Correct-By-Construction Generation of C code for seL4 Networking

Eric Smith (eric.smith@kestreltechnology.com)
Alessandro Coglio (coglio@kestreltechnology.com)

Kestrel Technology, LLC

Who we Are: Kestrel Technology, LLC

- Small business spun out of Kestrel Institute in 2000
- Applied R&D firm with ambitions to make software safer
- **Formal network stack synthesis (this project)**
- Formal analysis of software:
  - x86 binary analysis: Lift into logic for verification, matching, and analysis.
  - Formal Unit Tester: Supplement unit tests with small proofs about your code.

- CodeHawk static analyzer
  - For C, x86 binaries, Java
  - Emphasis on soundness
  - Now open source

Palo Alto, CA

http://kestreltechnology.com/

Verified Network Stack Synthesis: Project Overview

- Goal: Synthesize a high-assurance TCP/IP implementation for seL4.
  - Generate formally-verified C code.
  - Cover IP, TCP, UDP, etc.
  - Make the generated code open source.
  - Use/extend Kestrel’s open source synthesis tools.
- Benefits:
  - Will help others build secure systems on seL4.
  - Will help extend seL4's assurance case to larger systems.
  - Code will be provably immune to various attacks (subject to assumptions).
- Use ACL2 theorem prover
- Use Kestrel’s APT toolkit and ATC C code generator

Functionality to Synthesize

- Packet parsing and serialization
- The TCP state-machine
- Buffer / queue management
- Sequence numbers and re-transmitting lost packets
- Flow control
- Congestion control

- Approach: Start with a minimal feature set and add features over time.

- See our previous Summit talks for more details.

Project Status

- We have a detailed design for the network stack.
- We have a working C generator (ATC) and supporting transformation toolkit (APT).
- We have started writing ACL2 specifications for the network stack.

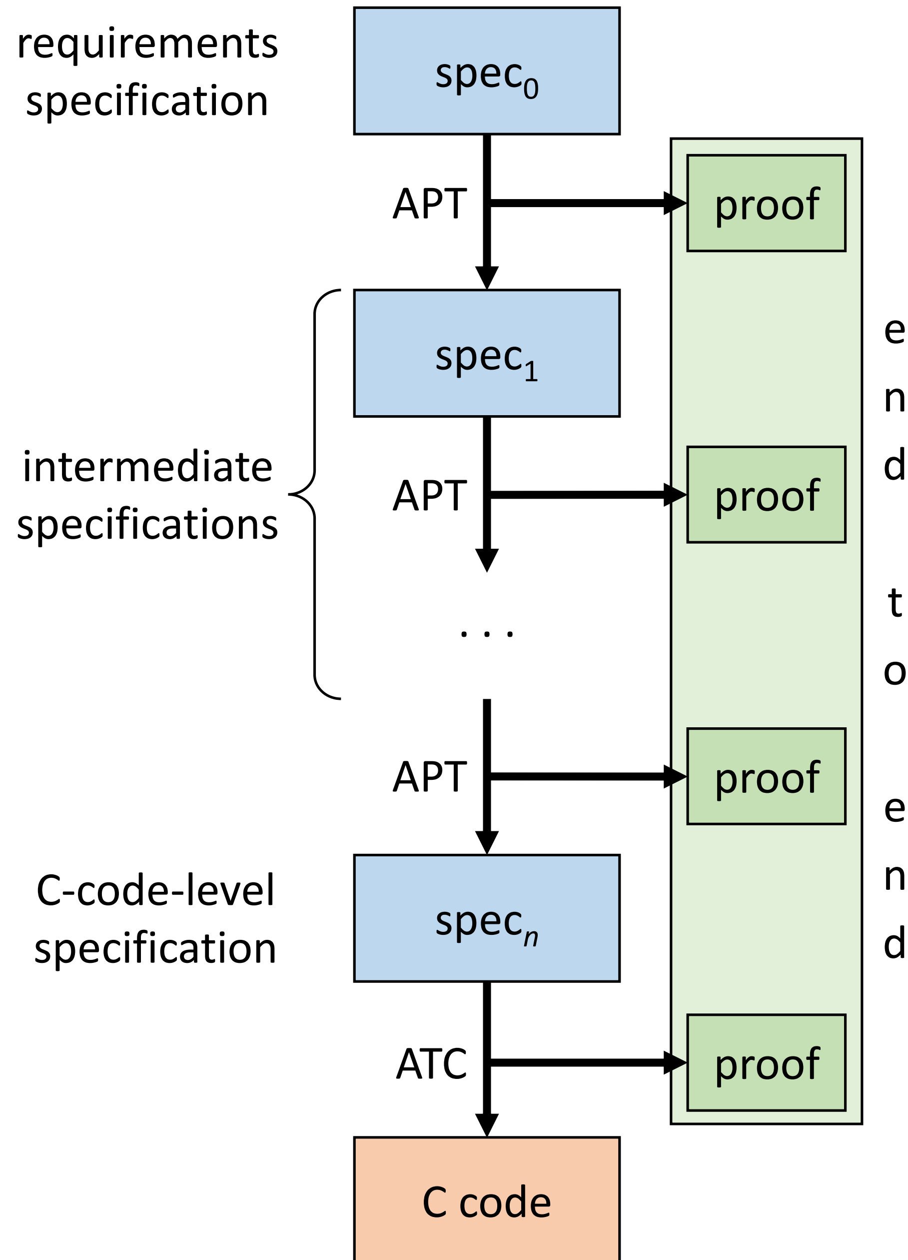
- Next steps: Finish writing specifications and apply APT and ATC to generate C code.

