Modal Verification Patterns for Systems Software

Ismail Kuru
ik335@drexel.edu
Drexel University
Philadelphia, PA, USA

Colin Gordon
csg63@drexel.edu
Drexel University
Philadelphia, PA, USA

Abstract

Although they differ in the functionality they offer, low-level systems exhibit certain patterns of design and utilization of computing resources. In this paper we examine how modalities have emerged as a common structure in formal verification of low-level software, and explain how many recent examples naturally share common structure in the relationship between the modalities and software features they are used to reason about. We explain how the concept of a *resource context* (a class of system resources to reason about) naturally corresponds to families of modal operators indexed by system data, and how this naturally leads to using modal assertions to describe *resource elements* (data in the relevant context).

CCS Concepts: • **Theory of computation → Modal and temporal logics**; **Logic and verification**; • **Software and its engineering → Operating systems**.

Keywords: separation logic, modal logic, verification, operating systems

1 Introduction

Although they differ in the functionality they offer, low-level systems exhibit certain patterns of design and utilization of computing resources. Systems software, in general, interfaces with an underlying computing substrate such that any software system at any higher level in the software stack can (at least indirectly) utilize the resources of the machine.

The last layer of software before the hardware is naturally critical to the correctness of an overall system, as essentially all software built on top of it assumes its correctness. And because hardware is complex and highly diverse, the implementation of those lowest layers of the software stack is

typically intricate and naturally error-prone, despite how critical its correctness is. Typically systems software has, as a primary focus, the task of *abstracting* from hardware details to simplify the construction of higher layers of the stack.

Interacting with computing resources is the essential point which shapes the design of low-level systems software. Exploiting these design choices has been an important field of study. This survey examines these common patterns in systems software, and their relationships to modalities. We believe that certain properties of modalities enable us to understand and address the verification challenges of these systems by tailoring specification and proof to the same patterns used to design the systems.

Contributions We argue that modal abstractions can be used to identify and abstract system verification challenges. We justify our perspective by discussing prior systems that have successfully used modalities for system verification, arguing that they fit into the verification design pattern we articulate, and explaining how this approach might apply to other systems' verification challenges.

- **Identifying System Verification Challenges**: We start with identifying common patterns in system verification: *virtualization, sharing, and translation.*
- **Introducing Modal *Resource Contexts***: Then we discuss the concept of *resource* which has already been an essential concept in the design of systems. Inspired by the concept of resource in systems, we define what a resource and its context are in the modal abstractions. For example, the *transaction* abstraction, which behaves like a container for in-memory dirty disk pages, can be considered a resource context for any dirty in-memory disk page resource associated with it.
- **Introducing *Nominals for System Resources***: Existing logics for systems software must frequently identify individual resources within a given context — nominalization (in the sense of hybrid logic) enables identifying a resource in its context. For example, a filesystem transaction is a context of resources for in-memory updated disk blocks which have not yet been saved to disk. The transaction identifier is used to associate a transaction with a disk block to persist so that, in case of a crash while persisting updated disk-blocks, the filesystem can rollback the *already persisted* disk-blocks of the transaction, and restore the previous consistent disk state. We discuss other examples to contextualize the existing systems surveyed.

- **Taxonomy of Current Modal Approaches in System Verification**: Based on these concepts, we summarize contemporary verification efforts using modal abstractions. We choose them from different domains, for example, reasoning about weak memory models and storage persistence [7–9, 11, 15, 16, 21, 37, 53, 54] and using different programming logics such as Iris [32] and Dafny [39] because we would like to justify that our *definitions* are neither domain nor programming logic dependent.

2 Background on Modal Logics for Programs, and Low-Level Systems Software

This section briefly recalls related concepts from modal logic of particular relevance to our taxonomy (Section 2.1), and enough general background on the systems software concepts organized by our taxonomy for the connections to modalities to be clear (Section 2.2).

2.1 Hybrid Logic, Dynamic Logic, and Nominals in Program Logics

Dynamic Logics as Program Logics. We believe that for the kinds of informal reasoning and the sorts of data structures discussed in the previous section, ideas from modal logic are a promising approach to formalization and verification. Broadly speaking modal logics incorporate *modal operators*, which take as arguments a proposition expected to be true in another time [44], place [3, 19–21, 40], circumstance or point-of-view [26, 29], and result in a proposition true in the *current* time, place, or circumstance at which the truth of the modal operator is being evaluated. Classic examples include modal necessity $\Box P$ describing that P is *necessarily* true, $G(P)$ meaning P is true *globally* (i.e., forever from this time onwards in temporal logics), or $K_i(P)$ describing that a particular participant i *knows* that P is true. The latter is an example of *multimodal* logic, where there is an family of modalities (modal operators) parameterized by some dimension of interest (there, participants).

