Compositionality and Observational Refinement for Linearizability with Crashes

ARTHUR OLIVEIRA VALE, Yale University, USA
ZHONGYE WANG, Yale University, USA
YIXUAN CHEN, Yale University, USA
PEIXIN YOU, Yale University, USA
ZHONG SHAO, Yale University, USA

Crash-safety is an important property of real systems, as the main functionality of some systems is resilience to crashes. Toward a compositional verification approach for crash-safety under full-system crashes, one observes that crashes propagate instantaneously to all components across all levels of abstraction, even to unspecified components, hindering compositionality. Furthermore, in the presence of concurrency, a correctness criterion that addresses both crashes *and* concurrency proves necessary. For this, several adaptations of linearizability have been suggested, each featuring different trade-offs between complexity and expressiveness. The recently proposed compositional linearizability framework shows that to achieve compositionality with linearizability, both a locality and observational refinement property are necessary. Despite that, no linearizability criterion with crashes has been proven to support an observational refinement property.

In this paper, we define a compositional model of concurrent computation with full-system crashes. We use this model to develop a compositional theory of linearizability with crashes, which reveals a criterion, *crash-aware linearizability*, as its inherent notion of linearizability and supports both locality and observational refinement. We then show that strict linearizability and durable linearizability factor through crash-aware linearizability as two different ways of translating between concurrent computation with and without crashes, enabling simple proofs of locality and observational refinement for a generalization of these two criteria. Then, we show how the theory can be connected with a program logic for durable and crash-aware linearizability, which gives the first program logic that verifies a form of linearizability with crashes. We showcase the advantages of compositionality by verifying a library facilitating programming persistent data structures and a fragment of a transactional interface for a file system.

CCS Concepts: • **Theory of computation → Parallel computing models**; Denotational semantics; **Program specifications**; **Program verification**; **Abstraction**; • **Computer systems organization → Reliability**.

Additional Key Words and Phrases: Crash-Aware Linearizability, Strict Linearizability, Durable Linearibility, Compositional Linearizability

ACM Reference Format:
Arthur Oliveira Vale, Zhongye Wang, Yixuan Chen, Peixin You, and Zhong Shao. 2024. Compositionality and Observational Refinement for Linearizability with Crashes. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 352 (October 2024), 115 pages. https://doi.org/10.1145/3689792

Authors' Contact Information: Arthur Oliveira Vale, Yale University, New Haven, USA, arthur.oliveiravale@yale.edu; Zhongye Wang, Yale University, New Haven, USA, zhongye.wang@yale.edu; Yixuan Chen, Yale University, New Haven, USA, yixuan.chen@yale.edu; Peixin You, Yale University, New Haven, USA, peixin.you@yale.edu; Zhong Shao, Yale University, New Haven, USA, zhong.shao@yale.edu.

ACM 2475-1421/2024/10-ART352
https://doi.org/10.1145/3689792

1 Introduction

In this paper, we develop a compositional account of linearizability under full-system crashes. By a full-system crash, we mean a crash that results in all agents of a system failing or being reset. This could result from a power outage, a user holding the power button on their computer, a fatal crash in an OS, a critical component failure, etc. By compositional, we mean that verified components can be freely composed vertically and horizontally so that the composed system is *correct by construction*, in that no side conditions are necessary to derive its correctness from the correctness of its components. As a result, we obtain a framework for verifying large-scale crash-aware systems against linearizability. To see why compositionality is important, consider one of our main examples: the FLiT library [38].

***The* FLiT *Library*.** Implementing persistent data structures, even when non-volatile memory (NVM) is available, is notoriously challenging. For instance, one of the challenges when programming with NVM is that it provides a buffered interface BCell. We can encapsulate the operations of a buffered memory cell in the following signature, where $\mathbf{1}$ stands for some singleton set (we will write $() \in \mathbf{1}$ if it is an argument, and $\mathsf{ok} \in \mathbf{1}$ if it is a return) and Val a set of memory values:

$$\mathsf{BCell} := \{\mathsf{load} : \mathbf{1} \rightarrow \mathsf{Val}, \mathsf{store} : \mathsf{Val} \rightarrow \mathbf{1}, \mathsf{flush} : \mathbf{1} \rightarrow \mathbf{1}\}$$

What this signature expresses is that BCell provides three operations: $\mathsf{load}()$, which takes unit $() \in \mathbf{1}$ as argument and returns some value in Val; $\mathsf{store}(v)$, which takes a value $v \in \mathsf{Val}$ as argument, and returns the unit $\mathsf{ok} \in \mathbf{1}$; and $\mathsf{flush}()$, which takes unit $()$ as argument and returns a unit ok. The signature BCell provides the syntax of the operations of a buffered memory cell. It must be paired with a specification defined later, which provides the semantics of the operations. Such a specification would state that stores are not guaranteed to persist immediately; instead, they are buffered and persist only when the buffer is non-deterministically flushed or explicitly flushed by a $\mathsf{flush}()$ invocation [33, 34]. In other words, once a crash happens, a load is only guaranteed to read a value no older than the latest flush. The explicit flush operation guarantees a buffer flush at a significant performance cost, so in practice, one would like to minimize its usage. For instance, in the trace (where α_0, α_1, α_2, and α_3 are the names of the agents performing the operations):

$$\boldsymbol{\alpha_0}{:}\mathsf{store}(0) \cdot \boldsymbol{\alpha_0}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_0}{:}\mathsf{flush}() \cdot \boldsymbol{\alpha_0}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_1}{:}\mathsf{store}(1) \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}{:}\mathsf{load}() \cdot \boldsymbol{\alpha_2}{:}v \cdot ↯ \cdot \boldsymbol{\alpha_3}{:}\mathsf{load}() \cdot \boldsymbol{\alpha_3}{:}v'$$

the value v must be $v = 1$, as 1 is currently the buffered value. Meanwhile, either $v' = 0$ (the value at the latest flush), or $v' = 1$ (which could have been non-deterministically flushed from the buffer). This non-determinism of the value of a load after a crash complicates programming with NVM.

Some works attempt to facilitate programming persistent data structures by providing more robust persistent objects than those available directly from the underlying NVM, which usually only provides buffered memory cells. One such work is FLiT, a C++ library which provides a wrapper for the BCell operations. Specifically, in its essence, FLiT provides an object with signature

$$\mathsf{FLiT} := \{\mathsf{load} : \mathbf{1} \rightarrow \mathsf{Val}, \mathsf{store} : \mathsf{Val} \rightarrow \mathbf{1}\}.$$

As is traditional in the linearizability literature, we use a set of valid concurrent traces ν' to represent objects. ν' may be further abstracted by providing a set ν of less concurrent traces (often atomic, i.e., traces where every invocation is immediately followed by its response) with respect to which the traces in ν' are linearizable[1]. In the context of durable linearizability [22], ν' also differs from its linearized specification ν in that ν' has explicit crashes while ν does not. More precisely, durable linearizability requires that $\mathsf{ops}(\nu')$, the crash-less specification obtained by removing all crash events from traces in ν', is linearizable (in the usual sense) w.r.t. ν.

We specify the FLiT object ν'_{FLiT} to be durably linearizable to ν_{FLiT}, the usual crash-less atomic memory cell. This should be understood as stating that the FLiT operations are persistent in ν'_{FLiT},

[1] We take the convention that a primed specification is a concrete specification, and the un-primed an abstract specification

meaning that a load after a crash does read the most recently written value, up to happens-before reordering. FLiT's implementation M_{FLiT}, which runs on top of a buffered memory cell object ν'_{BCell} and of a volatile counter object ν'_{Counter}, does this by: (1) always flushing stores; (2) using the counter to keep track of when flushes are necessary; (3) having loads only flush when the counter marks that a flush is necessary. The counter specified by ν'_{Counter} is volatile in that it lives in volatile memory, so after a crash, a new instance is created with the initial value of 0. The code M_{FLiT} for our simplified formulation of FLiT is found below in Fig. 1.

For instance, a buffered memory cell allows for the following trace:

$$\boldsymbol{\alpha_0}{:}\mathsf{store}(1) \cdot \boldsymbol{\alpha_1}{:}\mathsf{load}() \cdot \boldsymbol{\alpha_1}{:}1 \cdot ↯ \cdot \boldsymbol{\alpha_2}{:}\mathsf{load}() \cdot \boldsymbol{\alpha_2}{:}v$$

where either $v = 0$ (when the buffer containing 1 has not flushed before the crash) or $v = 1$ (when the buffer is flushed before the crash). If $v = 0$, the trace is not durably linearizable to the usual memory cell specification because a 0 is read after 1 is read with no store(0) to justify it.

```
Import B:BCell
Import C:Counter

load()                          store(v)
{   v ← B.load();               {   C.inc();
    if(C.get() != 0)                B.store(v);
    {  B.flush();  }                B.flush();
    return v;                       C.dec();
}                                   return; }
```

Fig. 1. FLiT Memory Cell Implementation M_{FLiT}

Meanwhile, when using FLiT, the call to store(1) must execute at least up to the B.store(1) invocation (as load() manages to read 1). This means that the call to store(1) will have executed C.inc(). Assuming it only executes up to receiving the response B.ok to its B.store(1) call (otherwise, it executes a flush). The $\boldsymbol{\alpha_1}$:load() call will execute to completion, so it will call B.load() and receive B.1 as response. Then, it will read 1 from C.get(), and will execute B.flush() before returning 1. Hence, when the crash ↯ happens, the buffered memory cell has been flushed, guaranteeing that any load() calls after the crash will read 1. Therefore, calling the memory operations using FLiT guarantees that $v = 1$.

The FLiT paper claims that: "Using the library's default mode makes any linearizable data structure durable [...]", which they do not prove. In fact, it is challenging to state this theorem without a compositional model of crash-aware computation, as it concerns discussing the composition of arbitrary clients with FLiT. In addition, even if such a compositional model were available, it must provide good support for durable linearizability and be closely connected with a concurrent compositional model without crashes, also providing good support for usual linearizability [20]. The reason for this is that this statement relates an implementation that assumes the usual concurrent memory and implements a linearizable object, with an implementation that runs on top of the crash-aware FLiT library and implements a durably linearizable object. No framework for verification of concurrent systems with crashes allows for the correctness of FLiT to be stated in full formality, much less for it to be proved and used to build provably correct durable components using a crash-less component (i.e., one whose specification does not involve crashes) which has been previously verified against a linearizability specification.

Using our compositional account of linearizability with crashes, we prove the following FLiT correctness theorem (ν'_{Cell} is any crash-less object Herlihy-Wing linearizable to ν_{FLiT}).

Proposition 1.1 (FLiT Correctness). *For any object signature E, writing $\nu'_{\mathsf{Mem}} := \otimes_{i \in I} \nu'_{\mathsf{Cell}}$ for the horizontal composition of several memory cells, if $\nu'_{\mathsf{Mem}}; M$ is an object linearizable to ν_E then, writing $\nu'_{\mathsf{BMem}} := \otimes_{i \in I} \nu'_{\mathsf{BCell}}$, it follows that $\nu'_{\mathsf{BMem}}; M_{\mathsf{FLiT}}; \mathsf{vol}(M)$ is durably linearizable to ν_E.*

The $- \otimes -$ operation stands for horizontal composition, which composes two objects into a single object, allowing operations from both components to be issued by a client. Therefore, ν'_{Mem} defines a memory array. M is code implementating a new object with signature E using the memory array ν'_{Mem}. The $-;-$ stands for vertical composition, so that $\nu'_{\mathsf{Mem}}; M$ stands for the object obtained by running the implementation M on top of the memory array. Similarly, ν'_{BMem} is a buffered memory array. $\mathsf{vol}(M)$ adds crash semantics to M by running it in each epoch (the period in between crashes), so that $v'_{\mathsf{BMem}}; M_{\mathsf{FLiT}}; \mathsf{vol}(M)$ is the object obtained by running M on top of the FLiT wrapper M_{FLiT} around the buffered memory array.

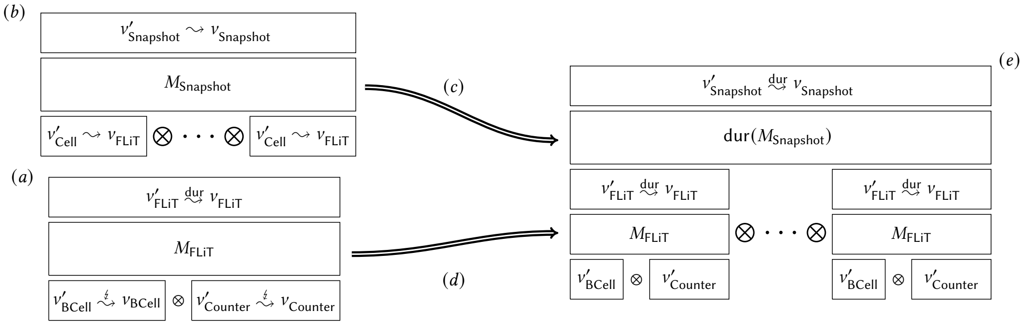

(b)
$v'_{\mathsf{Snapshot}} \rightsquigarrow v_{\mathsf{Snapshot}}$
M_{Snapshot}
$v'_{\mathsf{Cell}} \rightsquigarrow v_{\mathsf{FLiT}} \otimes \cdots \otimes v'_{\mathsf{Cell}} \rightsquigarrow v_{\mathsf{FLiT}}$
(c)
(e)
$v'_{\mathsf{Snapshot}} \overset{\mathsf{dur}}{\rightsquigarrow} v_{\mathsf{Snapshot}}$
$\mathsf{dur}(M_{\mathsf{Snapshot}})$
(a)
$v'_{\mathsf{FLiT}} \overset{\mathsf{dur}}{\rightsquigarrow} v_{\mathsf{FLiT}}$
M_{FLiT}
$v'_{\mathsf{BCell}} \overset{\lightning}{\rightsquigarrow} v_{\mathsf{BCell}} \otimes v'_{\mathsf{Counter}} \overset{\lightning}{\rightsquigarrow} v_{\mathsf{Counter}}$
(d)
$v'_{\mathsf{FLiT}} \overset{\mathsf{dur}}{\rightsquigarrow} v_{\mathsf{FLiT}}$
M_{FLiT}
$v'_{\mathsf{BCell}} \otimes v'_{\mathsf{Counter}}$
$\otimes \cdots \otimes$
$v'_{\mathsf{FLiT}} \overset{\mathsf{dur}}{\rightsquigarrow} v_{\mathsf{FLiT}}$
M_{FLiT}
$v'_{\mathsf{BCell}} \otimes v'_{\mathsf{Counter}}$

Fig. 2. (a) Using our program logic for durable linearizability, we verify the FLiT implementation; (b) Using the program logic for compositional linearizability we verify the crash-less snapshot object; (c) Using the FLiT correctness theorem, we lift the crash-less snapshot object into durably linearizable snapshot object running on top of a FLiT array; (d) Using vertical and horizontal composition, we obtain (e) a durably linearizable snapshot object running on top of an array of buffered memory cells and volatile counters.

We prove this by developing a compositional theory of durable linearizability which supports both locality and observational refinement (where we write $v' \overset{\mathsf{dur}}{\rightsquigarrow} v$ for "v' is durably linearizable to v"), proving locality and observational refinement properties for durable linearizability, and then introducing a program logic for verifying individual components to be durably linearizable. Using our program logic, we show that (depicted diagrammatically in Fig. 2 (a)):

Proposition 1.2. *$(v'_{\mathsf{Counter}} \otimes v'_{\mathsf{BCell}}); M_{\mathsf{FLiT}}$ is durably linearizable with respect to v_{FLiT}.*

$(v'_{\mathsf{Counter}} \otimes v'_{\mathsf{BCell}}); M_{\mathsf{FLiT}}$ is the object obtained by running the code in Fig. 1 on top of the volatile counter v'_{Counter} and the buffered memory cell v'_{BCell}. By using an observational refinement property for *crash-aware linearizability*, a novel linearizability criterion we introduce, we prove this using instead the linearized specifications for the counter and the buffered memory cell, greatly simplifying its proof by only considering atomic traces. By verifying that M_{FLiT} is durably linearizable, we can use locality (Prop. 1.4) and observational refinement (Prop. 1.3) to prove FLiT's correctness.

Proposition 1.3 (Observational Refinement). *An object $v'_A : A$ is durably linearizable to v_A if and only if whenever an implementation M implements a concurrent object linearizable to v_B using v_A, $\mathsf{vol}(M)$ implements an object durably linearizable to v_B using v'_A.*

Proposition 1.4 (Locality). *For $v'_A : A, v'_B : B$ and $v_A : A, v_B : B$:*

$$v'_A \overset{\mathsf{dur}}{\rightsquigarrow} v_A \text{ and } v'_B \overset{\mathsf{dur}}{\rightsquigarrow} v_B \text{ if and only if } v'_A \otimes v'_B \overset{\mathsf{dur}}{\rightsquigarrow} v_A \otimes v_B$$

While the original paper on durable linearizability claims it satisfies locality, it does not do so by formalizing horizontal composition. Meanwhile, our locality statement is directly formulated within our compositional model of computation with crashes, which is defined independent of any notion of linearizability. This makes our locality theorem much stronger as it interacts well with refinement and vertical composition. Observational refinement, however, has never been shown for any linearizability criteria with crashes. Our program logic is the first to verify any linearizability criteria with crashes. Moreover, as it is necessary to verify the FLiT library, our program logic can reason about external linearization points and helpings [26], even across crashes.

We further showcase the benefits of compositionality and of the FLiT correctness theorem by showing that we can lift a crash-less interval-sequential linearizable snapshot object [7] into a durable one (Fig. 2 (b)). We do this by first verifying the write-snapshot implementation M_{Snapshot} from Borowsky and Gafni [6] using the program logic of Oliveira Vale et al. [31, 32]. The implementation uses a (crash-less) memory cell object v'_{Cell} (which is Herlihy-Wing linearizable to v_{FLiT}) to implement an object with interface Snapshot with a single operation write_snapshot:

$$\mathsf{Snapshot} := \{\mathsf{write\_snapshot} : \mathsf{Val} \to \mathcal{P}(\mathsf{Val})\}$$

The operation write_snapshot writes the current value to the memory and returns a set of values that have been written to the object before. The implementation M_{Snapshot} uses one memory cell per agent $\alpha \in S$ in the snapshot system to implement the Snapshot object.

Using the soundness theorem for their program logic [31], we obtain a crash-less interval-sequential linearizable object. Because we formally connect our model and durable linearizability definition to their model, we can then use the FLiT correctness theorem to obtain that the write-snapshot object is interval-sequential *durably* linearizable in the model with crashes. Note that this also showcases that our linearizability criteria and program logic are all generalized to handle interval-sequential objects. We display this setup in Fig. 2 (e).

While durable linearizability is a good criterion for specifying persistent objects, it is inept at expressing objects with less persistent behaviors, such as volatile objects, buffered objects, or objects with hybrid crash behaviors (e.g., horizontal compositions of objects with different persistency guarantees). Therefore, we use the methodology of compositional linearizability [31] to derive the inherent notion of linearizability to our compositional model, which we call *crash-aware linearizability*. We show that this criterion, though simple, is novel to our work and satisfies locality and observational refinement. Then, we show that durable linearizability and strict linearizability factor through crash-aware linearizability as different ways to translate crash-aware linearizable objects to the crash-less model from compositional linearizability. We showcase that crash-aware linearizability is a robust verification criterion by verifying a fragment of a transactional file system interface featuring recovery and objects with many different persistency guarantees.

Summary of Main Contributions.

- A compositional model of concurrent computation with crashes directly connected to the model of crash-less computation used in the compositional linearizability paper [31].
- A novel linearizability criterion, which we call crash-aware linearizability, is apt for specifying objects with a variety of crash behaviors.
- Compositional formulations of strict and durable linearizability, in particular, generalizing them away from atomic specifications.
- Proofs of locality, formulated for the first time in a compositional style, for crash-aware, strict, and durable linearizability.
- The first proofs of observational refinement properties for any linearizability criterion with crashes, which we show for crash-aware, strict, and durable linearizability.
- Two variations of a program logic for showing linearizability of crash-aware components: one for crash-aware linearizability and the other for durable linearizability. This makes for the first program logic that can prove linearizability specifications for components with crashes.
- A proof of correctness for FLiT and a proof that the snapshot object of Borowsky and Gafni [6] is interval-sequential linearizable, yielding a verified durable interval-sequential snapshot object using the FLiT correctness theorem.

- A proof of correctness, against crash-aware linearizability, of a simplified file API, involving objects with a variety of crash-behaviors and a few layers to exemplify compositionality under heterogenous crash-behaviors.

We present a reduced treatment of our results, which emphasizes the main points and omits all proofs. A full account of our results may be found in our extensive appedix.

2 Three Linearizability Criteria under Crashes

We start the technical core of our paper by defining and contrasting three different linearizability criteria under crashes: *crash-aware linearizability*, *strict linearizability* and *durable linearizability*. We assume a crash model with full-system crashes, that is, a crash event crashes all agents in the system. This is appropriate for, for example, a multicore machine but not for a distributed system, which requires individual crashes for each node. It serves, however, as a crucial stepping stone toward a realistic compositional modeling of distributed systems with crashes, as each node is often a multi-threaded system over a multicore machine. We define the criteria formally but omit many technical details of the compositional model, which we explain later in §3.

2.1 Preliminaries

Our model is parametrized by a set Υ of agent names $\alpha \in \Upsilon$. Events look like $\boldsymbol{\alpha}{:}m$ denoting that agent α performs an invocation or response m. If M denotes the given set of events then $s \in M^*$ is said to be a *crash-less* well-formed trace if its projection $\pi_\alpha(s)$ to only events performed by α alternates between invocations and responses, and denote the set of all such traces by $\mathbb{P}^{\mathsf{conc}}_M$.

