

Modern software rests on a layer of code that almost no one reads and almost everyone trusts: the operating system kernel and the hardware interface beneath it. When a kernel manages memory, switches between processes, or commits a file to disk, it manipulates resources whose meaning depends on context: a memory address means one thing in one process and something else in another, and a disk block belongs to one snapshot but not the next. My research builds the formal foundations that let us state precisely what such code is supposed to do, and prove that it does it. The recurring theme is that the right way to reason about context-dependent systems code is with *modal logic*, in which the truth of a statement depends on the world in which it is evaluated. Systems code is full of such worlds (address spaces, filesystem snapshots, isolation domains, the orderings of a relaxed memory model), and modal abstractions turn that structure from an obstacle into a tool.

Two goals organize this work: *verification engineering* and *verification education*. Verifying a new system should not mean inventing a new logic. My PLOS 2025 paper, *Modal Verification Patterns for Systems Software* (with Colin S. Gordon), identifies recurring patterns (virtualization, sharing, translation): name the system's resource context and the right family of modalities follows, much as design patterns give an engineer a starting architecture. And the people who build systems must be able to read and write the specifications, not only the specialists who prove them: my specNL project turns controlled-English specifications into checked separation-logic proofs. The systems results below are where both goals are tested.

My doctoral work at Drexel University with Colin S. Gordon, funded by an NSF CAREER project on *Contingent Systems*, built the first formal foundations for several core pieces of low-level systems code in this way. I am now a postdoctoral researcher at the Aarhus University Center for Basic Research in Program Verification, working with Lars Birkedal, Jean Pichon-Pharabod, and Zongyuan Liu on program logics for Arm, and at the Center for R&D on Secure Computer Systems (CRADSEC), Research Organization of Information and Systems (ROIS), in Tokyo, working with Taro Sekiyama on trusted execution environments. All of my work is mechanized in the Rocq proof assistant on top of the Iris separation logic framework.

A taxonomy of patterns on resources and their contexts

The PLOS 2025 paper (co-located with SOSR) turns this approach into a multi-dimensional taxonomy, and it is how I choose problems. A modal abstraction for systems code is determined by a few independent design choices: the *resource elements* (the atomic facts, such as a points-to assertion), the *resource context* that collects them (an address space, a snapshot, an isolation domain), the strength of *nominalization* (whether a resource records its context exactly or only up to a bound), and the *context steps*, the operations that move a proof from one context to another. The same dimensions classify existing program logics (Perennial's crash modality, the fence modalities of FSL, the independence modality of NextGen, reference-counting modalities) and my own projects, which differ in their choice along each axis:

Project	Modality	Resource context	Elements	Context step
Virtual memory (OOPSLA'25)	$[r]P$	address space rooted at r	virtual points-to	address-space switch
Copy-on-write filesystem (under submission)	$@[r]P$	snapshot rooted at r	tree nodes, free pages	commit of a new root
Relaxed memory (Aarhus, under submission)	ordering modalities	position in the memory order	tied-to assertions	barriers, acquire/release
Trusted execution (CRADSEC, ongoing)	$@[d]P$	isolation domain d	physical points-to	world switch
Inline assembly (CRADSEC, ongoing)	$\lfloor P \rfloor$	the compiler's view of the machine	register and memory points-to	entering and leaving an asm block

Each row is a project below, and each empty cell in the table (for instance, a context step for caches or the TLB) is a research question. Read as an engineering method, the table is a checklist: a new system contributes a context and its steps, and the modality follows.

Address spaces: memory virtualization

Virtual memory is notoriously hard to verify, because the page tables that describe address translation are themselves stored in memory and accessed through translated addresses; earlier efforts handled a single address space, trusted large amounts of assembly, or reasoned over raw machine semantics. My OOPSLA 2025 paper introduces the modality $[r]P$, “ P holds in the address space rooted at r ,” together with virtual points-to assertions that definitionally mimic hardware address translation. Address spaces can then refer to one another, which makes it possible to verify instruction sequences that switch between them. All of its instruction rules for a fragment of supervisor-mode x86-64 are now proved sound against a model with four-level page tables.

Snapshots as address spaces: a copy-on-write filesystem

The taxonomy predicts that the same abstraction should apply wherever a system switches between containers of resources, and my paper *ModFs: Modal Abstractions for Snapshotting Copy-On-Write Filesystems* (with Gordon, under submission) confirms it. A copy-on-write filesystem keeps many snapshots, each a tree reachable from a root, and a superblock records which root is installed. We verify ModFs, written in Go and translated into Iris with Goose, using modalities in the style of *hybrid logic*: where hybrid logic's satisfaction operator asserts that a formula holds at a named world, our $@[r]P$ asserts that P holds in the snapshot rooted at r , with the names selecting data structures rather than Kripke worlds and the assertions backed by resources. The correspondence with the address-space work is exact: the superblock plays the role of the page-table root register, committing a new root is the disk's half of an address-space switch, and the crash condition is the disjunction of the old and new views of a single assertion. Separate modalities for the node tree, the snapshot map, and the free list let modules interact through relations between contexts rather than ownership bookkeeping.

Relaxed memory and realistic hardware

At Aarhus, with Zongyuan Liu, Freja Marott Crawford, Jean Pichon-Pharabod, and Lars Birkedal, I have worked on verifying the ticket lock of Android's pKVM hypervisor, which relies on mixed-size

accesses, against the relaxed, mixed-size memory model of Arm-A (under submission). Relaxed mixed-size behaviour breaks the reasoning patterns used for locks in stronger models, so we develop AxSL 2.0, a separation logic for Arm with tractable mixed-size reasoning, more expressive proof rules, and *ordering modalities* that hide the lock's synchronization from its clients, so that the lock satisfies a clean, higher-order modular specification. I will continue to advance this logic toward harder targets. The next step is to rebuild the ordering modalities as modal abstractions in the sense of the taxonomy, as was done for address spaces, so that causality holds by construction and deferral becomes a modality of its own, and then to add modalities for coherence and study how they compose.