Dynamic logic [45], a (multi)modal logic variant of weakest preconditions [2, 14, 23] which works with modalities of the form $[p](P)$, which states that *in the current program state, if* program p is run then afterwards P will hold (modulo non-termination).

Hybrid Logic. Hybrid logics [3, 4, 6, 19, 20] are a class of modal logics that add two primary new concepts to a logic. Nominals $\ell \in$ Loc uniquely identify points in a model (e.g., a particular state in the state space the logic is used to reason about), so there is exactly one point in the model where a nominal Loc (as a formula) is true. Satisfaction operators are modal operators indexed by nominals, which enable claims about the truth of another proposition at the

point in the model identified by a chosen nominal. Traditionally $@_\ell \varphi$ asserts that φ is true in the state identified by ℓ. Conceptually, given our context, we might like l to indicate truth in a particular data structure ℓ. However, most prior work on hybrid logics works in classical logics rather than a substructural setting like separation logic. Newer work [7–9, 11, 15, 16, 37, 53, 54]:w surveyed here adapts this idea to deal with different structures specified in separation logic.

Nominalization. Some classes of assertions benefit specifically from *naming* the explicit conditions where they are true (as opposed to simply requiring them to be true *somewhere* or *everywhere* as in the most classic modal operators). Because these named conditions strongly resemble the nominals of hybrid logic, we refer to the general idea of naming circumstances explicitly as *nominalization*, even though not all of the examples we discuss are actually hybrid logics.

An example use of state naming explicitly in the assertion appears in program logics with usage protocols (e.g. state transition systems resembling typestate [18, 52], which ensure the operations applied to some data occur in a specified order, and that invariants for various conceptual states of the data — such as a file being open or closed — are respected). Protocol assertions are *annotated with the name of the last (abstract) state at which the protocol is ensured*. A classic application of this idea to systems is Halpern et al.'s work adapting modal logics of knowledge to deal with distributed systems [24, 25, 27]. In most of that line of work, $\mathcal{K}_a(P)$ indicates that the node a in the system *knows* or *possesses knowledge of* P (for example, a Raft node may "know" a lower bound on the commit index). Alternatively, a modality $@_i(P)$ may represent that P is true of/at the specific node i [21] (e.g., that node i has physically stored a certain piece of data to reliable storage). These permit capturing specific concepts relevant to the correctness (and reasoning about correctness) of a certain class of systems, involving facts about specific named entities in the system.

In each of these cases, the fact that these facts are described using a modality M with the core modal property that if $P \Rightarrow Q$, then $M(P) \Rightarrow M(Q)$. This means, for example, that if a process p knows that the commit index is greater than 5 (e.g., $\mathcal{K}_p(\text{commitIndex} > 5)$) no extra work is required to conclude that the node knows it is greater than 3 (i.e., that $\mathcal{K}_p(\text{commitIndex} > 3)$), because this follows from standard properties of modal operators as described above. In contrast, if verification instead used a custom assertion

$$\text{minCommitIndex}(5)$$

to represent the former knowledge, one would need to separately provide custom reasoning to conclude

$$\text{minCommitIndex}(3)$$

As our survey shows, the value of the modal view (which grants both intuitive specifications and natural reasoning

principles) is becoming apparent in systems software verification.

2.2 Virtualization in Systems Software

One of the most common forms of abstraction provided by systems software is *virtualization*, which abstracts the relationship between conceptual and physical computing resources. Upper-levels of the software stack work with these conceptual — *virtual* — resources, while the lower levels of systems software deal with the translation of requests expressed in terms of virtual resources into operations on physical resources.

Operating System (OS) kernels virtualize memory locations and quantity (via virtual memory and paging [13]). Distributed language runtimes may virtualize addresses, even when processes may migrate across machines [31]. Filesystems virtualize locations on disk [5, 30, 49, 50]. Programs built on top of the corresponding systems software layer work logically at the level of these virtualized resources, and it makes sense to specify the systems software directly in terms of those abstractions.