We denote a crash event by $\lightning$. We say a trace $s \in (M + \lightning)^*$ is a well-formed *crash-aware* trace if it is of the form $s_1 \cdot \lightning \cdot s_2 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_n$ where each $s_i \in \mathbb{P}^{\mathsf{conc}}_M$. Given this decomposition, we define the number of epochs $\|s\|$ of s to be $\|s\| := n$. The trace s_i is called the i-th epoch of s and denoted by $\mathrm{epo}_i(s) := s_i$. We denote the set of all well-formed crash-aware traces over M by $\mathbb{P}^{\lightning}_M$.

As usual with linearizability, a specification is a non-empty, prefix-closed set of well-formed traces. If the specification ν only has crash-less traces, i.e. $\nu \subseteq \mathbb{P}^{\mathsf{conc}}_M$, we call it a *crash-less specification*, and if it has crash-aware traces, i.e. $\nu \subseteq \mathbb{P}^{\lightning}_M$, we call it a *crash-aware specification*.

Toward defining our linearizability criteria, we start by defining a rewrite system that models the preservation of happens-before ordering from the usual linearizability definition in a more localized way. This formulation has been used in many developments on linearizability [2, 14, 18, 31].

Definition 2.1. We define a string rewrite system $\rightsquigarrow$ with local rewrite rule:

$$s \cdot \boldsymbol{\alpha}{:}m \cdot \boldsymbol{\alpha'}{:}m' \cdot t \rightsquigarrow s \cdot \boldsymbol{\alpha'}{:}m' \cdot \boldsymbol{\alpha}{:}m \cdot t$$

whenever $\alpha \neq \alpha'$ and one of the following two conditions hold:

- m and m' are both invocations or both responses, or
- m is an invocation and m' is a response.

The definition of linearizability from the compositional linearizability paper is then given by:

Definition 2.2. A crash-less trace $s \in \mathbb{P}^{\mathsf{conc}}_M$ is linearizable to a crash-less trace $t \in \mathbb{P}^{\mathsf{conc}}_M$ when there exists a sequence of responses $s_P \in M^*$ and a sequence of invocations $s_O \in M^*$ such that $s \cdot s_P \rightsquigarrow t \cdot s_O$. We write $s \leadsto t$ when s is linearizable to t. We say a crash-less specification ν' linearizes to another one ν, written $\nu' \leadsto \nu$, when every trace $s \in \nu'$ linearizes to some trace $t \in \nu$.

Note that t is not required to be atomic, as in Herlihy-Wing linearizability, and that s_O is not required to contain every pending invocation of $s \cdot s_P$, unlike most definitions of linearizability. If t is an atomic trace, then this definition is equivalent to the original Herlihy-Wing definition [18, 31].

2.2 Linearizability Under Full-System Crashes

We now define crash-aware linearizability, the criterion we propose in this paper. It requires that each epoch of a trace s linearizes, in the crash-less sense, to the corresponding epoch of t.

Definition 2.3. A crash-aware trace $s \in \mathbb{P}^{\lightning}_M$ is *crash-aware linearizable* to a trace $t \in \mathbb{P}^{\lightning}_M$ when

$$\|s\| = \|t\| \qquad\qquad \text{and} \qquad\qquad \forall i \le \|s\|.\mathsf{epo}_i(s) \rightsquigarrow \mathsf{epo}_i(t)$$

We denote this as $s \overset{\lightning}{\rightsquigarrow} t$, extending the notation to specifications as with linearizability (Def. 2.2).

Observe that crash-aware linearizability relates crash-aware specifications to crash-aware specifications. This is unusual in the literature on linearizability under crashes, as the other criteria relate a crash-aware specification to a crash-less specification. We discuss the reasons for this later when we have defined two other linearizability criteria and can better compare them.

We now define strict linearizability [2]. Our definition differs from the original one in that it specializes it to full-system crashes (instead of allowing for each agent to crash independently), removes the notion of aborted executions, and generalizes away from atomicity to allow for non-atomic linearized specifications. The first two changes were already considered in Ben-David et al. [3] and make the criterion appropriate for the settings we are interested in, such as NVM and file systems. The later change goes along the lines of the way that Castañeda et al. [7] and Oliveira Vale et al. [31] generalize Herlihy-Wing linearizability [20]. If we restrict our definition so that the linearized trace must be atomic, we obtain the same criterion considered by Ben-David et al. [3].

Definition 2.4. For a crash-aware trace s, we define, whenever well-formed, the crash-less trace

$$\mathsf{ops}(s) := \mathsf{epo}_1(s) \cdot \mathsf{epo}_2(s) \cdot \ldots \cdot \mathsf{epo}_{\|s\|}(s)$$

We say a crash-aware trace $s \in \mathbb{P}^{\lightning}_M$ is *strictly linearizable* to a crash-less trace t, written $s \overset{\mathsf{str}}{\rightsquigarrow} t$, when there exists a crash-aware trace t' such that $s \overset{\lightning}{\rightsquigarrow} t'$ and $\mathsf{ops}(t') = t$.

Note that our definition of strict linearizability shows a clear factoring of strict linearizability as crash-aware linearizability followed by crash-removal.

The third and final linearizability criterion we consider here is *durable linearizability* [22]. Durable linearizability is more expressive than strict linearizability [3, 19] in that it considers more objects to be linearizable. This comes at the cost of the extra assumption on the model that new agent names are used in each epoch, which we call the *durability assumption.*

Definition 2.5. We say a crash-aware trace $s \in \mathbb{P}^{\lightning}_M$ is *durable* when:

$$\forall i, j \le \|s\|.i \ne j \implies \Upsilon(\mathsf{epo}_i(s)) \cap \Upsilon(\mathsf{epo}_j(s)) = \varnothing$$

where $\Upsilon(t)$ is the set of agents appearing in a trace t. We denote by $\mathbb{P}^{\mathsf{dur}}_M \subseteq \mathbb{P}^{\lightning}_M$ the subset of well-formed crash-aware traces that are durable.

When a trace s is durable, $\mathsf{ops}(s)$ is always a well-formed crash-less trace. Durable linearizability is then defined in terms of usual crash-less linearizability. Our definition, similarly to our definitions of the other linearizability criteria we consider, generalizes away from atomicity by allowing linearized traces to be non-atomic and allows for the specification of blocking objects, as it does not require all uncompleted pending invocations to be removed. It is, however, fully equivalent to the original definition of durable linearizability if we require that the linearized trace be atomic.

Definition 2.6. We say a durable trace $s \in \mathbb{P}^{\mathsf{dur}}_M$ is *durably linearizable*, written $s \overset{\mathsf{dur}}{\rightsquigarrow} t$, to a crash-less trace t when $\mathsf{ops}(s) \rightsquigarrow t$.

Note that durable linearizability corresponds to the inverse factoring to strict linearizability, one first removes crashes and then uses crash-less linearizability. These two factorings play an important technical role in our proofs. Moreover, it is possible to show that both criteria factor (in a different sense) through crash-aware linearizability.

Proposition 2.7.

- *If* $s' \overset{\lightning}{\rightsquigarrow} s$ *and* $s \overset{\mathsf{str}}{\rightsquigarrow} t$ *then* $s' \overset{\mathsf{str}}{\rightsquigarrow} t$
- *If* $s' \overset{\lightning}{\rightsquigarrow} s$ *and* $s \overset{\mathsf{dur}}{\rightsquigarrow} t$ *then* $s' \overset{\mathsf{dur}}{\rightsquigarrow} t$

Because of this fact, in practice, when verifying durably linearizable objects, we find it useful to use a crash-aware specification ν^{mid} satisfying: $\nu' \overset{\lightning}{\rightsquigarrow} \nu^{\mathsf{mid}}$ and $\nu^{\mathsf{mid}} \overset{\mathsf{dur}}{\rightsquigarrow} \nu$. This allows us to consider less concurrent traces within the linearized specification for ν' by linearizing as much as possible within each epoch of ν^{mid} first. This allows us to obtain the benefits of both crash-aware and durable linearizability simultaneously: by maintaining both ν^{mid} and ν we can still express durably linearizable specifications, but by manipulating ν^{mid} we achieve the same level of compositionality as crash-aware linearizability. This technique is not necessary for strict linearizability because we can just use crash-aware linearizability directly by always picking ν^{mid} so that $\mathsf{ops}(\nu^{\mathsf{mid}}) = \nu$.

2.3 Specifying a Buffered Memory Cell

In §1 we mentioned that we use crash-aware linearizability to specify a buffered memory cell with signature BCell. As an example, we define here what the linearized specification for a buffered memory cell implementation would be under crash-aware linearizability.

An example of a trace of a concrete buffered memory cell ν'_{BCell} is:

$$\boldsymbol{\alpha}_1{:}\mathsf{store}(1) \cdot \boldsymbol{\alpha}_2{:}\mathsf{load}() \cdot \boldsymbol{\alpha}_1{:}\mathsf{ok} \cdot \boldsymbol{\alpha}_2{:}0 \cdot \boldsymbol{\alpha}_2{:}\mathsf{flush}() \cdot \boldsymbol{\alpha}_1{:}\mathsf{store}(2) \cdot \boldsymbol{\alpha}_1{:}\mathsf{ok} \cdot \lightning \cdot \boldsymbol{\alpha}_3{:}\mathsf{load}() \cdot \boldsymbol{\alpha}_3{:}1$$

The trace above is crash-aware linearizable to the following trace, among others:

$$\boldsymbol{\alpha}_2{:}\mathsf{load}() \cdot \boldsymbol{\alpha}_2{:}0 \cdot \boldsymbol{\alpha}_1{:}\mathsf{store}(1) \cdot \boldsymbol{\alpha}_1{:}\mathsf{ok} \cdot \boldsymbol{\alpha}_2{:}\mathsf{flush}() \cdot \boldsymbol{\alpha}_2{:}\mathsf{ok} \cdot \boldsymbol{\alpha}_1{:}\mathsf{store}(2) \cdot \boldsymbol{\alpha}_1{:}\mathsf{ok} \cdot \lightning \cdot \boldsymbol{\alpha}_3{:}\mathsf{load}() \cdot \boldsymbol{\alpha}_3{:}1$$

We specify the semantics of the buffered memory cell by a set of traces ν'_{BCell} with only events that are allowed by the signature BCell. Crashes can happen at any point. To specify the correctness of ν'_{BCell} we require it to be crash-aware linearizable to the atomic linearized specification ν_{BCell}. Because we show observational refinement, we are able to leave ν'_{BCell} unspecified for the sake of verifying the FLiT implementation, as only the linearized specification will be necessary. The linearized specification ν_{BCell} is then defined by:

$$s \in \nu_{\mathsf{BCell}} \iff s \text{ is atomic} \wedge (\forall s_1, s_2. \forall v. s = s_1 \cdot \boldsymbol{\alpha}{:}\mathsf{load}() \cdot \boldsymbol{\alpha}{:}v \cdot s_2 \implies v \in \mathsf{snd}(\mathsf{mstate}(s_1)))$$

where $\mathsf{mstate}(s)$ assigns to an atomic complete trace s a set of pairs $\mathsf{mstate}(s) \subseteq \mathsf{Val} \times \mathsf{Val}$. A pair $(v_p, v_b) \in \mathsf{mstate}(s)$ consists of a possibility for a value v_p that has persisted and a value v_b currently in the buffer. $\mathsf{mstate}(s)$ is then the function inductively defined below ($v_0 \in \mathsf{Val}$ is an identified initial value for the memory cell):

$$\mathsf{mstate}(\epsilon) := \{(v_0, v_0)\} \qquad \mathsf{mstate}(s \cdot \lightning) := \{(v, v) \mid \exists v'.(v, v') \in \mathsf{mstate}(s)\}$$

$$\mathsf{mstate}(s \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok}) := \{(v', v) \mid \exists v''.(v', v'') \in \mathsf{mstate}(s)\} \cup \{(v, v)\}$$

$$\mathsf{mstate}(s \cdot \boldsymbol{\alpha}{:}\mathsf{load}() \cdot \boldsymbol{\alpha}{:}v) := \{(v', v) \mid (v', v) \in \mathsf{mstate}(s)\}$$

$$\mathsf{mstate}(s \cdot \boldsymbol{\alpha}{:}\mathsf{flush}() \cdot \boldsymbol{\alpha}{:}\mathsf{ok}) := \{(v, v) \mid (v', v) \in \mathsf{mstate}(s)\}$$

The function $\mathsf{snd}(p)$ projects into the second component v_b of the pair $p = (v_p, v_b)$, so that $\mathsf{snd}(\mathsf{mstate}(s)) = \{v_b \mid \exists v_p.(v_p, v_b) \in \mathsf{mstate}(s)\}$.

Note that we could have specified it instead using a labeled state transition system (LTS), in which case ν_{BCell} is the set of traces that start from the initial state of the LTS. Either way of defining ν_{BCell} defines the same set of traces.

2.4 Contrasting Crash-Aware Linearizability

We now compare crash-aware linearizability against strict and durable linearizability. We will not compare strict and durable linearizability against each other since they are not new to our work, and refer the interested reader to the following references [3, 19, 22]. We do briefly mention a key difference that applies to crash-aware linearizability as well. In strict and crash-aware linearizability, a pending invocation must be linearized *within* the epoch it was issued. Durable linearizability, however, allows for a pending invocation to be linearized in (essentially) a later epoch by allowing those pending invocations to be reordered after events from later epochs. This is what makes it more expressive than strict linearizability, allowing for more complex crash behaviors, such as recovering parts of a data structure only when they are demanded by a client, which could happen several epochs later. As explained in the remark at the end of §2.2 crash-aware linearizability interacts well with durable linearizability. This will be discussed further in §5.3.

As we saw, both durable and strict linearizability factor through crash-aware linearizability. The key difference between the two former criteria and the latter is that the former use crash-less linearized specifications, while the latter uses crash-aware linearized specifications. So let's refer to the former as *crash-unaware* criteria.

Crash-unaware criteria reduce the correctness of an object with crashes to that of an object without crashes. This makes them great at specifying objects with very strong persistency guarantees, that is, objects whose whole state (or almost) persists after a crash. But it makes them quite deficient at specifying objects with weaker persistency guarantees such as volatile objects (all of the state is lost on a crash), objects with hybrid persistency (part of the state is volatile and part of the state is persistent), or objects whose persistency features some degree of non-determinism (such as in buffered memory). Some of these issues were already known. For instance, in the original durable linearizability paper [22], it is noted that the criteria do not behave well when used to specify a buffered object, requiring them to define an *ad-hoc* notion of *buffered* durable linearizability which does not satisfy locality, making it not compositional.

Consider the simple problem of specifying the correctness of a volatile object. Given a crash-less specification v, we can construct a crash-aware specification $\mathsf{vol}(v)$ of a volatile version of that object by the Kleene algebra formula $\mathsf{vol}(v) := (v \cdot \lightning)^* \cdot v$.

For example, given the usual atomic counter specification v_{Counter}, the following trace is allowed by $\mathsf{vol}(v_{\mathsf{Counter}})$ (the subscript 1 under the Counter operations will be useful later):

$$s_1 = \boldsymbol{\alpha_1}{:}\mathsf{inc}_1 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \lightning \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_3}{:}0$$

Note that the crash move $\lightning$ plays a crucial role in the specification, as the counter only resets to 0 after a crash event (such as the last get event in s_1), making the linearized specification deterministic.

Under crash-aware linearizability, a concurrent object v' correctly implements a volatile version of a crash-less object v when $v' \overset{\lightning}{\rightsquigarrow} \mathsf{vol}(v)$. With our methods, it is easy to show that

Proposition 2.8. *If* $v' \rightsquigarrow v$ *then* $\mathsf{vol}(v') \overset{\lightning}{\rightsquigarrow} \mathsf{vol}(v)$.

A consequence of this is that if we have an implementation M that implements a crash-less object linearizable to v_B on top of a crash-less object v_A, then if we run M on each epoch on top of $\mathsf{vol}(v_A)$ then M implements an object crash-aware linearizable to $\mathsf{vol}(v_B)$. Formally,

$$v_A; M \rightsquigarrow v_B \implies \mathsf{vol}(v_A); \mathsf{vol}(M) \overset{\lightning}{\rightsquigarrow} \mathsf{vol}(v_B)$$

In other words, crash-aware linearizability is able to appropriately specify and characterize the correctness of volatile objects in a way that is useful to a client.

Now, consider what happens if we try to specify a volatile object using a crash-unaware criterion. A crash-unaware linearized specification will need to include both of the following traces:

$$\mathsf{ops}(s_1) = \boldsymbol{\alpha_1}{:}\mathsf{inc}_1 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_3}{:}0 \qquad \text{and} \qquad \boldsymbol{\alpha_1}{:}\mathsf{inc}_1 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_3}{:}1$$

so that the linearized specification under crash-unaware criteria must admit non-deterministically resetting the counter at any point. This can happen at any point, but the point at which it happens is not detectable in the linearized specification, which makes the specification quite weak. This means that even if some observational refinement theorem (*à la* Filipovic et al. [14]) holds for the crash-unaware criterion, the client to the linearized specification will need to contend with non-determinism, making the contextual refinement, and hence vertical composition, weaker.

This issue is compounded when considering horizontal composition. Both durable and strict linearizability are known to satisfy locality. However, those locality theorems introduce even more non-determinism into the resulting linearized specifications. Consider now a second trace s_2 for a second counter independent of the counter in play s_1

$$s_2 = \boldsymbol{\alpha_1}{:}\mathsf{inc}_2 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_2 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \lightning \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_2 \cdot \boldsymbol{\alpha_3}{:}0$$

Any trace in their parallel composition $s_1 \otimes s_2$ (the set of well-formed crash-aware interleavings of s_1 and s_2, defined in §3) synchronizes on the crash, so both counters reset their state at the same time. For example, the following crash-aware trace belongs to $s_1 \otimes s_2$:

$$\boldsymbol{\alpha_1}{:}\mathsf{inc}_1 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_1}{:}\mathsf{inc}_2 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_2 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \lightning \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_2 \cdot \boldsymbol{\alpha_3}{:}0 \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_3}{:}0$$

Meanwhile, the corresponding linearized specifications under durable or strict linearizability include these traces without the crash event, i.e., $\mathsf{ops}(s_1)$ and $\mathsf{ops}(s_2)$. Hence, the following trace is in their parallel composition $\mathsf{ops}(s_1) \otimes \mathsf{ops}(s_2)$ (the set of their well-formed crash-less interleavings):

$$\boldsymbol{\alpha_1}{:}\mathsf{inc}_1 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_1}{:}\mathsf{inc}_2 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_3}{:}0 \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_2 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_2 \cdot \boldsymbol{\alpha_3}{:}0$$

There is no trace of the concrete horizontally composed volatile counters that is linearizable to the trace above, as we must at least introduce a crash right before $\boldsymbol{\alpha_3}{:}\mathsf{get}_1$ to justify its return $\boldsymbol{\alpha_3}{:}0$:

$$\boldsymbol{\alpha_1}{:}\mathsf{inc}_1 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_1}{:}\mathsf{inc}_2 \cdot \boldsymbol{\alpha_1}{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \lightning \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_1 \cdot \boldsymbol{\alpha_3}{:}0 \cdot \boldsymbol{\alpha_2}{:}\mathsf{get}_2 \cdot \boldsymbol{\alpha_2}{:}1 \cdot \boldsymbol{\alpha_3}{:}\mathsf{get}_2 \cdot \boldsymbol{\alpha_3}{:}0$$

This makes the trace inconsistent with the semantics of the second counter, as the crash should also have reset it, so that $\boldsymbol{\alpha_2}{:}\mathsf{get}_2$ should not return $\boldsymbol{\alpha_2}{:}1$. The same kind of argument shows how crash-unaware criteria fail to accurately handle hybrid and buffered objects (all the traces above are valid for a buffered counter, for example).

3 A Concurrent Game Semantics with Crashes

So far, in §2 we focused on three linearizability criteria in a unstructured setup. For instance, we did not enforce typing on specifications. This will not be enough to achieve the degree of compositionality we seek, especially as we treat objects as open components.

In this section, we discuss our compositional model with crashes in detail. The model is defined using a simple game semantics. The reader not familiar with game semantics jargon will find the following approximation useful. A *game* A, B roughly corresponds to a type; a *move* of the game A corresponds to an event of type A which also has a *polarity*, i.e. its metadata (such as the name of the agent who issued it, and whether it is a move by the environment or by the system); a *play* over a game A is a trace of that type. Crucially, plays can have higher-order types (unlike in most trace semantics); in particular, we may form the affine implication game $A \multimap B$ (the type of code using an object of type A to implement one of type B) whose plays are well-formed traces involving moves from both A and B; a *strategy* σ of type A is the denotation of some computation, be it a state transition system, or the semantics of some code. It is represented as some prefix-closed set of plays[2] of its type A. Readers looking for comprehensive introductions to game semantics may

[2] It is folklore that prefix-closed sets of traces are in one-to-one correspondence with equivalence classes of transition systems under forward-backward simulation [27]. Therefore, all of our results translate to equivalent statements that hold up to forward-backward simulation. We use a presentation based on prefix-closed sets of traces as it aligns well with the typical treatment of linearizability, while simplifying many aspects of the presentation and of the compositional structure.

benefit from Abramsky and McCusker [1], Ghica [15], Hyland [21] though we warn that our model simplifies several aspects of these game semantics, which are not necessary for our purposes.

3.1 Games with Full-System Crashes

Definition 3.1 (Polarities and Moves). A *move set* consists of a set of *moves* M together with an assignment $\lambda : M \to \sum_{\alpha \in \Upsilon}\{O, P\}$, that is, every move is labeled with the agent who plays it and whether or not it is an environment (O) or a system (P) move. The elements of $\sum_{\alpha \in \Upsilon}\{O, P\}$ are called polarities and are denoted by $\boldsymbol{\alpha}{:}O$ or $\boldsymbol{\alpha}{:}P$.

