

The ART of Agentic Proof-Oriented Programming: Audits, Rubrics, and Templates

NIKHIL SWAMY, Microsoft Research, USA

TAHINA RAMANANANDRO, Microsoft Research, USA

GUIDO MARTÍNEZ, Microsoft Research, USA

GABRIEL EBNER, Microsoft Research, USA

SHUVENDU LAHIRI, Microsoft Research, USA

LEF IOANNIDIS, Microsoft Research, USA

AI-based program synthesis is now widespread. While AI agents churn out large amounts of code from informal natural language descriptions, comprehending this code and assessing its correctness with respect to a user’s intent is a major challenge. Formal proofs of software correctness may help, inasmuch as symbolic tools can check that the code conforms to more abstract specifications, allowing reviews to focus on vetting (hopefully simpler) specifications. With evidence emerging that AI agents can also develop specifications and proofs, having agents “vibe code” in proof-oriented languages may be a way forward. However, without proper guidance, agents use proof idioms and specifications that are complex and messy, and when confronted with reviewing messy specifications, one may not be much better off than trying to understand the code itself.

In this paper, we offer three principles to structure interactions between humans and agents, especially in the context of proof-oriented programming. The central question is how to *audit* the output of an agent: in addition to general software engineering techniques like code review and interoperability testing, we offer audits based on proof engineering techniques, including *small proof-oriented tests*. If the bulk of human effort is spent on audit, finding ways to reduce the burden of auditing is important. Towards that end, we offer *rubrics*, which draw on principles of abstraction and modularity in dependently typed programming to automate parts of the audit process. Finally, we offer *templates*, which provide patterns for agents to follow and generalize when authoring code, specifications, proofs, rubrics, and adjacent artifacts.

We describe our experience using these principles in a variety of case studies, including formalized proofs of textbook algorithms & data structures and a methodology for proofs of stateful interactive systems, with an application to the TLS-1.3 protocol. We hope our work spurs investigation into a more comprehensive theory of agentic proof-oriented programming, and the development of tools and techniques to support it.

1 Introduction

Building software by prompting AI agents in natural language to quickly produce large amounts of code, *vibe coding*, has moved from novelty to mainstream. However, the gain in software authoring velocity threatens to overwhelm our ability to review the resulting code, making such code hard to trust. Encouragingly, agents are now also capable of wielding program proof tools and can produce software in a variety of proof-oriented languages, including Dafny (Poesia et al. 2024), F* (Ioannidis et al. 2026), Lean (Feng et al. 2026; Morrison 2026), Rocq (Paraskevopoulou 2026), Verus (Yang et al. 2025), and others. Vibe coding with agents in proof-oriented languages could ease reviewing load, since reviewers need to only focus on specification review and on elements of the code (e.g., performance) not covered by the specification.

That agents perform so well at programming tasks has been widely attributed to their ability to exploit feedback from partial solutions and iterate until the task is complete. For example, given a test suite, agents will repeatedly revise their code to pass the test suite. Finding ways to structure

Authors’ Contact Information: Nikhil Swamy, Microsoft Research, Redmond, USA, nswamy@microsoft.com; Tahina Ramananandro, Microsoft Research, Redmond, USA, taramana@microsoft.com; Guido Martínez, Microsoft Research, Redmond, USA, mtzguido@gmail.com; Gabriel Ebner, Microsoft Research, Redmond, USA, gabrielebner@microsoft.com; Shuvendu Lahiri, Microsoft Research, Redmond, USA, shuvendu@microsoft.com; Lef Ioannidis, Microsoft Research, Redmond, USA, eioannidis@microsoft.com.

such feedback loops is key to productive agentic programming and proof-oriented languages are well-suited to this paradigm, with agents able to revise solution attempts based on feedback from proof checkers.

When the proof goal is set directly by an expert human, this makes for a compelling proof automation technique (since, usually, any proof of the goal will do). But when the goal (e.g., the top-level specification of a system), or the particular proof technique used (e.g., Hoare-style proof, relational proof, etc.) is left to the agent, humans must be able to judge whether or not what was proven matches what was intended. Without proper guidance, agents use proof idioms and specifications that are complex and messy, and when confronted with reviewing messy specifications, one may not be much better off than trying to understand the code itself.

As a first example, consider below the specification of heapsort from AutoCLRS, an agentic development formalizing a variety of textbook algorithms and data structures in F^* and Pulse. Without getting into too much detail, the specification states that heapsort sorts the first n elements of the integer array a , and that the rest of the array is unchanged.

```

fn heapsort (a: A.array int) (n: SZ.t)
  (#s0: erased (seq int) { SZ.v n ≤ A.length a ∧ length s0 == A.length a ∧ SZ.fits (2 * (length s0) + 2) })
requires A.pts_to a s0
ensures ∃* s cf. A.pts_to a s ** pure (
  length s == length s0 ∧ sorted_upto s (SZ.v n) ∧ permutation s0 s ∧
  (∀ (k:nat). SZ.v n ≤ k ∧ k < length s ⇒ index s k == index s0 k))

```

Judging whether or not this is a good specification is challenging, particularly as only a prefix of the array is sorted and the specification has some surprising preconditions (on $s0$, for example).

1.1 Audits, Rubrics, and Templates

Seeking to structure interactions between humans and agents, and aiming to steer such interactions towards high quality, comprehensible, and trustworthy results, we offer three concepts: *audits*, *rubrics*, and *templates*. While these concepts may be applicable to a variety of automation scenarios with agents, we focus on their relevance to proof-oriented programming.

Audits. Auditing an agent’s output is the central challenge in all forms of agentic interactions, aiming to judge if the agent’s output matches the user’s intent. With intent itself a fuzzy notion, one often employs a range of ad hoc auditing techniques, ranging from code review to testing—these remain important. However, for proof-oriented programs, much of the focus shifts from code audit to specification audit, and for this we offer a few simple techniques, including *small proof-oriented tests* (SPOTs) which check that a specification is precise enough to prove a test’s intended outcome.

For heapsort above, we had an agent generate this SPOT, which provides some confidence that the precondition is not too strong and the postcondition is sufficiently precise, at least on an easy-to-audit concrete input. Note, **assert** statements in F^* and Pulse are checked statically, so this is a proof that heapsort works as expected on this particular input (without running the code).

```

fn spot_heapsort () { let mut a = [[3;2;1]]; heapsort a 3sz; assert (A.pts_to a (of_list [1;2;3])); }

```

Rubrics. If the bulk of human effort is spent on auditing specifications, techniques to reduce the burden of auditing are important. Towards that end, we offer *rubrics*, which (from Merriam-Webster) are “a guide listing specific criteria for grading or scoring academic papers, projects, or tests”. Well-designed rubrics are key to setting strong feedback loops that produce outcomes that are easy to audit, and cover essential properties of a class of problems & solutions (e.g., sorting).

Proposed solutions can be checked mechanically for conformance to a rubric. For example, one could provide the type below as a rubric for functionally correct sorting algorithms on arrays.

```
type array_sort (a:Type) (ord:total_order a) =
  fn (arr:array a) (len:SZ.t) (#s:erased (seq a)) requires arr  $\mapsto$  s ** pure (length arr == SZ.v len)
  ensures  $\exists^* s'. \text{arr} \mapsto s' ** \text{pure} (\text{sorted ord } s' \wedge \text{permutation } s')$ 
```

Auditing this type to judge whether or not it is a good specification for sorting is important, and can be done independently of any particular sorting algorithm. SPOTs help here too: as shown below, one can use a symbolic test to audit `array_sort` on three-element lists for any order, which strikes a balance between judging rubrics abstractly and concretely.

```
fn spot_array_sort (a:Type) (leq:total_order a) (sort:array_sort a leq) (x y z:a { x  $\neq$  y  $\wedge$  y  $\neq$  z  $\wedge$  x  $\neq$  z })
{ let mut arr = [x;y;z]; sort arr 3sz;
  assert ( $\exists^* p \ q \ r. \text{arr} \mapsto [p; q; r] ** \text{pure} (p \text{ `leq` } q \wedge q \text{ `leq` } r \wedge \text{Set.of\_list } [x;y;z] == \text{Set.of\_list } [p;q;r])$ ) }
```

With such a rubric in place, a one-liner is sufficient to judge whether a particular definition is a correct sorting algorithm on integer arrays for a given order, e.g., here are two instances. Of course, specifications are not absolute, even when packaged into generic rubrics. One should still audit the code for other properties, e.g., performance, algorithmic fidelity, etc.

```
val heapsort_instance : array_sort int ( $\leq$ )      val insertionsort_instance : array_sort int ( $\leq$ )
```

Although this is a very simple example, the point is that proof-oriented languages offer many ways to abstract specifications, and employing those abstraction techniques to reduce auditing overhead is important—we will see more sophisticated rubrics later.

Templates. Whereas rubrics provide a way for a human to audit an agent’s output, a template is provided by a human to an agent to hint at the expected solution. For example, coming up with a good rubric requires careful thought and expertise. However, rubrics set for one problem can be reused as *templates* for other problems and agents appear very capable of generalizing from such patterns. For instance, presenting the rubric for sorting as template, agents can produce rubrics for other algorithms, including those with substantially different specification styles.

Templates are part of a rich area of work in the AI community on “in-context learning”, where an agent learns to solve a problem based on examples provided in the prompt (Brown et al. 2020). Automated ways to augment prompts with templates have also been proposed, under the banner of a class of techniques called *retrieval augmentation* (RAG) (Lewis et al. 2020). We do not discuss RAG techniques here, focusing instead on how to interactively craft useful templates with the assistance of agents.

1.2 Contributions

Another slogan for this paper might be *abstractions ease review and promote trust*. Our message is to structure human-agent interactions using well-designed proof-oriented abstractions. Such abstractions have always been useful, and they remain at least as useful when programming with agents. Eventually, we anticipate developing catalogs of rubrics & templates for a variety of problems and proofs styles which can be validated once, and automatically provided to agents (e.g., using RAG techniques) and reused in many contexts to produce easily auditable results. In summary, we make the following contributions:

Audits, Rubrics, and Templates. We introduce audits, rubrics, and templates and show how they can be used to structure interactions between humans and agents in proof-oriented programming, grounded in three new case studies: AutoCLRS, an agentic formalization of textbook algorithms & data structures; an enhancement to KernelBench, verified solutions to GPU kernel programming

benchmarks; and a framework for proofs of state machine refinement for stateful services applied to three protocols, TLS-1.3, TFTP, and YMODEM.

Audits using Small Proof-Oriented Tests (SPOTs). We introduce a new technique for auditing specifications, which we call small proof-oriented tests (SPOTs). SPOTs are small, concrete tests written against a specification, vetting that the specification is precise enough to prove the test’s expected outcome. We describe the use of SPOTs on a variety of specifications, illustrating their ability to discriminate between specifications that are precise and those that are not.