Access by Translation. Accessing virtualized resources via translation is a common way to virtualize notions of location (e.g., virtual memory addresses, inodes or object IDs instead of disk addresses [5, 30, 49, 50]). B-trees, page tables, and related structures both behave like maps, when the corresponding physical resources exist as such just in a different location. Control over the lookup process (e.g, handling the case of a missing translation entry) allows for additional flexibility, such as filling holes in sparse files, or demand paging (both from disk or lazily populating anonymous initially-zeroed mappings). Although the realization of these maps may differ from system to system based on the context (and sometimes hardware details), they are semantically — logically — partial maps, worth treating as such in verification.

2.2.1 Virtualization of Memory Locations.
A typical general purpose computer virtualizes memory resources in RAM. A program asking for a memory unit from an operating system kernel is served with a memory address that is *virtualized* such that it may be *mapped to* a physical RAM location or not. As our informal definition suggests, the common technique for *virtualization* of memory locations in OS kernels relies on *translating* (*mapping*) virtual addresses into physical ones. The conceptual address translation map is implemented with processor's page table trees as shown in Figure 1. In this Figure, we see a typical 4 levels of page tables, a virtual address on the top indexing into the different levels of the page table tree (the larger rectangles along the bottom are blocks of memory which serve as nodes of the tree along a particular lookup path). Fabrication of virtual addresses out of limited physical addresses is provided with entries in the page table tree where each points to a physical
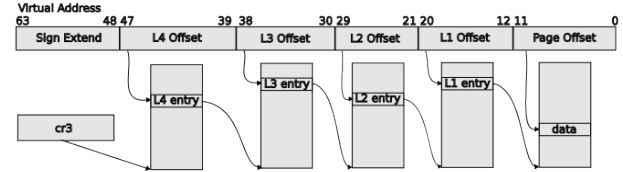

Figure 1. x86-64 page table lookups.

address aligned to 4KB (4096 byte) boundaries. An address translation requires traversal of a series of tables starting from level 4. The traversal ends in the level 1 page table with the final lookup to the actual page of physical memory holding the requested data, and the low-order 12 bits being used to index into this page.

A page table tree is referred to as a (virtual) *address space*, as it presents an abstraction of how different memory addresses could be related in space. This functionality is used by operating systems to present fictional collocation or separation of resources: a range of logically (virtually) contiguous memory addresses may be translated to a range of physically disparate locations in actual memory. The address space then acts as a kind of container for virtualized memory resources tied to that container.

A typical OS manages multiple address spaces at once. Each program is associated with a unique page table tree, which is stored in a specific hardware register (`cr3` in Figure 1). Using different mappings, which map only disjoint portions of physical memory in the map's codomain, is how the OS ensures memory isolation between processes.

2.2.2 Virtualized Resources in Filesystems.
Another important computer resource that is *virtualized* for access is disk blocks. Filesystems are software components that abstract disk access on behalf of normal programs Similar to how VMMs using page table trees to fabricate more memory addresses than what actually exists, filesystems use indexing maps for fabricating more *virtual disk page* addresses which are conventionally called *disk pages*, to be distinguished from physical *disk block* addresses. When a regular program accesses a file, it requests a specific logical offset from the start of that file's contents. As with virtual memory, the filesystem may map adjacent logical blocks of a single file to physically disparate parts of the disk (or even to different disks), though this translation mapping is implemented entirely in software (as opposed to the hardware cooperation that exists for memory virtualization).

Filesystems not only handle the address translation from a disk-page to a disk-block but also need to handle different *modes* of computing where to find disk resources: a disk block may live on a disk in stable storage, or the most up-to-date version of a block may live only in memory (for the filesystem to write to disk at a later time, batched for efficiency). To handle the consistency of the disk resources in different

modes of execution, filesystems employ different policies when indexing the disk resource. Which data-structure is going to be used to implement address translation from disk-pages to disk-blocks? What is going to be the *update policy* to the indexing map implementing address translation, which defines the *consistency policy* for the disk resources? Different filesystem implementations give different answers to these questions.

While exceptions exist [5, 30, 49, 50], most filesystems will continue to store a given block of a given file in the same physical disk block for as long as the file exists. These filesystems with in-place updates need to implement some form of *journalling* [10, 22] to ensure that a crash partway through an in-place update can be rolled back using an undo journal that records the original contents before in-place modification (so original contents could be restored after a crash) or completed using a write-ahead journal that first writes new data in unused space before *also* updating in place (so the journal entries could be re-applied after a crash). Because disk blocks associated with a file could now be in many states (on-disk, journaled but not updated in place, or both) the specification of these systems is complex. And because some states are only possible after a system failure, each line of filesystem code has two conceptual post-conditions: one referring to the successfully stored disk pages from a disk page container (e.g, a disk tree) , which can be uniquely identified by the last successful commit identifer (e.g., a transaction identifier or snapshot identifier), as the current view of the disk in the specification for continued execution, and one referring to the previous view of the disk in the specification for the stable state established in the event of a hardware failure immediately after that operation.

