "action code"

© 2021 Collins Aerospace, a Raytheon Technologies company. All rights reserved.