Most of the games we use in practice will be defined by first providing an effect signature. An effect signature is a collection of operations, or effects, $E = (e_i)_{i \in I}$ together with assignments $\mathsf{par}(-), \mathsf{ar}(-) : E \to \mathbf{Set}$ of a set of parameters $\mathsf{par}(e)$ and a set of return values $\mathsf{ar}(e)$ for each operation $e \in E$. This is conveniently described by the following notation.

$$E = \{e_i : \mathsf{par}(e_i) \to \mathsf{ar}(e_i) \mid i \in I\}$$

All the signatures defined in §1 are effect signatures. We call an Υ-indexed collection of effect signatures $E = (E[\alpha])_{\alpha \in \Upsilon}$ a concurrent effect signature. Given a concurrent effect signature E we define a corresponding move set as follows:

$$M_{\dagger E} := \textstyle\sum_{\alpha \in \Upsilon}(\sum_{e \in E[\alpha]}\mathsf{par}(e) + \sum_{e \in E[\alpha]}\mathsf{ar}(e))$$

$$\lambda_{\dagger E}(\boldsymbol{\alpha}{:}e(a)) := \boldsymbol{\alpha}{:}O,\ e \in E[\alpha] \wedge a \in \mathsf{par}(e) \qquad \lambda_{\dagger E}(\boldsymbol{\alpha}{:}v) := \boldsymbol{\alpha}{:}P,\ v \in \mathsf{ar}(e) \text{ for some } e \in E[\alpha]$$

in other words, moves in $M_{\dagger E}$ are either $\boldsymbol{\alpha}{:}e(a)$ for $e \in E[\alpha]$ and $a \in \mathsf{par}(e)$, in which case $\lambda_{\dagger E}(\boldsymbol{\alpha}{:}e(a)) = \boldsymbol{\alpha}{:}O$, or $\boldsymbol{\alpha}{:}v$ with $v \in \mathsf{ar}(e)$, in which case $\lambda_{\dagger E}(\boldsymbol{\alpha}{:}v) = \boldsymbol{\alpha}{:}P$.

Definition 3.2. We denote a crash by $\lightning$. Given a move set M we write $M^{\lightning}$ for its extension $M + \{\lightning\}$ with a crash move. We also extend its polarity function λ into $\lambda^{\lightning}$ with the assignment $\lambda^{\lightning}(\lightning) = \lightning$.

Recall that given a sequence $s \in M^*$, we write $\pi_\alpha(s)$ for the projection of s to its largest subsequence involving only events by $\alpha \in \Upsilon$.

Definition 3.3. A game $A = (M_A, \lambda_A, P_A)$ consists of a move set (M_A, λ_A) and a non-empty, prefix-closed set of well-formed crash-less plays $P_A \subseteq \mathbb{P}^{\mathsf{conc}}_{M_A}$ satisfying $P_A = \|_{\alpha \in \Upsilon}\ \pi_\alpha(P_A)$. We write $P^{\lightning}_A \subseteq \mathbb{P}^{\lightning}_{M_A}$ for the set $P^{\lightning}_A := (P_A \cdot \lightning)^* \cdot P_A$.

The set of plays P_A of a game A defines which plays are valid plays within an epoch. It is required to be an arbitrary parallel compositions of the sequential plays that each agent can perform. Meanwhile, $P^{\lightning}_A$, the corresponding set of crash-aware plays, is defined by simply allowing crashes to happen at any point in an epoch.

Some examples of crash-aware games are now due. The simplest game is the game $\mathbf{1} := (\varnothing, \varnothing, \{\epsilon\})$. The game $\mathbf{1}$ has no non-crash moves, and its only crash-aware plays are the empty sequence ϵ and sequences of crashes $\lightning \cdot \lightning \cdot \ldots \cdot \lightning$.

Another game is the game $\Sigma = ((\sum_{\alpha \in \Upsilon} q + a), (\sum_{\alpha \in \Upsilon} \boldsymbol{\alpha}{:}O + \boldsymbol{\alpha}{:}P), \|_{\alpha \in \Upsilon} \downarrow\{q \cdot a\})$ where $\downarrow -$ stands for prefix-closure. Unrolling this definition, every agent has two moves: an O-move q (question) and a P-move a (answer). The only valid sequential plays are $q \cdot a$ and its prefixes, and the valid plays for the game are interleavings of these sequential plays at each epoch, such as:

$$\boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha}{:}a \cdot \lightning \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}a \cdot \lightning \cdot \boldsymbol{\alpha'}{:}q$$

The most important kind of game for our examples are games $\dagger E$ generated by effect signatures E. We can extend the move set $(M_{\dagger E}, \lambda_{\dagger E})$ into a game with set of valid plays $P_{\dagger E}$ defined by:

$$P_{\dagger E} :=\|_{\alpha \in \Upsilon} \downarrow(\cup_{e \in E[\alpha]} \cup_{a \in \mathsf{par}(e)} \cup_{v \in \mathsf{ar}(e)} \boldsymbol{\alpha}{:}e(a) \cdot \boldsymbol{\alpha}{:}v)^*$$

That is to say, locally, each agent $\alpha \in \Upsilon$ is allowed to alternate between making a call to an effect $e(a)$ in $E[\alpha]$ or providing a response to the previously issued effect. For instance, recall that we defined, in §1, a signature BCell encoding the operations available to a buffered memory cell. This defines a concurrent effect signature $\mathsf{BCell}[\alpha] = \mathsf{BCell}$. The corresponding set of valid crash-aware plays $P^{\lightning}_{\dagger\mathsf{BCell}}$ includes all traces seen in §2.3.

3.2 Combining Games

We now define a few combinators on games. We start by defining a dualizing operation on move sets, which swaps the role of environment and system.

Definition 3.4 (Dual Move Set). Given a move set (M, λ) we define the moveset $(M^{\perp}, \lambda^{\perp})$ by $M^{\perp} := M$ and $\lambda^{\perp}(m) := \lambda(m)^{\perp}$, where $(\boldsymbol{\alpha}{:}O)^{\perp} := \boldsymbol{\alpha}{:}P$ and $(\boldsymbol{\alpha}{:}P)^{\perp} := \boldsymbol{\alpha}{:}O$.

In the context of games A, B, C, given $s \in \mathbb{P}^{\lightning}_{M_A+M_B}$ we define $s\restriction_{A,-} \in \mathbb{P}^{\lightning}_{M_A}$ and $s\restriction_{-,B} \in \mathbb{P}^{\lightning}_{M_B}$ to be the projections to the corresponding components of $M_A + M_B$, but keeping the crash moves in the projections too. Similarly, given $s \in \mathbb{P}^{\lightning}_{M_A+M_B+M_C}$, we write $s\restriction_{A,B,-}$, for the projection of s to its largest subsequence with only moves in A, B and crashes; we similarly define $s\restriction_{A,-,C}$ and $s\restriction_{-,B,C}$.

We now define horizontal composition of games, and the affine arrow.

Definition 3.5. Fix games A and B. We define the games $A \otimes B$ and $A \multimap B$ by the following data

$$M_{A\otimes B} := M_A + M_B \qquad \lambda_{A\otimes B} := \lambda_A + \lambda_B \qquad P_{A\otimes B} := \{s \in \mathbb{P}_{M_A+M_B} \mid s\restriction_{A,-} \in P_A \wedge s\restriction_{-,B} \in P_B\}$$

$$M_{A\multimap B} := M_A^{\perp} + M_B \qquad \lambda_{A\multimap B} := \lambda_A^{\perp} + \lambda_B \qquad P_{A\multimap B} := \{s \in \mathbb{P}_{M_A^{\perp}+M_B} \mid s\restriction_{A,-} \in P_A \wedge s\restriction_{-,B} \in P_B\}$$

It is implicit in this definition that by composing in parallel the two crash-aware plays, the resulting set of traces synchronizes the crash events, merging them into a single crash event and then producing any (locally sequential) parallel composition of the subtraces appearing in each epoch. Consider, for instance, the two plays below on the left, each of type Σ:

$$\begin{array}{cccccccc} \boldsymbol{\alpha}{:}q & \boldsymbol{\alpha}{:}a & \lightning & \boldsymbol{\alpha}{:}q & \boldsymbol{\alpha}{:}a & \lightning & \boldsymbol{\alpha'}{:}q & \boldsymbol{\alpha'}{:}a \\ & & & \multicolumn{2}{c}{\bigotimes} & & & \\ \boldsymbol{\alpha}{:}q & \boldsymbol{\alpha}{:}a & \lightning & \boldsymbol{\alpha'}{:}q & \boldsymbol{\alpha'}{:}a & \lightning & \boldsymbol{\alpha'}{:}q & \boldsymbol{\alpha'}{:}a \end{array} \Longrightarrow \begin{array}{cccccccc} \boldsymbol{\alpha}{:}q & \boldsymbol{\alpha}{:}a & | & \boldsymbol{\alpha}{:}q & \boldsymbol{\alpha}{:}a & | & \boldsymbol{\alpha'}{:}q & \boldsymbol{\alpha'}{:}a \\ \multicolumn{2}{c}{\otimes} & \lightning & \multicolumn{2}{c}{\otimes} & \lightning & \multicolumn{2}{c}{\otimes} \\ \boldsymbol{\alpha}{:}q & \boldsymbol{\alpha}{:}a & | & \boldsymbol{\alpha'}{:}q & \boldsymbol{\alpha'}{:}a & | & \boldsymbol{\alpha'}{:}q & \boldsymbol{\alpha'}{:}a \end{array}$$

The resulting set of traces synchronizes the crashes, as depicted on the right. For example, the following is a valid trace in their horizontal composition:

$$\boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}a \cdot \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}a \cdot \lightning \cdot \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha}{:}a \cdot \boldsymbol{\alpha'}{:}a \cdot \lightning \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha'}{:}a \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha'}{:}a$$

Similarly, consider the following play s of $\Sigma \multimap \Sigma$ (on the left):

$$\begin{array}{cccccccc|c} \Sigma & \boldsymbol{\alpha}{:}q & & \boldsymbol{\alpha'}{:}q & & | & \boldsymbol{\alpha'}{:}q & & s\restriction_{-,\Sigma} = \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \lightning \cdot \boldsymbol{\alpha'}{:}q \\ \rotatebox{90}{$\multimap$} & & & & & \lightning & & & \\ \Sigma & & \boldsymbol{\alpha}{:}q & & \boldsymbol{\alpha}{:}a & | & & \boldsymbol{\alpha'}{:}q & s\restriction_{\Sigma,-} = \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}a \cdot \lightning \cdot \boldsymbol{\alpha'}{:}q \end{array}$$

or, depicted sequentially: $\boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha}{:}a \cdot \lightning \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\alpha'}{:}q$. Note that the crash signal synchronizes across the source and target components of the play. This models that the crashes are *synchronous* across components (they happen in all components at once) and that they are *instantaneous* (it takes negligible time for the crash to propagate to components). On the right, above, we see the projections of s to the source and target components. Importantly, the crash event is retained in both projections, so it effectively belongs to both components.

3.3 Crash-Aware Strategies

We now define strategies, which are the denotations of both object specifications and code.

Definition 3.6 (Crash-Aware Strategy). A crash-aware strategy $\sigma : A$ over a game A is a non-empty, prefix-closed, subset $\sigma \subseteq P_A^{\lightning}$, which is moreover $\lightning$-receptive in that

$$\forall s \in \sigma . s \cdot \boldsymbol{\lightning} \in P_A^{\lightning} \implies s \cdot \boldsymbol{\lightning} \in \sigma$$

$\lightning$-receptivity models the usual assumption that crashes may non-deterministically happen at any point in an execution. It plays a crucial role in proving the locality property.

We specify the semantics of objects using strategies. For example, in §2.3 we specified the linearized buffered memory cell by a strategy $\nu_{\mathsf{BCell}} : \dagger\mathsf{BCell}$. The denotations of implementations, such as $M_{\mathsf{FLiT}} : \dagger\mathsf{BCell} \otimes \dagger\mathsf{Counter} \multimap \dagger\mathsf{FLiT}$ or $M_{\mathsf{Snapshot}} : \dagger\mathsf{Mem} \multimap \dagger\mathsf{Snapshot}$ mentioned in §1 are examples of strategies with the affine arrow type. Strategies of type $A \multimap B$ can be vertically composed, which amounts to the usual motto of "interaction + hiding".

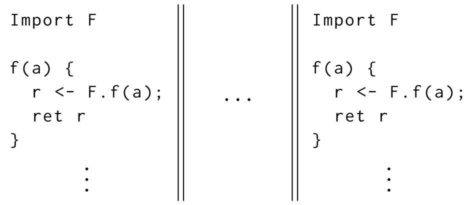
Definition 3.7. Given strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ we define their vertical composition $\sigma ; \tau : A \multimap C$ by: $\sigma ; \tau := \{ s\restriction_{A,-,C} \in \mathbb{P}^{\lightning}_{A \multimap C} \mid \exists s \in ((M_A + M_B + M_C)^{\lightning})^* . s\restriction_{A,B,-} \in \sigma \wedge s\restriction_{-,B,C} \in \tau \}$

Proposition 3.8. *Composition of crash-aware strategies is well-defined and associative.*

For the reader with familiarity with category theory, we can package all the information above:

Definition 3.9. We denote by $\underline{\textbf{Crash}}$ the semicategory of crash-aware games, with crash-aware strategies $\sigma : A \multimap B$ as morphisms between games A and B, and composition given by $- ; -$.

Unfortunately, and this is a common phenomenon in concurrency models, $\underline{\textbf{Crash}}$ does not assemble into a category, as the vertical composition operation $- ; -$ does not have a neutral element. That is, to say, there is no choice of strategies $\mathbf{id}_A : A \multimap A$ for which $\mathbf{id}_A ; \sigma ; \mathbf{id}_B = \sigma$ for every $\sigma : A \multimap B$. This issue is explained extensively in Oliveira Vale et al. [31] in the context of concurrent games.



```
Import F            ||          ||  Import F
                    ||          ||
f(a) {              ||          ||  f(a) {
  r <- F.f(a);      ||   ...    ||    r <- F.f(a);
  ret r             ||          ||    ret r
}                   ||          ||  }
       ⋮            ||          ||         ⋮
```

Fig. 3. Code corresponding to the copycat strategy $\mathsf{crashcopy}_{\dagger F \multimap \dagger F} : \dagger F \multimap \dagger F$

We follow the approach from compositional linearizability, and start by noting that there are obvious candidates $\mathsf{crashcopy}_A : A \multimap A$ for the neutral elements, which are called the copycat strategies and formalize the code seen in Fig. 3. The copycat strategy is *idempotent*, in that for all games A, $\mathsf{crashcopy}_A ; \mathsf{crashcopy}_A = \mathsf{crashcopy}_A$. This essentially means that the $\mathsf{crashcopy}_A$ at least behaves like a neutral element for itself. In fact, they behave like a neutral element with respect to any strategy which is a parallel composition of sequential strategies. This fact justifies defining a class of strategies that behaves well when composed with the copycat:

Definition 3.10. We say a strategy $\sigma : A \multimap B$ is saturated with respect to crashcopy when

$$\mathsf{crashcopy}_A ; \sigma ; \mathsf{crashcopy}_B = \sigma$$

Since crashcopy is idempotent, it is saturated. Moreover, by definition, crashcopy behaves as a neutral element for strategy composition of saturated strategies. It is also easy to see that saturated strategies compose. Note that this means that we can promote $\underline{\textbf{Crash}}$ to a category by restricting attention to these saturated strategies.

Saturation for concurrent strategies corresponds to, beyond O-receptivity (the environment can make valid moves whenever it wants), strategies that are insensitive to certain delays, which might be caused, for instance, if an agent is preempted. This is typically formalized using the rewrite system $- \rightsquigarrow -$ we defined in §2.1 and redefined now in light of our more detailed formalism.

Definition 3.11. Let $A = (M_A, P_A)$ be a game. We define a string rewrite system $(P_A, \rightsquigarrow_A)$ with local rewrite rules:

- $\forall m, m' \in M_A. \forall X \in \{O, P\}. \lambda_A(m) = \boldsymbol{\alpha}{:}X \wedge \lambda_A(m') = \boldsymbol{\alpha'}{:}X \wedge \alpha \neq \alpha' \implies m \cdot m' \rightsquigarrow_A m' \cdot m$
- $\forall m, m' \in M_A. \lambda_A(m) = \boldsymbol{\alpha}{:}O \wedge \lambda_A(m') = \boldsymbol{\alpha'}{:}P \wedge \alpha \neq \alpha' \implies m \cdot m' \rightsquigarrow_A m' \cdot m$

A concrete characterization of saturation for crash-aware strategies is possible, but we do not cover it here for the sake of space (see the appendix). We will soon see an equivalent characterization in terms of crash-aware linearizability, which will be sufficient for our purposes.

3.4 Refinement and Horizontal Composition

Before proceeding, we briefly address refinement and horizontal composition. We take as our notion of refinement behavior containment, $\sigma \subseteq \tau$, with joins given by set union. This makes all of the models we discuss into enriched (semi)categories over join semi-lattices, which means that:

Proposition 3.12. *Strategy composition $-;-$ is monotonic and join-preserving.*

For horizontal composition, recall that we have already defined a game $A \otimes B \in \underline{\mathbf{Crash}}$. The tensor can be extended to strategies $\sigma : A \multimap B$ and $\tau : A' \multimap B'$ by:

$$\sigma \otimes \tau := \{ s \in P_{A \otimes A' \multimap B \otimes B'} \mid s\restriction_{A \multimap B} \in \sigma \wedge s\restriction_{A' \multimap B'} \in \tau \}$$

Proposition 3.13. *Let* **Crash** *be the restriction of the semicategory* $\underline{\mathbf{Crash}}$ *to strategies saturated with respect to* $\mathsf{crashcopy}$. *Then,* $(\mathbf{Crash}, - \otimes -, \mathbf{1})$ *defines an enriched symmetric monoidal category.*

These definitions permit us to prove Prop. 3.13. This means that $- \otimes -$ defines a monotonic and join-preserving functor so that horizontal composition behaves well with respect to both vertical composition and refinement. This formalizes what we mean when we say that our model is compositional. It remains to extend this compositional structure to linearizability.

4 Three Linearizability Criteria Revisited

We now revisit the linearizability criteria discussed in §2 from the perspective of our just defined model and following ideas from compositional linearizability. In particular, we argue that their methodology recovers crash-aware linearizability as the notion of linearizability associated with the compositional structure of our model and use their general theorem around locality and observational refinement to obtain these results for crash-aware linearizability. Then, we extend these results to strict and durable linearizability by analyzing translations from our crash-aware model to the crash-less model from compositional linearizability.

4.1 Abstract Crash-Aware Linearizability

In §2 we defined a new linearizability criterion which we called *crash-aware linearizability* ($\overset{\lightning}{\leadsto}$). We, however, did not come up with this definition of linearizability. Instead, following the methodology of compositional linearizability, we have derived it from the structure of the model, $\underline{\mathbf{Crash}}$.

To understand this, we start by defining the operation $K_{\lightning}\ - : \underline{\mathbf{Crash}} \to \mathbf{Crash}$ by the formula

$$K_{\lightning}\ \tau := \mathsf{crashcopy}_A ; \tau ; \mathsf{crashcopy}_B$$

for $\tau : A \multimap B \in \underline{\mathbf{Crash}}$. This operation assigns to τ the smallest saturated strategy containing τ.

The framework of compositional linearizability proposes that the native notion of linearizability for crash-aware objects should be equivalent to the refinement $v' \subseteq K_{\lightning}\ v$. Indeed, we are able to show the following characterization of $K_{\lightning}\ -$, which provides a concrete characterization of $K_{\lightning}\ v$ as the set of all plays that are crash-aware linearizable w.r.t. v.

Proposition 4.1. *For any crash-aware strategy $v : A$ it holds that:* $K_{\lightning}\ v = \{ s \in \mathbb{P}^{\lightning}_A \mid \exists t \in v. s \overset{\lightning}{\leadsto} t \}$.

It follows immediately from this characterization that

Proposition 4.2. *ν' is crash-aware linearizable w.r.t. ν if and only if $\nu' \subseteq K_{\lightning}\ \nu$.*

This effectively turns linearizability into a refinement property. This has many benefits from the point of view of verification, as refinement techniques are well-understood. Moreover, since we derive it in this way, we may use the general category-theoretic result in Oliveira Vale et al. [31] to obtain locality and observational refinement.

Proposition 4.3 (Observational Refinement and Locality).

- *$\nu'_A : A$ is crash-aware linearizable w.r.t $\nu_A : A$ iff for all saturated $\sigma : A \multimap B$, $\qquad \nu'_A ; \sigma \subseteq \nu_A ; \sigma$*
- *For $\nu'_A : A, \nu'_B : B$ and $\nu_A : A, \nu_B : B$: $\nu'_A \overset{\lightning}{\rightsquigarrow} \nu_A$ and $\nu'_B \overset{\lightning}{\rightsquigarrow} \nu_B$ if and only if $\nu'_A \otimes \nu'_B \overset{\lightning}{\rightsquigarrow} \nu_A \otimes \nu_B$*

4.2 Compositional Verification of a File System Fragment

To showcase the benefits of compositionality and to show that crash-aware linearizability provides a flexible criterion for mixing objects with different, and complicated, crash behaviors, we verify against a crash-aware linearizable specification a fragment of a file API. Instead of providing a detailed description, we emphasize the salient aspects to our point (a detailed description is available in the appendix). The system also features recovery, our handling of which is discussed later (§5).

