FORMAL SYNTHESIS OF FILTER COMPONENTS FOR USE IN SECURITY-ENHANCING ARCHITECTURAL TRANSFORMATIONS

David S. Hardin and Konrad L. Slind

Applied Research and Technology
Collins Aerospace

with additional contributions by

Matthew Weis and Robby
Kansas State University

Collins Aerospace

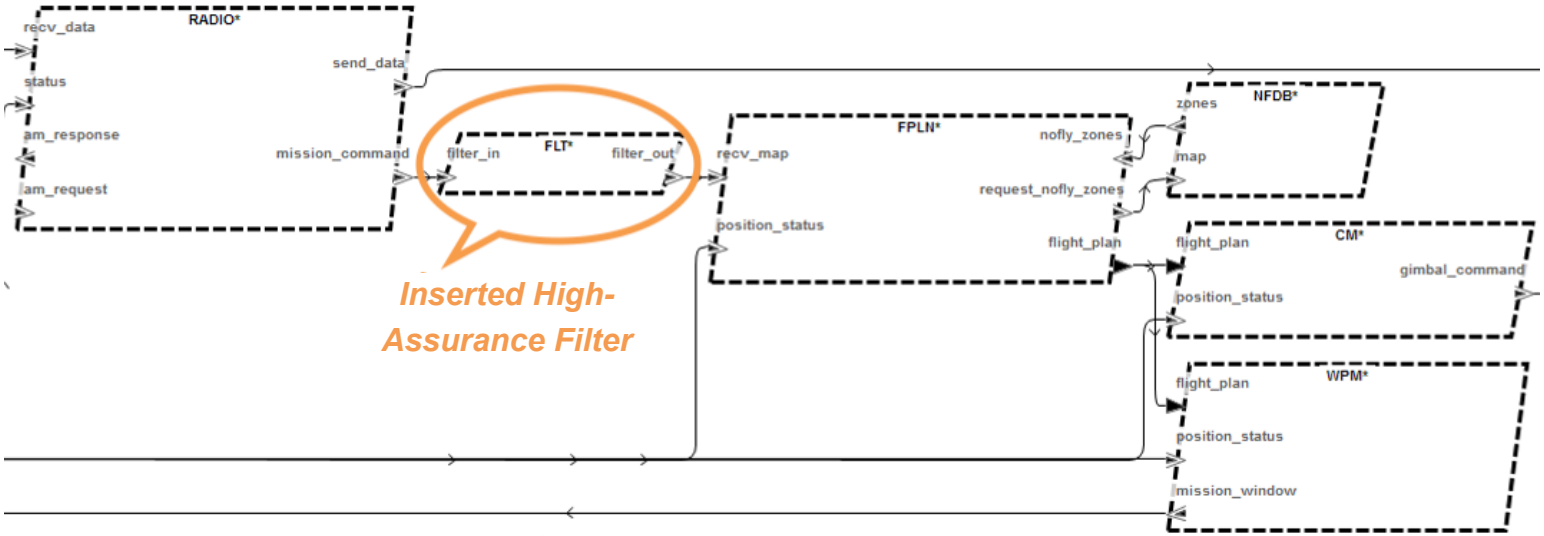
DISCLAIMER

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



CASE FORMAL MESSAGE SPECIFICATIONS

- We concur with the LangSec tenet that “the only path to trustworthy software that takes untrusted inputs is treating all valid or expected inputs as a formal language.”
  - Our work differs somewhat from many LangSec efforts in that we focus on providing standalone high-assurance message filter and monitor components
  - This is due to the fact that part of our mission statement on DARPA CASE is to provide security-enhancing components for existing systems, whose legacy components may not be able to be readily modified to introduce a LangSec-based parser as part of its codebase
- We have investigated three different formal message specification/synthesis approaches:
  - Regular Expressions
  - Context-Free Grammars
  - Contiguity Types
- We will examine each of these in turn