Focus of this talk: Kestrel's Tools for Generating Verified C Code

(See our previous Summit talks for more information about the planned application to TCP/IP.)

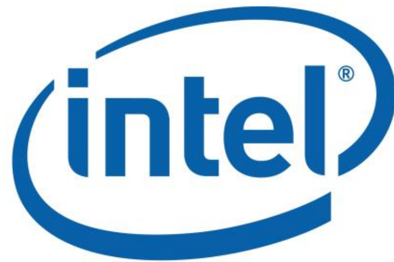
Synthesis of verified C code using APT and ATC, in ACL2:

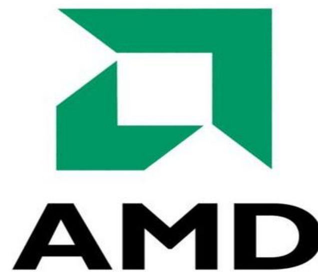
1. Write the high-level requirements specification in the ACL2 language.
2. Use APT transformations to refine the specification stepwise to a low-level form suitable for translation to C.
3. Use the ATC code generator to translate the refined ACL2 specification to C code.
4. Chain the proofs generated by APT and ATC to obtain an end-to-end proof.

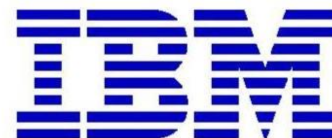


ACL2: Industrial-Strength Prover

- ACL2 is an industrial-strength theorem prover used by many companies.
- We are working with the ACL2 developers.







APT: Kestrel's Proof-Producing Software Transformation Toolkit

Kestrel's APT Toolkit for Program Synthesis

- APT tool suite = "**A**utomated **P**rogram **T**ransformations"
- Kestrel's newest software synthesis system
- Built on the ACL2 theorem prover (1989-present)
- Has roots in Kestrel's previous systems (Specware, KIDS, etc.) and work done at Stanford ca. 2005-2011
- Open source project under active development since 2014 (sponsors include DARPA, AFRL, Army, ONR)
- Partially (eventually all) open-source, at: https://github.com/acl2/acl2/tree/master/books/kestrel/apt

ABCDEFGHIJKLM NOPQRSTUVWXYZ Jump to append Search files

- Top
  - ACL2
  - Books
  - Boolean-reasoning
  - Debugging
  - Documentation
  - Hardware-verification
  - Interfacing-tools
  - Macro-libraries
  - Math
  - Projects
  - Proof-automation
  - Software-verification
  - Std
  - Testing-utilities

ACL2::kestrel-books ACL2::macro-libraries ACL2::projects

Apt

[books]/kestrel/apt/top.lisp

APT Package

APT (Automated Program Transformations) is a library of tools to transform programs and program specifications with automated support.

APT

The APT transformation tools operate on ACL2 artifacts (e.g. functions) and generate corresponding transformed artifacts along with theorems asserting the relationship (e.g. equivalence) between old and new artifacts. The APT transformation tools modify the ACL2 state only by submitting sound and conservative events; they cannot introduce unsoundness or inconsistency on their own.

APT can be used in *program synthesis*, to derive provably correct implementations from formal specifications via sequences of refinement steps carried out via transformations. The specifications may be declarative or executable. The APT transformations can synthesize executable implementations from declarative specifications, as well as optimize executable specifications or implementations. The APT transformations can also be used to generate a variety of diverse implementations of the same specification.

APT can also be used in *program analysis*, to help verify existing programs, suitably embedded in the ACL2 logic, by raising their level of abstraction via transformations that are inverses of the ones used in stepwise program refinement. The formal gap between a program and its specification can be bridged by applying top-down transformations to the specification and bottom-up transformations to the code, until they "meet in the middle".

APT enables the user to focus on the creative parts of the program synthesis or analysis process, leaving the more mechanical parts to the automation provided by the tools. The user guides the process by choosing which transformation to apply at each point and by supplying key theorems (e.g. applicability conditions of transformations), while APT takes care of the lower-level, error-prone details with speed and assurance.

The Community Books currently contain only some APT transformations. More transformations exist in Kestrel Institute's private files, but they will be eventually moved to the Community Books.

Also see the APT Project Web page.

Subtopics

Common-concepts
Concepts that are common to different APT transformations.

Common-options
Options that are common to different APT transformations.

Casesplit
APT case splitting transformation: rephrase a function definition by cases.

Expdata
APT expanded data transformation: change function arguments and results into expanded representations.

Isodata
APT isomorphic data transformation: change function arguments and results into isomorphic representations.

Parteval
APT partial evaluation transformation: specialize a function by setting one or more parameters to specified constant values.