2.2.3 Reference Counting Memory Reclamation.
It is safe to reclaim memory (for later reuse) when that memory allocation will no longer be used by the client program. Since that property itself is undecidable, sound heuristics are used instead. A common proxy for the program never accessing an allocation again is to check that there is no access path from data a program can trivially access (e.g., local variables stored on the stack) to a memory allocation — that is, when the allocation is no longer *reachable*. In that case the program *cannot* access the memory again, so the memory location can be reclaimed *soundly* — i.e., without violating any access through a reference to it.

To reclaim the allocated memory location that is no longer being accessed, many systems use a well-known method called *reference counting* in which each memory allocation includes not only its data but also a counter of how many pointers to that memory exist in the rest of the program. Any time a new reference to the allocated memory is created, the counter is incremented, and every time an existing reference is destroyed (by overwriting or dropping) the counter

is decremented.[1] These reference counts are a sound (but incomplete) way to approximate reachability: if a decrement of a reference count decrements the count to 0, no other references to that memory exist in the program, and the memory can be reclaimed.

Reasoning about the correctness of such code requires reasoning about these reachability relations and reference counts, relative to specific stack frames.

2.2.4 Weak Memory Models.
Multicore and multiprocessor systems do not directly write every memory update back to memory (or even a coherent hardware cache) individually, as this would be extremely slow. Instead, they effectively buffer reads and writes to and from memory locally within each CPU core [43][2] so that operations can be partially reordered and batched to improve performance. For most code — where the different CPUs are doing independent work — this is transparent to software. But fine-grained concurrent systems code which has multiple CPU cores simultaneously coordinating access to shared memory must reason about these (effective) bufferings and reorderings, whose details vary across CPU architectures.

Most of this coordination comes through reasoning about what *will be true after local operations are published to memory*, or what *will be true after remote operations become visible locally* [12, 15, 51]. Each of these flavors of contingent reasoning depend on use of certain *fence* or *memory barrier* operations, which block execution of a CPU until either all incoming or outgoing memory transactions are complete.

2.2.5 Actor Model.
An alternative approach to addressing the shared memory concurrency complication, i.e., data races, is to use actors [1, 28] by eliminating shared mutable state and enforcing updates on exclusive local mutable state via exchanging immutable messages among processes. This is a way of sequentializing the concurrent interleaved execution. Often correctness of one actor depends on *knowledge* about other actors. In a typical consensus algorithm, for example, an operation is committed if a quorum of nodes reach an agreement on it.

Therefore, as expected, the specification of an actor in the current view would refer to the other actors' states and has to validate that the shared knowledge amongst actors is preserved [21].

3 Contingency Decomposition of a System

In this section, we argue the position that *modalities* should be a go-to approach when specifying and verifying low-level systems code. We explain how the concept of a *resource context* helps guide the design of new modalities for verification

---

[1]This may be done using library functions for a reference counting system, or the operations may be inserted by a compiler.
[2]This is a layer below the better-known L1 cache, which is coherent across all CPUs in the system.

of systems code, and we justify our perspective by discussing prior systems that have used modalities for systems verification successfully, arguing that they fit into the verification design pattern we articulate, and explaining how this approach might apply to other systems verification challenges.

To explain our ideas in the general systems understanding, we briefly recap some of the background and themes our ideas build on again, casting them in a certain way to bring out the relevance of our philosophy.

3.1 Encoding Modalities in Program Logics

Program logics which would like to enjoy modal propositions need to have an underlying model that allows encoding of the propositions. A Kripke model can be roughly seen as a triple of set of worlds (e.g., program states), a binary relation between worlds (e.g., transition relations between states) and a state interpretation function. For a modality M, $M(\phi)$ is true in one state/world w if states related to w by the binary relation satisfy P. Whether this is all related states, or just one, or something else varies depending on what M is meant to model, but the modality is always tied to the relation of interest. For multimodal logics there are multiple relations. For example, for Dynamic Logic, there is a modality for each program, and the relation for running program "command" (in old program logic parlance) C is the relationship between states before and after executing C.

