Research Statement

My research is motivated by a simple goal: to design systems that are efficient, if necessary complex, and then express, maintain, and establish the rigorous formal foundations for establishing the safety and reliability of these software systems. I cultivate the answers to the following questions:

- What are the common patterns used in the design of efficient systems?
- What are the foundational techniques that allow us to understand these design patterns used in systems?
- How can we specify and verify real systems?
- How can we rigorously understand the behavior of proof for a system in different models and specifications?

Given that one of the main tools for building the software stack is *programming languages*, the answers that I try to propose as solutions to the above questions are usually from the field of *programming language theory*.

In practice, while tackling the problems in the domain of the above questions, I build a rigorous method, such as a type system, semantic models, or program logics for real software systems. It is clear that systems verification has drawn a lot of interest both from the systems field and from the programming languages and formal methods field. However, most of the efforts have traditionally failed to consider the challenging intricacies of real software systems, and the ignorance of these intricacies made them inapplicable to real-world programs and programming languages. For example:

- They ignored the right abstractions by ignoring the inherent nature of the abstracted resource: a virtualized memory address is *agnostic* to the address space it is in. Ignoring this inherent property would result in unnecessarily complicated specifications.
- They avoid the right abstractions, ignoring the inherent contingency of the resource contexts against their semantics: Although virtual memory addresses are agnostic to the address space they are in, they are only valid in a certain address space, and, more importantly, a virtual memory management operation switching address spaces requires us to refer to two address spaces in the specification in which we need assertions expressive enough to have claims such as *"resources valid under the current world"*.
- They assumed the use of simplified machine models, ignoring the virtualization of memory addresses in the existence of shared page tables. This would end up defining the translation of a virtual address in terms of the physical page table address lookup, which is *unsound* against any realistic typical machine model.
- Most of the system verification efforts focus only on completing the proof effort under certain assumptions. They ignore the aspects of maintaining the proof against changing specifications.

In my research, I have developed a range of new formal methods that avoid these kinds of simplifying assumptions, thus making it feasible to compositionally verify realistic software systems that were previously out of reach of any rigorous proof. While doing so, my strategy has always been shaped around finding the right abstractions. It could be highly expressive program logic inspired by modal logic and its derivatives [9, 7], or a type system expressing highly subtle concurrency in a low-level system [23]. At the moment, my work revolves primarily around an NSF Career project I have led during my Ph.D. study called Contingent Systems (https://github.com/Contingent-Systems), in which I have built the first formal foundations for low-level systems code using mainly the *modal abstractions*. This project ties together several strands of work that I have been (and will continue) pursuing in the areas of types, semantics, realistic machine models, concurrent separation logic, modal logic, interactive theorem proving, protocol-based reasoning, specification languages and techniques, and verification patterns for low-level systems. In this research statement, I will first give a brief overview of the projects that make up the project called Contingent Systems, and then conclude with a discussion of several outstanding research challenges that I intend to address in future and ongoing work.

Modal Abstractions for Location Virtualization [24, 22]

vKern: A Program Logic for Understanding Memory Virtualization

Virtual memory management (VMM) subsystems are inside operating system (OS) kernels to virtualize the addresses of memory regions in order to isolate untrusted processes, ensure process isolation, and implement demand-paging and copy-on-write behaviors for performance and resource controls. Bugs in these systems can cause kernel crashes. VMM code is a critical piece of general-purpose OS kernels, but its verification is challenging due to the hardware interface (mappings are updated via writes to memory locations, using addresses that are themselves virtualized). Prior work on VMM verification has either only handled a single address space, trusted significant pieces of assembly code, or resorted to direct reasoning over machine semantics rather than exposing a clean logical interface. In this project, I introduced a modal abstraction to describe the truth of assertions relative to a specific virtual address space, allowing different address spaces to refer to each other and enabling verification of instruction sequences manipulating multiple address spaces. Using them effectively requires working with other assertions, such as points-to assertions relative to a given address space. The program logic (x64Iris) therefore defines virtual points-to assertions, which mimic hardware address translation, relative to a page table root. This project includes a demonstration of the approach with essential and challenging functionality of VMM code in a running kernel, showing that our approach handles examples beyond what previous work [19, 20] can address, including reasoning about a sequence of instructions as it changes address spaces.

x64Iris: A Machine Model Enabling Memory Virtualization

All definitions and theorems for vKern including the operational model of a RISC-like fragment of supervisor-mode x86-64 (based on [1]) and a logic as an instantiation of the Iris framework, are mechanized inside Coq. The soundness proof for x64Iris relies on individual specification proofs given to the instructions against the machine model concerned with memory address virtualization.