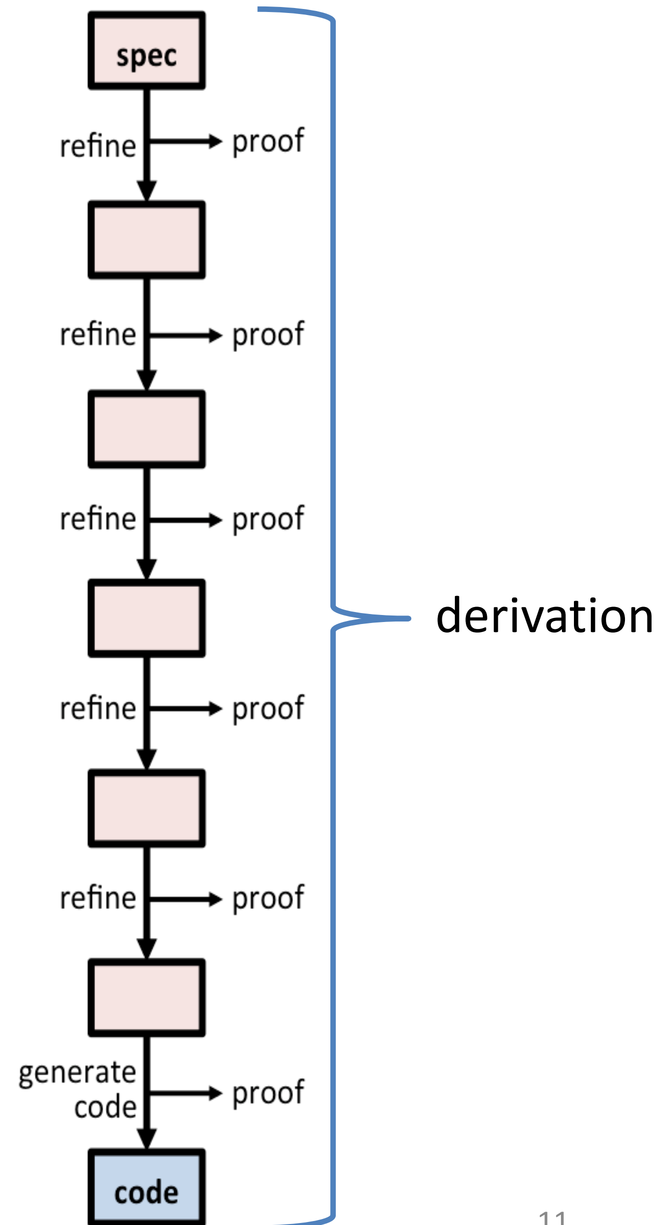
Restrict
APT domain restriction transformation: restrict the effective domain of a function.

Schemalg
APT schematic algorithm transformation: refine a specification by constraining a function to have a certain algorithmic form.

Simplify
Simplify the definition of a given function.