Rubrics for Worst-case Complexity Proofs. By combining ghost state with parametricity, we provide rubrics for complexity-aware specifications of algorithms, including algorithms with functional correctness proofs and data structures proven using functional refinement. As far as we know, this is a novel technique to structure complexity proofs in *shallowly embedded* program logics, although similar ideas have been explored in deeply embedded logics with more sophisticated applications.

Rubrics for State-Machine Refinement Proofs. A large class of systems can be modeled as communicating state machines. Yet, proofs of implementation refinement for such systems employ a variety of styles that can be hard to audit. We present a typeclass for state-machine refinement proofs in separation logic, and demonstrate its use on a small example to serve as a template for larger developments. We also develop and use a library for temporal logic proofs (LTL) of safety and liveness on state machine specifications.

All the code, proofs, and specifications described in this paper were authored by agents, totaling around 450,000 lines of F^* and Pulse, available in the supplement. Our experiments were done with GitHub Copilot CLI with a variety of models, primarily GPT-5.5 and Opus 4.8, using the FStarLang/proof-copilot plugin, which provides some basic information about F^* and Pulse as skills to the agent. However, we explicitly de-emphasize the particular agent framework and model used, as these change rapidly. Instead, we focus on how to structure interactions between humans and agents, especially on understandability of agent outputs, which will remain a central concern even as agents evolve. While almost all of the code, specifications, and proofs in the paper were authored by agents, all the text in the paper was authored by humans, except for §5.5, where an agent wrote a first draft which we then adapted.

2 Background on F^* and Pulse

We will use F^* (Swamy et al. 2016) and Pulse (Swamy et al. 2025) throughout this paper, though we expect our techniques to be applicable to other proof-oriented languages as well. We provide a brief overview of F^* and Pulse here, though we also explain the relevant features and syntax of these languages as we go along.

F^* is a dependently typed, proof-oriented language based on an extensional variant of type theory with a countable hierarchy of predicative universes. Though it emphasizes practical programming over mathematical developments, its expressive power is comparable to other proof assistants, including Lean, Rocq, or Agda. In service of practical proof-oriented programming, F^* offers *refinement* types (for lightweight specification), and *erased* types to model ghost variables commonly used in program proofs. F^* also offers *effects*, including the effect of divergence for Turing-complete programming, a *ghost* effect for erasable computations, and a user-defined effect system for custom effects. Perhaps the most distinctive feature of F^* is its use of SMT solvers to discharge proof obligations, which allows for a degree of automation in proofs, though F^* also has a tactic and metaprogramming system for when SMT solving is insufficient.

Basic syntax. F^* syntax is roughly modeled on OCaml (**val**, **let**, **match**, etc.). Binding occurrences b take the form $x:t$, declaring a variable x at type t ; or $\#x:t$ indicating that the binding is for

an implicit argument. The syntax $\lambda b_1 \dots b_n \rightarrow t$ introduces a lambda abstraction (metavariable t ranges over both types and terms), whereas $b_1 \rightarrow \dots \rightarrow b_n \rightarrow C$ is the shape of a curried function type with *computation type* C (more about them shortly). Refinement types are written $b\{t\}$, e.g., the type $x:\text{int}\{x \geq 0\}$ represents natural numbers. As usual, we omit the type in a binding when it can be inferred; and for non-dependent function types, we omit the variable name. E.g., the type $\#a:\text{Type} \rightarrow \#m:\text{nat} \rightarrow \#n:\text{nat} \rightarrow \text{vec } a \ m \rightarrow \text{vec } a \ n \rightarrow \text{vec } a \ (m + n)$ represents the append function on vectors, where the two explicit arguments and the return type depend on the three implicit arguments marked with ‘#’. We mostly omit implicit binders, except when needed for clarity, treating all unbound variables in types as prenex quantified, writing the type of append as just $\text{vec } a \ m \rightarrow \text{vec } a \ n \rightarrow \text{vec } a \ (m + n)$. We also omit universe annotations.

PULSE is an embedded DSL in F^* , allowing one to write imperative, concurrent programs with mutable shared state, and to prove their functional correctness using a concurrent separation logic called PulseCore (Ebner et al. 2025). Programs in Pulse use a superset of F^* ’s syntax, including custom notation for imperative constructs, e.g., **let mut**, **while**, **break**, etc. Programs verified in Pulse can be extracted to C, Rust, or OCaml code, depending on the constructs used.

3 Formalizing Algorithms & Data Structures with Complexity Bounds

We used agents to formalize textbook algorithms and data structures from *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein (CLRS). The initial prompt was just a single line “pick a collection of algorithms and data structures from CLRS to implement with proofs of functional correctness in F^* and Pulse”, and the agent picked around 50 algorithms and data structures spread across 22 chapters. However, making sense of what was produced and reaching a point where the resulting formalization proves meaningful properties took several weeks of iteration with agents. This experience has informed our thinking, and forced us to develop techniques to facilitate auditing agentic specifications (e.g., SPOTs and rubrics) and also to guide agents effectively using templates.

3.1 SPOTs for Vetting Specifications & Setting Feedback Loops

A recurring challenge in formalizing CLRS is finding a scalable way to review the more than 50 algorithms and data structures and to identify where their specifications are too weak. SPOTs give us a way to probe specification precision at scale. For each verified algorithm, we have an agent write a small, concrete, *verified* test that exercises the API on a tiny instance (e.g., sorting a concrete three-element list) and proves that the test always succeeds. Because the test is concrete and small, its expected outcome is easy to judge; because it must be *proved* to succeed, it doubles as a check of whether the specification is precise enough to support client-side reasoning. Such tests are cheap to review, serve as executable API samples, and can also shake out modeling bugs by actually running the verified code. The observation that provable tests are a useful way to evaluate specifications is not new (Goldweber et al. 2024; Lahiri 2024); what makes it practical at scale here is that agents can author the SPOTs and their proofs, closing a feedback loop that was previously prohibitively expensive to drive by hand.

Applying this methodology across CLRS was revealing. An initial run of agent-authored SPOTs surfaced interesting specification gaps in eight algorithms, while the remaining ones were judged to already have sufficiently precise specifications. The gaps that turned up are exactly the kind that are easy to miss in a manual review. For example, the linear-probing hash table returns a boolean indicating whether an insertion succeeded; its insert had a precise postcondition when the result was true and proved the table unchanged when the result was false, but it never guaranteed *when* false could occur, so the implementation was free to always fail even with empty slots available. A SPOT that inserts a key and then searches for it caught this under-specification immediately.

More dramatically, Prim’s minimum-spanning-tree algorithm carried a postcondition that captured only array shapes and boundary values and no structural minimum-spanning-tree property at all (spanning, acyclicity, minimality); a SPOT on a small three-vertex triangle graph exposed that one simply could not prove the output to be the expected minimal spanning tree.

Crucially, SPOTs are not just a diagnostic but a feedback mechanism that closes the loop for agentic proof-oriented programming. After a SPOT flags a gap, we provide specification critique in natural language and have agents iteratively strengthen the specification and repair the associated proofs. For the hash table, this yielded a revised, straight-line SPOT in which insertion followed by search is guaranteed to find the key; for Prim, the specification was strengthened to capture the necessary MST properties, and the revised SPOT proves that the output tree on the triangle graph contains exactly the expected edges with total weight three.

Indeed, with SPOTs as part of the feedback loop, agents were able to fix the specifications and proofs for all the initially identified eight cases. However, SPOTs are still an underapproximate technique: they prove the preciseness of the specification on several small test cases. One still needs to audit the specification for the general case, and techniques to help with that are needed.

3.2 Rubrics for Sorting Algorithms with Worst-Case Complexity Bounds

CLRS has many sorting algorithms. We had agents write SPOTs for each of their specifications, but were still confronted with judging the correctness of specifications for several related, but subtly different, algorithms. Especially troublesome to review are the complexity bounds: CLRS has a strong focus on asymptotic worst-case time complexity, and AutoCLRS tries to also cover complexity bounds, though the bounds it provides are not asymptotic, e.g., they include the explicit constant factors in the bounds, which adds to the reviewing load. We exploit the use of abstraction features available in F^* and Pulse to structure specifications using typeclasses, and use these classes as rubrics to mechanically audit properties of several algorithms at once.

For example, here below is a specification of insertion sort from AutoCLRS (slightly simplified): we describe it below, also covering the syntax in detail.

```

1 module GR = Pulse.Lib.GhostReference
2 fn insertion_sort (a: array int) (n: SZ.t) (ctr: GR.ref nat) (#s0: erased (Seq.seq int)) (#c0: erased nat)
3 requires a  $\mapsto$  s0 ** ctr  $\mapsto$  c0
4 requires pure (A.length a == SZ.v n)
5 ensures  $\exists^* s$  cf. a  $\mapsto$  s ** ctr  $\mapsto$  cf ** pure (sorted s  $\wedge$  permutation s0 s  $\wedge$  (cf - c0  $\leq$  n*(n-1)/2))

```

At line 1, we define GR as an alias for the module Pulse.Lib.GhostReference. Like other modern separation logics, Pulse supports many forms of ghost state, i.e., state that is computationally irrelevant and is erased before program execution. The GR module allows one to allocate mutable state in a "ghost heap". The rest of the listing defines a signature of the function insertion_sort. The parameters of insertion_sort include an array a and its length n (as before), but now also include a ghost reference ctr, which is used to count operations to be accounted for by the complexity bound. The parameters s0 and c0 are ghost parameters used as logical variables in the specification.

The precondition at line 3 is a separation logic assertion, stating that the array a points to a sequence of integers s0, and that, separately, the ghost reference ctr points to a natural number c0. The notation ** in Pulse stands for separating conjunction. The precondition at line 4 asserts that the length of the array a is equal to n, as we saw before in our array_sort rubric from the Introduction. Multiple preconditions (**requires**) and postconditions (**ensures**) are understood as being conjoined with separating conjunction. Additionally, we write **preserves** for a predicate that is both required and ensured.

```

295 let ticks_t = MR.mref #nat (≤)
296
297 let iorder (#a:Type) (ord:total_order a) (ctr:ticks_t) =
298   fn (x y:a) (#i:erased nat) requires ctr ↦ i returns o:order{o== x `ord.compare` y} ensures ctr ↦ (i + 1)
299
300 type iarray_sort (bound: nat → nat) =
301   fn (a:Type) (arr:A.array a) (len:SZ.t) (ctr:ticks_t) (#ord:erased (total_order a)) (iord:iorder ord ctr) #s #i
302   requires arr ↦ s ** ctr ↦ i ** pure (A.length arr == SZ.v len)
303   ensures ∃* s' j. arr ↦ s' ** ctr ↦ j ** pure (sorted ord s' ∧ permutation s s' ∧ j ≤ i + bound (length s))
304
305
306
307

```

Fig. 1. A typeclass for complexity-aware specifications of sorting algorithms

The postcondition at line 5 states that there exists a sequence s and a natural number cf such that the array a points to s , the counter points to cf , s is a sorted permutation of s_0 , and the difference between cf and c_0 is at most $n \cdot (n-1)/2$, i.e., `insertion_sort` is functionally correct and the ghost counter has been "ticked" at most $n \cdot (n-1)/2$ times.