Modal Abstractions for Protocol Modularity [22]

specmachine: Logical Framework for Modularity of State-Transition-Systems

While extending the kernel functionality, OS abstraction layers impose certain protocols shaped around proper hiding and fabricating states and operations, e.g., Virtual File System (VFS) enabling different filesystem implementations to co-exist. This project introduces a single-form logical abstraction for specifying these layers behaving as protocols and abstracted as state transition systems (STS)es. STSes are an increasingly popular means of specifying and verifying fine-grained concurrent programs. Unlike more traditional rely- guarantee-based approaches, they allow interference to be conveniently treated as a resource, transferred between threads, or even stored in other resources. However, existing STS systems [10, 45, 46, 43, 42, 17, 21, 36, 40, 34] leave the traditional Hoare-style rule of consequence weaker than before. Code involving STSes is verified against one particular

STS, making it unusable with other similar transition systems, even when one is contained in the other. This project extends the notion of entailment for STS-based logics to incorporate a form of bisimulation (as on Kripke structures [14]) between STS systems into an extended rule of consequence. This extension allows significant additional code reuse, as I demonstrate through a number of examples. The formulation in specmachine is abstract and relies only upon the existence of a protocol state update rule in the logic, so the soundness proof implies that the new rule introduced is admissible in a range of recent logics, such as CaReSL [45], and Iris [18].

Verification Patterns for Systems

Understanding the practice of *proof engineering* effort is as important as finding the right abstractions (vKern and vAmplify projects) in specifications or giving a foundation for proof reusability (specmachine project). I started a project to achieve the following goals:

- ○ giving a framework to the patterns in the system verification effort (step taken [25]).
- ○ giving a logical framework to design proof layouts with required specification instances. (not started)

Identifying Patterns via Nominals, Resources, and Contexts [22, 25]

Building program logics has drawn a lot of attention for low-level systems over the years. Most of the time, what makes them unique is their custom-tailored approach to the challenging aspect of the system [48, 11, 12, 5, 6, 47]. "Modal Verification Patterns for Systems paper argues the foundations of how modalities, in the sense of modal logic, should be a go-to approach when specifying and verifying low-level system code. This paper explains how the concept of a resource context helps guide the design of new modalities for verification of systems code and justifies the perspective by discussing prior systems that have used modalities for systems verification successfully, arguing that they fit into the verification design patterns articulated, and explaining how this approach might apply to other systems verification challenges.

A Logical Framework for Unified Families of Modalities for Systems

The next step in this project is to give abstract characterizations to each family of modalities described in *patterns* paper [25] and a powerful form of bisimulation between protocols described in my thesis [22]. Ideally, this would be a single set of rules and parameters for each modality (a set for location virtualization and a set for caching) and a single rule for treating subtyping among protocols. (not started)

Capturing Contingency at Type Assertions: rcutypes [22, 23]

Deferred memory reclamation algorithms such as Hazard Pointers [33] and Read-Copy-Update [29, 28, 30, 35, 27] have drawn much attention because using the machine resource as much as possible is one of the main concerns of a concurrent system programmer. These techniques rely on the concept of a grace period: nodes that should be freed are placed on a deferred free list, and all threads obey a protocol to ensure that the deallocating thread can detect when all possible readers have completed their use of the object. This provides an approach to safe deallocation, but only when these subtle protocols are implemented correctly. Several recent program logics [44, 15, 32] have focused on this challenge. There are many steps involved in rcutypes project, some of which have already been completed.