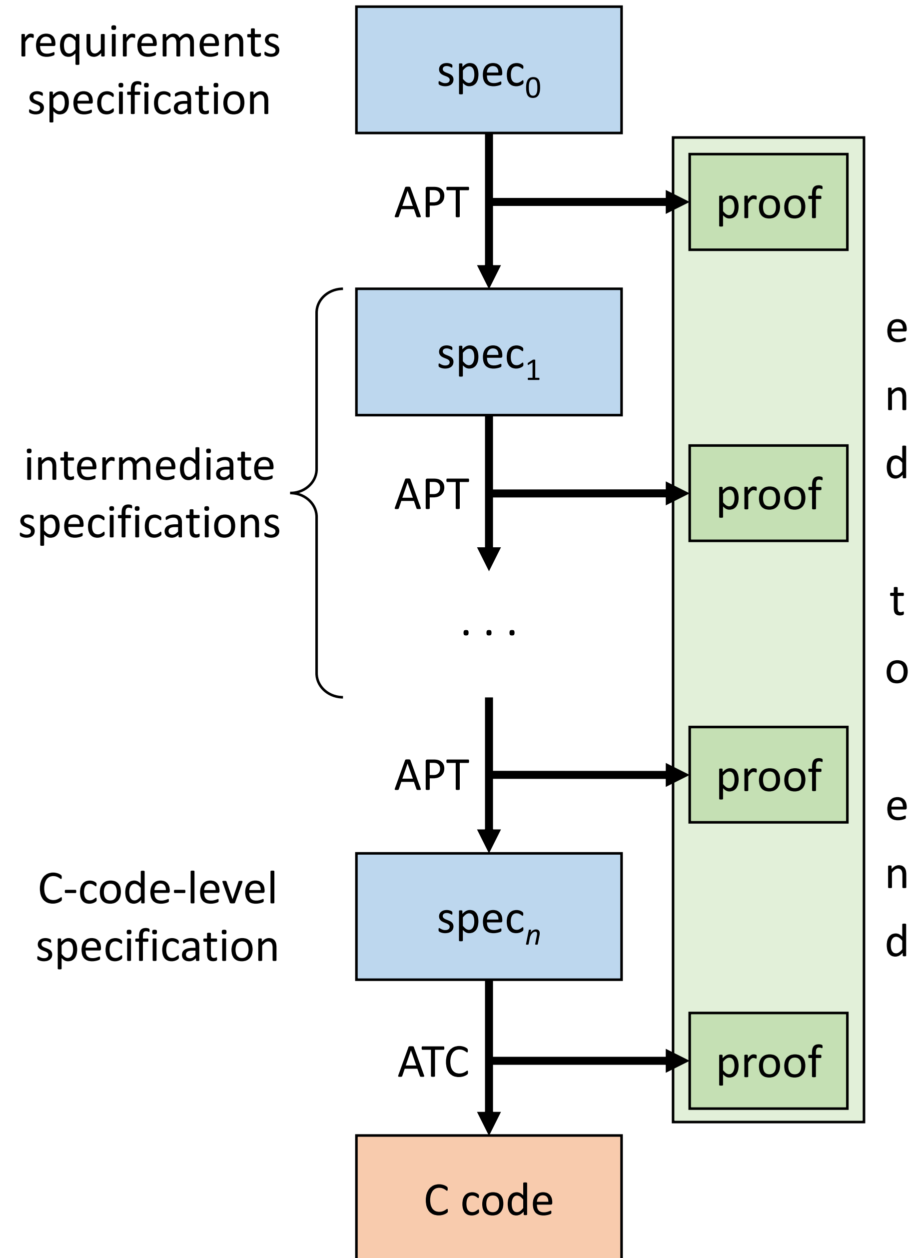
Software Synthesis with the APT toolkit

- Derive code from a specification step-by-step
- Refinements performed by automated transformations
  - Introduce data types / algorithms
  - Introduce optimizations systematically
  - Specialize code
  - Target code to a particular platform
  - Obfuscate code, if desired
- Each step generates a proof
- Chain together all proofs to obtain a proof that the code implements the spec



Synthesis of verified C code using APT and ATC, in ACL2:

1. Write the high-level requirements specification in the ACL2 language.
2. Use APT transformations to refine the specification stepwise to a low-level form suitable for translation to C.
3. Use the ATC code generator to translate the refined ACL2 specification to C code.
4. Chain the proofs generated by APT and ATC to obtain an end-to-end proof.



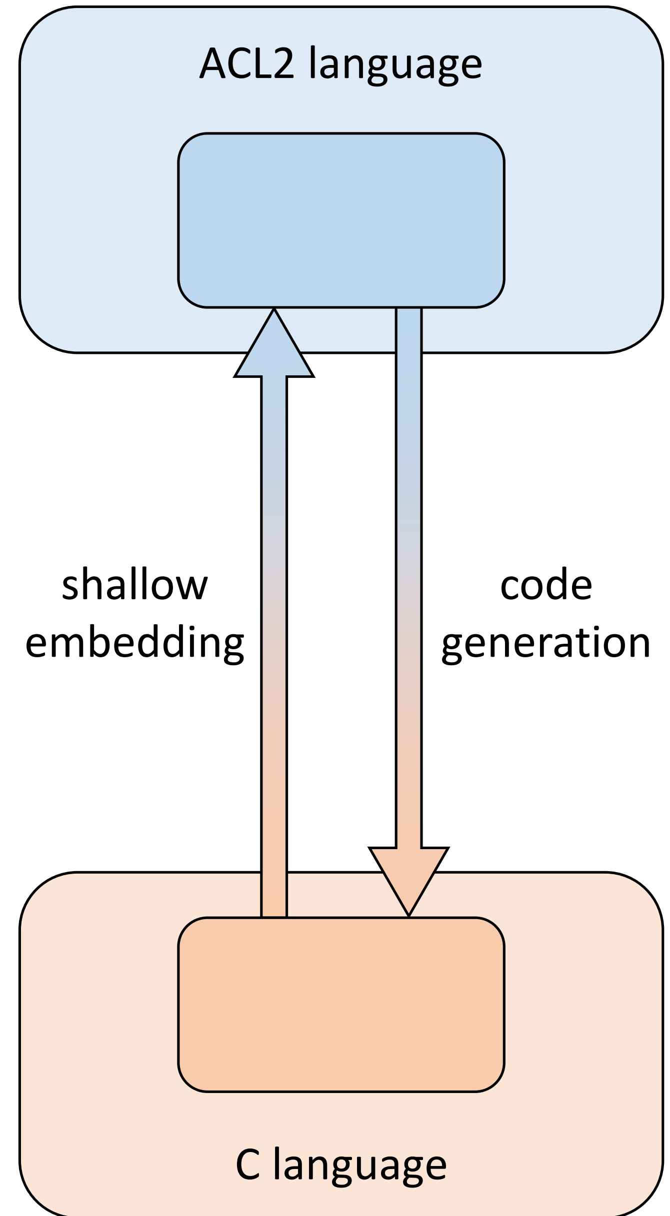
ATC: Kestrel's Proof-Producing C Generator

ATC (= ACL2 To C): C Code Generator for ACL2

- Translates a subset of ACL2 to a subset of C.
- Generates ACL2 theorems asserting the correctness of the generated C code w.r.t. the ACL2 code, based on a formalization of the subset of C in ACL2.
- Is a *verifying*, not *verified*, code generator.
- Implemented in the ACL2 programming language.
- Documented at user and implementation level.
- Open-source, available in the ACL2 Community Books (i.e. library) on GitHub.
- Uses an *inverse shallow embedding* approach.

ATC's inverse shallow embedding approach:

- Define a shallow embedding of a subset of C in ACL2:
  - ACL2 representations of the C constructs in that subset.
  - Looks like "C written in ACL2".
- ATC recognizes the image of the embedding and translates it "back" to the corresponding C code.
- The ACL2-to-C translation is *relatively* simple.
- APT transformations must refine the ACL2 code into something in the image of the embedding.
- The user has full control on the generated C code.