Encoding Hoare Triples. Following Pratt's original representation [45], we can encode a Hoare triple $\{P\}C\{Q\}$ (asserting that if P holds in some state, then executing C will yield a state where Q holds) as $P \rightarrow [C](Q)$. However, for this to hold for a specific programming language and assertion language, the modal representation requires a semantics in terms of an underlying model.

Conventionally, a Kripke model with its constituent must be picked. Gordon provides a Kripke model for an actor-based hybrid modal logic [21], in which the transition relation is indexed by a particular command and a particular actor, relating pre- and post-states of a specific actor referenced by a nominal executing the command. Gordon encodes this hybrid logic as a library in the Dafny program logic [39] with the help of two state invariants and rely-guarantee assertions checking the stability of invariants. Wagner et al. take a similar approach to the design of their separation logic and utilize a Kripke model for reference-counted memory reasoning principles [54].

On the other hand, program logics such as Iris [32] utilizes Kripke models for their separation algebras (a collection of CMRAs as an algebraic cousin of a BI-algebra [42]), modeling modalities for both heap extension and step indexing [41]. However in general, many modalities can be given proof-theoretic semantics directly within a logic. This is the case for Iris's weakest precondition modality (which is defined within the Iris logic itself), as well as for a wide range of other

modalities in the literature [47]. These modalities are typically multi-modal, with a form of roughly $M \overline{ctxt} \phi \triangleq \ldots \rightarrow \phi \overline{ctxt}$, such that the evaluation of the modality incorporates the $\overline{ctxt}$ arguments into the evaluation of ϕ.

Shallow Modalities in Separation Logic. In a substructural setting [35, 46] the implication $\rightarrow$ is typically replaced with substructural implication $-\!*$. This definition also reveals a subtlety about the denotation of assertions ϕ — they are effectively *predicates* of some kind, functions from some possible world into some logical algebra such as a BI-algebra [42]. Taking this interpretation literally gives rise to a convenient way to embed new modalities in Iris in a way well-suited to many use cases. If B is a type with a BI-algebra structure, then any space of predicate functions (e.g., val $\rightarrow B$ as a simplification) also carries a BI-algebra structure, lifted from B, often called the *pointwise lifting* of the algebra. A number of projects for systems software verification [7–9, 11, 15, 16, 37, 53, 54] now take advantage of this, explicitly or implicitly lifting the BI-algebra structure from Iris itself to Iris predicates used as assertions within modalities. Connecting this to Iris's typeclasses for interactive proofs [34, 36] then permits working within the embedded logic just as if working in Iris.

In addition, the modal definitions themselves may represent ownership of resources in a substructural setting [17, 33, 46]. This allows modalities to simultaneously represent ownership of certain resources used for interpreting the modality itself, as well as whatever resource ownership is implied by the modal argument.

3.2 Decomposing a System into its Constituents *Contingently*

Many existing program logics for system verifications have a common structure, which maps to modalities with a couple extra dimensions of design. We summarize our discussion of these logics in Table 1. We discuss, based on examples, common aspects of how we intuitively think about correctness of systems code in many contexts, articulate those pieces, and call out the commonalities across a range of systems.

3.3 Resource

Consider first the address space abstraction in an OS kernel. An address space of a process is a container of virtual addresses referencing data in memory. One would expect to have *points-to* assertions from separation logic to specify *ownership* of a memory reference pointing to some data. In the classic setting [48], such a points-to assertion $x \mapsto v$ asserts ownership of memory location x, and that the contents of that memory location are v. But that ownership is relative to a specific address space — a specific container. We tend to think directly about what is true *in an address space*, with the simplest piece being an association between a virtual

Table 1. Modal Decomposition of Program-Logics.

Modality	Context	Elements	Nominalization	Context Steps
Post-Crash [7–9]	$\diamond\, P$	$\ell \mapsto_n^{\overline{Y}} v$	Strong	Crash Recovery
NextGen [53]	$\overset{t}{\hookrightarrow}\, P$	Own (t(a))	Strong	Determined Based on the Model*
StackRegion* [53]	$\overset{ICut^n}{\hookrightarrow}\, P$	$\boxed{\text{n}}\ \ell \mapsto v$	Strong	Alloc and Return to/from stack
Actor [21]	$@_\iota\, P$	Variable values	Weak	Send Message
Memory-Fence [11, 15, 16]	$\triangle_\pi$ and ∇_π	$\ell \mapsto v$	Weak	Fence Acquire and Release
Address Space [37]	$[r]P$	$\ell \mapsto v$	Weak	Address Space Switch
Ref-Count [54]	$@_\ell\, P$	$\ell_1 \mapsto v$	Weak	Allocating, Dropping and Sharing a Reference

*The StackRegion Modality is an instance of NextGen (called the Independence Modality in [53]).

address and the data it points to. We call the simplest piece, in this and other examples, the *resource element*:

Definition 3.1 (Resource Elements). The simplest atomic facts we want to work with in a particular setting, specific to that setting.