REGULAR EXPRESSIONS

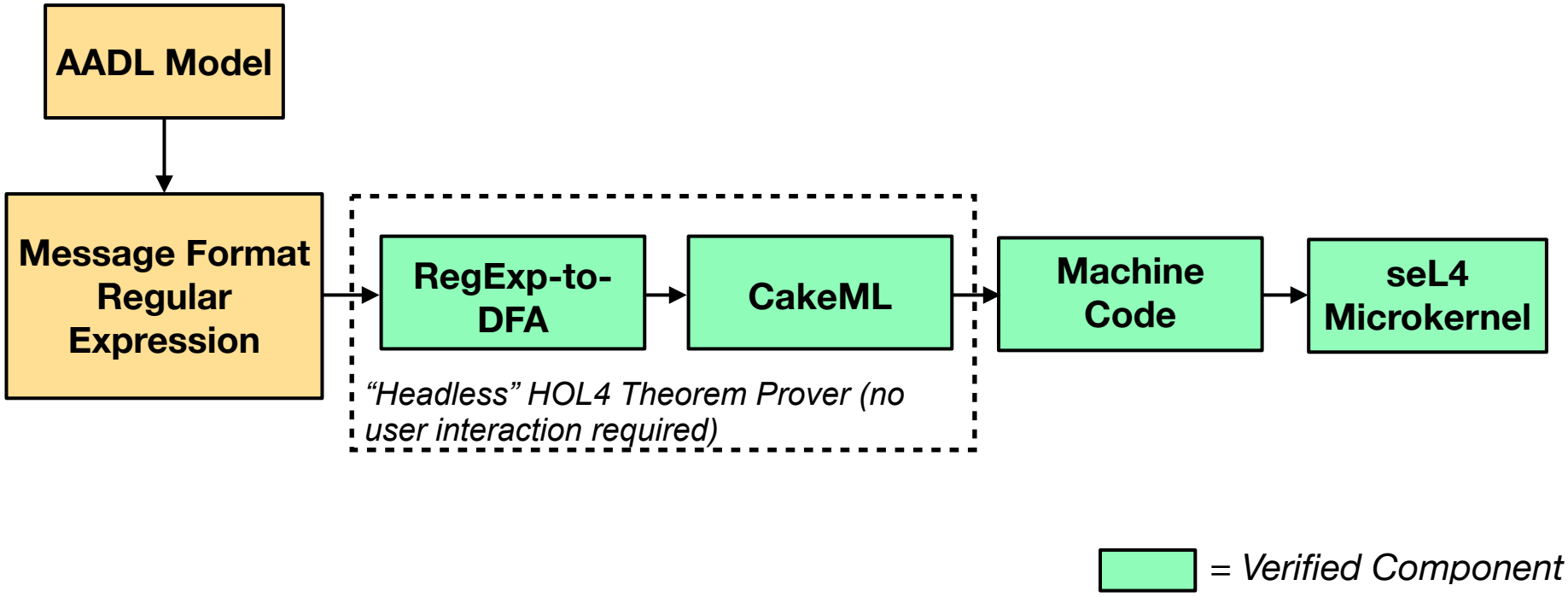
MESSAGE FORMATS FROM AADL MODELS

- Our AADL models feature assume/guarantee contracts expressed in the AGREE AADL annex
- We have built a translation of AGREE component input contracts to regular expressions, and provide proof tools for verifying semantical properties of the generated regular expressions.
- We call this tool SPLAT (Semantic Properties for Language and Automata Theory), since it combines formal language theory with specifications on the interpretation of the flat strings comprising a message format
- SPLAT also provides automated synthesis of message handling binary code using the facilities of the CakeML verified compiler

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

CASE VERIFIED FILTER SYNTHESIS TOOL FLOW (REGEX)



CONTEXT-FREE GRAMMARS

EXAMPLE: JSON MESSAGE FORMAT

- Use case: a UAV air-ground communications system employs JSON (JavaScript Object Notation) to encode certain messages sent between the ground-based control station and the UAV