The style of instrumenting code with "ticks" to count operations and then bounding the number of ticks is a common technique in formalizing complexity bounds, e.g., [Danielsson \(2008\)](#) employs a similar technique in Agda. However, auditing this style of specification for complexity bounds has a significant drawback, since now the code itself is part of the specification: one has to review the code to check that all the operations of interest were accounted for (e.g., every comparison ticks the counter) and, further, that the counter is never decremented. This defeats the main goal of specification review, which is to allow one to focus on the specification and not the code.

Better Rubrics with Abstraction. We can do better by designing a rubric for complexity-aware specifications of algorithms. Our idea is to combine two elements: first, make use of *monotonic* ghost state in Pulse to make sure that the counter is only ever advanced; and, second, to use parametricity to ensure that all operations of interest are properly accounted for. Figure 1 shows a typeclass for complexity-aware specifications of sorting algorithms.

Monotonic Ghost State. The type `ticks_t` is a monotonic ghost reference: using ghost state constructions based on PCMs (partial commutative monoids), the Pulse libraries provide the type `MR.mref`, such that for any preorder p on a type a , `MR.mref #a p` is the type of a mutable ghost heap cell whose contents are guaranteed to only evolve according to the preorder p . So, `MR.mref #nat (≤)` is a mutable ghost reference that can never be decremented. Using `ticks_t` for our operation counter solves one of the problems: we no longer have to review the code to make sure that the counter is always advanced.

Instrumenting Comparisons with Ticks. The next problem is to ensure that every operation of interest is properly accounted for. The main idea of our construction is to abstract over all the operations that one wants to account for, providing instrumented versions that advance the tick counter appropriately. For sorting algorithms, we typically want to count comparisons. So, we define an `iorder`, the type of a Pulse function that compares x and y according to the pure total order `ord`, but also increments a tick counter `ctr` from i to $i+1$.

Tying it Together with Parametricity. Finally, we define `iarray_sort bound`, the type of a sorting algorithm on arrays whose worst-case complexity bound (on comparisons) is described by `bound`. The type is *parametric* in the type a of array elements. In contrast to the `array_sort` type from the Introduction, where instances could be specialized to particular types and orders (e.g., `array_sort int (≤)`),

this is the type of a sorting function that is polymorphic in the array type a and the order ord . Importantly, the ord has type erased ($total_order\ a$), meaning that it can be used in specifications and ghost code, but not for concrete comparisons. For that, the signature provides $iord:iorder\ ord\ ctr$, an instrumented version of ord associated specifically with the tick counter ctr . As far as we are aware, the use of parametricity with ghost state for operation counting is novel, and enables complexity proofs for shallowly embedded program logics.

The rest of the specification should be relatively familiar: we require ownership of the array and the counter and that the length is correct; the postcondition ensures that the array is a sorted permutation of the input, and that the tick counter has been advanced by at most the bound (a function of the length of the input). Of course, one should audit the rubric to make sure it captures the intended behavior. For this, we can use the same SPOTs as shown before, perhaps augmented with assertions about the complexity bound.

Using the Rubric. This type makes for a much better rubric for complexity-aware sorting. By parametricity, we are assured that every comparison operation is accounted for, and, by monotonicity, that the counter is never decremented. Here, below, are instances for four sorting algorithms, each audited at a glance as an instance of our typeclass, ensuring functional correctness with the expected complexity bound. The bounds on `heap_sort` and `merge_sort` are messier than one would like (e.g., one could remove the branching on $n > 0$), but these are the bounds produced by an agent and we show them as is. Note also that one can easily derive instances of `array_sort` from `iarray_sort`, so the instrumentation and genericity do not come at the cost of usability.

```

val insertion_sort : iarray_sort ( $\lambda n \rightarrow n * (n - 1) / 2$ )
val quicksort : iarray_sort ( $\lambda n \rightarrow n * (n - 1) / 2$ )
val merge_sort : iarray_sort ( $\lambda n \rightarrow$  if  $n > 0$  then  $4 * n * \log2\_ceil\ n + 4 * n$  else  $0$ )
val heap_sort : iarray_sort ( $\lambda n \rightarrow$  if  $n > 0$  then  $(n/2) * (2 * \log2\_floor\ n) + (n-1) * (2 * \log2\_floor\ n)$  else  $0$ )

```

Auditing Implementations. Our point is that by reusing insights from programming language theory, one can design rubrics that impose stronger constraints on agents, setting better feedback loops, and produce outputs that are easier to audit. But rubrics do not replace audits altogether, e.g., this rubric says nothing about space complexity, recursion depth, etc. Besides, simply knowing that, say, `insertion_sort` has the type of a correct, quadratic-time sorting algorithm doesn't tell us that it is actually an insertion sort. Notice that `quicksort` has exactly the same type, so the rubric alone does not have the power to distinguish the two. For this, we need to audit the implementation itself, looking at the code for hallmarks of each algorithm—for this, one may prefer to look at monomorphic, extracted C code. For algorithmic formalizations, this is particularly important, as we have observed agents are cheating in this regard, e.g., producing an implementation of insertion sort and claiming that it is quicksort.

3.3 Rubrics as Templates: Abstracting Memory Operations for Complexity Bounds

We have seen how to use parametricity to abstract over comparisons in sorting algorithms. Using the rubric for sorting, we had agents generate rubrics for several other algorithm families in the same style, e.g., search data structures proven using functional refinement. But, not all complexity arguments involve counting just the pure operations like comparison. For example, linked list implementations are fully parametric in the payload type, and a complexity analysis usually counts the number of memory operations (e.g., reads, writes, and allocations). Applying our methodology to this setting requires libraries that abstract over instrumented versions of the memory operations.

Working with an agent, and using our previous rubrics as templates, we developed a type-class for complexity-aware operations on heap references. The Pulse libraries provide a type

Pulse.Lib.Box.box a, the type of heap references, and the typeclass below abstracts it, for an arbitrary (strictly positive) type constructor box,¹ instrumenting each of its operations with complexity tracking. This time, instead of using a tick counter, we asked the agent to use *time credits* (Pottier et al. 2024) based on Pulse libraries that provide such abstractions. We show read and write, but the class also includes alloc and free, which are similar.

```
class box_class (cr:credits_t) (box:([@@strictly_positive] _:Type → Type)) = {
  pts_to: #a:Type → box a → a → slprop;

  read: fn (#a:Type) (b:box a) (#v:erased a)
    requires pts_to b v ** credits cr read_cost returns u:a ensures pts_to b v ** pure (u == reveal v);

  write: fn (#a:Type) (b:box a) (u:a) (#v:erased a) (#i:erased nat)
    requires pts_to b v ** credits cr write_cost ensures pts_to b u; ...}
```

Using this class, one can define rubrics for data structures where complexity bounds count memory operations. For example, here is a (partial) rubric for linked-list-like structures, a record with several fields parameterized by an element type, a representation type (with just list as the model), a predicate owns relating the two, and complexity bounds for each operation.

```
type llist_structure (cr:credits_t) (box:(Type → Type)) (boxes:box_class cr box) (a:Type) (repr:Type)
  (owns: repr → list a → slprop) (insert_head_bound append_tail_bound ...: nat → nat) = { ... }
```

The fields include, e.g., insert_head and append_tail, which are specified in terms of their actions on the list model and with complexity accounted for by the time credit. For example, here is the specification of insert_head and append_tail:

```
insert_head: fn (xs:repr) (x:a) (#l:erased (list a)) (#i:erased nat)
  requires owns xs | ** credits cr (insert_head_bound (length l)) returns xs':repr ensures owns xs' (x :: l)

append_tail: fn (xs:repr) (x:a) (#l:erased (list a)) (#i:erased nat)
  requires owns xs | ** credits cr (append_tail_bound (length l)) returns xs':repr ensures owns xs' (snoc l x)
```

With this rubric in place, instances of linked-list-like structures are shown below, establishing that both are functionally correct, that slist has a linear time append_tail, while for dlist append_tail is constant time. The distinction is characteristic of singly-linked and doubly-linked lists, where the latter typically also maintains a pointer to the tail of the list. Of course, one could also have a singly-linked list with a tail pointer to achieve the same complexity bound, so, as usual one has to audit the implementation for hallmarks of singly-linked or doubly-linked lists.

```
instance slist cr box (bc:box_class cr box) a : llist_structure cr box a (sll box a) owns bound_hd bound_tl
  where bound_hd _ = alloc_cost and bound_tl n = 2 * n + BC.alloc_cost
```

```
instance dlist cr box (bc:box_class cr box) a : llist_structure cr box a (dll box a) owns bound_bound
  where bound _ = alloc_cost + read_cost + write_cost
```

The point of this is to show that abstraction over instrumented operations can be quite general, and can even abstract over the memory operations. Supporting this requires a specification language with support for abstraction over type constructors & monotonic ghost state: but modern dependently typed languages, with shallowly embedded DSLs like Pulse, have all these features and more. Exploiting them to better structure audits, rubrics, and templates is a profitable way to proceed with agents in proof-oriented programming. Additionally, whereas humans may balk at

¹We omit positivity annotations from here on, for simplicity.

writing proofs against specifications that are parametric in even the heap operations, agents appear to handle them quite well.

4 Abstraction in Library Design for Agentic GPU Programming in Kuiper

We now turn our attention to a different domain: GPU kernel programming in Kuiper (Martínez et al. 2026). Kuiper is a framework for verified GPU programming, embedded as an extensible library in F^{*} and Pulse. Kuiper models the heterogeneous programming model of GPUs within separation logic, including the separate device and host memories, kernel launches and synchronization, and more advanced features like tensor cores. Recently, Martínez and Sorensen (2026) report using coding agents to build *verified* solutions in Kuiper to the first 100 challenges in the KernelBench suite, a set of benchmarks intended to gauge GPU-programming proficiency of LLMs. Agents were used both to state the specifications for each challenge and to implement the kernels. Throughout the process, the KernelBench test suite was used to gauge plausibility before attempting to verify anything (and before requesting any human feedback), illustrating the continued importance of testing in the auditing process. However, Martínez and Sorensen report that reviewing KernelBench specifications was a challenge, so we revisit their work through the lens of our ART framework.

Whereas in the previous section we argued for the importance of abstraction in proof-oriented language design in support of setting good rubrics, here we illustrate that abstraction in *library design* is important too.

4.1 A Rubric for GEMMs

Generalized matrix multiplication (GEMM) is a fundamental operation in many scientific computing and machine learning applications. Given matrices A , B and C of proper shapes, and real numbers α and β , a GEMM computes $C \leftarrow \alpha AB + \beta C$. Computing the product of A and B is the most interesting part of a GEMM implementation, so we will mostly focus on that here, writing our specs as $C \leftarrow AB$. The generalization to an actual GEMM is immediate.