The file system fragment involves four main objects: the file interface File, a disk interface Disk implemented using a disk array Disk$[N]$ with a finite number N of disks each with $S + 1$ blocks, and a write-ahead log Log. The signatures for File and Disk are given below:

$$\mathsf{File} := \left\{ \begin{array}{c} \mathsf{write} : \mathsf{block\_id} \times \mathsf{file\_id} \times \mathsf{block} \longrightarrow 1, \\ \mathsf{read} : \mathsf{block\_id} \times \mathsf{file\_id} \longrightarrow \mathsf{block}, \\ \mathsf{swap} : \mathsf{block\_id} \times \mathsf{block\_id} \times \mathsf{file\_id} \times \mathsf{file\_id} \longrightarrow 1 \end{array} \right\} \quad \mathsf{Disk} := \left\{ \begin{array}{c} \mathsf{write} : \mathsf{block\_id} \times \mathsf{block} \longrightarrow 1, \\ \mathsf{read} : \mathsf{block\_id} \longrightarrow \mathsf{block} \end{array} \right\}$$

The file interface exposes a two-level structure. At the first level lies a set of folders, each occupying a single disk block as its inode. For simplicity, the API uses block ids instead of strings to uniquely identify folders. Each folder contains a set of files identified by their file id. The swap operation swaps the pointers in the respective folders' inodes, which provides a symmetric move operation as seen in actual file systems. The write and read operations are as usual. The file interface is implemented on top of a disk, providing write and read operations to read and write to a block.

All the objects involved are specified using crash-aware linearizability. For instance, a single disk is specified as the horizontal composition of its blocks, using locality, guaranteeing that its concrete specification $\nu'_{\mathsf{Disk}} : \dagger\mathsf{Disk}$ is crash-aware linearizable to a specification $\nu_{\mathsf{Disk}} : \dagger\mathsf{Disk}$ which guarantees read and writes are persistent and atomic. The disk array specification $\nu'_{\mathsf{Disk}[N]}$ is required to be crash-aware linearizable to the horizontal composition of N atomic disk specifications $\nu_{\mathsf{Disk}[N]} := \otimes_{i \in [N]} \nu_{\mathsf{Disk}}$. The concrete object ν'_{File} is required to be crash-aware linearizable to a specification ν_{File} that ensures that writes, reads and swaps are persistent and seem to happen atomically. All the specifications also enforce that the recovery routines correctly reconstruct any relevant lost state after a crash.

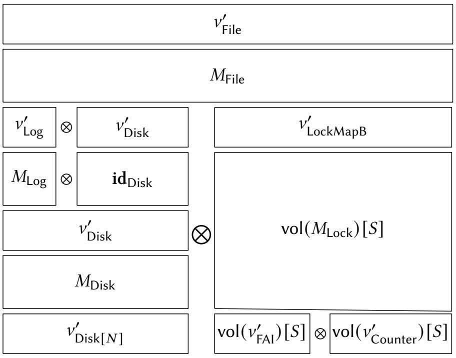

ν'_{File}
M_{File}
ν'_{Log}
$\otimes$
ν'_{Disk}
ν'_{LockMapB}
M_{Log}
$\otimes$
$\mathbf{id}_{\mathsf{Disk}}$
ν'_{Disk}
$\otimes$
$\mathsf{vol}(M_{\mathsf{Lock}})[S]$
M_{Disk}
$\nu'_{\mathsf{Disk}[N]}$
$\mathsf{vol}(\nu'_{\mathsf{FAI}})[S]$
$\otimes$
$\mathsf{vol}(\nu'_{\mathsf{Counter}})[S]$

Fig. 4. The structure of our File example.

We implement the replicated disk on top of the disk array by replicating writes to all the disks in the array in a specific order. Reads to the disk array non-deterministically choose a disk to read from, mimicking the behavior of a disk array controller. On a crash, a recovery procedure copies the contents of the first disks to all disks.

The File implementation M_{File} for write and read is mostly straight-forward. The swap operation requires special treatment for its atomicity. As swap operations need to update two different folders (and thus two different disk blocks), to ensure persistency, we record the operations in a write-ahead log v'_{Log} so that the recovery routine can finish incomplete operations. The log is itself implemented on top of a single block of the disk together with a volatile array and a volatile lock (omitted from Fig. 4). Since the disk is itself equivalent to the parallel composition of individual blocks, we use locality together with our compositionality properties (the symmetric monoidal structure of the model) to separate the part of the disk used for the log, from the rest of the disk.

The file system also uses a set of dynamically allocated locks v'_{LockMapB} to guarantee atomicity when writing to a block. These locks are volatile objects residing in memory that only last for the duration of a single File operation. Because of this, we use the verified lock from Oliveira Vale et al. [31] and lift it to a volatile object using Prop. 2.8, benefiting from the fact that we have connected our model to their model. The whole structure of the example is depicted in Fig. 4.

At this point it is worth remarking that even this small fragment of a file system features a mix of persistent objects, volatile objects, and objects that fit neither category well. Some of the objects involved are horizontal compositions of these objects, making them have hybrid crash behavior. We model all of these objects using crash-aware linearizability, which proves to be robust enough to verify the whole system compositionally.

4.3 Crash Abstraction

Recall that strict and durable linearizability relate a crash-aware concrete specification to a crash-less specification. In this section we develop conversions between these computational models, which serve as a building block for strict and durable linearizability. So, first, we briefly recall that:

Definition 4.4. Given a game A, a crash-less strategy $\sigma : A$ consists of a non-empty, prefix-closed set of plays $\sigma \subseteq P_A$.

The main difficulty in removing crashes from a play s is that the removal may generate traces that do not satisfy well-formedness. This happens when the same agent has a pending invocation in one epoch and also moves in a later epoch. So, in the definition of the operation $-^\flat$ (read *de-crash*, and the same as $\mathsf{ops}(-)$), the projections $\mathsf{ops}(s)$ are required to be well-formed plays.

Definition 4.5. Given a game $A = (M_A, \lambda_A, P_A)$ we define the game $A^\flat$, by:

$$M_{A^\flat} := M_A \qquad \lambda_{A^\flat}(m) := \lambda_A(m) \qquad P_{A^\flat} := (P_A)^* \cap \mathbb{P}^{\mathsf{conc}}_{M_A}$$

Given a crash-aware strategy $\sigma : A \in \underline{\mathbf{Crash}}$ we define the crash-less strategy $\sigma^\flat : A^\flat$ as below. Note that $-^\flat$ formalizes $\mathsf{ops}(-)$ (§2). It is also useful to provide a reverse operation $-^\sharp$, read *re-crash*, that lifts, in a persistent way, a crash-less strategy $\sigma : A^\flat$ into a strategy $\sigma^\sharp : A$.

$$\sigma^\flat := \{\mathsf{ops}(s) \in \mathbb{P}^{\mathsf{conc}}_{M_A} \mid s \in \sigma\} \qquad \sigma^\sharp := \{s \in \mathbb{P}^{\lightning}_{M_A} \mid \mathsf{ops}(s) \in \sigma\}$$

4.4 Strict Linearizability

Similarly to how Oliveira Vale et al. [31] characterizes linearizability by lifting a non-saturated strategy to a saturated strategy, we formalize strict linearizability by lifting a strategy without crashes into a strategy with crashes.

Definition 4.6. Given games A, B, we define the strict lift $\mathsf{str}(\sigma) : A \multimap B$ of a crash-less strategy $\sigma : A^\flat \multimap B^\flat$ as the crash-aware strategy: $\mathsf{str}(\sigma) := K_{\lightning}\ \sigma^\sharp$

It then turns out that, similarly to what we did for crash-aware linearizability, strict linearizability supports the following refinement-based characterization:

Proposition 4.7. *$\nu' : A$ is strictly linearizable to $\nu : A^\flat$ if and only if $\nu' \subseteq \mathsf{str}(\nu)$.*

We make use of this characterization to show the following observational refinement property:

Proposition 4.8 (Observational Refinement). *Suppose $\nu'_A : A$ is strictly linearizable to $\nu_A : A^\flat$ and that $\sigma : A^\flat \multimap B^\flat$ implements an object linearizable to $\nu_B : B^\flat$ using ν_A, i.e. $\nu_A ; \sigma \subseteq \nu_B$, then, $\mathsf{str}(\sigma)$ implements an object strictly linearizable to $\mathsf{str}(\nu_B)$ using ν'_A, i.e. $\nu'_A ; \mathsf{str}(\sigma) \subseteq \mathsf{str}(\nu_B)$.*

The reverse direction, unfortunately, does not hold, fundamentally because $\mathsf{str}(\mathsf{ccopy}_{A^\flat}) \neq \mathsf{crashcopy}_A$. By similar reasoning as the locality for crash-aware linearizability, we also obtain:

Proposition 4.9 (Locality). *For crash-aware strategies $\nu'_A : A, \nu'_B : B$ and crash-less strategies $\nu_A : A, \nu_B : B$: $\quad \nu'_A \subseteq \mathsf{str}(\nu_A)$ and $\nu'_B \subseteq \mathsf{str}(\nu_B)$ if and only if $\nu'_A \otimes \nu'_B \subseteq \mathsf{str}(\nu_A \otimes \nu_B)$*

4.5 Durable Linearizability

Recall that a crash-aware play (i.e., a trace) is durable when the set of agents on different epochs is disjoint. Given a game A, let P_A^{dur} be the subset of $P_A^{\lightning}$ containing only its durable plays. As we noted in §2, durable plays s have the important property that their de-crash $s^\flat$ is always defined. We call a crash-aware strategy *durable* if it only contains durable plays.

Now, for our refinement-based formulation, we define a durable lift $\mathsf{dur}(-)$, which assigns to a crash-less strategy $\nu : A^\flat$ the durable strategy $\mathsf{dur}(\nu) : A$ defined by $\mathsf{dur}(\nu) : A := (K_{\mathsf{Conc}}\ \nu)^\sharp \cap P_A^{\mathsf{dur}}$.

The operation $K_{\mathsf{Conc}}\ -$ in the formula is defined by Oliveira Vale et al. [31] similarly to $K_{\lightning}\ -$, but in the crash-less setting. It may be more intuitively understood through their result that:

$$K_{\mathsf{Conc}}\ \nu = \{s \in P_{A^\flat} \mid \exists t \in \nu . s \rightsquigarrow t\}$$

that is to say, $K_{\mathsf{Conc}}\ -$ assigns to a crash-less strategy ν the smallest strategy containing ν that has all plays linearizable w.r.t. to ν. We observe that, indeed, $\mathsf{dur}(-)$ does provide an appropriate lifting operation for durable linearizability.

Proposition 4.10. *$\nu' : A$ is durably linearizable to $\nu : A^\flat$ if and only if $\nu' \subseteq \mathsf{dur}(\nu)$.*

This refinement characterization enables us to show observational refinement and locality.

Proposition 4.11 (Observational Refinement and Locality).

- *Let A, B be games. Then $\nu'_A : A$ is durably linearizable to $\nu_A : A^\flat$ if and only if whenever $\sigma : A^\flat \multimap B^\flat$ is a crash-less strategy implementing a crash-less object linearizable to ν_B using ν_A, then $\mathsf{dur}(\sigma) : A \multimap B$ implements an object durably linearizable to ν_B using ν'_A.*
- *For durable strategies $\nu'_A : A, \nu'_B : B$ and crash-less $\nu_A : A, \nu_B : B$: $\nu'_A \overset{\mathsf{dur}}{\rightsquigarrow} \nu_A$ and $\nu'_B \overset{\mathsf{dur}}{\rightsquigarrow} \nu_B$ if and only if $\nu'_A \otimes \nu'_B \overset{\mathsf{dur}}{\rightsquigarrow} \nu_A \otimes \nu_B$*

5 Program Logic

In this section, we present a program logic for verifying durable linearizability, which is based on rely-guarantee reasoning, crash Hoare logic and possibility reasoning. We first (§5.1) briefly discuss how to abstract away recovery. Then (§5.2) we define an object-agnostic imperative programming language. Lastly (§5.3) we demonstrate the key rules of the program logic. We refer readers to our appendix for its variation for verifying crash-aware linearizability, which is largely similar.

5.1 Recovery

We start by discussing a simple way of removing recovery events from a play, which is enough for our purposes. First, we fix a certain kind of signature for objects with recovery.

Definition 5.1. We define a recovery signature $E \circlearrowleft R$ to be the union of two effect signatures E for regular operations and R for recovery operations.

To simplify reasoning about programs with recovery, it is common to provide a way to remove the recovery events from the specification. In our setting, this is notoriously simple.

Definition 5.2. We say a strategy $\nu' : \dagger(E \circlearrowleft R)$ recovery-refines to $\nu : \dagger E$ when $\nu' \upharpoonright_E \subseteq \nu$.

It is straightforward to see that the following refinement theorem holds.

Proposition 5.3 (Recovery Refinement Theorem). *Suppose $\nu' : \dagger(E \circlearrowleft R)$ recovery-refines to $\nu : \dagger E$ and that $\sigma' : \dagger(E \circlearrowleft R) \multimap \dagger F$ then, for $\sigma : \dagger E \multimap \dagger F$,* $\quad \sigma' \upharpoonright_{\dagger E \multimap \dagger F} \subseteq \sigma \implies \nu'; \sigma' \subseteq \nu; \sigma$

5.2 Programming Language

5.2.1 Syntax. We start by defining a language Com for commands over some effect signature E.

$$\mathsf{Prim} := x \longleftarrow e(a) \mid \mathsf{assume}(\phi) \mid \mathsf{ret}\ v \qquad \mathsf{Com} := \mathsf{Prim} \mid \mathsf{Com}; \mathsf{Com} \mid \mathsf{Com} + \mathsf{Com} \mid \mathsf{Com}^* \mid \mathsf{skip}$$

Prim stands for primitive commands. The assignment command, $x \longleftarrow e(a)$, executes the effect $e \in E$ with argument a and stores the response to variable x in a local environment $\Delta \in \mathsf{Env}$. The assume command, $\mathsf{assume}(\phi)$, takes a boolean function ϕ over Δ and terminates the computation if it evaluates to False. We implement loops and if-statements using $\mathsf{assume}(-)$ in the usual way. The return command, ret v, stores the value v into a reserved variable res, and is executed once per invocation of a procedure. Com is the grammar of commands defined as usual in a Kleene algebra.

The implementation M of an object (with the effect signature $F \circlearrowleft R_F$) is defined as a collection of commands $M[\alpha]^f \in \mathsf{Com}$, $M = (M[\alpha])_{\alpha \in \Upsilon} = \left(M[\alpha]^f\right)_{\alpha \in \Upsilon, f \in F \cup R_F}$, which implements each method $f \in F \cup R_F$ per agent $\alpha \in \Upsilon$. Here F defines the overlay's regular procedures and R_F its recovery procedures. For simplicity, we require that there is only one recovery program r in R_F, i.e. $R_F = \{r : \mathbf{1} \longrightarrow \mathbf{1}\}$. We call $M[\alpha]$ a local implementation and $M \in \mathsf{CMod}$ a concurrent module, where CMod is the set of all concurrent modules.

5.2.2 Memory Model & Object State. Observe that our programming language is object-agnostic in that it operates over an arbitrary object of type E. This means that the language does not have a memory model baked in. Instead, the underlay object's effect signature E, over which the language is parameterized, determines which memory operations the user can perform. For example, to implement the FLiT memory cell in Fig. 1, one would use as the underlay a buffered memory cell with the BCell signature. Then, one can write a program with statements like $x \longleftarrow B.\mathsf{load}(); B.\mathsf{flush}()$ to manipulate the memory shared across threads.

We define the underlay state as $(\Delta, s) \in \mathsf{UndState}$, a tuple of a local environment Δ and a history $s \in P_{\dagger E}$. The local environment Δ is defined solely as a mapping from local variables to their values (with Δ_0 representing the empty local environment). The history s is a canonical representation for shared state, since it records all previous operations to the shared underlay object. One may reconstruct other more intuitive definitions of the shared state by defining an interpretation function over the trace s. For example, given the traces $p \in \nu_{\mathsf{FLiT}}$ of one FLiT memory cell, we can define the evaluation function $\mathsf{fstate} : \nu_{\mathsf{FLiT}} \longrightarrow \mathsf{Val}$ to compute the current value of the cell by reading the latest stored value. In particular, note that we may use the (atomic) linearized specification for FLiT because of observational refinement.

5.2.3 Semantics. Primitive commands B are interpreted as state transformers $[\![B]\!]_\alpha : \mathsf{UndState} \to \mathcal{P}(\mathsf{UndState})$ from a set of underlay states to a new set of states. The $[\![B]\!]_\alpha$ depends on α only in that it tags event it adds to the history with an agent identifier α. We then lift the state transformer $[\![B]\!]_\alpha$ to a thread-local small-step semantics $\langle C, \Delta, s\rangle \longrightarrow_\alpha \langle C', \Delta', s'\rangle$, which encodes how α steps through commands in a mostly standard way following the Kleene algebra structure of commands.

$$\longrightarrow\ \subseteq (\mathsf{Com} \times \mathsf{UndState}) \times \Upsilon \times (\mathsf{Com} \times \mathsf{UndState}) \qquad \twoheadrightarrow_{R_E}\ \subseteq (\mathsf{Cont} \times \mathsf{ModState}) \times \mathsf{CMod} \times (\mathsf{Cont} \times \mathsf{ModState})$$

$$\frac{f \in F \qquad a \in \mathsf{par}(f) \qquad \Delta' = \Delta[\alpha \mapsto [\mathsf{arg} \mapsto a]]}{\langle c[\alpha \mapsto \mathsf{idle}], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c[\alpha \mapsto M[\alpha]^f], \Delta', s \cdot \boldsymbol{\alpha}{:}f\rangle}\ \textsc{Inv} \qquad \frac{\langle C, \Delta, s\restriction_E\rangle \longrightarrow_\alpha \langle C', \Delta', s'\restriction_E\rangle}{\langle c[\alpha \mapsto C], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c[\alpha \mapsto C'], \Delta', s'\rangle}\ \textsc{Step}$$

$$\frac{\pi_\alpha(s\restriction_F) = p \cdot f \qquad \Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(f) \qquad \Delta' = \Delta[\alpha \mapsto \varnothing]}{\langle c[\alpha \mapsto \mathsf{skip}], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c[\alpha \mapsto \mathsf{idle}], \Delta', s \cdot \boldsymbol{\alpha}{:}v\rangle}\ \textsc{Ret}$$

$$\frac{\begin{array}{c}\forall \alpha \in s.c'[\alpha] = \mathsf{dead}\\ \forall \alpha \in \Upsilon.\alpha \notin s \Rightarrow c'[\alpha] = \mathsf{halt}\end{array}}{\langle c, \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c', \Delta_0, s \cdot \lightning\rangle}\ \textsc{Crash} \qquad \frac{\begin{array}{c}s = s' \cdot \lightning \qquad \vec{r} = \mathsf{perm}(R_E)\\ C = \mathsf{sequence}(\vec{r}, M[\alpha]^r)\end{array}}{\langle c[\alpha \mapsto \mathsf{halt}], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c[\alpha \mapsto C], \Delta, s \cdot \boldsymbol{\alpha}{:}r\rangle}\ \textsc{StartRec}$$

$$\frac{\begin{array}{c}\pi_\alpha(s\restriction_{F \cup R_F}) = s' \cdot r \qquad \Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(r) \qquad \Delta' = \Delta[\alpha \mapsto \varnothing]\\ \forall \alpha \in \Upsilon.c[\alpha] = \mathsf{dead} \Rightarrow c'[\alpha] = \mathsf{dead} \qquad \forall \alpha \in \Upsilon.c[\alpha] \neq \mathsf{dead} \Rightarrow c'[\alpha] = \mathsf{idle}\end{array}}{\langle c[\alpha \mapsto \mathsf{skip}], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c', \Delta', s \cdot \boldsymbol{\alpha}{:}v\rangle}\ \textsc{EndRec}$$

$$\text{where}\ \ \mathsf{sequence}(\vec{r}, C) = \begin{cases} C & \vec{r} = \epsilon \\ (x_r \leftarrow r(a)); \mathsf{sequence}(\vec{r}', C) & \vec{r} = r \cdot \vec{r}' \wedge a \in \mathsf{par}(r) \wedge \mathsf{reserved}(x_r) \end{cases}$$

Fig. 5. Local Small-Step Semantics ($\longrightarrow$) and Module Small-step semantics ($\twoheadrightarrow_{R_E}$)

In Fig. 5, we lift this local small-step semantics to a concurrent module small-step semantics $\langle c, \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c', \Delta', s'\rangle$, which takes a continuation $c \in \mathsf{Cont} := \Upsilon \to \{\mathsf{idle}, \mathsf{skip}, \mathsf{dead}, \mathsf{halt}\} + \mathsf{Com}$ and a module state $(\Delta, s) \in \mathsf{ModState} := (\Upsilon \to \mathsf{Env}) \times P_{\dagger(E \circlearrowright R_E) \multimap \dagger(F \circlearrowright R_F)}$ containing local environments for all agents and the global trace of the system. The first three rules come from the semantics in Oliveira Vale et al. [31] to handle mainly the execution of regular procedures:

Inv Allows a new invocation of any overlay operation f in an idle thread and appends the new invocation to the end of s.

Step Non-deterministically chooses some thread that is running a program C and performs a thread local small-step in that thread with its effect applied to the concurrent module state.

Ret Allows any thread that has finished its program to return to idle by appending the return value as a response to the end of s and clearing $\Delta[\alpha]$.

We add three highlighted rules to handle crashes and recoveries:

Crash Allows for crashes to happen at any time, resetting local environments to Δ_0 for all agents, marking all the previously active agents as dead and all remaining ones as halt.

StartRec Non-deterministically selects a halted thread α and starts the recovery phase by using C as its continuation, which sequentially runs first a permutation of underlay recoveries ($\vec{r} = \mathsf{perm}(R_E)$) and then the overlay recovery $M[\alpha]^r$. This is achieved by using sequence to sequence a list of commands (note that $\mathsf{reserved}(x_r)$ simply means that x_r is a reserved variable). During the recovery phase, other threads must wait for α to finish the recovery before their executions. The execution of α follows the Step rule.