This specification may state the requirements in any form.

requirements specification

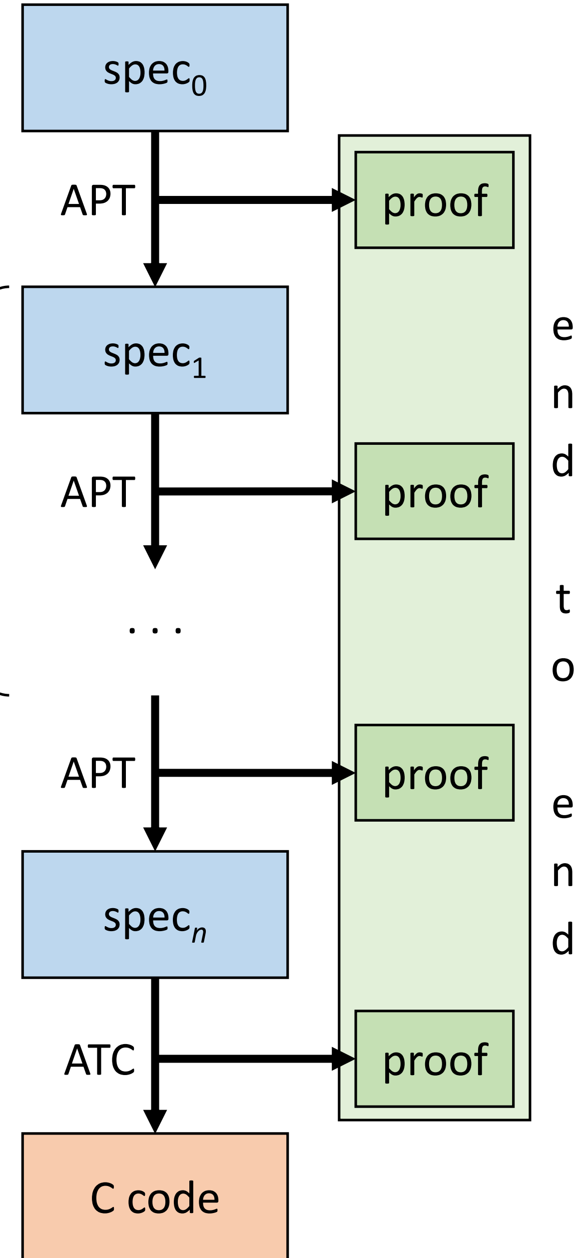
These transformation steps must turn $spec_0$ into a $spec_n$ in the image of the embedding.

intermediate specifications

This specification must be in the image of the embedding.

C-code-level specification

This is the C code represented by the specification $spec_n$.



A Simple Concrete Example of “C Written in ACL2”...

ACL2 function definition

“domain” of the function

ACL2 representation of the *signed int* type

```
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard (and (sintp |x|)
                              (sintp |y|)
                              (sintp |z|)
                              ...))) ; more restrictions
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```

ACL2 representation of *signed int* addition

ACL2 representation of *signed int* multiplication

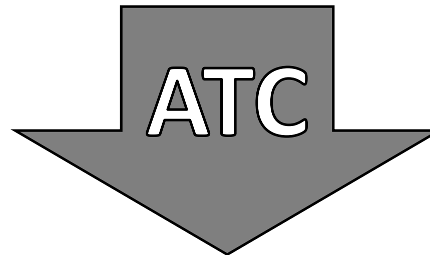
ACL2 representation of *signed int* subtraction

ACL2 representation of the C *signed int* constant 3 in decimal (i.e. 10) base

‘sint’ = signed int

... and Its Translation to C...

```
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard (and (sintp |x|)
                              (sintp |y|)
                              (sintp |z|)
                              ...))) ; more restrictions
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```



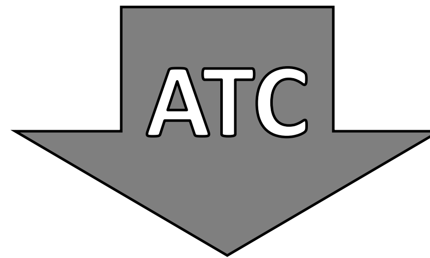
note the clear correspondence

```
int f(int x, int y, int z) {
    return (x + y) * (z - 3);
}
```

... along with its Formal Proof

```
(defun |f| (|x| |y| |z|) ...)
```

(same as before)




```
(defthm |f|-correct
  (implies (and (sintp |x|)
                (sintp |y|)
                (sintp |z|)
                ...)
           (equal (exec-fun (ident "f")
                            (list |x| |y| |z|)
                            ...)
                  (... (|f| |x| |y| |z|)
                       ...))))
```

assuming the guard (domain) of the function...

... if we run the C function...

... we get the same result as the ACL2 function

elisions for focusing on the essentials

Basis for the Formal Proofs: Deep Embedding of C in ACL2

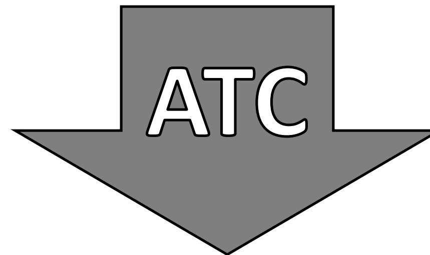
- Formalization of the C subset in ACL2:
  - Abstract syntax.
    - Captures syntax after C preprocessing.
    - Consists of ASTs of algebraic data types.
  - Static semantics.
    - Checking functions on ASTs.
    - Ensures the C code compiles.
  - Dynamic semantics.
    - Big-step interpretive operational semantics.
    - E.g. **`exec-fun`** shown in the theorem in the previous slide.
  - Pretty-printer, used for code generation.
- Overlaps with shallow embedding of the C subset in ACL2 (e.g. model of integer types and operations).