By definition, the resource elements are specific to some limited domain or setting. For example, knowing that a certain address points to a 32-bit signed integer representing 3 is knowledge restricted to a certain address space. In general, we call these domains that any resource element is tied to *resource contexts*:

Definition 3.2 (Resource Context). A resource context is an abstraction, context, or container of resource elements of the same type, e.g., an address space of a process.

We discuss a range of examples for each of these in turn.

Table 1 gives additional examples of systems and their corresponding resource elements and contexts where these elements reside, though none of the work in that table analyzes itself according to the structure we are giving.

Except for the Post-Crash-Modality, one can think of the resource contexts in the first column in Table 1 as containers for the corresponding resource elements in the second column.

Stack Regions. When reasoning about stack frame contents, the resource element would be a stack-memory points-to assertion ($\boxed{\text{n}}\ \ell \mapsto v$) indicating that a certain offset into stack region n holds value v. The regions (frames) themselves are the contexts.

Virtual Memory. For virtual memory management, a *virtual-points-to*, $\ell \mapsto v$, is an ownership assertion pairing a virtual address (ℓ) with data (v) in an address space, which is a resource context for virtual address mappings, $\ell \mapsto v$, and uniquely identified with a root address. Logically, specifying an address space switch [37] from the address space with the root address $r2$ to the one with the root address $r1$

$$\{[r1]P * Q\}cr3 := r1\{P * [r2]Q\}_{\text{cr3}}$$

requires having the resource elements loaded into the memory

$$P \triangleq \ell_1 \mapsto v...$$

to be in the resource context of the address space modality with the root $r1$, and, once the new address space is loaded—i.e., cr3 is loaded with the other address space root address—the resource elements (Q) that were loaded in the previous view of the memory are now introduced to the resource context with the root $r2$.

Weak-Memory. When considering weak memory models, we consider address-value mappings of thread-local views in C11 memory model [38]). Modalities $\triangle_\pi(P)$ and $\nabla_\pi(P)$ with P represent the memory address-value mappings that held before or will hold after certain memory fence operations by thread π. The elements are points-to assertions, each specific to views of memory after those fence operations.

Reference Counting. When dealing with reference counting APIs, we may care to specify reachability of memory nodes ($\ell \mapsto v$) in a certain context defined by a shared root address. A shared memory address ℓ can be the root of the graph ($@_\ell\ (\ell_1 \mapsto v_1 * ... * \ell_j \mapsto v_j)$) that can be a container of memory nodes ($\ell_1 \mapsto v$) that are reachable from the root ℓ.

Actors. When specifying the consistency of an actor in the current view, we need to refer to other actors' state: $@_\iota(P)$ asserts that P is true in the actor referenced actor ID ι. Assertions may mention local variables and state at the designated actor by name, so the interpretation of variables in assertions depends on which actor the assertion applies to.

Post-Crash. The resource element of Post-Crash Modality is not obvious in Table 1, and needs a bit of explanation. Perennial, based on the Iris logic, has both disk-points-to assertions $d[\text{p}] \mapsto_n v$ (for a specific disk d) and in-memory points-to assertions $\ell \mapsto_n^{\overline{Y}}$. Perennial crash-recovery logic book-keeps resource names (can be thought of as logical

variables) $\overline{\gamma}$ to identify which assertions (resource elements) remain valid after a crash — these assertions are only usable while the names in $\overline{\gamma}$ are valid, and a crash resets them, discarding assumptions about volatile state. A subtlety of the notion of a resource context is that, unlike the earlier examples, the context does not need to be a literal data structure. It can instead be (various forms of) a set of executions, as in the Post-Crash and NextGen modalities. The Post-Crash modality $\diamond$ expresses that the assertion P will be true after a crash discards all unstable storage (i.e., RAM). This was the inspiration for the NextGen modality, which is in fact a framework for defining "after-t" modalities, where t is an transformation of the global state.[3]

3.4 Nominals for Systems Resource Contexts

Finally, another design point is the question of whether or not resource element assertions must explicitly track their corresponding context, or if they implicitly pick up their context from where they are used.