Kuiper has several implementations of GEMMs, with subtle differences between them. The most basic versions spawn separate threads for each output cell, and compute the dot product of the corresponding row of A and column of B in the naive form. However, as their logic becomes more complicated, GEMMs may have extra preconditions on the input. For instance, a common technique to reduce the amount of memory transfers is to *tile* the matrices, and have each block of threads compute tiles of the output. Here, it must be the case that the dimensions of the matrices are multiples of the tile size.²

To capture this variety, we define a generic `gemm` type for algorithms parametric in the type `et` of matrix elements and their *in-memory representation* `repA`, `repB`, `repC` (e.g., row-major or column-major). The type is indexed by a *size requirement predicate* which is a precondition on the matrix dimensions. This allows different algorithms to vary such preconditions while sharing the same overall type. The style of proof used here is similar to the functional refinement style described in the previous section. We have an in-memory representation (`array2`) and a pure model (`ematrix`), and we prove that the in-memory representation is a refinement of the pure model. The type states that the output array `c` is a refinement of the matrix product of the pure models of the input arrays `a` and `b`. Kuiper also includes a notion of approximation for floating-point numbers, but we do not use it here to keep the presentation simpler.

```
type gemm (size_req : size_req_t) =
  fn (#et : Type0) { scalar et } (repA repB repC : trepr2) (m n k : szp)
    (a : array2 et (repA m k)) (b : array2 et (repB k n)) (c : array2 et (repC m n))
```

²Avoiding this restriction is possible, but requires extra logic that hurts performance.

```

491   (#eA #eB #eC : ematrix et _ _) (#fA #fB : perm)
492   preserves a ↦ Frac fA eA ** b ↦ Frac fB eB
493   requires pure (size_req m n k) ** c ↦ eC
494   ensures c ↦ MS.matmul eA eB

```

Kuiper had several existing implementations of GEMMs, which we could easily coalesce into this rubric, ensuring they all satisfy the same functional correctness specification, albeit with slightly different size requirements. We present only a few here. The last two are tiled GEMMs, and require divisibility (‘/?’) of the matrix dimensions by the tile size, as well as not exceeding the maximum number of blocks and threads allowed by the GPU.

```

501 val gemm_naive0 : gemm (λ m n k → m * n ≤ max_blocks)
502 val gemm_naive : gemm (λ m n k → m * n ≤ max_blocks * max_threads)
503 val gemm_tiled tile : gemm (λ m n k → tile /? m ∧ tile /? n ∧ tile /? k ∧ (m/tile) * (n/tile) ≤ max_blocks)
504 val gemm_shm tile : gemm (λ m n k → tile /? m ∧ tile /? n ∧ tile /? k ∧ (m/tile) * (n/tile) ≤ max_blocks)

```

Auditing the Rubric. How can we know that these implementations are correct? While they clearly have the gemm type, one would need to audit the definitions of the array2 type, the ematrix type, the MS.matmul function, and the entire virtual array stack to make sure it is really meaningful. A skeptical reader could indeed mistrust it! To bring some grounding, we turn again to SPOTs: we aim to prove that any element of this type is really a GEMM, statically, on some small representative examples. We do so as follows: we first define two simple 2×2 matrices and their product, which we can show matches the result of calling MS.matmul:

```

514 let mA : ematrix int 2 2 = mk2 (function 0,0 → 1 | 0,1 → 2 | 1,0 → 3 | 1,1 → 4)
515 let mB : ematrix int 2 2 = mk2 (function 0,0 → 5 | 0,1 → 6 | 1,0 → 7 | 1,1 → 8)
516 let mAB : ematrix int 2 2 = mk2 (function 0,0 → 19 | 0,1 → 22 | 1,0 → 43 | 1,1 → 50)
517 let mAB_matches_matmul : Lemma (mAB == MS.matmul mA mB) = ...

```

Then, for a given $g : \text{gemm}$, we call it on flat arrays containing the data of mA and mB in row-major order and prove, statically, that the output is the row-major representation of mAB. This can be done by *abstracting* a flat array as an array2 with any given layout, operating on it, and *concretizing* it back down. Abstraction and concretization are both ghost operations; nothing happens at runtime.

```

523 fn spot_gemm22 (#size_req : size_req_t) (g : gemm size_req) (a b c : larray u32 4)
524   preserves a ↦ seq![1; 2; 3; 4] ** b ↦ seq![5; 6; 7; 8]
525   requires c ↦ _ ** pure (size_req 2 2 2)
526   ensures c ↦ seq![19; 22; 43; 50]
527 {
528   let mA : array2 u32 (row_major 2 2) = array2_abstract a;
529   let mB : array2 u32 (row_major 2 2) = array2_abstract b;
530   let mC : array2 u32 (row_major 2 2) = array2_abstract c;
531   g ___ 2 2 2 mA mB mC;
532   array2_concr mA; array2_concr mB; array2_concr mC;
533   with s'. assert c ↦ s';
534   assert pure (s' == seq![19; 22; 43; 50]); }

```

Having reached this point, the only non-ghost line of code is the call to g , and we have statically proven it fills c with the correct row-major representation of mAB. The large stack of definitions in support of virtual arrays, views, layouts, matrices, etc., need not be trusted at all for this fact.

Using this new, audited rubric, we’ve used coding agents to implement several more variants of GEMMs (mostly by reordering how memory accesses occur, without using more complex features) and obtain immediate confidence that they are correct, and that their preconditions are sensible.

4.2 Enabling Audits: Picking the Right Abstractions

An interesting situation occurred when the coding agents were asked to generalize matrix multiplication into a “batched” version, that is, where the inputs and outputs are arrays of matrices, and the effect is to fill the output with the pointwise matrix multiplication of the inputs. Intuitively, the inputs and outputs are *three*-dimensional arrays, where the first dimension is the batch, and the other two are those of each matrix in the batch.

However, Kuiper had no notion of a three-dimensional array, and hence the existing GEMMs could not be used directly, nor was there a three-dimensional mathematical matrix to be used in the specification. So Martínez and Sorensen’s agents wrote a version working over flat arrays (of length $B \times M \times K$, $B \times K \times N$ and $B \times M \times N$, where B is the batch dimension). That agents use whatever is available to solve a task is remarkable, but without the right abstractions the result is hard to make sense of, e.g., this is the pure specification for batched matrix multiplication:

```
#push-options "-z3rlimit_120"
let batched_gemm_result (batch rows shared cols : pos) (va vb : seq f32)
  : Pure (seq f32) (requires Seq.length va == batch * rows * shared ∧ Seq.length vb == batch * shared * cols)
  = Seq.init (batch * rows * cols) (λ tid →
    let rc = rows * cols in let bi = tid / rc in
    let ri = (tid % rc) / cols in let ci = tid % cols in
    batched_dot_up_to va vb (bi * rows * shared + ri * shared) (bi * shared * cols) ci cols shared)
```

where `batched_dot_up_to` is a function that computes the relevant dot product of elements of `va` and `vb`.

The code happened to be right, but this specification is very far from pleasant for a human to review due to its involved indexing and lack of reuse of existing concepts. The increase of the `z3rlimit` is also a sign that all the indexing arithmetic in the specification makes it difficult to even typecheck. Furthermore, this definition also fixes batched-row-major layout for all three arrays, something that is only evident by interpreting the indexing in the specification. Clearly, this is a terrible way of doing things. This is not a specification that a human should sign off on, nor be maintained in a codebase.

The right thing to do here is to generalize the matrix abstraction in Kuiper to arbitrary-dimensional tensors, of which “batched matrices” are a special case. This is a conceptually straightforward, but technically non-trivial change: it requires defining tensor *shapes* as an inductive type, defining tensors (and their abstract view) parametrized by them, working with dependently-typed index types, and making sure this all verifies robustly and extracts to efficient CUDA code. While agents were useful to automate some leaf-level proofs, the bulk of this extension was human-driven library design and implementation.

Once finished, we provided the new tensor library abstraction to the agents. This allowed them to write programs using a new `array3` type (a special case of a tensor), while also remaining layout-polymorphic. The specification-level counterparts for an `array3`, i.e., `ematrix3`, are layout-independent pure values. So, the specification for a batched matrix multiplication can be written much more succinctly as:

```
let batched_matmul (#et:Type) {l scalar et l} #batch #rows #shared #cols
  (a : ematrix3 et batch rows shared) (b : ematrix3 et batch shared cols) : ematrix3 et batch rows cols =
  ematrix3_mk λ i j k → macc (MS.matmul (ematrix3_page a i) (ematrix3_page b i)) j k
```

where m_{ij} is the contents of the i -th row and j -th column of the matrix m , and $ematrix3_page\ i$ is the i -th submatrix of the three-dimensional array a (i.e., it fixes the first dimension to i and returns a two-dimensional array). This specification 1) is much more readable; 2) is easy for an expert to judge as correct because it clearly states that the i -th submatrix of the output is the product of the i -th submatrices of the inputs; 3) is layout-independent, and 4) reuses the existing `MS.matmul` definition, reducing duplication and enabling reuse of the theorems proved for it. The code is easily updated to this new specification (it in fact becomes simpler, as the indexing is now via abstract batch-row-column tuples instead of flat integers) and trivially layout-polymorphic. Verification time is also reduced.

Agents are often happy to grind at the goal in question without regard for how it interoperates with the rest of the development, or whether the solution is “elegant”. Providing them with templates that encourage the use of abstraction produces results that are easier to understand. One may imagine that future agents might invent abstractions and use them, but it will remain important for humans to understand those abstractions and how they fit into the overall design of the system.

4.3 Auditing and Performance

Audits remain important in this setting too. Kuiper specifications do not, at the moment, capture non-functional aspects related to performance, the features used by the kernels (e.g., are tensor cores used?) or even the number of kernels launched by a function. We can foresee adding support for all this to Kuiper, but auditing the source code for these properties is relatively straightforward. Further, the performance characteristics of GPUs are extremely complicated and poorly understood, and it seems unlikely one can do both tractable and precise performance reasoning at all.

In the case of Kuiper, audits can also be performed on the generated CUDA code instead of the (significantly more verbose) Pulse source. After a module verifies and extracts, it produces first-order CUDA code without any proof annotations. Some non-functional properties immediately stand out at this stage: Does the kernel use shared memory? How many nested loops are there? Where is synchronization used? How many kernels are launched? Despite the high polymorphism, these properties can be ascertained by inspecting a single instance of generated CUDA.

5 Rubrics and Templates for Stateful Services

In contrast to textbook algorithms or GPU kernels, many real-world applications are stateful services: long-running processes that often maintain some internal state (either in memory or persisted, say, to a database), and where multiple services may interact with each other over a network. How is one to build such services in a proof-oriented way, and how can agents help? In this section, we explore one way to do this, using state transition systems as an abstract, mathematical model for such services and a technique called *state-machine refinement* to build verified implementations.

We start small, developing a very simple *calculator service*, which still has many of the properties we want for such services in general—it receives requests from clients, parses those requests, maintains some internal state, processes requests, and responds by serializing a message back to the client. By starting small, we aim to develop well-understood rubrics and templates which we can then use to develop more complex services.