Avoiding Undefined Behavior in the Previously Shown Example

```
(defun |f| (|x| |y| |z|)
  (declare (xargs :guard (and (sintp |x|)
                              (sintp |y|)
                              (sintp |z|)
                              ...))) ; more restrictions
  (mul-sint-sint (add-sint-sint |x| |y|)
                 (sub-sint-sint |z| (sint-dec-const 3))))
```

restrictions ensuring that *signed int* results are always in range

these functions generate proof obligations that require results to be in range



buffer overflows are handled in the same way

```
int f(int x, int y, int z) {
    return (x + y) * (z - 3);
}
```

the generated proofs guarantee that these operations never have undefined behavior

Overcoming the Challenges in Formalizing the Semantics of C

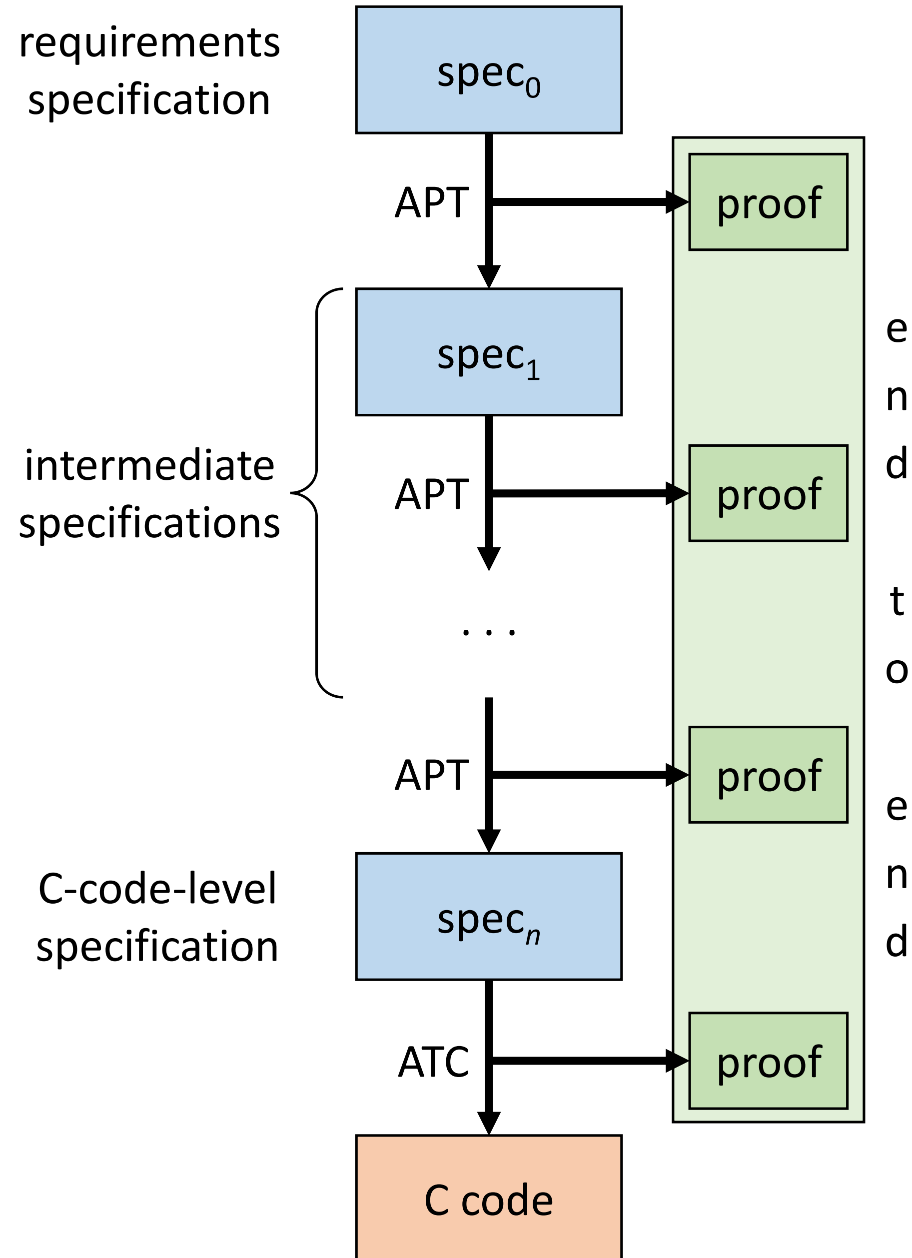
- Undefined behavior according to ISO/IEC C18, including buffer overflows.
  - Generate only code with predictable behavior.
  - Model undefined behavior as yielding a distinguished 'undefined' result, distinct from all the possible well-defined results, e.g. $x + y$ yields 'undefined' if x and y are signed integers and the result does not fit into the type.
  - Generate formal proofs showing that the code never yields the 'undefined' result ➔ no buffer overflows.
- Implementation-defined behavior, which may be actually necessary for some low-level code.
  - Parameterize some aspects of the formal semantics.
  - Instantiate the parameters according to the known implementation.
  - Relativize the proofs to the implementation's assumptions.