EndRec When the recovery finishes, any agent that is not dead becomes idle, so that the system can now run normally. To enforce the durable assumption, dead agents will no longer run.

We define the denotation of a module by the formula below as the set of traces generated by the small-step semantics from the initial configuration, where c_0 is the initial continuation (every agent is idle) and Δ_0 is the initial environment where every agent has an empty local environment.

$$[\![M]\!]_{R_E} := \{s \mid \exists c \in \mathsf{Cont}, \Delta \in (\Upsilon \to \mathsf{Env}).\langle c_0, \Delta_0, \epsilon\rangle \twoheadrightarrow^{M}_{R_E} \langle c, \Delta, s\rangle\} \subseteq P_{\dagger(E \circlearrowright R_E) \multimap \dagger(F \circlearrowright R_F)}$$

5.3 A Program Logic for Durable Objects

5.3.1 Interfaces. The interface of a crash-aware linearizable object E is a (round bracket) tuple.

$$(\nu'_E : \dagger(E \circlearrowleft R_E), \nu_E : \dagger E) \qquad \text{s.t.} \qquad \nu'_E {\upharpoonright_E} \overset{\lightning}{\rightsquigarrow} \nu_E$$

ν'_E is the concrete specification containing all possible traces the object can produce, including crash and recovery events, and ν_E is the linearized specification after removing recovery events.

Similarly, we define the interface of a durable linearizable object E as the (angle bracket) tuple but with a major difference: the durable interface's linearized specification ν_E is crash-less.

$$\langle \nu'_E : \dagger(E \circlearrowleft R_E), \nu_E : \dagger E \rangle \qquad \text{s.t.} \qquad \nu'_E {\upharpoonright_E} \overset{\text{dur}}{\rightsquigarrow} \nu_E$$

The objective of our program logic is to establish the judgment

$$\vdash M : (\nu'_E, \nu_E) \longrightarrow (\nu'_F, \nu_F) \qquad \text{or} \qquad \vdash M : (\nu'_E, \nu_E) \longrightarrow \langle \nu'_F, \nu_F \rangle$$

which means under the assumption that the implementation M implements F with either a crash-aware interface (ν'_F, ν_F) (the variation described in our appendix) or a durable interface $\langle \nu'_F, \nu_F \rangle$ (the variation we describe here), using the crash-aware underlay E with interface (ν'_E, ν_E). The concrete specification ν'_F is defined by running an implementation M above ν'_E, i.e., $\nu'_F = \nu'_E ; [\![M]\!]_{R_E}$. The program logic's soundness guarantees the validity of the crash-aware/durable overlay interface. In this context, (ν'_E, ν_E) is called M's *underlay*, while $\langle \nu'_F, \nu_F \rangle$ is called M's *overlay*.

The specifications $\nu'_E, \nu_E, \nu'_F, \nu_F$ are fixed in the program logic. For simplicity, we take them as a parameter in all that follows and omit the parametrization in the concrete proof rules.

5.3.2 The Rely-Guarantee Crash Linearizability Hoare Logic (CLHL).

Configurations & Assertions. CLHL uses as proof configurations triples $(\Delta, s, \rho) \in \mathsf{Config} := \mathsf{ModState} \times \mathsf{Poss}$, where $\rho \in \mathsf{Poss}$, called a possibility, is a play of type $\dagger F$ linearizable w.r.t. ν_F. A configuration is valid when s is durably linearizable to ρ and ρ is linearizable w.r.t. ν_F. This ensures that the concrete trace s is always durably linearizable with respect to ν_F after the recovery refinement. Pre-conditions P, post-conditions Q, and crash conditions $Q_{\lightning}$ are given by sets of configurations, while rely conditions $\mathcal{R}$ and guarantee conditions $\mathcal{G}$ are relations over Config.

Top Level Rules. The top level rule Object Impl proves that M implements the overlay $\langle \nu'_F, \nu_F \rangle$ using the underlay (ν'_E, ν_E). It requires the prover to find an object invariant $I : \Upsilon \longrightarrow \mathcal{P}(\mathsf{Config})$ for the implementation and then verify regular procedures and the recovery separately.

$$\frac{\begin{array}{c}\forall \alpha, \alpha' \in \Upsilon . \alpha \neq \alpha' \Rightarrow \mathcal{G}[\alpha] \cup \mathsf{invoke}_{\alpha}(-) \cup \mathsf{return}_{\alpha}(-) \subseteq \mathcal{R}[\alpha'] \\ \forall \alpha \in \Upsilon . \mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash^{F}_{\alpha} M[\alpha] \qquad \forall \alpha \in \Upsilon . I \vDash^{R}_{\alpha} M[\alpha]\end{array}}{\vdash M : (\nu'_E, \nu_E) \longrightarrow \langle \nu'_F, \nu_F \rangle} \ \textsc{Object Impl}$$

Verifying Regular Procedures. To verify a concurrent object, the Object Impl rule requires finding appropriate rely $\mathcal{R}$ and guarantee $\mathcal{G}$ for the object. The rely $\mathcal{R}[\alpha']$ of an agent models the interference of other threads in the executions and therefore must take into account at least invocations, returns, and the guarantee of other agents α (specified, respectively, by $\mathsf{invoke}_{\alpha}(-)$, $\mathsf{return}_{\alpha}(-)$, and $\mathcal{G}[\alpha]$). The prover needs to show $\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash^{F}_{\alpha} M[\alpha]$, which asserts that when α runs regular methods in F, assuming other threads behave according to $\mathcal{R}[\alpha]$, α will behave according to $\mathcal{G}[\alpha]$, and $I[\alpha]$ is satisfied when the thread α is idle.

The Local Impl rule proves this judgment by splitting $I[\alpha]$ into conjunctions of $P[\alpha]^f$, each specifying the pre-condition of a method invocation, and then proving a series of objectives ($- \circ -$ stands for relational composition).

$$\frac{\begin{array}{c}I[\alpha] = \cap_{f \in F} P[\alpha]^f \qquad \forall f \in F . (\Delta_0, \epsilon, \epsilon) \in P[\alpha]^f \qquad \forall f \in F . \mathsf{stable}(\mathcal{R}[\alpha], P[\alpha]^f) \\ \forall f \in F . \mathcal{R}[\alpha], \mathcal{G}[\alpha] \vDash^{f}_{\alpha} \{P[\alpha]^f\} M[\alpha]^f \{Q[\alpha]^f\}\{\top\} \qquad \forall f \in F . \mathsf{return}_{\alpha}(f) \circ Q[\alpha]^f \subseteq I[\alpha]\end{array}}{\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash_{\alpha} M_F[\alpha]} \ \textsc{Local Impl}$$

Firstly, each pre-condition $P[\alpha]^f$ needs to include the initial configuration and must be stable under interferences $\mathcal{R}[\alpha]$ of the environment, which implies the invariant $I[\alpha]$ to be stable.

Then, the prover needs to show that $\mathcal{R}[\alpha], \mathcal{G}[\alpha] \vDash^f_\alpha \{P[\alpha]^f\}\ M[\alpha]^f\ \{Q[\alpha]^f\}\{\top\}$ is satisfied for each method f. The hexad $\mathcal{R}, \mathcal{G} \vDash^f_\alpha \{P\}\ C\ \{Q\}\{Q_{\lightning}\}$ means that given states satisfying P, running the program C on thread α in an environment with interference in $\mathcal{R}$ will produce actions in $\mathcal{G}$, and if it terminates normally, the state will satisfy Q, and if it crashes, the state will satisfy $Q_{\lightning}$. A hexad is proved with proof rules introduced later. It is worth mentioning that *there is no need to explicitly specify and prove a crash condition for any regular method*, and we can simply put $\top$ as the crash condition. This is true because:

(1) The guarantee $\mathcal{G}[\alpha]$ of the current thread is included in any other thread's rely $\mathcal{R}[\alpha']$, and therefore any step during the execution of any method in thread α is captured in $\mathcal{R}[\alpha']$.
(2) For any other thread α', its invariant $I[\alpha']$ is stable w.r.t. $\mathcal{R}[\alpha']$, which means any state after any execution step of any method in thread α (captured in $\mathcal{R}[\alpha']$) is in $I[\alpha']$.
(3) Therefore, the state of thread α will satisfy any other thread's invariant $I[\alpha']$ at any time (including the point of crash), and the crash condition in α can be derived from $I[\alpha']$.

Lastly, after finishing the execution of a method and returning from it, the invariant $I[\alpha]$ needs to be satisfied so that the current thread can still access the object by invoking its procedures.

Verifying Recovery. Then, to ensure the durability of the object, provers need to show $I \vDash^R_\alpha M[\alpha]$, which means whenever a crash happens, the execution of the recovery on any thread α can restore the program state to satisfy the object invariant I. It can be verified via the Recover rule.

$$\frac{\begin{array}{cc}\mathsf{ID}, \top \vDash^r_\alpha \{P_r[\alpha]\}M[\alpha]^r\{Q_r[\alpha]\}\{Q_{\lightning}[\alpha]\} & Q_{\lightning}[\alpha] \subseteq P_r[\alpha] \\ \cup_{\alpha'\in\Upsilon} I[\alpha'] \Rightarrow_{\lightning} Q_{\lightning}[\alpha] & \mathsf{return}_\alpha(r) \circ Q_r[\alpha] \subseteq \cap_{\alpha'\in\Upsilon} I[\alpha']\end{array}}{I \vDash^R_\alpha M[\alpha]}\ \textsc{Recover Impl}$$

The prover needs to find a recovery pre-condition P_r, a recovery post-condition Q_r, and a crash condition $Q_{\lightning}$ for the recovery program, and prove $\mathsf{ID}, \top \vDash^r_\alpha \{P_r[\alpha]\}M[\alpha]^r\{Q_r[\alpha]\}\{Q_{\lightning}[\alpha]\}$, which means running the recovery program $M[\alpha]^r$ from states in $P_r[\alpha]$ will either recover the system into states in $Q_r[\alpha]$ or crash into states in $Q_{\lightning}[\alpha]$. Since the recovery program always runs after a crash, the crash condition $Q_{\lightning}$ needs to imply P_r. But as the recovery program executes sequentially, with no interference from other threads, the rely and guarantee for it are ID and $\top$.

The invariant $I[\alpha']$ serves as the crash condition of other threads. Therefore, we require that all $I[\alpha']$ crash into the crash condition $Q_{\lightning}$ of the recovery program. The crash-into relation ($\Rightarrow_{\lightning}$) amounts to implication after adding a crash: $I \Rightarrow_{\lightning} Q_{\lightning} \iff \forall(\Delta, s, \rho) \in I.(\Delta_0, s \cdot \lightning, \rho) \in Q_{\lightning}$.

Lastly, after the execution of the recovery, the system is restored and ready to run, so the program state after the recovery's return needs to imply the invariant $I[\alpha']$ of any thread α'.

The Core Proof Rule. According to these top-level rules, proofs of both the regular procedures and the recovery boil down to proofs of hexads like $\mathcal{R}, \mathcal{G} \vDash_\alpha \{P\}C\{Q\}\{Q_{\lightning}\}$. Among CLHL proof rules for the hexad, the core proof rule for proving the durable linearizability is the Prim rule, which we focus on in this section and refer readers to the appendix for other proof rules, which are standard.

$$\frac{P \Rightarrow_{\lightning} Q_{\lightning} \quad Q \Rightarrow_{\lightning} Q_{\lightning} \quad Q_{\lightning} \Rightarrow_{\lightning} Q_{\lightning} \quad \mathsf{stable}(\mathcal{R}, P) \quad \mathsf{stable}(\mathcal{R}, Q) \quad \mathcal{G} \vdash_\alpha \{P\}B\{Q\}}{\mathcal{R}, \mathcal{G} \vDash_\alpha \{P\}B\{Q\}\{Q_{\lightning}\}}\ \textsc{Prim}$$

There are three groups of Prim rule's premises. Firstly, as crashes can happen at any point, the pre-/post-condition and the crash condition should be able to crash into ($\Rightarrow_{\lightning}$) the crash condition. Then, as any rely-guarantee logic, the pre-/post-condition needs to be stable w.r.t. the rely $\mathcal{R}$.

$$\begin{array}{ll}\mathcal{G} \vdash_\alpha \{P\}B\{Q\} \iff \forall \Delta, s, \rho, \Delta', s'.((\Delta, s, \rho) \in P \wedge (\Delta', s') \in [\![B]\!]_\alpha(\Delta, s) \cap v_E) & \text{where } \rho \dashrightarrow \rho' \iff \\ \qquad \Rightarrow (\exists \rho'.(\Delta', s', \rho') \in Q \wedge (\Delta, s, \rho)\mathcal{G}(\Delta', s', \rho') \wedge \rho \dashrightarrow \rho') & \exists t_P \in (M^P_F)^*.\rho \cdot t_P \rightsquigarrow_{\dagger F} \rho'\end{array}$$

Lastly, we need to prove the commit rule $\mathcal{G} \vdash_\alpha \{P\}B\{Q\}$ for the primitive command B. It states that after a step from a state in P made by the command B the new state will satisfy the post-condition Q and the guarantee. This step may be the commitment point of some pending operations. To maintain the invariant that s is durably linearizable to ρ, the commit rule allows an angelic linearization update, $\rho \dashrightarrow \rho'$, where provers can append several response events to ρ and rewrite it according to $\rightsquigarrow_{\dagger F}$ to obtain ρ', a new possibility that s linearizes into. Moreover, since possibility updates are recorded in $\mathcal{G}$, its effect is visible to any other thread. Only a careful choice of possibility updates will respect other threads' relies and prove that this object is indeed durably linearizable.

Soundness. CLHL is justified by the following soundness theorem.

Proposition 5.4 (Soundness). *If* $\vdash M : (v'_E, v_E) \longrightarrow \langle v'_F, v_F \rangle$ *is provable, and* (v'_E, v_E) *is a valid crash-aware interface, and* $v'_F = v'_E; [\![M]\!]_{R_E}$*, then* $\langle v'_F, v_F \rangle$ *is a valid durable interface.*

5.4 Examples Revisited

In this section, we present some high-level proof ideas of the FLiT example and demonstrate the usage of the program logic. The FLiT object is built above the buffered memory cell BCell.

The Buffered Cell. The buffered memory cell's concrete traces in v'_{BCell} are crash-aware linearizable to its specification v_{BCell}, which we can define through an interpretation function, mstate : $v_{\mathsf{BCell}} \longrightarrow \mathcal{P}(\mathsf{Val} \times \mathsf{Val})$, which computes the set of all possible combinations of the persisted value (the first component) and the buffered value (the second component), as seen in §2.3.

Using mstate, the specification v_{BCell} is essentially defined as the set of traces that can step from the initial state, the singleton set $\{(v_0, v_0)\}$, to some non-empty state, with the step function below. The sets on the two sides of the arrow are the value of mstate before and after appending the events to the trace.

$$S \xrightarrow{\boldsymbol{\alpha}:\mathsf{store}(v)\cdot\boldsymbol{\alpha}:\mathsf{ok}} \{(v_p, v) \mid (v_p, v_b) \in S\} \cup \{(v, v)\} \qquad S \xrightarrow{\boldsymbol{\alpha}:\mathsf{flush}\cdot\boldsymbol{\alpha}:\mathsf{ok}} \{(v_b, v_b) \mid (v_p, v_b) \in S\}$$

$$S \xrightarrow{\lightning} \{(v_p, v_p) \mid (v_p, v_b) \in S\} \qquad S \xrightarrow{\boldsymbol{\alpha}:\mathsf{load}\cdot\boldsymbol{\alpha}:\mathsf{ok}(v)} \{(v_p, v) | (v_p, v) \in S\}$$

- When a store operation finishes, there are two possible outcomes: the value may have been stored only to the buffered content, while the persisted content remains the same as before the store; the value may be persisted, making the buffered content the same as the persisted one.
- When a flush operation finishes, the buffered value gets flushed into the persisted part. Since after each store operation, the buffered content is uniquely determined (synchronized), after a consequent flush operation, the content of mstate is uniquely determined.
- When a crash $\lightning$ happens, the buffered content is lost, and after the crash, the buffered content is overwritten by the persisted value, which may have various possibilities because a flush may not have happened before the crash. As a result, the uniqueness of the buffered content no longer holds after the crash and is un-synchronized. *The non-determinism brought by* store *and* $\lightning$ *is the first challenge of the* FLiT *proof and the reason we define* mstate *in this way.*
- When a load operation finishes, the actual buffered content is determined and all future load will not observe other possibilities of the buffered content. As we will explain later, this behavior makes the load operation an external linearization point of buffered operations before a crash. *The helping mechanism, especially helpings across crashes, is the second challenge of the* FLiT *proof.* The returned value must be consistent with at least one possible buffered content in S. Otherwise, the post-state is an empty set and this trace will not be accepted in v_{BCell}.

To use CLHL to verify the FLiT overlay, we need an invariant I that links the overlay and underlay states and is maintained by any program step. Depending on the current buffered memory cell state, we split the invariant into three cases. (1) When the buffered content v_b is synchronized and persisted (the Flushed state), then the overlay state fstate(ρ) should also be v_b, i.e., the store operation that writes this v_b is durably linearized. (2) When the buffered content v_b is synchronized but not persisted (the Unflushed state), we use a ghost list B to buffer the pending overlay store(v_b) operations in order, so future operations can help linearize it when the value gets persisted. (3) When a crash happens (the Unsynced state), the buffered content v_b is un-synchronized and corresponds to some store(v_b) operation in the ghost list B, in case it has persisted, or is equal to the current overlay state fstate(ρ), when none of the buffered operations persisted.

As a result, the proof configuration now becomes $(\Delta, s, \rho, B) \in \mathsf{ModState}\times\mathsf{Poss}\times M_F^*$. According to the Object Impl rule, we need to find rely and guarantee conditions verifying $\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \models_\alpha^F M[\alpha]$ for the load and store operations and $I \models_\alpha^R M[\alpha]$ for an empty recovery procedure.

5.4.1 *Regular Procedure Proofs.*

To prove regular procedures through the Local Impl rule, we must find the pre-/post-conditions corresponding to each procedure and prove their Hoare quadruples. For the FLiT implementation, we prove Hoare quadruple (1) and (2) for the load and store operations.

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{\mathsf{invoke}_\alpha(\mathsf{load}) \circ I\}\mathsf{load}()\{\mathsf{returned}_\alpha(\mathsf{load}) \circ I\}\{\top\} \tag{1}$$

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{\mathsf{invoke}_\alpha(\mathsf{store}) \circ I\}\mathsf{store}()\{\mathsf{returned}_\alpha(\mathsf{store}) \circ I\}\{\top\} \tag{2}$$

The invoke and returned relations are defined below. The invoke simply adds an invocation (by clients of the overlay object) to the procedure f to the end of s and ρ. The returned asserts the returned result recorded in Δ is consistent with the one linearized in ρ by the prover.

$$(\Delta, s, \rho)\mathsf{invoke}_\alpha(f)(\Delta', s', \rho') \iff \begin{pmatrix} (\Delta, s, \rho) \in \mathsf{idle}_\alpha \wedge \exists a.\Delta'(\alpha) = [\mathsf{arg} \mapsto a] \wedge \\ \forall \alpha' \neq \alpha.\Delta'(\alpha') = \Delta(\alpha) \wedge s' = s \cdot \boldsymbol{\alpha}{:}f \wedge \rho' = \rho \cdot \boldsymbol{\alpha}{:}f \end{pmatrix}$$

$$(\Delta, s, \rho)\mathsf{returned}_\alpha(f)(\Delta', s', \rho') \iff (\Delta', s', \rho') = (\Delta, s, \rho) \wedge \exists v \in \mathsf{ar}(f).\Delta(\alpha)(\mathsf{ret}) = v \wedge \mathsf{last}(\pi_\alpha(\rho)) = \boldsymbol{\alpha}{:}v$$

These quadruples are proved by mainly using the Prim rule to step through primitive commands. In most of the cases, the underlay load/store operations only add pending overlay operations to the list B, and a consequent flush operation makes sure they are persisted and helps operations in B linearize. The Counter object prevents unnecessary flushes in this process but is not the main complexity of the FLiT object, and thus we refer readers to the appendix for its treatment.

Figure 6 shows the proof outline for the load operation, which we use as an example for demonstration. The program contains two potential linearization points, line 2 and line 5, and we show how to use the Prim rule to complete proofs and find linearizations at these points.