Strong nominalization is the case where resource elements must explicitly include the identity of their intended context, while *weak nominalization* occurs when the resource elements implicitly pick up the relevant context from how they are used. The first three modalities in Table 1 are strongly nominalized, with the resource elements generally carrying identifiers of a specific modality usage.[4] The rest are weakly nominalized.

This choice trades off complexity against flexibility and scoping constraints. Strongly nominalized elements track additional specifier/prover-visible book-keeping data. But in exchange for carrying those identifiers of their context with them, strongly nominalized elements can be used together under any modality. For example, one can use the StackRegion modality to talk about two different stack frames simultaneously for code which accesses multiple stack frames: $\boxed{n}\ \ell \mapsto v * \boxed{n+1}\ \ell' \mapsto v'$. Using a given strongly nominalized assertion element under different modalities for different frames does not change its meaning. Likewise, weak-memory specific modalities existentially quantify over other views, related to the "current" view (the one where the current thread's assertions are evaluated), and evaluate a resource (a thread-local view of an address-value mapping) with respect to those other views. Given that views are identified by *nominals*, referring to all the other views makes weak-memory modalities strongly nominalized.

By contrast, weakly nominalized elements are more concise, but then make talking about multiple contexts together marginally more complex: changing which modality an assertion is used with drastically changes its meaning. In the

case of the Ref-Count modality, $@_\ell(\ell_1 \mapsto v)$ says that ℓ points to a reference count wrapping ℓ_1, while placing the $\ell_1 \mapsto v$ under a jump modality for a different location entails talking about a different region of memory.

In general, use cases where code frequently manipulates small parts of multiple contexts together should prefer strong nominalization, while use cases where usually larger portions of a single context are at issue should prefer weak nominalization.

4 Conclusion

We have argued that there is a systematic pattern emerging in the use of modalities in the verification of low-level systems software, and explained how a range of modern work fits into this pattern. While most of our examples come from public verification work carried out in Iris, we have one example of these ideas applying in Dafny. We are applying these ideas in ongoing work using different modalities for different parts of a copy-on-write filesystem (e.g., for assertions true in different snapshots), and are observing more structured specifications as a result. So not only are the modalities observably present, but they are demonstrably useful for conducting proofs of systems software. We believe that these essential patterns in the designs of modal abstractions for systems verification constitute a fundamental concept when working out how to specify different kinds of systems code.

References

[1] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT press.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner H ähnle, and Mattias Ulbrich. 2025. The Many Uses of Dynamic Logic. In *Go Where the Bugs Are: Essays Dedicated to Wolfgang Reif on the Occasion of His 65th Birthday*. Springer, 56–82.

[3] Carlos Areces, Patrick Blackburn, and Maarten Marx. 2001. Hybrid logics: Characterization, interpolation and complexity. *The Journal of Symbolic Logic* 66, 3 (2001), 977–1010.

[4] Patrick Blackburn and Jerry Seligman. 1995. Hybrid languages. *Journal of Logic, Language and Information* 4, 3 (1995), 251–272.

[5] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. The Zettabyte File System. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies (USENIX FAST)*.

[6] Torben Braüner. 2010. *Hybrid logic and its proof-theory*. Vol. 37. Springer Science & Business Media.

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

[8] Tej Chajed, Joseph Tassarotti, and Contributors. 2023. Post-crash modality in Perennial's Coq Mechanization. https://github.com/mit-pdos/perennial/blob/master/src/goose_lang/crash_modality.v

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

---

[3] The transformations are subject to some technical constraints that are unrelated to our point here.

[4] The Post-Crash modality does not look like this in the presentation here; technically the definition of $\diamond$ quantifies over names $\overline{\gamma}$ internally, dealing with sets of possible contexts.

[10] Sailesh Chutani, Owen T Anderson, Michael L Kazar, Bruce W Leverett, W Anthony Mason, and Robert N Sidebotham. 1992. The Episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*. San Fransisco, CA, USA, 43–60.

[11] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. *Proc. ACM Program. Lang.* 4, POPL, Article 34 (Dec. 2019), 29 pages. doi:10.1145/3371102

[12] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. 2022. Compass: strong and compositional library specifications in relaxed memory separation logic. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 792–808.