Subset of C Supported by ATC

- Standard signed and unsigned integer types.
- Monodimensional "whole" arrays of such types.
- All integer operations and conversions (casts).
- Arrays reads and (destructive) writes.
- Local variable declarations and assignments.
- Conditional statements and expressions.
- Non-strict conditional operations.
- Loops (with **`while`**).
- Functions that side-effect arrays and may return **`void`**.
- More features to come, e.g. **`struct`**s.

Synthesis of verified C code using APT and ATC, in ACL2:

1. Write the high-level requirements specification in the ACL2 language.
2. Use APT transformations to refine the specification stepwise to a low-level form suitable for translation to C.
3. Use the ATC code generator to translate the refined ACL2 specification to C code.
4. Chain the proofs generated by APT and ATC to obtain an end-to-end proof.



Using APT Transformations Before Calling ATC

Generating the input to ATC

ATC's input:
- Must express everything using ATC's C types and operators
- Must obey various additional restrictions

These are deliberate design decisions.

Example input to ATC (array sum):

```
(defun |sum_loop| (|array| |n| |r| |len|)
  (declare ...)
  (if (mbt (equal (c::uint->get |len|)
                  (len (c::uint-array->elements |array|))))
      (if (mbt (c::uintp |r|))
          (if (mbt (and (c::uint-arrayp |array|)
                        (c::uintp |n|)
                        (not
                         (c::boolean-from-sint (c::lt-uint-uint |len| |n|)))))
              (if (c::boolean-from-sint (c::lt-uint-uint |n| |len|))
                  (let*
                      ((|r| (c::assign
                             (c::add-uint-uint
                              |r|
                              (c::uint-array-read-uint |array| |n|))))
                       (|n| (c::assign (c::add-uint-uint (c::uint-dec-const 1)
                                                          |n|))))
                    (|sum_loop| |array| |n| |r| |len|))
                (mv |r| |n|))
            (mv |r| |n|))
        (mv (c::uint-dec-const 0) |n|))
    (mv (c::uint-dec-const 0) |n|)))

(defun |sum| (|array| |len|)
  (declare ...)
  (let ((|n| (c::declar (c::uint-dec-const '0))))
       (let ((|r| (c::declar (c::uint-dec-const '0))))
            (mv-let (|r| |n|)
              (|sum_loop| |array| |n| |r| |len|)
              (declare (ignore |n|))
              |r|))))
```

- You could write the ATC input by hand, and prove properties of it.
- Better to generate it using APT!

Writing a Spec

Common approach:

1. Create an executable ACL2 function using convenient ACL2 types (integers, bit-vectors)
   - Possibly take from a library
   - Possibly derive from a non-executable specification
2. Create a wrapper function that traffics in C types. It should:
   1. Take arguments that are C types.
   2. Immediately convert arguments to the ACL2 types used in the spec.
   3. Call the spec function.
   4. Convert the result back to a C type before returning it.

Example:

```
(defund sum-spec (array len)
  (declare … (c::uint-arrayp array) …
             (equal len (c::uint (c::uint-array-length array))))
  (uint-from-bv (sum-list (integer-list-from-uint-array array))))
```

- “All” that remains is to transform this function into an equivalent function that ATC accepts!

Rewriting to Introduce C operators

Consider an expression like this:

Convert to C type

Convert from C type

```
(uint-from-bv (… lots of computations on bvs … (bv-from-uint x) …)
```

- Goal is to convert all operation to C operations.
- Approach: Use rewriting to "push" the conversions toward each other (the APT `simplify` transformation). Example rewrite rule:

```
(defthm uint-from-bv-of-bvplus-32
  (implies (and (unsigned-byte-p 32 x)
                (unsigned-byte-p 32 y))
           (equal (uint-from-bv (bvplus 32 x y))
                  (c::add-uint-uint (uint-from-bv x)
                                    (uint-from-bv y)))))
```

- **Key concept: Pushing a conversion inside an operation turns it into a C operation.** Do this repeatedly (`simplify` transformation).
- Eventually, the 2 conversions meet and disappear (because they are inverses).