5.1 A Calculator Service

Consider a simple calculator service whose internal state is a stack of integers, and which processes messages of the form `request` and produces `response` values:

```
type request = | Push of int | Peek | Add | Sub | Mul | Div
type response = | Ok | Result of int | Error
```

where `Ok` indicates that the request was processed successfully; `Result n` is the response to a `Peek` request; and `Error` indicates that the request could not be processed (e.g., an `Add` request when the stack is empty, a division by zero, or when the stack exceeds a certain size). A pure specification of the behavior of this service is given by a simple state transition function, of which we show just a few cases:

```

type calc_stack = list int
let max_stack_size = 10
let step (s:calc_stack) (req:request) : calc_stack & response = match req with
| Push x → if L.length s ≥ max_stack_size then (s, Error) else (x :: s, Ok)
| Peek → (match s with [] → (s, Error) | x :: _ → (s, Result x)) ...

```

Hopefully, this specification is simple enough to understand and audit by a human. We also need a way to parse requests and serialize responses, so that the service can interact with clients over a network. We define serialization and parsing functions `parse_request` and `serialize_response`, defining a wire format for our service (a 5-byte format with a one-byte tag and a four-byte payload).

In principle, we now have all we need: these purely functional specifications are executable, and one could imagine standing up a functional service by extracting this code to OCaml, reading bytes off the network, parsing, transitioning the state, and serializing the response in a loop. While this would work for a simple calculator service, for more complex protocols this is not always what one wants. For example, one would often prefer a more efficient implementation in a low-level language like C or Rust, without a garbage collector. Or the state transition might not be executable and may be specified as a relation, leaving freedom in how the transition is implemented. What we'd like instead is to implement the service in a proof-oriented language like PULSE, with low-level control over memory and other resources, and to prove that the implementation *refines* the abstract state machine specification.

5.2 A Pulse Implementation

You can ask an agent to implement the calculator service in PULSE, proving that it refines the abstract state machine specification. Here is the signature of one solution it produced: we don't expect you to read it in detail; suffice to know that it is the type of a single request handler, receiving the request in an input buffer `req_buf` and writing its response in `resp_buf`, while transitioning the server state `srv`.

```

fn process_request (srv: server_state) (req_buf resp_buf: Vec.vec U8.t) (#log0: erased calc_log)
requires server_exactly srv log0 ** req_buf ↦ 'req_bytes ** resp_buf ↦ 'resp_bytes **
  pure (length 'req_bytes == 5 ∧ length 'resp_bytes == 5 ∧ parse_request 'req_bytes ≠ None)
ensures ∃* resp_bytes1 log1.
  server_exactly srv log1 ** req_buf ↦ 'req_bytes ** resp_buf ↦ resp_bytes1 **
  pure (log_single_step log0 log1 ∧
    log1.input_bytes `Seq.equal` append log0.input_bytes 'req_bytes ∧
    log1.output_bytes `Seq.equal` append log0.output_bytes resp_bytes1 ∧
    calc_frame_network_step_ok log0 'req_bytes log1 resp_bytes1)

```

Is this a correct implementation of the calculator service? It does compile to C and correctly runs a suite of network tests, but how can one be sure? One has to read the specification carefully, and check that the preconditions and postconditions are what one expects. For an arbitrary, agent-generated specification, this is a non-trivial task. Can we have a more structured way to ensure that the implementation is correct, without needing to read the entire specification and implementation? We can, by setting a better, generic rubric.

5.3 State Machine Refinement

We would like a way to specify that a PULSE implementation of a service refines an abstract state machine. There are many ways to do this, but we want to settle on a generic, reusable structure that can be reviewed once by a human, and then used as a rubric for agents to implement other services in a similar way. It will take us a few steps to get there, so a roadmap: (1) we first define a class for an abstract state machine, with typed inputs, outputs, and a transition relation; (2) then, we define a notion of a wire format, with parsers and serializers, and compose it with our abstract state machine to get a *wire-level* state machine specification; and (3), finally, we define a typeclass for a PULSE implementation of an imperative event handler whose type signature ensures that it processes events in accordance with the state machine specification. Note, this process of arriving at the state machine refinement class was itself done using a collaborative process between an agent and a human, emphasizing our focus on structured interaction between humans and agents. A few initial attempts at having the agent write it produced results that we found unacceptable. Eventually, we directed the agent to write exactly the signature we wanted for state machine refinement.

5.3.1 State Machine Class. For generality, our state machine class processes two kinds of input events: “wire events” received over a network, and “local events” generated by the service itself. Similarly, it produces two kinds of outputs. For our calculator service there are no local events, but we include them for generality.

```

type event (msg:Type) (loc_in:Type) = | WireEvent of msg | LocalEvent of loc_in

type step_output (msg:Type) (loc_out:Type) = { so_wire: list msg; so_locals: list loc_out }

class state_machine (state msg loc_in loc_out:Type) =
{ sm_initial_state: state; sm_step: state → event msg loc_in → state → step_output msg loc_out → prop }
```

A trace element is a triple of a state, the inputs and outputs, and a trace is a list of trace elements. Reachability between states is defined as the existence of a trace to transition between them—the definition is standard, so we omit it here.

5.3.2 Wire-Level State Machines. The `wire_format` class below defines a wire format for a type `msg` with parser and serializer specification (`ghost`) functions, and a proof that parsing is an inverse of serialization.

```

class wire_format (msg:Type) =
{ wf_serialize: msg → bytes; wf_parse: bytes → option (msg & bytes);
  wf_parse_serialize_exact: msg:msg → Lemma (wf_parse (wf_serialize msg) == Some (msg, empty)) }
```

Given a wire format and a state machine we can define a wire-format state machine (`wfsm`), whose inputs and outputs are raw bytes rather than messages or events—a state machine defines the end-to-end observable behavior of a service.

```

class wfsm (state msg loc_in loc_out:Type) =
{ sm: state_machine state msg loc_in loc_out; wire_fmt: wire_format msg; }
```

Using a `wfsm`, a wire-level state machine trace is defined via a `valid_byte_trace` predicate, which states that there is a trace starting at the initial state such that the input bytes parse as the trace’s input messages and the output bytes equal the serialization of the trace’s wire outputs. This predicate is the main one we’ll use to connect the PULSE implementation to a state machine specification.

5.3.3 *Implementation Refinement.* Our final step is a typeclass to relate a PULSE implementation of an event handler to a wire-level state machine. The class `pi impl st msg loc_in loc_out` (for protocol implementation) is parameterized by an implementation type `impl` and the types of the state, wire messages, local events, and local outputs. The class includes the following key fields:

- `system:wfsm st msg loc_in loc_out`, defining the wire-level state machine specification the implementation is supposed to refine.
- invariant `i received sent st`, a separation logic predicate that ties the implementation to the specification, where `received` and `sent` record the full history of bytes on the wire so far, in both directions, and `st` is the current state of the state machine. This is made precise by the next field.
- invariant `_valid`, a ghost function that states that if invariant `i received sent st` holds, then one can derive `valid_byte_trace`, i.e., that the implementation is in a state corresponding to a valid trace of the state machine from the initial state on the bytes received, producing the bytes sent.

invariant `_valid`: **ghost fn** (`i:impl`) (`received sent:bytes`) (`st:state`)

preserves invariant `i received sent st`

ensures pure (`WFSM.valid_byte_trace (pi_system i) received st sent empty`)

These are the key elements of our state-machine refinement class (there are some technicalities with regard to monotonicity, which we cover shortly). With this in place, we can define signatures for the event handlers, both for wire events and local events—we show the wire event handler only, as the local event handler is similar.

The structure of the wire event handler is similar to what we showed for `process_request` for our calculator sample, which the agent had access to as a template. The idea is for `pi_process_network` below to receive a network buffer with the raw bytes of the input and to write its result into an output buffer (decoupling it from the actual network interface). The signature below shows `pi_process_network` receiving as arguments a handle to the implementation state (`impl`) and the two buffers with their length—its first precondition requires ownership of the buffers and that their lengths are correct. The main precondition is the invariant, tying the implementation to the state machine specification. The event handler returns a result, and re-establishes the invariant for some new state and history and returns ownership to the buffers, with the input buffer unmodified (line 6).

```

1 pi_process_network: fn (i:impl) (input out:array U8.t) (ilen olen:SZ.t) (#received sent st in0 out0:erased _)
2   requires input ↦ in0 ** out ↦ out0 ** buffers_lengths_wf in0 ilen out0 olen
3   requires invariant i received sent st
4   returns result:process_result
5   ensures  $\exists^*$  received1 sent1 st1 out1 consumed wire_outputs loc_outs.
6     invariant i received1 sent1 st1 ** input ↦ in0 ** out ↦ out1 **
7     pure (network_process_correct i .. loc_outs)

```

Tying it all together, at line 7 we have a pure predicate `network_process_correct`, which relates the old state to the new state, the inputs and outputs, as sketched below. The `process_result` contains a status, the number of bytes consumed from the input, and the number of bytes produced to the output. We focus on just the success case, although the other cases are also fully specified:

type `process_status` = `| StepOk | NeedMoreInput | ParseFailed ...`

let `network_process_correct .. (result:process_result) .. : prop =`

match `result.process_status` **with**

```

785 | StepOk → ∃ msg residual produced.
786   consumed_by_parse wire_fmt (input_bytes input ilen) msg consumed residual ∧
787   SZ.v result.process_consumed_len == Seq.length consumed ∧
788   step st0 (WireEvent msg) st1 (step_output wire_outputs loc_outs) ∧
789   Seq.equal produced (serialize_all wire_fmt wire_outputs) ∧
790   output_written out_bytes result.process_produced_len produced ∧
791   Seq.equal received1 (Seq.append received0 consumed) ∧
792   Seq.equal sent1 (Seq.append sent0 produced)
793 | ...

```

In the success case (StepOk), the implementation has consumed some bytes from the input, parsing them into msg leaving some residual bytes; the state machine has stepped from st0 to st1 on msg, producing wire and local outputs serialized to the output buffer; and the new indices on invariant exactly record the extended inputs and outputs, and the new state.

Monotonicity. There is one more technicality: monotonicity. We know invariant i received sent st implies that the implementation is always in a state that corresponds to running the state machine from the initial state on the bytes received so far. But nothing tells us the implementation could not suddenly rewrite history and claim it had received different bytes and jump to another arbitrary state. We need to ensure that once the implementation has processed some inputs, it commits to the history so far. For this we add three fields: pi_snapshot i received sent st, a duplicable predicate stating the implementation was once in state st after consuming received and producing sent; pi_take_snapshot to record the current state; and pi_recall_snapshot to prove that the current state is consistent with (“ahead of”) a snapshot, ensuring an implementation cannot rewrite history. This style of modeling monotonicity is common in separation logic; there are libraries in PULSE for it based on ghost state, and we reuse it here.