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

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

- In order to create such a well-formedness filter for JSON, we need to perform both *lexical* and *syntactic* level analysis on any candidate JSON message, to ensure that it is legal JSON

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
  - We combine the individual regular expressions for our JSON tokens to create an overall DFA table for the lexer

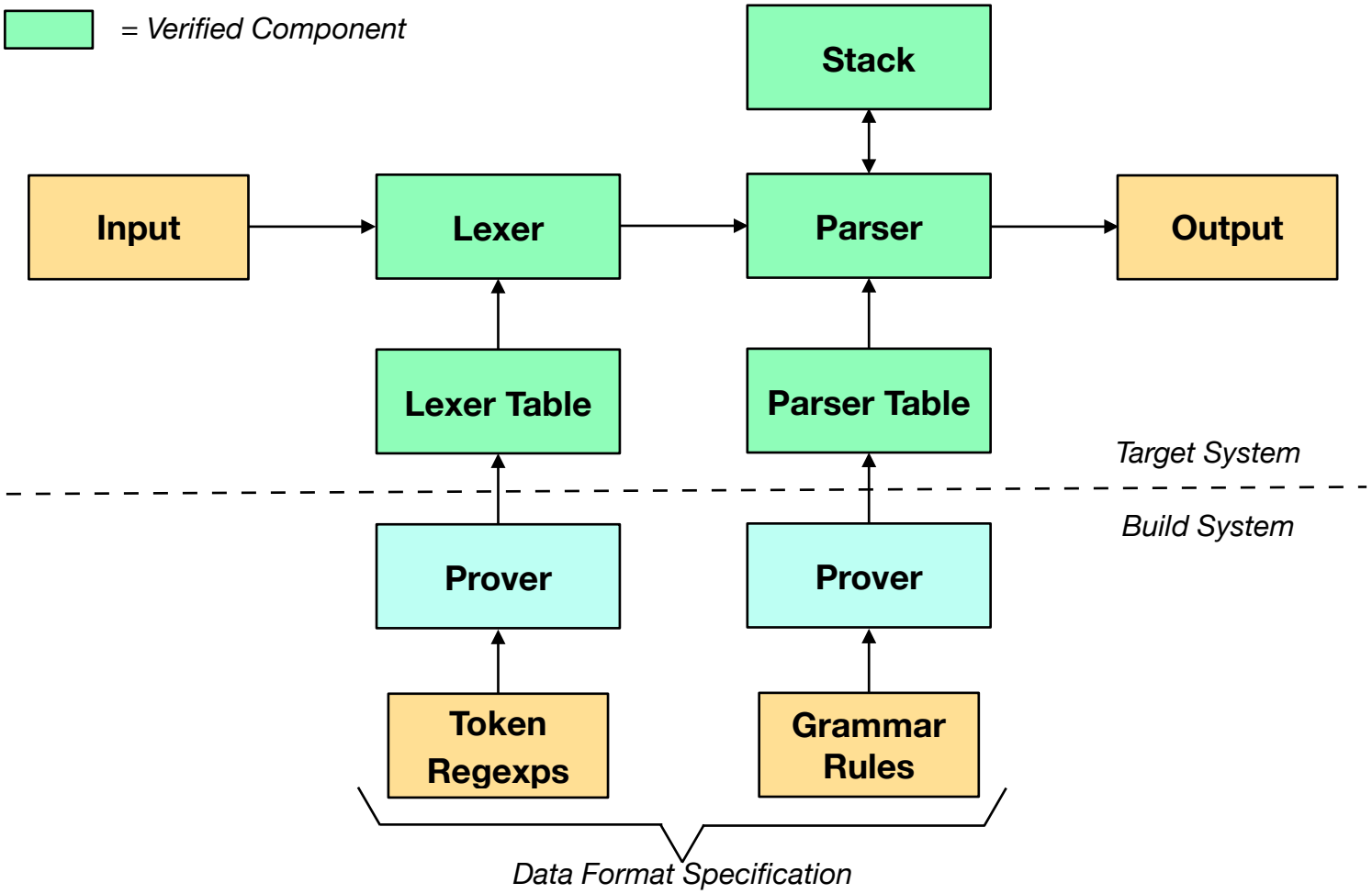
SYNTACTIC ANALYSIS

- 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

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

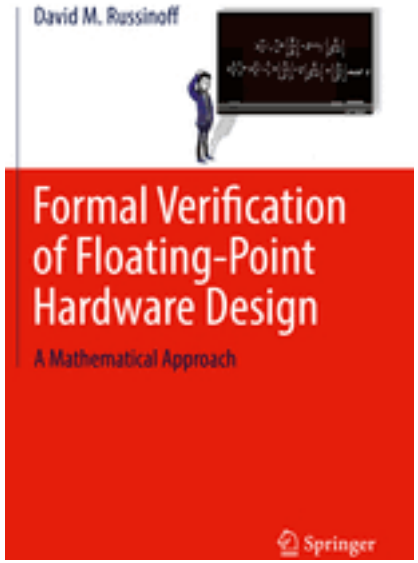
- We wish to create a lexical/syntactic filter for JSON using hardware/software co-design and co-assurance techniques