The underlay load operation at line 2 may execute from three different situations depending on the object state (Flushed, Unflushed, Unsynced). We choose to perform three different updates to the possibility ρ and the ghost list B and illustrate them through guarantee conditions below, which record the effects of these updates on proof configurations.

$$(s, \rho, B)\,\mathcal{G}_{\text{load-f}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v.\mathsf{Flushed}(s, B) \wedge (v, v) \in \mathsf{mstate}(s{\restriction}_{\mathsf{BCell}^{\lightning}}) \wedge v = \mathsf{fstate}(\rho) \wedge \\ \mathsf{lin}(\rho') = \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \wedge B' = \epsilon \wedge s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \end{pmatrix} \tag{3}$$

$$(s, \rho, B)\,\mathcal{G}_{\text{load-uf}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v.\mathsf{Unflushed}(s, B) \wedge \mathsf{last}(B{\restriction}_{\mathsf{store}}) = \mathsf{store}(v) \wedge \\ B' = B \cdot \boldsymbol{\alpha}{:}\mathsf{load} \wedge \rho' = \rho \wedge s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \end{pmatrix} \tag{4}$$

$$(s, \rho, B)\,\mathcal{G}_{\text{load-us}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v, B_1, B_2.\mathsf{Unsynced}(s, B) \wedge (v, v) \in \mathsf{mstate}(s{\restriction}_{\mathsf{BCell}^{\lightning}}) \wedge \\ s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \wedge \begin{pmatrix} (B = B_1 \cdot B_2 \wedge \mathsf{last}(B_1) = \mathsf{store}(v)) \\ \vee(\mathsf{fstate}(\rho) = v \wedge B_1 = \epsilon) \end{pmatrix} \wedge \\ \mathsf{lin}(\rho') = \mathsf{merge}(\mathsf{lin}(\rho), B_1) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \wedge B' = \epsilon \end{pmatrix} \tag{5}$$

```
$\{\mathsf{invoke}_\alpha(\mathsf{load}) \circ I\}$
load(){
    $\{I \wedge \boldsymbol{\alpha}{:}\mathsf{load} \in s_O \wedge (\mathsf{Flushed} \vee \mathsf{Unflushed} \vee \mathsf{Unsynced})\}$ // $P_{\mathsf{load}}$
    $v \leftarrow M.\mathsf{load}();$ // load-f/load-uf/load-us
    $\left\{I \wedge \begin{pmatrix} (\mathsf{Flushed} \wedge \mathsf{last}(\pi_\alpha(\rho)) = v) \vee \\ (\mathsf{Unflushed} \wedge (\exists B'. B' \cdot \boldsymbol{\alpha}'{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \vee \mathsf{last}(\pi_\alpha(\rho)) = v)) \end{pmatrix}\right\}$ // $Q_{\mathsf{load}}$
    $n \leftarrow C.\mathsf{get}();$
    $\left\{I \wedge \begin{pmatrix} (n = 0 \wedge \mathsf{last}(\pi_\alpha(\rho)) = v) \vee \\ (n \neq 0 \wedge (\exists B'. B' \cdot \boldsymbol{\alpha}'{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \vee \mathsf{last}(\pi_\alpha(\rho)) = v)) \end{pmatrix} \wedge (\mathsf{Flushed} \vee \mathsf{Unflushed})\right\}$
    if($n \neq 0$){
        $\{I \wedge (\mathsf{Flushed} \vee \mathsf{Unflushed}) \wedge (\exists B'. B' \cdot \boldsymbol{\alpha}'{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \vee \mathsf{last}(\pi_\alpha(\rho)) = v)\}$
        $M.\mathsf{flush}();$ // flush
        $\{I \wedge \mathsf{last}(\pi_\alpha(\rho)) = v\}$
    }
    $\{I \wedge \mathsf{last}(\pi_\alpha(\rho)) = v\}$
    ret $v$
}
$\{\mathsf{returned}_\alpha(\mathsf{load}) \circ I\}\ \{\top\}$
```

Fig. 6. A Proof Snippet of the load operation of FLiT Memory Cell

Load from Flushed State. When the underlay memory cell is at the Flushed state, i.e., there are no buffered operations and $B = \epsilon$, then, the current memory content $\mathsf{fstate}(\rho)$ is exactly the same as the content in the underlay memory cell v. Therefore, we can simply extend the linearized prefix $\mathsf{lin}(\rho)$ in ρ with $\boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v$ by reordering the pending load to the place and add the response as (3).

Load from Unflushed State. When the underlay memory is at the Unflushed state, there are different possible values for the persisted content. Although the underlay load will load the most recently buffered value v, we do not know whether v has been persisted or not. If a crash happens before returning from the current overlay load, this value may be lost from the memory and we are not supposed to linearize $\boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v$ to $\mathsf{lin}(\rho)$. Therefore, instead of linearizing it at this point, we choose to append the pending load to the buffered list B so that a subsequent flush operation from either the current program or other threads can help linearize it as (4).

Load from Unsynced State. The most special case is when the load is executed after a crash with some buffered store not flushed yet. As explained before, both the buffered and the persisted contents may have various values depending on previously buffered stores. The load operation will determine the actual content in the memory cell, which reveals and linearizes the operations that are persisted before the crash, making it an *external linearization point across crashes.*

Figure 7 shows an example of this kind of load operation. After a buffered store(2) operation, the persisted data has not been synchronized with the buffered value 2 since no flush has been performed, and the system crashes at this moment, resulting in a state with unknown content of the buffered cell. Just like (4), buffered store operations will be put into the list B instead of directly linearized into ρ. If the result of the load operation following the recovery is 2, like in this example, it implies that the buffered store operation has been persisted before the crash, and thus we can linearize the store(2) cached in B followed by the current load operation. In the other case, where the load after recovery gets 1, we know the buffered store operation failed to persist, and thus we do not linearize the store(2) and instead remove it from the list B.

We follow this pattern and modify the proof configuration as (5). We maintain as an object invariant that any persisted value in the underlay memory corresponds to some store in B or $\mathsf{lin}(\rho)$. Based on the return value v of the underlay load, we decide how to handle buffered operations

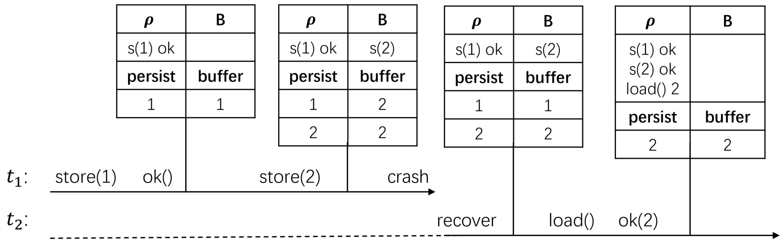


Fig. 7. External Linearization Point and Crash: the tables above the timeline show the content of the linearized trace ρ and the ghost list B in the first two rows and the mstate content in the remaining rows. $s(-)$ is a shorthand for the store$(-)$ operation.

in B. If v is the result of some store(v) in B, then we know this store(v) has persisted before the crash, and we linearize all operations B_1 (by reordering them before the crash, adding responses to invocations in B_1 and putting them after their corresponding invocations) in B preceding this store into $\mathsf{lin}(\rho)$ along with the current load operation and discard what remains in B.

Then by merging these three branches into one Hoare quadruple through the disjunction rule and weakening the post-condition to the stable Q_{load}, we prove the Hoare quadruple

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \models_\alpha \{P_{\mathsf{load}}\} v \leftarrow M.\mathsf{load}()\{Q_{\mathsf{load}}\}\{\top\}$$

at line 2 in Figure 6. According to the Prim rule, the quadruple is provable because we can prove $\mathcal{G} \vdash_\alpha \{P_{\mathsf{load}}\} v \leftarrow M.\mathsf{load}()\{Q_{\mathsf{load}}\}$ by our reasoning in previous paragraphs, i.e., any update obeys the rewrite relation $\rightsquigarrow_{\dagger F}$, and other entailments and stability checks are all true.

The post-condition Q_{load} indicates that either the current load is linearized and it is obvious that the returned value v is equal to the linearized value v, or the state is unflushed and the current load is buffered in B. In the second case, the proof that remains to be done for the rest of commands is still non-trivial. Specifically, the current load may be linearized by some external operations, or it will be linearized when the flush at line 5 takes place and we need to prove it is a valid linearization step. The proof of either case will follow the outline in Figure 6 and we can prove (1). We can also prove (2) and we refer readers to the appendix for its detailed proof.

5.4.2 Recovery Procedure Proof. The FLiT object has no recovery procedure, and therefore we use the empty recovery signature $R_\varnothing := \{r_\varnothing\}$ with the recovery program, $M[\alpha]^{r_\varnothing} :=$ `r() { ret ok }`. According to the Recover Impl rule, we need to prove the hexad $\mathsf{ID}, \top \vDash_\alpha \{I\} M[\alpha]^{r_\varnothing}\{I\}\{I\}$ for $r_\varnothing$, which reduces to the idempotence of the invariant w.r.t. crashes, i.e., $I \Rightarrow_{\lightning} I$.

As we have shown $\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \models^F_\alpha M[\alpha]$ and $I \models^R_\alpha M[\alpha]$ for any $\alpha \in \Upsilon$, according to the Object Impl rule, we prove $\vdash M_{\mathsf{FLiT}} : (v'_{\mathsf{BCell}} \otimes v'_{\mathsf{Counter}}, v_{\mathsf{BCell}} \otimes v_{\mathsf{Counter}}) \longrightarrow \langle v'_{\mathsf{FLiT}}, v_{\mathsf{FLiT}} \rangle$, i.e., the FLiT memory cell is durably linearizable. Based on the FLiT memory cell, we implement a durable version of the one-shot write-snapshot object [6], a famous interval-sequential [7] concurrent object. We prove its linearizability using the logic in Oliveira Vale et al. [31] and use the FLiT correctness theorem 1.1 to derive its durable linearizability.

We also prove the transactional file system to be crash-aware linearizable with the crash-aware linearizability variant of CLHL. It demonstrates CLHL's ability to verify non-trivial recoveries, and to decompose complicated systems into multiple layers with simpler proofs and then easily compose these proofs to obtain the originally challenging proof of the entire system.

6 Related Works

Game Semantics. Our game semantics model is directly based on that of Ghica and Murawski [16], Oliveira Vale et al. [31], and our use of object-based game semantics traces back to Oliveira Vale et al. [30], Reddy [35, 36]. To develop our crash-aware model, we indirectly made use of insights from Mellies [28]. In its goal of describing systems written in imperative languages, our game semantics is related to some of the work by Ghica and Tzevelekos [17], Koenig and Shao [25]. It is important to note that crashes are not accurately modeled as a separate computational agent responsible for issuing crashes: crashes are instantaneous and pervasive, synchronous across components, are not invoked, and are unimplemented. Because of this, our crash-aware model is rather unorthodox in that it breaks the tradition of having only two players (Opponent and Proponent) by adding an extra player for crash events. It models crash events differently from usual moves in traditional game models by having crash events happen instantaneously and synchronously across all components, while typically, a move belongs to a single component and happens mostly asynchronously. As far as we are aware, this is the first game semantics of its kind. Because of this, while we build on the model from Oliveira Vale et al. [31] and benefit significantly from the theory there, our model needs to address the intrusive effects of properly modeling crashes.

Linearizability with Crashes. We already discussed some of the history of linearizability criteria with crashes throughout the paper [2, 4, 19, 22]. In our paper, we address strict linearizability (in the context of full-system crashes) and durable linearizability. We generalize both of them by not requiring the linearized specifications to be atomic and by allowing for blocking objects. This makes our variations of these linearizability criteria closer to interval-sequential linearizability [7]. We formulate these criteria in the style of compositional linearizability [31], which is novel. This allows us to give simple proofs of locality, develop a compositional verification framework around these criteria, give the first proof of observational refinement properties for these two criteria, and provide a counterpart to the analogous result proved for Herlihy-Wing linearizability [14] and for compositional linearizability [31]. We also discover that the inherent notion of linearizability to crash-aware objects is the linearizability criterion we called crash-aware linearizability (§4) satisfying locality and observational refinement. Although related to strict linearizability, it does not appear elsewhere. We note that while crash-aware linearizability is the compositional linearizability [31] one gets from our model $\underline{\mathbf{Crash}}$, our formulations of strict and durable linearizability impose new challenges and new structures, in particular, because they relate two distinct models of computation (concurrency with and without crashes). We conjecture that this different structure can be reconciled with that from compositional linearizability through a weakening of the notion of a Grothendieck fibration, following ideas from functorial refinement [29].

Verification with Crashes. There are approaches for verifying systems with crashes that do not involve linearizability. Much of the work on this line has been done in the context of file system verification. A perhaps notable start is the development of Crash Hoare Logic [11], later refined into recovery refinement [8], and generalized to handle concurrent systems [9, 10]. Of these, only Chajed et al. [8], which only handles sequential systems, formally proves a refinement theorem that enables building large systems. The later variants that handle concurrency lack such a contextual refinement theorem. These works, different from ours, have been mechanized.

Another important work is Khyzha and Lahav [24], which proves a contextual refinement theorem for programs with crashes. Quite interesting is the fact that their approach is reminiscent of that used by Oliveira Vale et al. [31] and by us, in that they define a notion of refinement by composition with a "Most General Client". This most general client seems to be a special case of the copycat strategies that appear in our game models. Since they do this using operational semantics, we believe their work is further evidence of the practicality of our approach. Moreover, their programming language features a buffered memory interface with global flushes, which our example does not. Despite the similarities, they only address linearizability by providing a few examples where linearizability specifications can be encoded in their framework, but they do not describe a generic framework to do so, nor prove a formal connection with linearizability. Modeling a memory model with global flushes in our model is straightforward: its specification is almost the same as our buffered memory cell arrays, but with a requirement of proving a memory separation property, like they had to do. We do not do this here as it was not required for our examples.

A recent line of work proves linearizability specifications, but only for a single component [13], and focuses on data structures implemented on top of NVM only. It is quite impressive in that it assumes a weak memory model, which requires handling weak consistency models, which we do not. Despite that, they do not provide a program logic and are closer to axiomatic approaches, which could hinder scalability. It is unlikely that their framework could be generalized to a compositional verification methodology without significant effort.

Concurrently to our work Bodenmüller et al. [5] verified the FLiT library and have a mechanized proof of correctness. Part of their simulation-based technique is reminiscent of our use of refinement and $\mathrm{dur}(-)$, which they define as a specific transformation of a state-transition system into another and do not note its relationship to the structure of some compositional model (which they do not develop). Their technique is restricted to durable linearizability w.r.t. atomic specifications and is specialized in verifying persistency libraries over NVM. Our work is, therefore, significantly more general in scope. Our FLiT correctness theorem shows that linearizable objects in the sense of Oliveira Vale et al. [31] are transformed into durable linearizable libraries in our sense, and therefore applies even to non-atomic and blocking objects, proving a stronger correctness theorem for FLiT (in fact, stronger than the FLiT author's informal claim of correctness, for the same reasons).

Our program logic is the first to verify a linearizability criterion with crashes. It is based on Khyzha et al. [23], Oliveira Vale et al. [31], and takes inspiration from Crash Hoare Logic and Argosy [8]. It differs from the aforementioned works in that it proves durable, and crash-aware linearizability specifications. The compositional framework, which we directly connect with our program logic, is the only one that simultaneously provides refinement, linearizability specifications, and vertical and horizontal composition. Our theory allows us to state the correctness of systems like FLiT [38]. We also show we can verify a simplified variant of a file system interface. Note that previous file system interfaces are not verified against linearizability specifications, which are deemed as more intuitive than the kind of specifications one gets from HOCAP style specifications [12, 37].

Acknowledgments

We thank the reviewers of the current and previous iterations of this work for their thoughtful revisions. This material is based upon work supported in part by NSF grants 2313433, 2019285, and 1763399, and by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Samson Abramsky and Guy McCusker. 1999. Game Semantics. In *Computational Logic*, Ulrich Berger and Helmut Schwichtenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–55. https://doi.org/10.1007/978-3-642-58622-4_1

[2] Marcos K Aguilera and Svend Frølund. 2003. Strict linearizability and the power of aborting. *Technical Report HPL-2003-241* (2003).

[3] Naama Ben-David, Michal Friedman, and Yuanhao Wei. 2022. Brief Announcement: Survey of Persistent Memory Correctness Conditions. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 41:1–41:4. https://doi.org/10.4230/LIPIcs.DISC.2022.41

[4] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 2016. Robust Shared Objects for Non-Volatile Main Memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 46)*, Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru (Eds.). Schloss Dagstuhl, Dagstuhl, Germany, 20:1–20:17. https://doi.org/10.4230/LIPIcs.OPODIS.2015.20

[5] Stefan Bodenmüller, John Derrick, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. 2024. A Fully Verified Persistency Library. In *Verification, Model Checking, and Abstract Interpretation*, Rayna Dimitrova, Ori Lahav, and Sebastian Wolff (Eds.). Springer Nature Switzerland, Cham, 26–47. https://doi.org/10.1007/978-3-031-50521-8_2

[6] Elizabeth Borowsky and Eli Gafni. 1993. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing* (Ithaca, New York, USA) *(PODC '93)*. Association for Computing Machinery, New York, NY, USA, 41–51. https://doi.org/10.1145/164051.164056

[7] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2015. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363* (Tokyo, Japan) *(DISC 2015)*. Springer-Verlag, Berlin, Heidelberg, 420–435. https://doi.org/10.1007/978-3-662-48653-5_28

[8] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Argosy: verifying layered storage systems with recovery refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1054–1068. https://doi.org/10.1145/3314221.3314585

[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. https://doi.org/10.1145/3341301.3359632

[10] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 423–439.

[11] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

[12] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

[13] Emanuele D'Osualdo, Azalea Raad, and Viktor Vafeiadis. 2023. The Path to Durable Linearizability. *Proc. ACM Program. Lang.* 7, POPL, Article 26 (jan 2023), 27 pages. https://doi.org/10.1145/3571219

[14] Ivana Filipovic, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. 2010. Abstraction for Concurrent Objects. *Theor. Comput. Sci.* 411, 51–52 (dec 2010), 4379–4398. https://doi.org/10.1016/j.tcs.2010.09.021

[15] Dan R. Ghica. 2019. The far side of the cube. *CoRR* abs/1908.04291 (2019). arXiv:1908.04291 http://arxiv.org/abs/1908.04291

[16] Dan R. Ghica and Andrzej S. Murawski. 2004. Angelic Semantics of Fine-Grained Concurrency. In *Foundations of Software Science and Computation Structures*, Igor Walukiewicz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 211–225. https://doi.org/10.1016/j.apal.2007.10.005

[17] Dan R. Ghica and Nikos Tzevelekos. 2012. A System-Level Game Semantics. *Electronic Notes in Theoretical Computer Science* 286 (2012), 191–211. https://doi.org/10.1016/j.entcs.2012.08.013 Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII).

[18] Éric Goubault, Jérémy Ledent, and Samuel Mimram. 2018. Concurrent Specifications Beyond Linearizability. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 125)*, Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 28:1–28:16. https://doi.org/10.4230/LIPIcs.OPODIS.2018.28

[19] Rachid Guerraoui and Ron R. Levy. 2004. Robust Emulations of Shared Memory in a Crash-Recovery Model. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04) (ICDCS '04)*. IEEE Computer Society, USA, 400–407. https://doi.org/10.1109/ICDCS.2004.1281605

[20] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

[21] Martin Hyland. 1997. Game Semantics. In *Semantics and Logics of Computation*, Andrew M. Pitts and P.Editors Dybjer (Eds.). Cambridge University Press, Cambridge, UK, 131–184. https://doi.org/10.1017/CBO9780511526619.005

[22] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[23] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew Parkinson. 2017. Proving Linearizability Using Partial Orders. In *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017* (Uppsala, Sweden). Springer-Verlag, Berlin, Heidelberg, 639–667. https://doi.org/10.1007/978-3-662-54434-1_24

[24] Artem Khyzha and Ori Lahav. 2022. Abstraction for Crash-Resilient Objects. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 262–289. https://doi.org/10.1007/978-3-030-99336-8_10

[25] Jérémie Koenig and Zhong Shao. 2020. Refinement-Based Game Semantics for Certified Abstraction Layers. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 633–647. https://doi.org/10.1145/3373718.3394799

[26] Hongjin Liang and Xinyu Feng. 2013. Modular verification of linearizability with non-fixed linearization points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 459–470. https://doi.org/10.1145/2491956.2462189

[27] Nancy Lynch and Frits Vaandrager. 1996. Forward and Backward Simulations. *Inf. Comput.* 128, 1 (jul 1996), 1–25. https://doi.org/10.1006/inco.1996.0060

[28] Paul-André Mellies. 2019. Categorical Combinatorics of Scheduling and Synchronization in Game Semantics. *Proc. ACM Program. Lang.* 3, POPL, Article 23 (jan 2019), 30 pages. https://doi.org/10.1145/3290336

[29] Paul-André Melliès and Noam Zeilberger. 2015. Functors are Type Refinement Systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 3–16. https://doi.org/10.1145/2676726.2676970

[30] Arthur Oliveira Vale, Paul-André Melliès, Zhong Shao, Jérémie Koenig, and Léo Stefanesco. 2022. Layered and Object-Based Game Semantics. *Proc. ACM Program. Lang.* 6, POPL, Article 42 (jan 2022), 32 pages. https://doi.org/10.1145/3498703

[31] Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. 2023. A Compositional Theory of Linearizability. *Proc. ACM Program. Lang.* 7, POPL, Article 38 (jan 2023), 32 pages. https://doi.org/10.1145/3571231

[32] Arthur Oliveira Vale, Zhong Shao, and Yixuan Chen. 2024. A Compositional Theory of Linearizability. *J. ACM* 71, 2, Article 14 (apr 2024), 107 pages. https://doi.org/10.1145/3643668

[33] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (dec 2019), 31 pages. https://doi.org/10.1145/3371079

[34] Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 135 (oct 2019), 27 pages. https://doi.org/10.1145/3360561

[35] Uday S. Reddy. 1993. *A Linear Logic Model of State.* Technical Report. Dept. of Computer Science, UIUC, Urbana, IL.

[36] Uday S. Reddy. 1996. Global State Considered Unnecessary: An Introduction to Object-Based Semantics. *LISP Symb. Comput.* 9, 1 (1996), 7–76. https://doi.org/10.1007/978-1-4757-3851-3_9

[37] Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–168. https://doi.org/10.1007/978-3-642-54833-8_9

[38] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: a library for simple and efficient persistent algorithms. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 309–321. https://doi.org/10.1145/3503221.3508436

Summary of the Appendices

A Provides a more detailed account of the sequential, concurrent and crash-aware game models used in the main paper.

B Gives our extended treatment of crash-aware linearizability.

C Prepares the ground for strict and durable linearizability by discussing the recrash and decrash operations.

D Gives our treatment of strict linearizability, including proofs of locality and an observational refinement property for it.

E Gives our extended treatment of durable linearizability.

F Defines more carefully the kinds of strategies that serve as denotations of the imperative programs in our program logic section.

G Gives the full account of our program logic for durable linearizability, including the full definition of our programming language and operational semantics, all the program logic rules, as well as the proof of soundness.

H Serves as a continuation of Appendix G by giving a small modification of the program logic there so that it shows crash-aware linearizability instead of durable linearizability.