Whereas our previous rubrics were for proofs of specific classes of algorithms, this rubric encapsulates a pattern for a broad class of proofs—many proofs are state machine refinements, and this rubric imposes a common style for their top-level structure to enable easier audit. Instances of this class can use many variations internally (e.g., they may model ghost state and histories in many different ways, employ concurrency internally, use different memory management strategies etc.), leaving an agent lots of freedom, but providing enough constraints at the top-level audit boundary for a human to make sense of it. Of course, in order to judge whether the rubric itself is correct, one needs some expertise in separation logic and to read the definitions carefully. But, one can do this in the abstract, without worrying about the details of any particular service. Furthermore, once such a rubric is blessed, a user can review an implementation of a stateful service purely at the level of the state machine specification that it refines, without needing to understand separation logic at all.

5.4 Refined Calculator Service

Setting up this rubric may seem like a lot of work, but it is a one-time cost, and once done it can be reused for many services, especially since state machine refinement is such a common pattern. Providing it as a rubric to an agent for our calculator service works well and produces an instance of the class, as shown below:

```

829 instance calc_server : pi src ... = { system = calc_frame_wire_format_state_machine; ...
830

```

The only thing we need to review is that the system field is indeed the correct state machine for the calculator service. The rest of the class ensures there is a PULSE implementation that refines it.

That implementation and proof are about 4,500 lines of F^* and PULSE code, but we do not need to inspect any of it.

Even knowing that it refines the pure state machine, one should also apply other auditing steps, including basic testing. We also provide verified code (not shown) to turn a protocol event handler into a full implementation, tied to a TCP channel, with a top-level event-handling loop that repeatedly calls the event handler and sends the results back to the client. The resulting verified code extracts to about 300 lines of C code, and together with a few dozen lines of C code to actually open a TCP socket, this code correctly runs a suite of network tests.

5.5 Reasoning About State Machine Specifications using LTL

For the calculator service, the state machine is simple enough that one can check it just by inspection, but for more complex services we will need to do more. Our refinement framework opens up some possibilities for reasoning about specifications. Once implementation refinement is discharged in separation logic (§5.3), all remaining questions about a service reduce to questions about its pure state-machine specification. Beyond auditing that specification by inspection, we would like to *prove* properties of it, and in particular properties of how two endpoints interact over time. To this end, we used agents to build a small, protocol-independent temporal-logic library (Common.Temporal), which we now illustrate on the calculator.

A temporal logic over transition systems. The library is parameterized by an abstract state type S and a step relation $\rightarrow \subseteq S \times S$. A *path* is an infinite sequence of states $\pi : \mathbb{N} \rightarrow S$, and a *run* is a path in which consecutive states are related, $\pi_i \rightarrow \pi_{i+1}$ for all i ; $\pi^{|n}$ denotes the suffix $i \mapsto \pi_{i+n}$. On top of paths we define the standard LTL operators as predicates on runs— $G \phi$ (always), $F \phi$ (eventually), $X \phi$ (next), $\phi \cup \psi$ (until)—and the branching path quantifiers AG , EF , AF , where $AG \phi$ at a state s means “on every run starting from s , ϕ holds at every position”.

A calculator client to pair with the server. To make things a bit more interesting, and also to serve as a template for TLS, we added a client side to the calculator. The client state machine is the mirror image of the server, a two-phase machine that is either `ClientIdle`, ready to issue the next request, or `ClientAwaiting b`, having sent request frame b and awaiting its response. Where the server receives requests and sends responses, the client sends requests and receives responses, tracking a “predicted server” log of all completed round-trips; `client_stack` is the calculator stack of that log and `server_stack` the server’s. Crucially, the client’s receive step performs literally the same computation as the server’s request processing (`lemma_rcv_completed_eq`), which is what makes the two logs re-converge on delivery.

To reason about the entire system, we compose the client and server machines into a single system whose state is a triple (c, ℓ, κ) of the client state c , the server log ℓ , and a *channel* κ modeling the network. Because the protocol is strict request/response lockstep, at most one message is ever in flight, so the channel is one of `Quiet` (nothing in flight), `InReq b` (request frame b sent but not yet processed), or `InResp r` (response r produced but not yet received), and the three transitions are *issue*, *serve*, and *rcv*. The single-slot channel bakes in our delivery assumptions: the network is reliable and order-preserving—messages are never dropped, duplicated, or reordered—while delivery may be delayed arbitrarily (modeled by a stuttering step in which the channel does not advance). Under these assumptions safety holds unconditionally; liveness additionally assumes *fairness*, i.e., that no message stays in flight forever.

A simple safety theorem: agreement under quiescence. One might hope to prove AG (`client_stack = server_stack`), that the two endpoints always agree on the calculator stack. This is *false*: while a request or response is in flight the stacks lag by exactly the in-flight work. We say the session is

quiescent when the channel is Quiet, i.e., there is no request or response in flight and both endpoints have caught up. The theorem we prove is then $\text{AG}(\text{quiescent} \implies \text{client_stack} = \text{server_stack})$, which states that on every run from the initial state, whenever the session is quiescent the client and server agree on the current stack.

The proof is done entirely by an agent, which, rather than reasoning about paths directly, proves a structural invariant `system_inv` that pins down the exact lag (Quiet and `InReq`: the server log equals the client's completed log; `InResp`: the server is exactly one processed request ahead). This invariant is *inductive*—it holds initially and is preserved by every transition—so it holds on every reachable state. A soundness bridge lemma, `lemma_ag_of_invariant`, then lifts any predicate that holds on all reachable states to a genuine path-based AG. Thus a cheap inductive-invariant proof discharges a real temporal property, and agreement follows because `system_inv` forces `server == client.completed` exactly in the quiescent case.

Beyond safety: next-step and liveness. The library also lets us state liveness and next-step properties in the same vocabulary: we prove an X property (from an idle, quiescent state the *next* state necessarily has a request in flight, since issuing is the only enabled move) and, under fairness, a liveness property $\text{AG}(\text{in flight} \implies \text{F quiescent})$ —every outstanding request is eventually completed—proved by well-founded recursion on a channel rank (`Quiet` = 0, `InResp` = 1, `InReq` = 2) that fairness forces to decrease.

In summary, with state machine refinement coupled with temporal reasoning on the specifications, one can provide abstract high-level specifications (independent of separation logic) that are easier to review and promote trust. So far, we have used our LTL framework only on the calculator example, though we believe it could also be applied to more sophisticated protocols in the future.

6 Case Study: TLS 1.3

We went to some trouble in the last section to create a rubric for state machines and their implementation refinements. Here, we put that rubric to work by using it to structure a verified implementation of the TLS-1.3 protocol. Transport Layer Security (TLS) is a widely used cryptographic protocol for secure communication over the Internet. TLS-1.3 is the latest version, specified in RFC 8446.³ Verifying implementations of TLS has been a long-standing challenge, due to the complexity of the protocol and the need to reason about cryptographic properties. For example, Project Everest (Ahman et al. 2026) targeted the verification of TLS and produced verified implementations of many of its components using F^* . However, some parts of the protocol were not fully verified—in particular, the main state machine proof of the TLS handshake protocol was not completed. A verified executable implementation of TLS-1.3 is still considered a challenging goal, and we aimed to see if we could use our techniques to get agents to build one.

TLS supports many modes and optimizations. We focus on verifying a single, commonly used profile of the protocol, reusing some verified components from Project Everest (notably the cryptographic primitives) and focusing on the main handshake protocol in the following mode: 1-RTT handshake with no 0-RTT support; no session resumption; X25519 key exchange; and ChaCha20-Poly1305 as the symmetric cipher, with SHA-256 as the hash function. This is a limited profile for TLS, but a common one and widely supported.

6.1 A Roadmap for an Agentic Implementation

An agentic implementation of TLS-1.3 took about 2 weeks of agentic programming to get a first verified version. Getting there took some planning and structure, with audits, rubrics, and templates playing a central role.

³<https://datatracker.ietf.org/doc/html/rfc8446>

Templates. We used the implementation of the Calculator Server as a template for an agent, instructing it to first implement a TLS-1.3 client, and then using the client as a template for the server. While it might seem like a leap too far to go from a calculator service to a TLS-1.3 client and server, our experience is that agents can handle such leaps quite well. A lesson: start small, and build up to more complex systems, using the simpler systems as templates for the more complex ones, though one might proceed much faster and with much bigger steps than a human would.

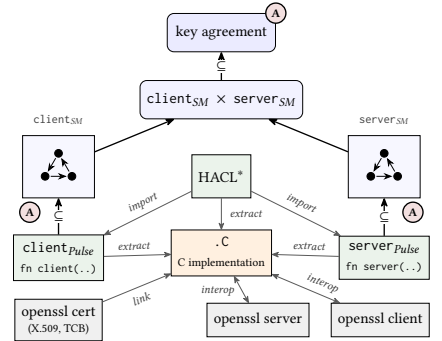
Rubrics. To ensure that the implementation follows a well-understood pattern and to keep things comprehensible for a human, we use the state machine typeclass to ensure that the Calculator Server and the TLS implementations are not just loosely related, but that they are both, formally, instances of the same rubric. Understanding the rubric alone, and convincing oneself that it enforces a state machine refinement, is enough to understand that a TLS implementation refines a state machine specification.

Audits. The state machine specification for the Calculator Server is just a few dozen lines and is easy to review. In contrast, the state machine specification for TLS, even for our limited profile, is on the order of a few thousand lines. While our rubric assures that the implementation refines this specification, there is still the question of whether the specification is correct. For this we use two main techniques:

- *Interoperability testing* A claimed implementation of TLS can only practically be considered an implementation of TLS if it can interoperate with other implementations. Our TLS implementation interoperates with OpenSSL in both directions.
- *State machine proofs* Knowing that the implementation refines the state machine, we can ignore the implementation and focus on the specification alone. We use agents to prove that the specification guarantees some useful properties. Notably, we prove that if a client and server both agree on the messages exchanged, and their handshake state machines have reached a state ready for application data, then they also agree on all the key material derived by the handshake key schedule.

Note, we do not prove any *cryptographic security* properties of the protocol, only functional correctness properties. Extending our state machine specification to also support security proofs would be an interesting direction to pursue in the future.

The figure alongside sketches the overall shape of our development. Reading it bottom-up, agents author the Pulse implementations and their refinement proofs; each proof discharges, in separation logic and checked by F^* , that the implementation *refines* ($\sqsubseteq$) a state-machine specification. Refinement composes upward: the product of the client and server state machines refines a top-level property, that the client and server derive the same keys on a successful handshake. Because the $\sqsubseteq$ edges are machine-checked, the human audit burden collapses onto the artifacts those edges *target*: the specifications and the top-level property, marked with a circled A. Rubrics (the state-machine refinement typeclass, which is audited once and for all) make the bottom-most edges cheap to check, while audits—SPOTs, proofs of specification properties, and, at the boundary with the outside world, interoperability testing of the extracted C against OpenSSL—discharge the remaining obligations. Trusted components, such as



the imported HACL* primitives and the OpenSSL certificate library used for X.509 validation, sit in the trusted computing base. We develop each ingredient of this picture in the sections that follow.