- We utilize the verified lexer generation and verified parser generation capabilities described earlier to create the lexer table and parser table for JSON (“verified data”)
- The table 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

JSON FILTER BUILT FROM HARDWARE/ SOFTWARE CO-ASSURANCE COMPONENTS



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
  - 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 in the standard ACL2 thereom prover distribution

ALGORITHMIC C

- The Algorithmic C datatypes “provide a basis for writing bit-accurate algorithms to be synthesized into hardware”

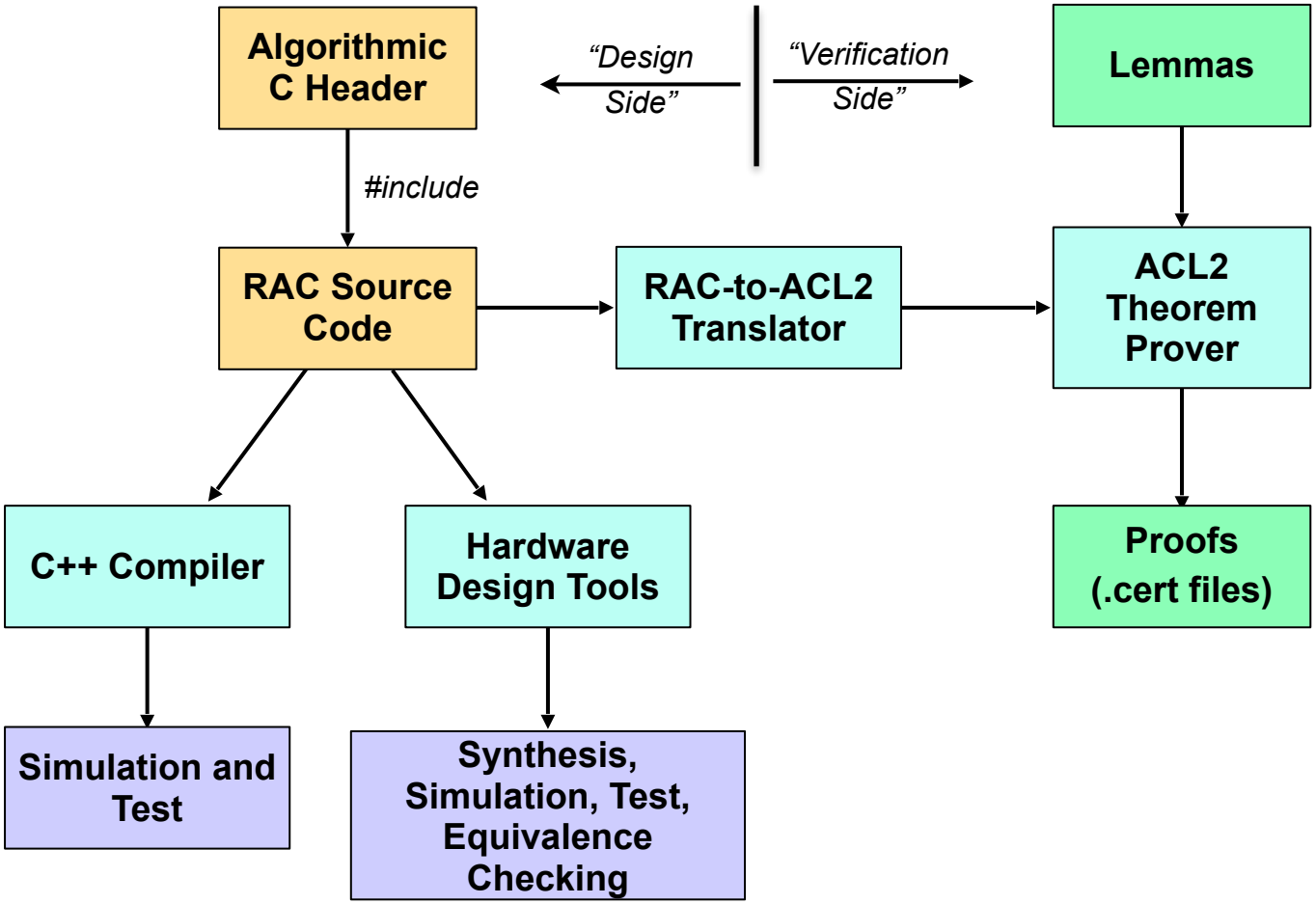
- Example use:
  - `typedef ac_int<112,false> ui112;`