I Collects the detailed specification and proofs in our program logic of the FLiT and File examples discussed in the main paper.

J Collects all of the proofs omitted elsewhere.

A A Concurrent Game Semantics with Crashes

In this section we define our game model with crashes. This does require a long exposition, and a few different game models. This proves necessary, as durable linearizability involves crash-aware objects as well as objects without crashes, which live in different models of computation. We start by recalling the definitions of sequential games Seq and concurrent games Conc appearing in Oliveira Vale et al. [31], slightly reformulating them in the process (§A.1). Then, we define our concurrent game model with full-system crashes (§A.2). We finish by discussing the issue of neutral elements and define the copycat strategy crashcopy that will play the role of the neutral element in our compositional model (§A.3), and provide a concrete characterization of saturation with respect to crashcopy.

A.1 Concurrent Games

We now recall the sequential and concurrent games used in Oliveira Vale et al. [31]. The sequential game mode is traditional in the game semantics literature [1, 21] except that our presentation differs in, at least at first, not requiring *O-receptivity*.

The concurrent game model is closely related to the model appearing in Ghica and Murawski [16]. We slightly generalize the model in Oliveira Vale et al. [31]. There, concurrent games are *homogenous* in that all agents play the same game locally, i.e. have access to copies the same operations. The model we define here is *hetererogenous* in the sense that it allows different agents to play different games locally.

At this point, we must note that our model of concurrent games is parametrized by a set Υ of agent names. We start by defining a notion of move of a game, and what it means to be a well-formed sequential and concurrent play.

Definition A.1 (Move Sets and Well-Formed Plays). We define the set of polarities for sequential games $\mathsf{Pol}^{\mathsf{seq}}$ and concurrent games $\mathsf{Pol}^{\mathsf{conc}}$ respectively as the sets:

$$\mathsf{Pol}^{\mathsf{seq}} := \{O, P\} \qquad\qquad \mathsf{Pol}^{\mathsf{conc}} := \sum_{\alpha\in\Upsilon} \mathsf{Pol}^{\mathsf{seq}}$$

We define a *move set* as a pair $(M, \lambda : M \longrightarrow \mathsf{Pol})$ of a set of *moves* and a polarity assigning function $\lambda : M \longrightarrow \mathsf{Pol}$ to a set of *polarities*. Given such a move set, we write M^{pol} for largest subset of M including only moves $m \in M$ such that $\lambda(m) = \mathsf{pol}$.

In the case of a move set $(M, \lambda : M \longrightarrow \mathsf{Pol}^{\mathsf{conc}})$ we also write M^{α} for the largest subset of M including only moves $\lambda(m) = \boldsymbol{\alpha}{:}\mathsf{pol}$ for any polarity $\mathsf{pol} \in \mathsf{Pol}^{\mathsf{seq}}$. Note that M^{α} defines a move set $(M^{\alpha}, \lambda^{\alpha} : M \longrightarrow \mathsf{Pol}^{\mathsf{seq}})$ by the polarity assignment

$$\lambda^{\alpha}(m) = \mathsf{pol} \iff \lambda(m) = \boldsymbol{\alpha}{:}\mathsf{pol}$$

Given a sequence $s \in M^*$ and a subset $S \subseteq \Upsilon$, we denote by $\pi_{\alpha}(s) \in M^*$ the largest subsequence of s containing only moves in M^{α}.

Given a move set $(M, \lambda : M \longrightarrow \mathsf{Pol}^{\mathsf{seq}})$ we write $\mathbb{P}^{\mathsf{seq}}_{M}$ for the set of sequences $s \in M^*$ that start with a move labelled by O, and alternate between O and P moves in the sense that

$$s = p \cdot m \cdot m' \cdot t \implies \lambda(m) \neq \lambda(m')$$

We call sequences in $\mathbb{P}^{\mathsf{seq}}_{M}$ well-formed sequential plays.

Similarly, given a move set $(M, \lambda : M \longrightarrow \mathsf{Pol}^{\mathsf{conc}})$ we write $\mathbb{P}^{\mathsf{conc}}_{M}$ for the set of sequences $s \in M^*$ which are locally sequential in that $\pi_{\alpha}(s) \in \mathbb{P}^{\mathsf{seq}}_{M^{\alpha}}$. We call sequences in $\mathbb{P}^{\mathsf{conc}}_{M}$ well-formed concurrent plays.

We are now ready to define the key notion of a game.

Definition A.2 (Sequential and Concurrent Games). A sequential game $A = ((M_A, \lambda_A), P_A)$ consists of a move set $(M_A, \lambda_A : M_A \longrightarrow \mathsf{Pol}^{\mathsf{seq}})$ and a non-empty, prefix-closed, subset $P_A \subseteq \mathbb{P}^{\mathsf{seq}}_{A}$, where we write $\mathbb{P}^{\mathsf{seq}}_{A}$ for $\mathbb{P}^{\mathsf{seq}}_{M_A}$.

A concurrent game $A = ((M_A, \lambda_A), P_A)$ consists of a move set $(M_A, \lambda_A : M_A \longrightarrow \mathsf{Pol}^{\mathsf{conc}}_{\Upsilon})$ and a non-empty, prefix-closed subset $P^{\alpha}_{A} \subseteq \mathbb{P}^{\mathsf{seq}}_{M^{\alpha}_{A}}$ for each $\alpha \in \Upsilon$ verifying that $P_A = \|_{\alpha \in \Upsilon} P^{\alpha}_{A}$

Given a game A we can recover the local game that α plays, written A^{α}, as the sequential game $A^{\alpha} = ((M^{\alpha}_{A}, \lambda^{\alpha}_{A}), \pi_{\alpha}(P_A))$. Note that by construction, given $s \in P_A$ it is necessarily the case that $\pi_{\alpha}(s) \in P_{A^{\alpha}}$.

Conversely, given an Υ-indexed collection of sequential games $A = (A[\alpha])_{\alpha \in \Upsilon}$ we define the concurrent game $\mathrm{Conc}\ A$ by the data

$$M_{\mathrm{Conc}\ A} := \sum_{\alpha \in \Upsilon} M_{A[\alpha]} \qquad \lambda_{\mathrm{Conc}\ A} := \sum_{\alpha \in \Upsilon} \lambda_{A[\alpha]} \qquad P_{\mathrm{Conc}\ A} := \Big\|_{\alpha \in \Upsilon} \boldsymbol{\alpha}{:}P_{A[\alpha]}$$

when the context permits, we write A instead $\mathrm{Conc}\ A$. It is useful to note that given a concurrent game A, we always have $A \cong (A^{\alpha})_{\alpha \in \Upsilon}$, where the isomorphism holds up to renaming some moves. So, up-to isomorphism, every concurrent game may be seen as a collection $A = (A[\alpha])_{\alpha \in \Upsilon}$ of sequential games.

An example of a sequential game is the unit game Σ, in which Opponent is able to ask a question q and Proponent may answer the unique answer available to it a. Formally, its moves are $M_{\Sigma} := \{q, a\}$ where q is an Opponent move, i.e. $\lambda_{\Sigma}(q) = O$, and a is a Proponent move, i.e. $\lambda_{\Sigma}(a)$, while its set of plays is given by

$$P_{\Sigma} := \{\epsilon \quad , \quad q \quad , \quad q \cdot a\}$$

We can use the sequential unit game to define an Υ-indexed collection Σ with $\Sigma[\alpha] := \Sigma$. The corresponding game Σ simply allows all agents in Υ to play an instance of Σ locally, so that its plays are sequentially consistent interleavings of plays of Σ labelled by the agent issuing the corresponding moves. For example, for $\alpha \neq \alpha'$ both agents in Υ, the following sequence a play of Σ:

$$\begin{tikzcd}
\boldsymbol{\alpha}{:}q \arrow[rrr, bend right] & \boldsymbol{\alpha'}{:}q \arrow[r] & \boldsymbol{\alpha'}{:}a & \boldsymbol{\alpha}{:}a
\end{tikzcd}$$

where the arrows are merely a visual aid in keeping track of the individual threads of computation.

Definition A.3 (Strategies). For a sequential/concurrent game A, a sequential/concurrent *strategy* σ over A, written $\sigma : A$, is a non-empty, prefix-closed subset $\sigma \subseteq P_A$.

Definition A.4 (Dual Move Set). Given a move set $(M, \lambda : M \longrightarrow \mathsf{Pol}^{\mathsf{seq}})$ we define $(M^\perp, \lambda^\perp : M^\perp \longrightarrow \mathsf{Pol}^{\mathsf{seq}})$ by

$$M^\perp := M \qquad\qquad \lambda^\perp(m) := (\lambda(m))^\perp$$

where $O^\perp := P$ and $P^\perp := O$.

Given a move set $(M, \lambda : M \longrightarrow \mathsf{Pol}^{\mathsf{conc}})$ we define $(M^\perp, \lambda^\perp : M^\perp \longrightarrow \mathsf{Pol}^{\mathsf{conc}})$ by

$$M^\perp := M \qquad\qquad \lambda^\perp(m) := (\lambda(m))^\perp$$

where $(\boldsymbol{\alpha}\mathbf{:}\mathsf{pol})^\perp := \boldsymbol{\alpha}\mathbf{:}\mathsf{pol}^\perp$ for $\mathsf{pol} \in \mathsf{Pol}^{\mathsf{seq}}$.

Definition A.5 (Tensor and Affine Implication). Given sequential/concurrent games A and B we define the sequential/concurrent games $A \otimes B$ and $A \multimap B$ by the following data

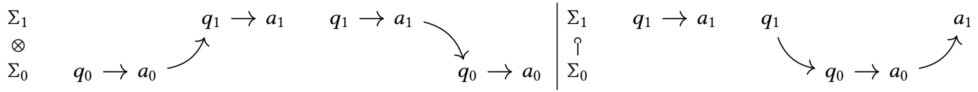
$$M_{A\otimes B} := M_A + M_B \qquad \lambda_{A\otimes B} := \lambda_A + \lambda_B \qquad P_{A\otimes B} := \{s \in \mathbb{P}_{A\otimes B} \mid s\restriction_{A,-} \in P_A \wedge s\restriction_{-,B} \in P_B\}$$

$$M_{A\multimap B} := M_A^\perp + M_B \qquad \lambda_{A\multimap B} := \lambda_A^\perp + \lambda_B \qquad P_{A\multimap B} := \{s \in \mathbb{P}_{A\multimap B} \mid s\restriction_{A,-} \in P_A \wedge s\restriction_{-,B} \in P_B\}$$

where $-\restriction_{A,-} : (M_A + M_B)^* \longrightarrow M_A$ assigns to a sequence s its largest subsequence involving only elements in M_A, and analogously for $-\restriction_B : (M_A + M_B)^* \longrightarrow M_B$.

The plays of $A \otimes B$ are essentially plays of A and B interleaved in a sequentially consistent way, so that $A \otimes B$ corresponds to independent horizontal composition. The game $A \multimap B$ meanwhile corresponds to switching the roles of Opponent and Proponent in A and then taking the tensor with B, which is the traditional way of modelling the affine implication type in game models.

As a matter of illustration, the maximal plays (under prefix ordering) for the sequential games $\Sigma_0 \otimes \Sigma_1$ (the two plays on the left) and $\Sigma_0 \multimap \Sigma_1$ (the two plays on the right) are depicted below. We denote by Σ_0, Σ_1 the two components of these types, both of which are instances of the game Σ. We will similarly add an index to the moves of each component.


Σ_1
$\otimes$
Σ_0
$q_0 \longrightarrow a_0$
$q_1 \longrightarrow a_1$
$q_1 \longrightarrow a_1$
$q_0 \longrightarrow a_0$
Σ_1
$\multimap$
Σ_0
$q_1 \longrightarrow a_1$
q_1
$q_0 \longrightarrow a_0$
a_1

Observe that in the game $\Sigma \otimes \Sigma$ Opponent can choose to start in either component, while in the game $\Sigma \multimap \Sigma$ Opponent must start in the target component (Σ_1) due to the flip of polarity in the source component (Σ_0). In $\Sigma \otimes \Sigma$ only Opponent may switch components, while in $\Sigma \multimap \Sigma$ only Proponent may switch components because of alternation (these are typically called the switching conditions of sequential games). Their concurrent variants $\Sigma \otimes \Sigma$ and $\Sigma \multimap \Sigma$ merely allow for any sequentially consistent interleaving of the plays above, labelled by the agents who are performing each move.

Definition A.6. For sequential/concurrent games A, B and C, we define the set of sequential/concurrent interaction plays as

$$\mathsf{int}(A, B, C) := \{s \in (M_A + M_B + M_C)^* \mid s\restriction_{A,B,-} \in P_{A\multimap B} \wedge s\restriction_{-,B,C} \in P_{B\multimap C}\}$$

where $-\restriction_{A,B,-} : (M_A + M_B + M_C)^* \longrightarrow (M_A + M_B)^*$ assigns to s the largest subseqeunce of s involving only events in M_A and M_B, and analogously for the projection $-\restriction_{-,B,C}$.

Given sequential/concurrent strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ we define their set of interactions by

$$\mathsf{int}(\sigma, \tau) := \{s \in \mathsf{int}(A, B, C) \mid s\restriction_{A,B,-} \in \sigma \wedge s\restriction_{-,B,C} \in \tau\}$$

and their composition, for sequential strategies and concurrent strategies, respectively:

$$\sigma;\tau := \{s\restriction_{A,-,C} \in \mathbb{P}^{\mathsf{seq}}_{A\multimap C} \mid s \in \mathsf{int}(\sigma,\tau)\} \qquad \sigma;\tau := \{s\restriction_{A,-,C} \in \mathbb{P}^{\mathsf{conc}}_{A\multimap C} \mid s \in \mathsf{int}(\sigma,\tau)\}$$

where, $-\restriction_{A,-,C} : (M_A + M_B + M_C)^* \longrightarrow (M_A + M_C)^*$ assigns to s the largest subsequence of s containing only moves in M_A and M_C.

Proposition A.7. *Composition of sequential/concurrent strategies is well-defined and associative.*

Prop. A.7, proved by Oliveira Vale et al. [31], means that sequential games, and concurrent games both assemble into semicategories. Recall that a semicategory is essentially a category without the requirements about the neutral element for composition.

Definition A.8. We call $\underline{\mathbf{Seq}}$ the semicategory of sequential games and sequential strategies, and $\underline{\mathbf{Conc}}$ the semicategory of concurrent games and concurrent strategies.

An important class of games for us, as they will make for the types of our concrete objects and will play a key rule in defining the semantics of imperative code, are games generated by effect signatures. This follows an approach for modeling imperative programs started in Koenig and Shao [25], Oliveira Vale et al. [30]. First, we recall the definition of effect signature and define a notion of concurrent effect signature.

Definition A.9. An effect signature is given by a collection of operations, or effects, $E = (e_i)_{i\in I}$ together with an assignments $\mathsf{par}(-), \mathsf{ar}(-) : E \longrightarrow \mathbf{Set}$ of a set of parameters $\mathsf{par}(e)$ and a set of return values $\mathsf{ar}(e)$ for each operation $e \in E$. This is conveniently described by the following notation:

$$E = \{e_i : \mathsf{par}(e_i) \to \mathsf{ar}(e_i) \mid i \in I\}$$

For a given set of agents Υ we call an Υ-indexed collection of effect signatures $E = (E[\alpha])_{\alpha\in\Upsilon}$ a concurrent effect signature.

The corresponding games associated sequential and concurrent games associated to effect signatures are as follows.

Definition A.10. Given an effect signature E, define the game $\mathbf{Seq}(E) \in \underline{\mathbf{Seq}}$ by the data:

$$M_{\mathbf{Seq}(E)} := \sum_{e\in E} \mathsf{par}(e) + \sum_{e\in E} \mathsf{ar}(e) \qquad \lambda_{\mathbf{Seq}(E)} := O + P \qquad P_{\mathbf{Seq}(E)} := \downarrow \cup_{e\in E} \cup_{a\in\mathsf{par}(e)} \cup_{v\in\mathsf{ar}(e)} e(a)\cdot v$$

we take the freedom of writing E for $\mathrm{Seq}(E)$.

Then, given a concurrent effect signature E, its corresponding game $\mathbf{Conc}(E)$ in $\underline{\mathbf{Conc}}$ is the game Conc E. We will often write just E for Conc E.

The typical play of E looks like the following (on the left), where $e \in E$ is an effect, $a \in \mathsf{par}(e)$ and $v \in \mathsf{ar}(e)$.

$$E : e(a) \longrightarrow v \qquad\qquad \dagger E : e_1(a_1) \to v_1 \to e_2(a_2) \to v_2 \to \ldots \to e_n(a_n) \to v_n$$

Note that E only allows for a single effect of E to be issued. We can lift such a game E to a game $\dagger E$ (read “replay E”) that allows several effects of E to be invoked in sequence. Its plays, depicted above on the right, consist of sequences of invocations $e_i \in E$ with argument $a_i \in \mathsf{par}(e_i)$ alternating with their responses $v_i \in \mathsf{ar}(e_i)$. We will define the replay modality $\dagger-$ formally later. This informal description will suffice until then. The concurrent variant E, as usual, just allows each agent to play an instance of the corresponding sequential game $E[\alpha]$ in a sequentially consistent fashion, while $\dagger E$ similarly allows each agent to play $\dagger E[\alpha]$ locally.

A.2 Games with Full-System Crashes

We are now ready to define our model of concurrent computation with full-system crashes. We follow the same structure as §A.1.

Definition A.11 (Polarities with Crashes). We define the set of polarities for crash-aware games $\mathsf{Pol}^{\lightning}$ as the set $\mathsf{Pol}^{\lightning} := \mathsf{Pol}^{\mathsf{conc}} + \{\lightning\}$.

Note that crash-aware games will effectively have an extra player which is responsible for issuing crash signals, which are represented by moves labelled by $\lightning$ and will be treated differently from O and P moves.

Definition A.12 (Move Set and Well-Formed Plays). We define a crash-aware move set to be a move set $(M, \lambda : M \longrightarrow \mathsf{Pol}^{\lightning})$ such that $M^{\lightning}$ is a singleton set.

We write M^{Υ} for the largest subset of M including only moves $m \in M$ such that $\lambda(m) \neq \lightning$, i.e. $M^{\Upsilon} = \cup_{\alpha \in \Upsilon} M^{\alpha}$. Note that M^{Υ} always defines a move set $(M^{\Upsilon}, \lambda^{\Upsilon} : M^{\Upsilon} \longrightarrow \mathsf{Pol}^{\mathsf{conc}})$ by the polarity assignment $\lambda^{\Upsilon}(m) := \lambda(m)$.

Note that a sequence $s \in M^*$ is of the form

$$s_1 \cdot \boldsymbol{\lightning} \cdot s_2 \cdot \boldsymbol{\lightning} \cdot \ldots \cdot \boldsymbol{\lightning} \cdot s_{n+1}$$

where every $s_i \in (M^{\Upsilon})^*$ and $\boldsymbol{\lightning} \in M_A^{\lightning}$. We define $\|s\|$ to be $n + 1$. We also define the operation $\mathsf{epo}_i(-)$ which assigns to s the sequence $\mathsf{epo}_i(s) := s_i$, called the i-th *epoch* of s. If $i > \|s\|$ then we take the convention that $\mathsf{epo}_i(s) := \epsilon$.

We say a sequence $s \in M^*$ is a well-formed crash-aware play when, for every $i \in \mathbb{N}$, $\mathsf{epo}_i(s) \in \mathbb{P}^{\mathsf{conc}}_{M^{\Upsilon}}$. We denote by $\mathbb{P}^{\lightning}_{M}$ the set of all well-formed crash-aware plays over the move set $(M, \lambda : M \longrightarrow \mathsf{Pol}^{\lightning})$.

Note that in the definition of the well-formedness of crash-aware plays we implicitly already enforce that crashes affect the entire system, as when a crash happens the entire system resets back to a P-position. This means that after a crash, the first move for all agents is an O move, no matter what was the last move by that agent in the previous epoch. We are now ready to define crash-aware games. In the definition,and for the remainder of this paper, we will make frequent use of the usual Kleene algebra over sequences. We will also denote by $\boldsymbol{\lightning}$ the unique crash event $\boldsymbol{\lightning} \in M^{\lightning}$, for any crash-aware move set, what allows us to ignore the actual name of the crash event. While the assumption that $M_A^{\lightning}$ is a singleton is not necessary, allowing for several crash events makes several of the definitions more involved. Since for our practical purposes one crash event is enough, we opt for this simpler presentation.

Definition A.13 (Crash-Aware Game). A crash-aware game $A = (M_A, \lambda_A, P_A)$ consists of a move set $(M_A, \lambda_A : M_A \longrightarrow \mathsf{Pol}^{\lightning})$ and a non-empty, prefix-closed subset $P_A^{\Upsilon} \subseteq \mathbb{P}_A^{\mathsf{conc}}$ making

$$P_A = (P_A^{\Upsilon} \cdot \boldsymbol{\lightning})^* \cdot P_A^{\Upsilon} \subseteq \mathbb{P}^{\lightning}_{M_A}$$

Note that any crash-aware game A defines a concurrent game $A^{\Upsilon} := ((M_A^{\Upsilon}, \lambda_A^{\Upsilon}), P_A^{\Upsilon}) \in \underline{\mathbf{Conc}}$.

Conversely, given a concurrent game $A = ((M_A, \lambda_A), P_A)$ we can construct a crash-aware game $A^{\lightning} := ((M_A + \{\boldsymbol{\lightning}\}, \lambda_A + \lightning), (P_A \cdot \boldsymbol{\lightning})^* \cdot P_A)$ where we write $\lightning$ for the constant function $\lightning : \{\boldsymbol{\lightning}\} \longrightarrow \{\lightning\}$. This game has every agent $\alpha \in \Upsilon$ playing the concurrent game A. It is useful to observe that given a crash-aware game A, $(A^{\Upsilon})^{\lightning} \cong A$.