Using isodata to change functions' argument types and return types

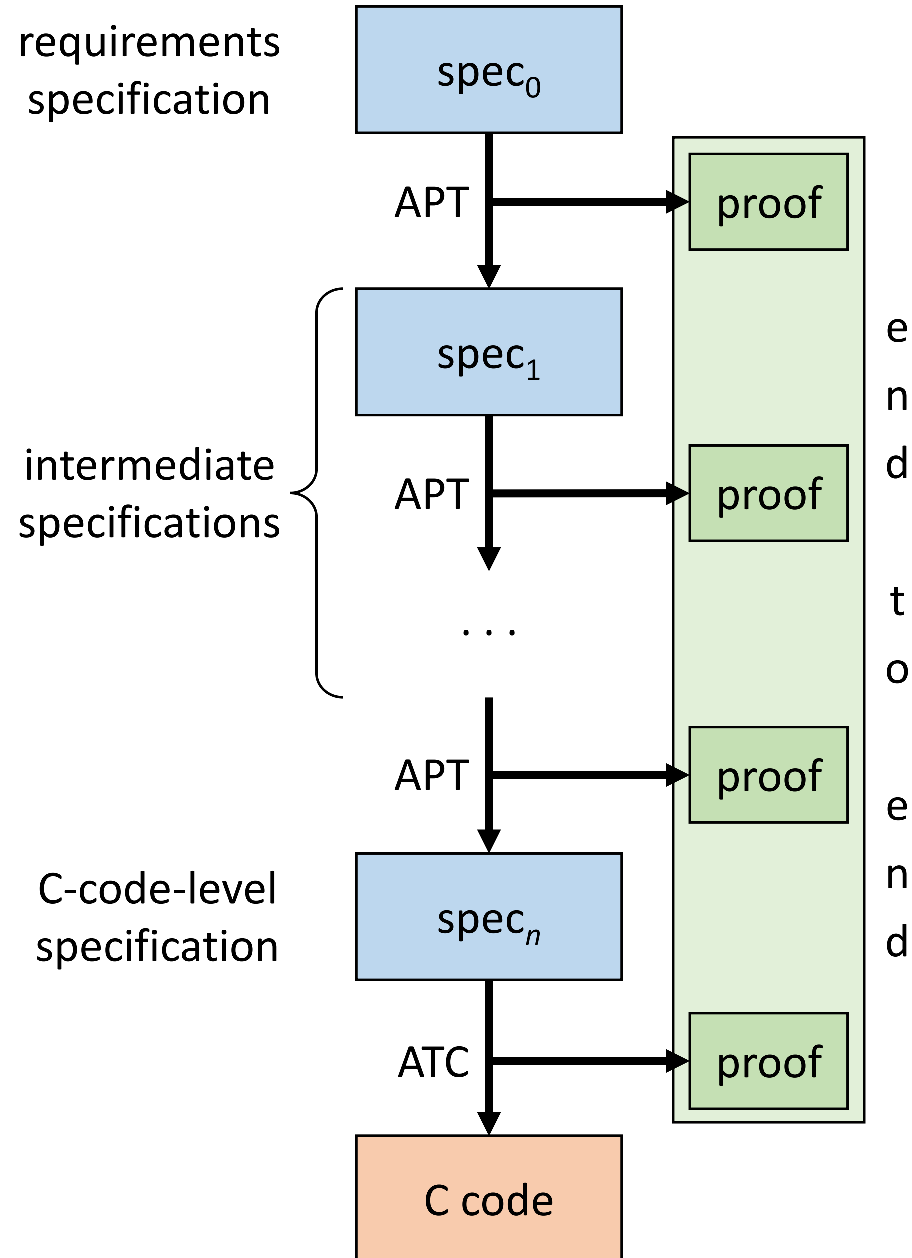
- The preceding approach works to convert expressions to C expressions.
- What about entire functions?
- Apply APT's `isodata` transformation
  - Use a known isomorphism (from our library) between other ACL2 types (integers, BVs) and C types (e.g., `uint-from-bv`, `bv-from-uint`).
  - Given a spec function, `isodata` creates an analogous function that traffics in C types.
  - Roughly speaking, the new function:
    - Immediately converts its arguments to the old types
    - Processes them using the old operations.
    - Converts to the new type before returning or passing values to a recursive call.
  - Now the body is an expression that includes conversions back and forth: So apply the approach from the previous slide!
  - Isodata generates a theorem that replaces calls of the old function with calls of the new function
    - With appropriate conversions (usually get simplified away)
- Repeated calls to `isodata` and `simplify` can transform the entire spec to traffic in C types, as required by ATC.

Other transformations that often precede calls to ATC

`ATC input requirement`	APT transformation to satisfy it
Recursive functions must be tail recursive, so they can become loops.	`tailrec`
Names must be legal C names (function names, param names, variable names).	`rename-for-c`
Only one local variable can be bound in each `let`.	`serialize-lambdas`
Local variables must be annotated to distinguish declarations from subsequent assignments.	`annotate-c-locals`
Array writes cannot be nested.	`remove-nesting`
Arguments passed in a recursive call must be exactly the function's formals (perhaps modified by an overarching `let`).	`let-bind-formals-in-calls`
A loop function must return all values that it changes.	`add-return-values`
A function must have the right `if`-structure for `tailrec`.	`combine-ifs`
Any `mbt` ("must be true") tests must come first and cannot be negated.	`arrange-ifs-for-c-loop`
Values returned by loop function must be properly caught.	`reconstruct-mv-lets`

Synthesis of verified C code using APT and ATC, in ACL2:

1. Write the high-level requirements specification in the ACL2 language.
2. Use APT transformations to refine the specification stepwise to a low-level form suitable for translation to C.
3. Use the ATC code generator to translate the refined ACL2 specification to C code.
4. Chain the proofs generated by APT and ATC to obtain an end-to-end proof.



Next Steps

- Continue improving APT transformations
- Open source remaining APT transformations
  - `simplify`, `tailrec`, and `isodata` are already open source
- Apply APT and ATC to synthesize the network stack

Code:

- https://github.com/acl2/acl2/tree/master/books/kestrel/apt
- https://github.com/acl2/acl2/tree/master/books/kestrel/c/atc

Docs:

- http://acl2.org/manual/index.html?topic=APT____APT
- http://acl2.org/manual/index.html?topic=C____ATC

Conclusion

- Formally verified TCP/IP code for seL4 will support building secure, verified systems.
- The APT synthesis toolkit and the ATC C generator have independent value for generating verified C code.

- Thanks to our sponsors at DARPA and the Army!