  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

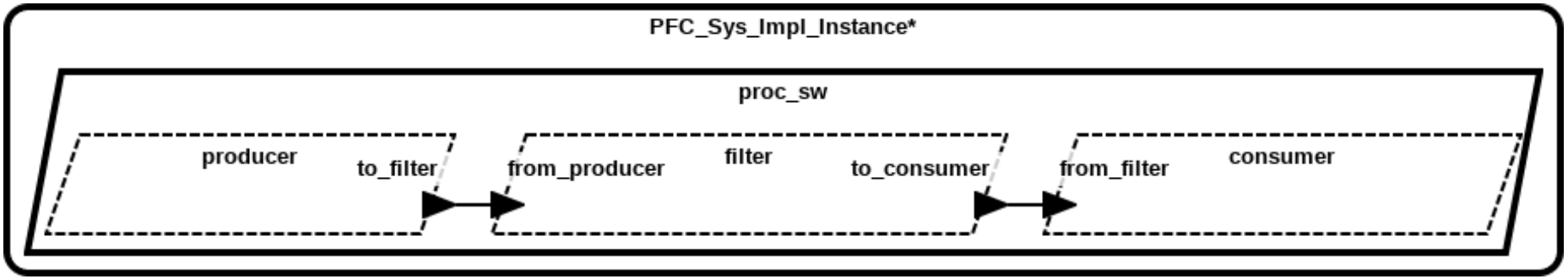
RESTRICTED ALGORITHMIC C TOOLCHAIN



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

Demo: Synthesize Hardware for CASE-generated filter

AADL Model:

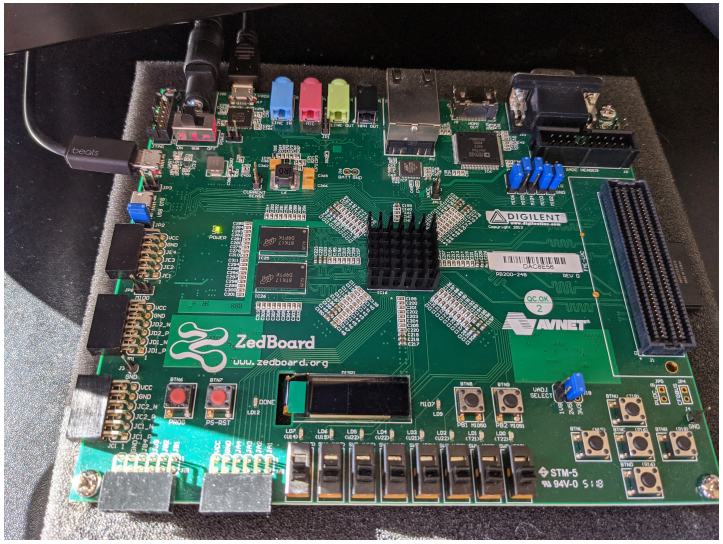


…mapped to
Linux software process

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

…mapped to
Linux software process

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])
```

CONTIGUITY TYPES

SELF-DESCRIBING MESSAGE APPROACHES

- Any successful message format description language must deal with self-describing message formats
- The self-describing aspect, unfortunately, defeats the use of standard lexer and parser technology
- Regular languages are insufficient, and context-free grammars can't capture the dependency of one field (or collection of fields) on another
- Context-sensitive grammars are a possibility, but there is little tool support
- Parser combinators allow hand-rolled parsers to be very quickly constructed, and can be adapted to support the accumulation and propagation of the necessary dependency information
- Chlipala and others have done work in this area using dependent types in Coq

- We have decided to take a Formal Language Theory approach since there is a strong connection between messages (flat strings of bits/bytes) and the strings of formal language theory

CONTIGUITY TYPES

- Contiguity Types attempt to bridge the gap between standard data structures (base types, arrays, records) and Formal Language Theory
- We start with a collection of base types and close under making records, arrays, and unions:

- $base ::= bool \mid char \mid u8 \mid u16 \mid u32 \mid u64 \mid i16 \mid i32 \mid i64 \mid float \mid double$

  $\tau ::= base \mid \textbf{\textit{Recd}}\ (f_1 : \tau_1)\ \ldots (f_n : \tau_n) \mid \textbf{\textit{Array}}\ \tau\ \text{exp} \mid \textbf{\textit{Union}}\ (bexp_1 : \tau_1)\ \ldots (bexp_n : \tau_n)$

- The Array construct uses arithmetic expressions to describe the number of elements in the array
- The Union construct uses boolean expressions to decide which choice to take
- In both cases the expressions can refer to earlier elements in the message
- L-values and R-values provide a comprehensive notation for such references.
  - L-values can occur on the left-hand side of an assignment, R-values on the right, e.g., A[0].foo := bar.B[i] + z;

EXAMPLE CONTIGUITY TYPES

- Record type, variable-length array of floats, variable-length 2D array of bytes:

```
{ A : Char
  B : Float
  C : Double
  D : i64
  E : u32 [1024]
 }
```

```
{ len : u16
  elts : Float [len]
}
```

```
{ Dims : { rows : u32
           cols : u32
         }
  Image : u8 [Dims.rows * Dims.cols]
}
```

- Two versions of a GPS coordinate, selected by a Release field:

```
{Release : u8
 contents : Union {
     (Release <= 14 --> {lat : Double, lon : Double, alt : u32})
     (Release >  14  --> {lat : Double, lon : Double, alt : u32,
                          altTy : AltitudeType, status : Bool})}
}
```

ASSERTIONS ON MESSAGES

- Contiguity types can express in-message assertions:
- $\boldsymbol{Assert}\ bexp = \boldsymbol{Union}\ (bexp,\ \boldsymbol{SKIP})\ (\neg bexp,\ \boldsymbol{VOID})$
- $\boldsymbol{SKIP} = \boldsymbol{Recd}\ [\ ]$
- $\boldsymbol{VOID} = \boldsymbol{Union}\ (\boldsymbol{false},\ u8)$

- In words: "If $bexp$ is true, continue without consuming any bytes, else fail"

- We can place assertions inside contiguity types that will be checked when the message handler runs

- Example: macro for variable-length arrays with bounds