- ○ Static Type System: rcutypes is a static type system to ensure correct use of RCU memory management: nodes removed from a data structure are always scheduled for subsequent deallocation, and nodes are scheduled for deallocation at most once. Our type system enforces *locality* on heap mutations—one heap node at a time—to preserve the well-formedness properties of data structures, including the well-known *acyclicity*. (finished)

- Semantic Soundness Proof: As part of our soundness proof, we give an abstract semantics for RCU memory management primitives that captures the fundamental law of RCU [23]. (finished)
- Experiment: rcutypes are used to verify a singly linked list and a binary search tree for the first time. (finished)
- Proof of Abstract RCU Model Abstracting other RCU Models: In rcutypes, an abstract model of RCU primitives is given. There are many different deferred memory reclamation models, which, at this point, are believed to be abstracted by our abstract RCU model. This needs a proof. (not started)
- Implementation: rcutypes are going to be implemented for clang. (not started)

Ongoing Projects

vAmplify: Modal Abstractions as Summarizations for Universal Facts

The modalities presented in the virtual memory manager verification (vKern project) speak about the contingency from a particular local view: virtual points to assertions that are *relatively* valid *under* an address space with a unique *root* address. With the aim of looking for another application setting for this type of modal principle, ModFs is implemented: a fully functional, modern (most importantly including chunking, dynamically sized nodes and splitting, and implemented in 5K lines of Go source code) copy-on-write filesystem (CoW). Semantically, in ModFS (as in other CoW filesystems [**?**, 4, 37, 38, 13]) requires updates to those metadata blocks that previously referred to the old block locations to instead refer to the newly allocated blocks. This must be repeated until the allocation/update of the new chunks percolates up to the root (including the root), through a process called *write-amplification*. This project includes multiple steps:

- Operational model for ModFs: Translation into an operational model using Goose [6] is required so that we can specify this operational model with the modal reasoning principles built on top of Iris [18]. (finished)
- Proving subcomponents: Modern features such as chunking, freelists, etc., need to be specified and proven. (finished).
- $[root]P$ for snapshots: Like VMMs that navigate between different address spaces, a CoW filesystem switches from the old to the new root, yielding consistent views of the filesystem at different points in time, each commonly referred to as a *snapshot*. A CoW filesystem treats the snapshot abstraction as VMMs treat the address-space abstraction, a container of the relevant resources. Like VMMs loading a process's memory mappings to be the current view of the memory by setting the control register to show the unique root address of the page-table tree of the process, the filesystem atomically switches between consistent on-disk states by updating the oldest so-called *super* chunk at a fixed location on disk. Super chunks are the first two blocks of the disk dedicated to keeping the state of the file system, which is composed of some metadata on the file system and, most importantly, the root chunk of the file system index tree. (in progress: *write-amplification* proof with axioms on boundaries *nodes* and its backing *chunks* is finished.)

Proofs for Rust against Virtualizations

The Rust programming language has drawn so much attention amongst the system programmers because its type system encodes memory safety invariants. Consequently, it is claimed that a type-safe Rust program is claimed to be safe w.r.t. the memory safety invariants defined in the type safety. As part of RustBelt [8, 16], there has been a line of papers with proofs mostly showing a subset of Rust type *sound*. However, these proofs are made against machine models that ignore memory address virtualization. This project is aiming to put the proofs, both client and type-soundness, under investigation to obtain a more in-depth understanding about how rigorous the properties Rust type ensures in tandem with RustBelt proofs holding this claim. At this stage, I am extending RustBelt's memory model with a virtual memory address translation project. As a next step, I will try to do the proofs already done against the machine model, ignoring memory virtualization. A proof effort against x86 emitted from Rust code would be a straightforward step to take, but, *ideally*, I should lift the proof effort in x64Iris to Rust intermediate language.

Foundations for Drawing Verified Systems [22]

Explaining both the specification and the proof of a system is as important as applying them to get the system verified. This was a need for me when I wanted to write and explain intricate protocols in the specmachine project. Then I developed an informal but systematic way of drawing protocol-based reasoning and bisimulations amongst them. These graphs represent the STS specifications, including *rely/guarantee* relations and bisimulations embedding these relations in one state machine to another, both of which are instances of Iris resource algebras. In this work, I explore giving formal foundations first to the systemic diagrammatic approach I used in specmachine project, then extend it. In this regard, string diagrams [39, 3, 2], have been the foundation for different types of semantics but have never been used as a tool to draw real system specifications whose specification and proof can reside in separation logic like Iris.