6.2 Specifying the State Machine

Specifying the TLS state machine, even for just the limited profile we are considering, is a substantial piece of work. However, from their background knowledge of the TLS protocol, and using RFC 8446 as a reference, agents produce a specification seemingly without difficulty. Initial versions of the state machine specification were too permissive, and the specification was refined multiple times throughout the process, particularly when trying to prove properties of the specification—we describe some of this in the next section.

We also had the agents implement first the client state machine only, and then generalize it to support multiple roles to also cover the server in a single specification. We do not show the state machine in detail at all, but provide a glimpse of it to give a sense of its structure.

The state of the protocol is `connection_state`, which includes a model of the protocol state, a log of the messages sent and received on the wire, and a log of the events that have occurred. The connection model itself has many components, reflecting the state of the various sub-protocols that are part of TLS, including the initial configuration and the handshake state machine.

```
type connection_state = { cs_model:connection_model; cs_wire_log:wire_log; cs_event_log:list conn_event }
type connection_model =
{ config: connection_config; control: connection_control_state; record: record_layer_state;
  handshake: handshake_state; application: application_state; failure: option tls_error; }
```

The handshake state machine is perhaps the most interesting part: it records various parts of the handshake, e.g., the messages sent and received so far, the state of the key schedule, and the transcript of the handshake. The transition relation is given by `client_step`. Unlike for the calculator, the transition relation is not a simple function of the current state but is instead a *relation* between the current state, the event, the next state, and the outputs; there are both wire events and local events; and the core of the transition relation is a notion of legal transitions.

We show a part of a legal transition produced by a local event, which includes cases for deriving keys, callbacks to certificate validation, and other local events. The transition shown below might appear too permissive, e.g., a `LocalVerifyCertificateSignature cv` event could be issued even if the certificate signature is invalid. However, the transitions are guarded by a `legal_loc_in` predicate, as shown below, ensuring that `LocalVerifyCertificateSignature cv` can only be fired if the certificate signature is indeed valid according to `C.verify_signature`, a pure function specifying the behavior of a signature verification algorithm.

```
let legal_loc_in (model:connection_model) (ev:loc_in) : prop =
let hs = model.handshake in
match ev, model.control with
| LocalVerifyCertificateSignature cv, ControlHandshaking HsCertificateVerifyReceived →
  model.config.config_role == ClientEndpoint ∧
  (match hs.hs_validated_peer, hs.hs_certificate_verify, hs.hs_buffers.hb_certificate_verify_input with
  | Some peer, Some stored_cv, Some input →
    stored_cv == cv ∧ C.verify_signature cv.scheme peer.leaf_public_key input cv.signature
  | _, _ → False)
```

Our point is that this is a very detailed specification of the TLS state machine. Auditing it in its entirety for compliance with the RFC (an informal document) is a significant task, so we do not make any claims about that. Interoperability testing helps somewhat: it at least shows that the

state machine specification permits enough behaviors to allow an implementation that refines it to interoperate with OpenSSL. However, it does not show that the state machine is not too permissive, i.e., that it does not allow incorrect behaviors. So we try to come at it from another angle, by proving that the state machine specification satisfies some useful properties.

6.3 Auditing with Key Agreement Proofs

One way to audit whether this is a good specification is to try to prove that it satisfies various useful properties. One property we aim to ensure is that if the client and server agree on the messages exchanged, then they also agree on the session key derived by the handshake. We had an agent prove this derived key agreement property, and with several rounds of refinement on exactly how it should be formalized—interactive intent formalization with an agent requires a human to have a level of fluency with formal notation and also some amount of domain knowledge (in this case, TLS) to judge whether what is proven is indeed what one wants to prove. This is the statement of the lemma we arrived at, and we describe it in detail below.

```
val client_server_agreement_from_paired_successful_traces (client server:connection_state) : Lemma
(requires
  sent_tls_messages client.event_log == received_tls_messages server.event_log  $\wedge$ 
  sent_tls_messages server.event_log == received_tls_messages client.event_log  $\wedge$ 
  client_driver_application_ready client  $\wedge$  server_driver_application_ready server  $\wedge$ 
  client_trace_shape client  $\wedge$  server_trace_shape server )
(ensures supported_profile_client_server_key_material_agrees client server  $\wedge$ 
  peer_record_material_agrees (traffic_id TrafficApplication ClientTraffic) client server  $\wedge$ 
  peer_record_material_agrees (traffic_id TrafficApplication ServerTraffic) client server)
```

The first two preconditions are relatively straightforward, i.e., the traces are paired, i.e., the client’s sent messages are received by the server and vice versa (this is a functional correctness theorem rather than a cryptographic one, so we have no notion of an adversary). The remaining preconditions are more subtle.

At line 5, the precondition `client_driver_application_ready` packages a lot. It states that the client state satisfies the state machine invariant, notably that the state is reachable from the initial state by running the event log recorded in state. It also states that the client is in the `ControlApplicationData` state and that the keys installed in the record-layer sub-protocol are exactly the keys derived by the key schedule. The precondition on the server is similar.

At line 6, we have preconditions on the shapes of the client and server traces. We describe the client precondition, and the server precondition is similar. First, it requires that the trace contain no CCS (ChangeCipherSpec) messages or key update events—CCS messages are a legacy feature of TLS-1.2 that are not used in TLS-1.3, although they remain in the TLS-1.3 specification for backward compatibility; key update events are also not supported by this theorem, though they would be useful to add in the future. Next, it requires that the trace be split into a handshake prefix and an application-data suffix, where the handshake prefix is the first point at which the state machine enters the `ControlApplicationData` state, and the application-data suffix is the remainder of the trace (which remains in the `ControlApplicationData` state).

Under these preconditions, the postcondition guarantees that the client and server agree on all the key material derived by the handshake key schedule, including the handshake traffic keys, the application traffic keys, and the master secret. This is a crucial property for the correctness of the protocol, based on which one could further analyze its cryptographic security, since it is a prerequisite to showing that both parties can communicate securely using the same keys.

Assumptions. Underlying these proofs are also some assumptions about the underlying cryptographic primitives: that X25519 key agreement is symmetric (if the client and server have the same public keys, they derive the same shared secret), and that ChaCha20-Poly1305 is a functionally correct authenticated encryption scheme (opening a sealed plaintext returns the plaintext). Both properties could be proven from the underlying primitives; for this development, we focused on using pre-existing verified cryptographic primitives from HACL* (Zinzindohoué et al. 2017) and assumed these properties, rather than verifying these properties again from scratch. We also used libraries from OpenSSL for certificate validation, and assumed that they are functionally correct.

6.4 Specifying the TLS Wire Formats

Nothing a priori prevents agents from designing the wire format specification directly in pure F* and writing verified parsers and serializers directly in PULSE. However, we instead chose to use EverParse (Ramananandro et al. 2019), with support for automatic generation of verified parsers and serializers with separation logic in PULSE (Ramananandro et al. 2025).

Using EverParse has two benefits. First, given a data format specification, EverParse provides not only the formal specifications and verified implementations of parsers and serializers, but also proofs of roundtrip between the parser and the serializer, needed by the `wire_format` class, and more (e.g. parser injectivity by virtue of unique binary representation), thus relieving agents from completing any such proofs. EverParse also serves as a rubric, since it only generates secure parsers & serializers. Second, the specification produced by the AI agents for the TLS handshake message format is suitable both for EverParse consumption and for human auditing, by virtue of EverParse using notations inspired from those of the IETF TLS 1.3 standard (Rescorla 2018, §3). This makes auditing easier by reducing the likelihood of interoperability failures due to wire format mismatch.

6.5 Auditing with Rubric Compliance & Interoperability Testing

While we have a large state machine specification, our typeclass for state machine refinement gives us an easy way to check that the PULSE implementation of this TLS state machine is indeed a correct refinement. Reviewing this is just a matter of checking that the implementation satisfies the typeclass, which is just a few lines of code. A similar instance is provided for the server, and both instances are checked to ensure that they satisfy the typeclass, guaranteeing that they are indeed refinements of the state machine specification.

```
let client_pi : pi ... = { system = client_system; ...
```

Finally, to confirm that we have truly implemented a TLS-1.3 client and server, we perform interoperability testing with OpenSSL. Our implementation extracts to about 45,000 lines of C code (of which about 15,000 lines are generated by EverParse for wire format parsers and serializers). We compile this to C, linking the HACL* libraries for cryptographic primitives, and OpenSSL libraries for certificate validation. Together with a few hundred lines of unverified C code to implement a top-level main function to launch a client or server, configure the network, and perform some basic testing to negotiate a connection, the resulting test client and server interoperate with OpenSSL.

One would want to audit this further for a number of other properties, e.g., performance profiling and side-channel resistance (although the HACL* cryptographic primitives already have some side-channel resistance guarantees). We don't mean to claim that our TLS client and server are production ready, though it is a good start. What is important is that with a careful choice of templates (starting with small demonstrations like the Calculator Server), rubrics (to easily check that an implementation is a refinement of a state machine), and audits (e.g., by proving specification properties, and by interoperability testing), we can use agents to produce a verified implementation of a complex protocol like TLS-1.3 and have some confidence that it is correct.

7 Missteps, Reflections, and Validation

The development of our ART principles arose from a long series of experiments with agents, starting with AutoCLRS, and continuing through to TLS. Along the way, we made several missteps, and learning from them helped us articulate the ART approach. We highlight a few of our key learnings. We also present a clean slate application of ART to a family of file-transfer protocols, aiming to get a sense of how one would apply ART to a new agentic proof-oriented project today.

7.1 Missteps & Learnings

Wire Formats First. If one uses a verified parser/serializer generator (e.g. EverParse) for the wire formats, it is preferable to let agents define the wire format first, in the data description language of the generator, and then use the generated message types in the state machine specifications and implementations, rather than letting AI agents produce those types themselves when defining the state machines. We did not do this for TLS, allowing the agent to produce the state machine first. When trying to connect this eventually with the types produced by EverParse, we ran into a number of difficulties, including the need to produce bridge code. Viewed through the lens of ART, it is a misstep because it allowed the agents to invent the event types for the state machine, without any constraint on how those types relate to the wire format, e.g., the types chosen may not satisfy the needed roundtrip properties, as prescribed by the wire format class.

Parallel Work. For AutoCLRS, we had the agents work in parallel on several chapters. This led to a lot of replicated work by the agents, and worse, replicated reviewing work for humans. Parallel work also restricts the possibility of using earlier successful attempts as templates. In retrospect, setting rubrics for shared algorithms first would have reduced the overall reviewing effort, and may also have promoted the use of shared abstractions among the various chapters. While it is alluring to set a one-line prompt to an agent, thinking up front about ART may lead to overall less work and better results.

Maintenance. While we have been able to repeatedly add features and proofs to our code bases, we do not yet have experience to judge long-term maintainability of agentic code bases. What is clear is that the scale of our proofs is already such that using agents to maintain the code going forward is unavoidable. Given agents' ability to iterate effectively with a clear objective function, in a code base with proofs, refactorings that do not change the top-level theorem can be confidently attempted. However, open-ended refactorings with prompts to simply improve the code without clear guidance appear unlikely to succeed, at least with the current models.