```
BoundedArray contig bound =
   {len : u16
    len-check : Assert(len <= bound)
    elts : contig [len]
  }
```

ALGORITHMS FOR CONTIGUITY TYPES

- **Decoding**. A decoder breaks a sequence of bytes up and puts the pieces into a useful data structure, typically a parse tree.
- **Filtering**. Computes an answer to the question: “does a sequence of bytes meet the specification of a given contiguity type”. This is an instance of the language recognition problem in Formal Language Theory.
- **Serialization**. Given a contiguity type, synthesize a function that writes a compact binary version of a data structure to a message.
- **Test generation**. Given a contiguity type, generate byte sequences that do (or do not) meet its specification and feed the sequences to implementations in order to observe their behavior.
- **Learning**. Given training sets of message buffers that are accepted/rejected by an implementation, attempt to discover a contiguity type exactly capturing the entire set of messages.

- Thus far, we have implementations for decoders, filters, and test generation:
  - $\boldsymbol{decode} : \tau \rightarrow bytes \rightarrow parsetree$
  - $\boldsymbol{filter} : \tau \rightarrow bytes \rightarrow bool$
  - $\boldsymbol{testgen} : \tau \rightarrow bytes\ list$

SEMANTICS OF CONTIGUITY TYPES

- Contiguity types look like “normal” programming language types, but they are in fact formal languages
- Each base type is characterized by its width, and denotes all byte-strings of that width; thus the meaning of base type i64 is “all strings of length 8 bytes”
- The meaning of $\boldsymbol{Recd}\ (f_1 : \tau_1) \cdots (f_n : \tau_n)$ is the concatenation of the languages denoted by $\tau_1 \ldots \tau_n$. (*)
- The meaning of $\boldsymbol{Array}\ \tau\ exp$ is “evaluate exp to get a natural number n; the result is the n-ary concatenation of the language denoted by τ" . (*)
- The meaning of $\boldsymbol{Union}\ (bexp_1, \tau_1) \cdots (bexp_n, \tau_n)$ is “if there is exactly one $(bexp_k)$ that evaluates to true, return the language denoted by τ_k: else, return the empty language $\varnothing$”
- Expression evaluation (and Boolean expression evaluation) is done with respect to an environment that maps L-values to slices of the message

- (*) $\boldsymbol{Recd}$ and $\boldsymbol{Array}$ are accumulative: the meaning of later fields (or array elements) can depend on the values of earlier fields

FORMALIZATION

- We have defined contig-types and the matcher and filter functions over them in HOL4
- These have been proved to terminate
- The matcher function $\boldsymbol{match} : \tau \rightarrow bytes \rightarrow \theta\ option$ attempts to match the contig-type against the byte array
- If successful, match returns the built-up environment θ which binds locations specified by τ to segments of the byte array
- A simple version of the correctness of the matcher can be stated as

$$\boldsymbol{match}\ \tau\ bytes = \boldsymbol{Some}\ \theta \implies \theta(\tau) = bytes$$

- Also needed is a theorem relating the language of a contig-type with the matcher's behavior:

$$\left(\exists \theta .\ \ \boldsymbol{match}\ \tau\ bytes = \boldsymbol{Some}\ \theta\right) \iff bytes \in Lang(\tau)$$

- This proof effort is underway

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 implementations automatically synthesized using the verified CakeML compiler
- We have described the use of regular expressions, context-free languages, as well as a novel contiguity type specification language, to formally describe message formats
- We have conducted formal synthesis for wellformedness checkers including verified DFA generation, table-driven lexer generation, and table-driven parser generation
- 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
- In future work, we will continue to enhance our verified synthesis tools for increasingly complex systems
- We will continue to evolve the contiguity type specification language, developing a compiler for contiguity type specifications, and bridging the gap with conventional parsing technology based on context-free grammars