Future Projects

Modal Abstractions in Interaction: Specifying the Marriage of Virtual-Memory and Filesystem

VMM and the file system are tightly coupled. The file contents are buffered in pages mapped to the kernel address space when accessed via system calls, in pages mapped to process address spaces when accessed directly (via mmap), and in physical pages mapped in both ways when accessed in both ways: using virtual addresses aliasing to keep the two views in sync. In FreeBSD [31] and Solaris [26] (both derived from 4.3 BSD), and others, the primary cache of the filesystem is actually a borrowed chunk from the free list of the VMM, and the VMM can reclaim clean (synchronized with disk) portions of the filesystem to satisfy allocation requests. Filesystems use a VMM-provided cache of file contents, which is synced to disk by the filesystem at the request of the VMM (pageout daemon), a subtle circular dependency, since the file may be a general swap file. I think that specifying this circular dependency is going to be interesting, first as a system verification challenge and then for the potential for constructing interesting reasoning principles.

Modal Abstraction on More Elaborative Machine Models: Caching

Caching is another recurring concept across many parts of an OS kernel (and other systems software). The plan is to focus our attention on 4 forms: file system caching, TLB, processor data cache, and hardware store buffers that lead to the weak memory model present on multicore x86-64 processors (a variant of Total Store Order: TSO [41]). This establishes a setting to explore the key question in modal logics: how different modalities interact. Caching modality for TLB obviously needs to interact with another type of modality, i.e., address translation. The data cache and store buffer modalities interact: flushing the store buffers moves data into the traditional memory hierarchy.

Design Principles for a Systems Language

Based on my expertise in vKern and vAmplify projects, I have built understanding and intuition on the essential abstractions needed in a language for programming a general-purpose OS kernel. From the language design point of view, how much of these low-level logical constructs (including modal ones and more) can be lifted into the language level or be part of the intermediate representation?

Publications

AMD. AMD64 architecture programmer's manual, volume 2: System programming, January 2023. Revision 3.40.

Filippo Bonchi, Alessandro Di Giorgio, and Elena Di Lavore. A diagrammatic algebra for program logics. In Parosh Aziz Abdulla and Delia Kesner, editors, *Foundations of Software Science and Computation Structures*, pages 308–330, Cham, 2025. Springer Nature Switzerland.

Filippo Bonchi, Alessandro Di Giorgio, and Alessio Santamaria. Deconstructing the calculus of relations with tape diagrams. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571257.

Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

Tej Chajed. *Verifying a concurrent, crash-safe file system with sequential reasoning*. Ph.d. dissertation, Machetutes Institute of Technology, Cambridge, MA, 2022. Available at https://dspace.mit.edu/handle/1721.1/144578.

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 243–258, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3341301.3359632.

Kaustuv Chaudhuri, Joëlle Despeyroux, Carlos Olarte, and Elaine Pimentel. Hybrid linear logic, revisited. *Mathematical Structures in Computer Science*, 29(8):1151–1176, 2019.

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371102.

Joëlle Despeyroux and Kaustuv Chaudhuri. A hybrid linear logic for constrained transition systems. In *Post-Proceedings of the 9th Intl. Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 150–168. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *European Conference on Object-Oriented Programming*, ECOOP 2010, pages 504–528. Springer, 2010.

Marko Doko and Viktor Vafeiadis. A program logic for c11 memory fences. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, page 413–430, Berlin, Heidelberg, 2016. Springer-Verlag. `doi:10.1007/978-3-662-49122-5_20`.

Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with fsl++. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings*, page 448–475, Berlin, Heidelberg, 2017. Springer-Verlag. `doi:10.1007/978-3-662-54434-1_17`.

Dave Hitz, James Lau, and Michael A Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, pages 10–5555, 1994.

George Edward Hughes and Max J Cresswell. *A new introduction to modal logic*. 1996.

Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. Modular verification of safe memory reclamation in concurrent separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. `doi:10.1145/3622827`.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. `doi:10.1145/3158154`.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2676726.2676980`, `doi:10.1145/2676726.2676980`.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, 2015.

Rafal Kolanski and Gerwin Klein. Mapped separation logic. In Natarajan Shankar, Jim Woodcock, editor, *Verified Software: Theories, Tools and Experiments*, pages 15–29, Toronto, Canada, oct 2008. Springer.