7.2 Validation: File Transfer Protocols

We validated our ART principles by applying them from scratch, along with our lessons learned, to verified file transfer protocol implementations, such as TFTP or YMODEM. We picked these as additional examples to instantiate our generic state machine refinement framework. A proof of each of these would be substantial work for a human, but using the calculator again as a template, as well as the TLS implementation, and building on the rubrics we already had developed, we were able to quickly produce verified interoperable implementations of these protocols: YMODEM in less than two days, and TFTP in 7 hours.

We first let agents define the rubric as a typeclass providing an additional layer of abstraction for file transfer protocols, with a theorem stating that, if the protocol reaches a successful end, then the client receives the file it requested from the server. This theorem is the critical piece that lets us audit the rubric, illustrating that rubrics can compose, with the typeclass for file transfer refining the wire format state machine (wfs_m). Then, we let agents use the calculator service and

TLS implementations as templates to produce the implementation of the first protocol, YMODEM, with auditing by interoperability testing with lrzsz.⁴ The implementation is about 10,000 lines of verified code, specification, and proof, including 1,500 lines of code produced by EverParse. Finally, we let agents use YMODEM as a template to produce an implementation of TFTP, with auditing by interoperability testing with tftp-hpa and tftpd-hpa.⁵ This time, the implementation is about 5,000 lines of verified code, specification, and proof, and does not use EverParse, though in both cases, we instructed the agents to produce the wire format first.

We are encouraged by these results to attempt broader scale applications, e.g., using our ART approach to build verified implementations of a large collection of RFCs. While we have been able to use agents to quickly build verified implementations, pushing further to produce production-quality verified code is the next challenge.

8 Related Work

Language models are increasingly applied to formal proof. Using them for proof automation, particularly of leaf goals, is now widespread. Tactic-level search in GPT-f (Polu and Sutskever 2020), informal-sketch guidance in Draft-Sketch-Prove (Jiang et al. 2023), whole-proof generation with error-driven repair in Baldur (First et al. 2023), and premise retrieval in LeanDojo (Yang et al. 2023) synthesize proofs toward a human-fixed goal, as do dafny-annotator (Poesia et al. 2024) and AutoVerus (Yang et al. 2025) when filling annotations for a *given* specification. Our work instead focuses first on the synthesis and posterior auditing of agent-proposed specifications themselves, via SPOTs, rubrics, and then on using templates to help the agents reach an acceptable solution effectively. More related to our work, Feng et al. (2026) validate synthesized specifications by property-based testing (which requires executable specifications); and Paraskevopoulou (2026) reports a *silently weakened* theorem caught only by chance. We hope SPOTs and rubrics help audit non-executable specifications more systematically.

A second strand studies specification quality, the problem central to our work. Clover (Sun et al. 2024) checks consistency among code, docstring, and annotations, whereas our SPOTs statically probe whether a specification is precise. IronSpec (Goldweber et al. 2024) is closest to SPOTs, vetting specifications through spec-testing proofs, sanity checks, and mutation testing, albeit for human-authored Dafny specs, but it does not cover tests that expose relations among different APIs or our checking of generic specifications, as we do for our rubrics. Lahiri (2024) symbolically tests whether a postcondition captures intent, the direct lineage for SPOTs. The co-audit agenda (Gordon et al. 2023) frames tools helping humans double-check AI output, and ART is an instance of co-audit.

Our TLS-1.3 case study inherits a lineage of expert-verified systems: Project Everest (Ahman et al. 2026) pursued a verified drop-in HTTPS stack, and miTLS (Bhargavan et al. 2013) is a verified TLS-1.2 reference implementation with a machine-checked cryptographic-security proof. Delignat-Lavaud et al. (2017) proved the TLS-1.3 *record layer*; we target a *different* layer, the handshake state machine, with an agent-authored proof in Pulse. For it we use a state-machine refinement methodology similar in spirit to IronFleet (Hawblitzel et al. 2017), but recast in separation logic, with a generic formulation of state machine refinement as a typeclass.

9 Conclusion

There is a growing chorus of voices claiming that, with agents, natural language is the only programming language we’ll need. While it is true that agents are remarkable at generating code from informal intent, and perhaps humans will cease to write code, we strongly believe that

⁴<https://ohse.de/uwe/software/lrzsz.html>

⁵<https://git.kernel.org/pub/scm/network/tftp/tftp-hpa.git/>

programming language design and abstractions are necessary both to delimit agent behavior, and perhaps more importantly, as a prerequisite for trust, for humans to be able to effectively comprehend at least the specifications. We have offered some techniques for structuring agent interactions, under our ART umbrella, and we hope to spur further research into specification language development and spec comprehension.

References

- Danel Ahman, Karthikeyan Bhargavan, Barry Bond, Jay Bosamiya, Christopher Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Aymeric Fromherz, Sydney Gibson, Chris Hawblitzel, Cătălin Hrițcu, Markulf Kohlweiss, Guido Martínez, Haobin Ni, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, Exequiel Rivas, Nikhil Swamy, and Santiago Zanella-Béguelin. 2026. Project Everest: Perspectives from Developing Industrial-Grade High-Assurance Software. *ACM Trans. Program. Lang. Syst.* 48, 2, Article 9 (June 2026), 64 pages. <https://doi.org/10.1145/3805702>
- Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and P. Strub. 2013. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy*. 445–459.
- Tom B. Brown, Benjamin Mann, Nick Ryder, et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*. <https://arxiv.org/abs/2005.14165>
- Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. *SIGPLAN Not.* 43, 1 (Jan. 2008), 133–144. <https://doi.org/10.1145/1328897.1328457>
- Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean-Karim Zinzindohoué. 2017. Implementing and Proving the TLS 1.3 Record Layer. In *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 463–482. <https://doi.org/10.1109/SP.2017.58>
- Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1516–1539. <https://doi.org/10.1145/3729311>
- Yueyang Feng, Dipesh Kafle, Vladimir Gladshtein, Vitaly Kurin, George Pirlea, Qiyuan Zhao, Peter Müller, and Ilya Sergey. 2026. Certified Program Synthesis with a Multi-Modal Verifier. <https://doi.org/10.48550/arXiv.2604.16584> arXiv:2604.16584 [cs.SE]
- Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, 1229–1241. <https://doi.org/10.1145/3611643.3616243>
- Eli Goldweber, Weixin Yu, Seyed Armin Vakil Ghahani, and Manos Kapritsos. 2024. IronSpec: Increasing the Reliability of Formal Specifications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024)*. USENIX Association, 875–891. <https://www.usenix.org/conference/osdi24/presentation/goldweber>
- Andrew D. Gordon, Carina Negreanu, José Cambronero, Rasika Chakravarthy, Ian Drosos, Hao Fang, Bhaskar Mitra, Hannah Richardson, Advait Sarkar, Stephanie Simmons, Jack Williams, and Ben Zorn. 2023. Co-audit: tools to help humans double-check AI-generated content. arXiv:2310.01297 [cs.HC] <https://arxiv.org/abs/2310.01297>
- Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2017. IronFleet: proving safety and liveness of practical distributed systems. *Commun. ACM* 60, 7 (2017), 83–92. <https://doi.org/10.1145/3068608>
- Eleftherios Ioannidis, Nikhil Swamy, Gabriel Ebner, Matthai Philipose, and Tahina Ramananandro. 2026. Proofs Promptly: Proof-Oriented Programming with AI Agents (Experience Report). *Proc. ACM Program. Lang.* 10, ICFP, Article 311 (2026). <https://doi.org/10.1145/3828709>
- Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations (ICLR)*. <https://arxiv.org/abs/2210.12283>
- Shuvendu K. Lahiri. 2024. Evaluating LLM-Driven User-Intent Formalization for Verification-Aware Languages. In *Formal Methods in Computer-Aided Design (FMCAD 2024)*. TU Wien Academic Press, 142–147. https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_21
- Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*. <https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html>
- Guido Martínez, Bastian Köpcke, Jonáš Fiala, Gabriel Ebner, Tahina Ramananandro, Michel Steuwer, Tyler Sorensen, and Nikhil Swamy. 2026. Kuiper: Correct and Efficient GPU Programming with Dependent Types and Separation Logic. *Proc.*

- ACM Program. Lang. 10, PLDI, Article 202 (June 2026), 25 pages. <https://doi.org/10.1145/3808280>
- Guido Martínez and Tyler Sorensen. 2026. The Next Frontier for AI-Generated Kernels: Correctness. In *Proceedings of the 2026 ACM SIGPLAN International Workshop on Principles of Agentic Engineering* (Boulder, CO, USA) (PAGe '26). Association for Computing Machinery, New York, NY, USA, 25–37. <https://doi.org/10.1145/3819802.3820580>
- Kim Morrison. 2026. lean-zip: A Formally Verified zlib Implementation in Lean 4. <https://github.com/kim-em/lean-zip>. Accessed: 2026-07-09.
- Zoe Paraskevopoulou. 2026. Machine-Generated, Machine-Checked Proofs for a Verified Compiler (Experience Report). <https://doi.org/10.48550/arXiv.2602.20082> arXiv:2602.20082 [cs.PL]
- Gabriel Poesia, Chloe Loughridge, and Nada Amin. 2024. dafny-annotator: AI-Assisted Verification of Dafny Programs. CoRR abs/2411.15143 (2024). <https://doi.org/10.48550/arXiv.2411.15143> arXiv:2411.15143 [cs.SE]
- Stanislas Polu and Ilya Sutskever. 2020. Generative Language Modeling for Automated Theorem Proving. arXiv:2009.03393 [cs.LG] <https://arxiv.org/abs/2009.03393>
- François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proc. ACM Program. Lang.* 8, POPL, Article 50 (Jan. 2024), 27 pages. <https://doi.org/10.1145/3632892>
- Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) (USENIX Security 2019). USENIX Association, USA, 1465–1482.
- Tahina Ramananandro, Gabriel Ebner, Guido Martínez, and Nikhil Swamy. 2025. Secure Parsing and Serializing with Separation Logic Applied to CBOR, CDDL, and COSE. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) (CCS '25). Association for Computing Machinery, New York, NY, USA, 2744–2758. <https://doi.org/10.1145/3719027.3765120>
- Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. <https://doi.org/10.17487/RFC8446>
- Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. arXiv:2310.17807 [cs.AI] <https://arxiv.org/abs/2310.17807>
- Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Association for Computing Machinery, New York, NY, USA, 256–270. <https://doi.org/10.1145/2837614.2837655>
- Nikhil Swamy, Aseem Rastogi, Gabriel Ebner, Guido Martínez, Tahina Ramananandro, and the Pulse team. 2025. Pulse: Proof-oriented Programming in Concurrent Separation Logic. <https://fstar-lang.org/tutorial/book/pulse/pulse.html>. Chapter of the F* tutorial. Accessed: 2026-07-09.
- Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu K. Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 3454–3482. <https://doi.org/10.1145/3763174>
- Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*. <https://arxiv.org/abs/2306.15626>
- Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, New York, NY, USA, 1789–1806. <https://doi.org/10.1145/3133956.3134043>