[13] Peter J. Denning. 1970. Virtual Memory. *ACM Comput. Surv.* 2, 3 (Sept. 1970), 153–189. doi:10.1145/356571.356573

[14] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457. http://doi.acm.org/10.1145/360933.360975

[15] Marko Doko and Viktor Vafeiadis. 2016. A Program Logic for C11 Memory Fences. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation* (St. Petersburg, FL, USA). doi:10.1007/978-3-662-49122-5_20

[16] Marko Doko and Viktor Vafeiadis. 2017. Tackling Real-Life Relaxed Concurrency with FSL++. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017* (Uppsala, Sweden). doi:10.1007/978-3-662-54434-1_17

[17] Kosta Došen. 1992. Modal translations in substructural logics. *Journal of Philosophical Logic* 21, 3 (1992), 283–336.

[18] Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 4 (2014), 1–44.

[19] George Gargov and Valentin Goranko. 1993. Modal logic with names. *Journal of Philosophical Logic* 22, 6 (1993), 607–636.

[20] Valentin Goranko. 1996. Hierarchies of modal and temporal logics with reference pointers. *Journal of Logic, Language and Information* 5, 1 (1996), 1–24.

[21] Colin S Gordon. 2019. Modal assertions for actor correctness. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 11–20.

[22] Robert Hagmann. 1987. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*. 155–162.

[23] Reiner Hähnle. 2022. *Dijkstra's Legacy on Program Verification* (1 ed.). Association for Computing Machinery, New York, NY, USA, 105–140. https://doi.org/10.1145/3544585.3544593

[24] Joseph Y Halpern. 2017. *Reasoning about uncertainty*. MIT press.

[25] Joseph Y Halpern and Ronald Fagin. 1989. Modelling knowledge and action in distributed systems. *Distributed computing* 3 (1989), 159–177.

[26] Joseph Y Halpern and Yoram Moses. 1985. A guide to the modal logics of knowledge and belief. In *Proceedings of the 9th international joint conference on Artificial intelligence-Volume 1*. Morgan Kaufmann Publishers Inc., 480–490.

[27] Joseph Y Halpern and Yoram Moses. 1990. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)* 37, 3 (1990), 549–587.

[28] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. 1973. Actor induction and meta-evaluation. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 153–168. doi:10.1145/512927.512942

[29] Jaakko Hintikka. 1962. *Knowledge and belief*. Cornell University Press.

[30] Dave Hitz, James Lau, and Michael A Malcolm. 1994. File System Design for an NFS File Server Appliance.. In *USENIX Winter*, Vol. 94.

[31] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. 1988. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 109–133.

[32] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20.

[33] Norihiro Kamide. 2002. Kripke semantics for modal substructural logics. *Journal of Logic, Language and Information* 11, 4 (2002), 453–470.

[34] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–30.

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

[36] Robert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*. http://cs.au.dk/~birke/papers/ipm-conf.pdf

[37] Ismail Kuru and Colin S. Gordon. 2025. Modal Abstractions for Virtualizing Memory Addresses. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025). doi:10.1145/3763134

[38] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++ 11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 618–632.

[39] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 348–370.

[40] Tom Murphy, Karl Crary, Robert Harper, and Frank Pfenning. 2004. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004*. IEEE, 286–295.

[41] Hiroshi Nakano. 2000. A Modality for Recursion. In *Logic in Computer Science (LICS)*. 255–266. http://www602.math.ryukoku.ac.jp/~nakano/papers/modality-lics00.ps.gz

[42] Peter W O'Hearn and David J Pym. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244.

[43] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *Theorem Proving in Higher Order Logics*. Springer, 391–407.

[44] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*. ieee, 46–57.

[45] Vaughan R Pratt. 1976. Semantical consideration on Floyd-Hoare logic. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*. IEEE, 109–121.

[46] Greg Restall. 1993. Modalities in substructural logics. *Logique et Analyse* 36, 141/142 (1993), 25–38.

[47] Greg Restall. 2002. *An introduction to substructural logics*. Routledge.

[48] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, Washington, DC, USA, 55–74. http://dl.acm.org/citation.cfm?id=645683.664578

[49] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. doi:10.1145/2501620.2501623

[50] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. doi:10.1145/146941.146943

[51] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. 2015. A separation Logic for Fictional Sequential Consistency. In *European Symposium on Programming*. Springer, 736–761.

[52] Robert E Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE transactions on software engineering* 1 (1986), 157–171.

[53] Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. 2025. 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* (Denver, CO, USA) *(CPP '25)*. 83–97. doi:10.1145/3703595.3705876

[54] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 315 (Oct. 2024), 30 pages. doi:10.1145/3689755