Rafal Kolanski and Gerwin Klein. Types, maps and separation logic. In S. Berghofer, T. Nipkow, C. Urban, M. Wenzel, editor, *International Conference on Theorem Proving in Higher Order Logics*, pages 276–292, Munich, Germany, aug 2009. Springer.

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *European Symposium on Programming*, pages 696–723. Springer, 2017.

Ismail Kuru. *Modal abstractions for operating system kernels*. PhD thesis, Drexel University, 2025. doi:10.17918/00010934.

Ismail Kuru and Colin S. Gordon. Safe deferred memory reclamation with types. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 88–116. Springer, 2019. doi:10.1007/978-3-030-17184-1_4.

Ismail Kuru and Colin S. Gordon. Modal abstractions for virtualizing memory addresses, 2024. URL: https://arxiv.org/abs/2307.14471, arXiv:2307.14471.

Ismail Kuru and Colin S. Gordon. Modal verification patterns for systems, 2025. URL: https://arxiv.org/abs/2506.01719, arXiv:2506.01719.

Richard McDougall and Jim Mauro. *Solaris internals: Solaris 10 and OpenSolaris kernel architecture*. Pearson Education, 2006.

Paul E. McKenney. N4037: Non-transactional implementation of atomic tree move, May 2014. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4037.pdf.

Paul E. McKenney. Some examples of kernel-hacker informal correctness reasoning. Technical Report paulmck.2015.06.17a, 2015. URL: http://www2.rdrop.com/users/paulmck/techreports/IntroRCU.2015.06.17a.pdf.

Paul E. Mckenney, Jonathan Appavoo, Andi Kleen, O. Krieger, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *In Ottawa Linux Symposium*, pages 338–367, 2001.

Paul E. Mckenney and Silas Boyd-wickizer. Rcu usage in the linux kernel: One decade later, September 2012. URL: http://rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf.

Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The Design and Implementation of the FreeBSD Operating System, Second Edition*. Pearson Education, 2014.

Roland Meyer and Sebastian Wolff. Decoupling lock-free data structures from memory reclamation for static analysis. *PACMPL*, 3(POPL):58:1–58:31, 2019. URL: https://dl.acm.org/citation.cfm?id=3290371.

Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004. URL: http://dx.doi.org/10.1109/TPDS.2004.8, doi:10.1109/TPDS.2004.8.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming*, ESOP 2014, pages 290–310. Springer, 2014.

Lai Jiangshan Paul E. McKenney, Mathieu Desnoyers and Josh Triplett. The rcu-barrier menagerie, November 2016. URL: `https://lwn.net/Articles/573497/`.

Azalea Raad, Jules Villard, and Philippa Gardner. Colosl: Concurrent local subjective logic. In *ESOP*, pages 710–735, 2015.

Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992. URL: `http://doi.acm.org/10.1145/146941.146943`, `doi:10.1145/146941.146943`.

Ralph Sarkis and Fabio Zanasi. String diagrams for graded monoidal theories with an application to imprecise probability, 2025. URL: `https://arxiv.org/abs/2501.18404`, `arXiv:2501.18404`.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015). ACM*, 2015.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *European Symposium on Programming*, ESOP 2014, pages 149–168. Springer, 2014.

Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 110–120, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2737924.2737992`, `doi:10.1145/2737924.2737992`.

Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 110–120, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2737924.2737992`, `doi:10.1145/2737924.2737992`.

Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 377–390, New York, NY, USA, 2013. ACM. URL: `http://doi.acm.org/10.1145/2500365.2500600`, `doi:10.1145/2500365.2500600`.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. Gps: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of the 2014 ACM International*

Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, pages 691–707, New York, NY, USA, 2014. ACM. URL: `http://doi.acm.org/10.1145/2660193.2660243`, `doi:10.1145/2660193.2660243`.

Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. The nextgen modality: A modality for non-frame-preserving updates in separation logic. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '25, page 83–97, New York, NY, USA, 2025. Association for Computing Machinery. `doi:10.1145/3703595.3705876`.

Andrew Wagner, Zachary Eisbach, and Amal Ahmed. Realistic realizability: Specifying abis you can count on. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. `doi:10.1145/3689755`.