Transferring proofs between protocols

Kernels are layered, and each layer imposes a protocol. Protocols are naturally modeled as state transition systems, a standard tool for specifying fine-grained concurrent programs, but a proof against one transition system cannot be reused against a similar one, even when one refines or contains the other. In work with Gordon now under submission, I package a bisimulation between transition systems as a *resource morphism*: an embedding of a source protocol into a target under which a Hoare triple proved against the target transfers to the source, with the program text unchanged. The transfer carries across object languages related by a real compiler and across arbitrarily many protocol commits. It runs on copy-on-write and the kernel/user boundary over my x86-64 virtual-memory model, and on a third-party mechanized semantics of the Linux eBPF instruction set, where using that semantics verbatim exposed defects in it. Everything is mechanized with no axioms beyond those of Iris.

Types for safe memory reclamation

My ESOP 2019 paper gave a type system guaranteeing the safe use of read-copy-update (RCU) memory reclamation, the first memory-safety proofs for RCU linked lists and trees without full verification. Its follow-up, with Gordon and under submission, shows that RCU, hazard pointers, epoch-based reclamation, and reference counting are one discipline with three components: a thread has announced that it may touch a node, a node has been retired, and a retired node may now be freed. A single type system for memory safety is that discipline written as syntax. The development is mechanized in Rocq without axioms, and mechanizing the original system uncovered twenty-two defects in it. A Clang-based checker infers the types with no new annotations and runs on the Linux kernel's own RCU list headers.

Isolation and the hardware/software boundary

At CRADSEC, with Taro Sekiyama, I am building a trusted-execution-environment instantiation of Iris on RISC-V. A tower of Iris languages sits on the official Sail RISC-V semantics and reaches up to the GlobalPlatform TEE interface. A modality $@[d]P$ over isolation domains (the security monitor, the normal world, and each enclave) is enforced by the hardware's physical memory protection, and world switches are its context steps. The model is OP-TEE's user trusted applications (UTAs), with its secure-storage application as the first case study. In a related project with Sekiyama and Keisuke Nishimura, a research intern I advised at the National Institute of Informatics in summer 2026, we are giving inline assembly a semantics in which C and assembly share a single program state, with a modality that records what the compiler may assume about the machine, mechanized in Iris.

Verification education and tooling

Two projects, both nearly complete, turn these results into tools for building verified systems and for teaching how to build them. **iIris** is a layered Iris program logic for x86-64 with assembly, macro, and structured intermediate layers, each a separate Iris language whose proof rules compose down through an inline-assembly boundary, with a verified encoder, decoder, and compiler, building on my OOPSLA 2025 development. With the modal abstractions being carried over to xv6 on Sail RISC-V, the aim is a single setting in which an operating system can be both written and proved, with modal abstractions as its specification vocabulary. **specNL** (with Gordon) is the education half of the program: it brings proofs to software engineering through natural language. It is a categorical calculus in which a parse of a controlled-English specification is a separation-logic proof, with an elaborator written and certified in Rocq that turns English into the derivation it then checks. The modal patterns of the taxonomy appear in it as words: from five English tokens specNL reproduces, and proves equal to, a definition from the published virtual-memory logic. A playground site with a step-by-step tutorial and worked examples (crash recovery in Go, an address-space switch on x86-64), and editor support that shows the proof obligation a sentence produces in Rocq, Go, Rust, C, or x86-64, are the materials I bring to a verification course: students state the patterns before they prove them.

Future directions

Several directions follow from the taxonomy. The first is the parts of the machine that are *lazy*, *stale*, and *believed*: demand paging, the TLB, and accessed and dirty bits. In ongoing work, an agreement modality makes stale translations unstateable, so the logic enforces TLB invalidation by construction. The second is caching in its other forms (processor caches, store buffers), each naturally a modality, and the coupling between virtual memory and the filesystem, each of which caches the other. The third, and the most foundational, is how these modalities compose: a single logical framework in which families of modalities are generated from their position in the taxonomy rather than designed one system at a time.

Research group, students, and collaboration

Each of the projects above is a thesis-sized problem, and I would build a group of Ph.D. students around them. Their central questions can also be appreciated through familiar systems concepts before the full formal apparatus is mastered, so undergraduates can take part:

- *Specifying a teaching kernel*. Students write English and then formal specifications for xv6 system calls and test the gap between them; this needs only an operating systems course.
- *Seeing address translation*. Students build a page-table simulator and visualizer, then state and check the invariants that make translation safe.
- *Specifications in English*. Students extend specNL with new controlled-English constructions and check that each sentence yields the proof obligation they intended, combining linguistics, software engineering, and logic.
- *First mechanized proofs*. Advanced students verify small concurrent data structures in Rocq and Iris, building on material from discrete mathematics and programming languages.

specNL also opens a direction beyond computing. Its grammar comes from formal linguistics, and I would like to deepen it with semanticists, so that scientists in other fields can write specifications in their own words and have them checked: a researcher in biology, for instance, stating what a lab

protocol or analysis pipeline must guarantee about the samples, reagents, and data it consumes. Protocols that consume and split resources in context are exactly what separation logic describes. I would seek funding from the NSF's SHF, CNS, SaTC, and CAREER programs, and from DARPA and industry programs on high-assurance software.