So, for example, the crash-aware version of Σ is given by $\Sigma^{\lightning}$. Then, an example of play of $\Sigma^{\lightning}$ is:

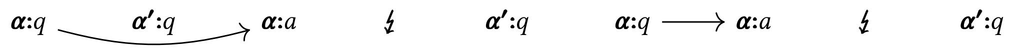

$\boldsymbol{\alpha}{:}q$
$\boldsymbol{\alpha'}{:}q$
$\boldsymbol{\alpha}{:}a$
$\lightning$
$\boldsymbol{\alpha'}{:}q$
$\boldsymbol{\alpha}{:}q$
$\boldsymbol{\alpha}{:}a$
$\lightning$
$\boldsymbol{\alpha'}{:}q$

Fig. 8. Example of a play of $\Sigma_{\lightning}$.

Note that the main well-formedness constraint about plays of crash-aware games is that in each epoch they play a well-formed concurrent play. i.e. a locally sequential play.

Definition A.14 (Crash-Aware Strategy). A strategy $\sigma : A$ over a game A is a non-empty, prefix-closed, subset $\sigma \subseteq P_A$, which is moreover $\lightning$-receptive in that

$$\forall s \in \sigma . s \cdot \boldsymbol{\lightning} \in P_A \implies s \cdot \boldsymbol{\lightning} \in \sigma$$

The $\lightning$-receptivity property of strategies models the usual assumption that crashes may non-deterministically happen at any point in an execution of a program. Surprisingly, it plays a crucial role in proving the symmetric monoidal structure of crash-aware games.

Observe that in the definition of the tensor product and the affine implication for sequential and concurrent games the disjoint union of move sets plays a crucial role. In order to correctly model the instantaneous and synchronous behavior of crashes, we must treat crash signals differently when computing the disjoint union of move sets. For this, we define a smash product which behaves like the disjoint union for O and P moves, but that merges crash signals together. It will also be necessary to redefine projections to take this merger into account.

Definition A.15 (Smash Product). Given move sets (M_A, λ_A) and (M_B, λ_B) we define their smash product $(M_A +_{\lightning} M_B, \lambda_A +_{\lightning} \lambda_B)$ by

$$M_A +_{\lightning} M_B := M_A^{\Upsilon} + M_B^{\Upsilon} + \boldsymbol{\lightning} \qquad \text{and} \qquad \lambda_A +_{\lightning} \lambda_B := \lambda_A + \lambda_B + \lightning$$

where $\lightning$ stands for the constant function to $\lightning \in \mathsf{Pol}^{\lightning}$.

Given $s \in \mathbb{P}_{M_A +_{\lightning} M_B}$ we define $s\upharpoonright_{A,-} \in \mathbb{P}_{M_A}$ and $s\upharpoonright_{-,B} \in \mathbb{P}_{M_B}$ to be the projections to the corresponding components of $M_A +_{\lightning} M_B$. What this means is that $-\upharpoonright_{M_A,-}$ and $-\upharpoonright_{-,M_B}$ are respectively generated by the maps in **Set** below using the universal property of the free monoids M_A^* and M_B^* respectively:

$$-\upharpoonright_{M_A,-} := \mathbf{id}_{M_A^{\Upsilon}} + \epsilon + \mathbf{id}_1 : M_A +_{\lightning} M_B \longrightarrow M_A^* \qquad -\upharpoonright_{-,M_B} := \epsilon + \mathbf{id}_{M_B^{\Upsilon}} + \mathbf{id}_1 : M_A +_{\lightning} M_B \longrightarrow M_B^*$$

where ϵ is the constant function to the empty sequence, and $\mathbf{id}_S$ is the identity for the set S.

Definition A.16 (Dual Move Set). Given a move set (M, λ) we define the moveset $(M^{\perp}, \lambda^{\perp})$ by

$$M^{\perp} := M \qquad \lambda^{\perp}(m) := \lambda(m)^{\perp}$$

where $(\boldsymbol{\alpha}{:}\mathsf{pol})^{\perp} := \boldsymbol{\alpha}{:}\mathsf{pol}^{\perp}$ for $\mathsf{pol} \in \mathsf{Pol}^{\mathsf{seq}}$, and $\lightning^{\perp} := \lightning$.

In the context of games A and B, as opposed to move sets, we write $-\upharpoonright_{A,-}$ and $-\upharpoonright_{-,B}$ for $-\upharpoonright_{M_A,-}$ and $-\upharpoonright_{-,M_B}$ respectively. We will also write $-\upharpoonright_{A,B,-}$ and $-\upharpoonright_{-,B,C}$ for, respectively,

$$-\upharpoonright_{M_A +_{\lightning} M_B,-} : (M_A +_{\lightning} M_B +_{\lightning} M_C)^* \longrightarrow (M_A +_{\lightning} M_B)^* \qquad -\upharpoonright_{-,M_B +_{\lightning} M_C} : (M_A +_{\lightning} M_B +_{\lightning} M_C)^* \longrightarrow (M_B +_{\lightning} M_C)^*$$

We also take the opportunity to define a projection

$$-\upharpoonright_{A,-,C} : (M_A +_{\lightning} M_B +_{\lightning} M_C)^* \longrightarrow (M_A +_{\lightning} M_C)^*$$

as the monoid homomorphism associated by the universal property of the free monoid $(M_A +_{\lightning} M_C)^*$ to the mapping $p_{A,-,C} : M_A +_{\lightning} M_B +_{\lightning} M_C \longrightarrow (M_A +_{\lightning} M_C)^*$ in **Set**:

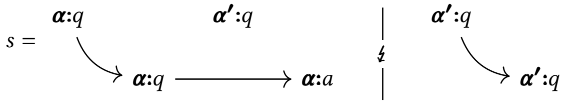
$$p_{A,-,C}(m) = \begin{cases} m, & m \in M_A^{\Upsilon} + M_C^{\Upsilon} \\ \epsilon, & m \in M_B^{\Upsilon} \\ \boldsymbol{\lightning}, & (\lambda_A +_{\lightning} \lambda_B +_{\lightning} \lambda_C)(m) = \lightning \end{cases}$$

Definition A.17. Fix games A and B. We define the games $A \otimes B$ and $A \multimap B$ by the following data

$$M_{A\otimes B} := M_A +_{\lightning} M_B \qquad \lambda_{A\otimes B} := \lambda_A +_{\lightning} \lambda_B \qquad P_{A\otimes B} := \{s \in \mathbb{P}_{M_A +_{\lightning} M_B} \mid s{\upharpoonright}_{A,-} \in P_A \wedge s{\upharpoonright}_{-,B} \in P_B\}$$

$$M_{A\multimap B} := M_A^{\perp} +_{\lightning} M_B \qquad \lambda_{A\multimap B} := \lambda_A^{\perp} +_{\lightning} \lambda_B \qquad P_{A\multimap B} := \{s \in \mathbb{P}_{M_A^{\perp} +_{\lightning} M_B} \mid s{\upharpoonright}_{A,-} \in P_A \wedge s{\upharpoonright}_{-,B} \in P_B\}$$

It is in this change from a disjoint union to the smash product, and in the corresponding modification to projections that lies some of the assumptions about the behavior of crashes. Consider the following play s of $\Sigma^{\lightning} \multimap \Sigma^{\lightning}$ (on the left):



$s =$
$\boldsymbol{\alpha}{:}q$
$\boldsymbol{\alpha}{:}q$
$\boldsymbol{\alpha}{:}a$
$\boldsymbol{\alpha'}{:}q$
$\lightning$
$\boldsymbol{\alpha'}{:}q$
$\boldsymbol{\alpha'}{:}q$

$$s{\upharpoonright}_{-,\Sigma_{\lightning}} = \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha'}{:}q \cdot \boldsymbol{\lightning} \cdot \boldsymbol{\alpha'}{:}q$$

$$s{\upharpoonright}_{\Sigma_{\lightning},-} = \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}a \cdot \boldsymbol{\lightning} \cdot \boldsymbol{\alpha'}{:}q$$

Note that the crash signal synchronize across the the source and target components of the play. This simultaneously models that the crashes are *synchronous* across components (they happen in all components at once) and that they are *instantaneous* (it takes negligible time for the crash to propagate to other components). On the right, above, we see the projections of s to the source and target components. Importantly, the crash event is retained in both projections, so that the crash event $\boldsymbol{\lightning}$ of $\Sigma^{\lightning} \multimap \Sigma^{\lightning}$ effectively belongs to both components.

Definition A.18. We define the set of interaction plays as

$$\mathsf{int}(A, B, C) := \{s \in (M_A +_{\lightning} M_B +_{\lightning} M_C)^* \mid s{\upharpoonright}_{A,B,-} \in P_{A\multimap B} \wedge s{\upharpoonright}_{-,B,C} \in P_{B\multimap C}\}$$

Given strategies $\sigma : A \multimap B$ and $\tau : B \multimap C$ we define their set of interactions by

$$\mathsf{int}(\sigma, \tau) := \{s \in \mathsf{int}(A, B, C) \mid s{\upharpoonright}_{A,B,-} \in \sigma \wedge s{\upharpoonright}_{-,B,C} \in \tau\}$$

and their composition

$$\sigma; \tau := \{s{\upharpoonright}_{A,-,C} \in \mathbb{P}^{\lightning}_{A\multimap C} \mid s \in \mathsf{int}(\sigma, \tau)\}$$

Proposition A.19. *Composition of crash-aware strategies is well-defined and associative.*

Definition A.20. We denote by **Crash** the semicategory of crash-aware games, with crash-aware strategies $\sigma : A \multimap B$ as morphisms between games A and B, and composition given by $-;-$.

We take the opportunity to define the crash-aware version of E as $E^{\lightning}$. We similarly define the game $\dagger E^{\lightning}$ as $(\dagger E)^{\lightning}$.

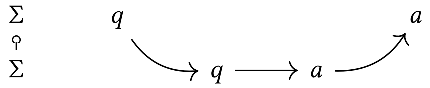
A.3 The Copycat Strategies and Saturation

None of **Seq**, **Conc** or **Crash** assemble into categories for the corresponding composition operations $-;-$ do not have a neutral element. That is, to say, there is no choice of strategies $\mathbf{id}_A : A \multimap A$ for which $\mathbf{id}_A; \sigma; \mathbf{id}_B = \sigma$ for every $\sigma : A \multimap B$. On the other hand, there are clear candidates for such a neutral element, which are called the copycat strategies.

The sequential copycat strategy $\mathsf{seqcopy}_A : A \multimap A$ intuitively replicates O-moves in the target component as the same move in the source component, and P-moves in the source component to the same move in the target component. Formally, it is defined by:

$$\mathsf{seqcopy}_A := \{s \in P_{A\multimap A} \mid \forall p \sqsubseteq_{\mathsf{even}} s. p{\upharpoonright}_{A_1} = p{\upharpoonright}_{A_2}\}$$

where we write $\sqsubseteq$ for the prefix relation and $\sqsubseteq_{\mathsf{even}}$ for the even-length prefix relation. For Σ the maximal play of the copycat strategy $\mathsf{seqcopy}_{\Sigma}$ is the play below (on the left):



```
Import Σ

q () {
  a <- q
  ret a
}
```

Fig. 9. Maximal play of seqcopy_Σ (left) and corresponding pseudocode (right)

This play corresponds to complete execution of the code we display on the right, which implements Σ by importing another instance of Σ.

The concurrent copycat then $\text{ccopy}_A : A \multimap A$ merely has every agent $\alpha \in \Upsilon$ play the sequential copycat for its corresponding game $\text{seqcopy}_{A^\alpha} : A^\alpha \multimap A^\alpha$:

$$\text{ccopy}_A := \{s \in P_{A \multimap A} \mid \pi_\alpha(s) \in \text{seqcopy}_{A^\alpha}\}$$

Finally, the crash-aware copycat $\text{crashcopy}_A : A \multimap A$ just plays $\text{ccopy}_{A^\Upsilon}$ within each epoch:

$$\text{crashcopy}_A := (\text{ccopy}_{A^\Upsilon} \cdot ↯)^* \cdot \text{ccopy}_{A^\Upsilon}$$

It turns out that in all cases the corresponding copycat strategy can lead to emergent behavior after composition, which prevents the models from being compositional. Concretely, writing copy generically for any of the copycat strategies, it can happen that for some $\sigma : A \multimap B$, $\sigma \subset \text{copy}_A; \sigma; \text{copy}_B$ strictly. This issue is explained extensively in Oliveira Vale et al. [31] in the context of concurrent games, and the situation is the same for crash-aware games, so we refer the reader there for a more detailed account of the issue.

The solution they present which turns out to be deeply related to linearizability, is to note that the copycat strategy is *idempotent*, in that for all games A in the corresponding models

$$\text{copy}_A; \text{copy}_A = \text{copy}_A$$

This essentially means that the copy_A at least behaves like a neutral element for itself. With that fact at hand, we define a class of strategies that behave well when composed with the copycat.

Definition A.21. We say a strategy $\sigma : A \multimap B$ is saturated with respect to the copycat strategy copy when

$$\text{copy}_A; \sigma; \text{copy}_B = \sigma$$

It is not hard to see that the fact that the copycat strategy is idempotent ensures that composing saturated strategies yields a saturated strategy, and that the copycat does behave like a neutral element for saturated strategies. This means that we can promote the semicategories we have defined to categories by restricting attention to these saturated strategies.

Definition A.22. We define the categories **Seq**, **Conc** and **Crash** as the restrictions, respectively, of the semicategories $\underline{\textbf{Seq}}$, $\underline{\textbf{Conc}}$ and $\underline{\textbf{Crash}}$ to strategies saturated with respect to the corresponding copycat strategies seqcopy, ccopy and crashcopy.

It is folklore that, concretely, saturation for sequential strategies is equivalent to O-receptivity. That is, a strategy is saturated if and only if it accepts O-moves whenever the environment is allowed to make them. For concurrent games the story for saturation is more complicated, and corresponds to, beyond O-receptivity, strategies that are insensitive to certain delays, which might be caused, for instance, if an agent is preempted. This is typically formalized using a rewrite system $- \rightsquigarrow -$, which figures prominently in the concrete formulation of linearizability appearing in Oliveira Vale et al. [31], so naturally it will also play a key role in our own work on linearizability, and we take the opportunity to define it now.

Definition A.23. Let $A = (M_A, P_A)$ be a concurrent game. We define a string rewrite system $(P_A, \rightsquigarrow_A)$ with local rewrite rules:

- $\forall m, m' \in M_A.\forall \alpha, \alpha' \in \Upsilon.\forall X \in \{O, P\}.\alpha \neq \alpha' \wedge \lambda_A(m) = \boldsymbol{\alpha}{:}X \wedge \lambda_A(m') = \boldsymbol{\alpha'}{:}X \implies m{\cdot}m' \rightsquigarrow_A m' \cdot m$
- $\forall o, p \in M_A.\forall \alpha, \alpha' \in \Upsilon.\alpha \neq \alpha' \wedge \lambda_A(o) = \boldsymbol{\alpha}{:}O \wedge \lambda_A(p) = \boldsymbol{\alpha'}{:}P \implies o \cdot p \rightsquigarrow_A p \cdot o$

For crash-aware strategies, the concrete characterization is only slightly more involved. We do not cover it here for the sake of space. We will soon see an equivalent characterization in terms of a linearizability criterion, which will be sufficient for our purposes.

A.4 Refinement and Horizontal Composition

Before proceeding, we briefly address refinement and horizontal composition. We take as our notion of refinement behavior containment, $\sigma \subseteq \tau$, with joins given by set union. This makes all of the models we have discussed so far into enriched (semi)categories over join semi-lattices. Specifically, this means that

Proposition A.24. *Strategy composition $-;-$ is monotonic and join-preserving.*

Now, for horizontal composition, recall that we have already defined a game $A \otimes B \in \underline{\mathbf{Crash}}$ given games A and B in $\underline{\mathbf{Crash}}$. The tensor defines a semifunctor as follows, where $\sigma : A \multimap B$ and $\tau : A' \multimap B'$:

$$\sigma \otimes \tau := \{s \in P_{A \otimes A' \multimap B \otimes B'} \mid s{\upharpoonright_{A \multimap B}} \in \sigma \wedge s{\upharpoonright_{A' \multimap B'}} \in \tau\}$$

Note that these projections are defined just like in the usual concurrent case except for its behaviour on crashes $\lightning$ which is given by:

$$\lightning{\upharpoonright_{A \multimap B}} = \lightning \qquad\qquad \lightning{\upharpoonright_{A' \multimap B'}} = \lightning$$

Moreover, the game $\mathbf{1}$ is given by the following data

$$M_{\mathbf{1}} = \{\lightning\} \qquad\qquad \lambda_{\mathbf{1}}(\lightning) = \lightning \qquad\qquad P_{\mathbf{1}} = \lightning^*$$

These definitions permit us to prove that

Proposition A.25. $(\mathbf{Crash}, - \otimes -, \mathbf{1})$ *defines an enriched symmetric monoidal category.*

This means, in particular, that $- \otimes -$ defines a monotonic and join-preserving functor, so that horizontal composition behaves well with respect to both vertical composition and refinement.

B Crash-Aware Linearizability

Compositional linearizability provides an account of linearizability based on an operation K_{Conc} $-$ converting strategies in $\underline{\mathbf{Conc}}$ to strategies in $\mathbf{Conc}$. This operation, comes with the abstract construction defining $\mathbf{Conc}$, so that, as $\mathbf{Crash}$ follows the same construction, there is a counterpart $K_{\lightning}$ $-$ in $\mathbf{Crash}$. In this section, we give a concrete characterization of the notion of linearizability associated to $K_{\lightning}$ $-$, which is closely related to strict linearizability. Then, we present the equivalence with observational refinement and the locality property for it.

B.1 Crash-Aware Linearizability

We start by defining the operation $K_{\lightning}$ $- : \underline{\mathbf{Crash}} \to \mathbf{Crash}$ by the formula, for $\tau : A \multimap B \in \underline{\mathbf{Crash}}$

$$K_{\lightning}\ \tau := \mathsf{crashcopy}_A ; \sigma ; \mathsf{crashcopy}_B$$

Intuitively, this operation assigns to σ the smallest saturated strategy containing σ. $K_{\lightning}$ $-$ has the important property that it is an oplax semifunctor, i.e.

Proposition B.1. *For all $\sigma : A \multimap B$ and $\tau : B \multimap C$ in $\underline{\textbf{Crash}}$,* $\quad K_{\lightning}\ (\sigma;\tau) \subseteq K_{\lightning}\ \sigma; K_{\lightning}\ \tau.$

The framework of compositional linearizability proposes that the native notion of linearizability for crash-aware objects should be equivalent to the refinement $v' \subseteq K_{\lightning}\ v$. So for the remainder of this section, our goal is to concretely characterize this refinement in terms of plays. For this, we find it useful to recall the concrete formulation of compositional linearizability. This notion of linearizability is a slight generalization of interval-sequential linearizability, and in particular does not require that the linearized specification be atomic nor that all pending operations be removed.

Definition B.2. For $A \in \textbf{Conc}$, a play $s \in P_A$ is linearizable to a play $t \in P_A$ when there exists a sequence of P-moves $s_P \in (M_A^P)^*$ and a sequence of O-moves $s_O \in (M_A^O)^*$ such that $s \cdot s_P \rightsquigarrow\!\!\!\rightarrow_A t \cdot s_O$. We write $s \rightsquigarrow t$ when s is linearizable to t. We say $s \in P_A$ is linearizable with respect to a strategy $v : A$, written $s \rightsquigarrow v$ when there exists $t \in v$ such that $s \rightsquigarrow t$. We say a strategy $v' : A$ is linearizable with respect to a strategy $v : A$, written $v' \rightsquigarrow v$, when for every play $s \in v'$, $s \rightsquigarrow v$.

We use linearizability in **Conc** to define crash-aware linearizability. It is straight-forward: s crash-aware linearizes to t when their corresponding epochs linearize to each other.

Definition B.3. For $A \in \underline{\textbf{Crash}}$, we say a play $s \in P_A$ crash-aware linearizes to a play $t \in P_A$ when

$$\|s\| = \|t\| \qquad\qquad \text{and} \qquad\qquad \forall i \leq \|s\|.\mathsf{epo}_i(s) \rightsquigarrow \mathsf{epo}_i(t)$$

and write $s \overset{\lightning}{\rightsquigarrow} t$ when this holds, and extend the notation as we did for linearizability (see Def. B.2).

We now discuss a few examples of crash-aware linearizability. A first example is volatile objects. For this, we find it useful to define a functor $\mathsf{Vol}\ -$, defined by $\mathsf{Vol}\ A := A^{\lightning}$ on games. Meanwhile, given a strategy $\sigma : A \multimap B \in \underline{\textbf{Conc}}$ we define the strategy $\mathsf{vol}(\sigma) \in \underline{\textbf{Crash}}$ as $\mathsf{vol}(\sigma) := (\sigma \cdot \lightning)^* \cdot \sigma$. The functor is named $\mathsf{vol}(-)$ because we use it to define volatile objects, as given a strategy $v_E : \dagger E \in \textbf{Conc}$, $\mathsf{vol}(v_E) : \dagger E^{\lightning}$ describes the object that behaves as v_E within each epoch, and in particular resets the object to its initial state on a crash. It is easy to see that:

Proposition B.4. *For $v' : A \in \textbf{Conc}$ and $v : A \in \underline{\textbf{Conc}}$, if $v' \rightsquigarrow v$ then* $\mathsf{vol}(v') \overset{\lightning}{\rightsquigarrow} \mathsf{vol}(v)$.

The main result of this section is the following characterization of $K_{\lightning}\ -$.

Proposition B.5.

$$K_{\lightning}\ \tau = \{s \in P_{A \multimap B} \mid s \textit{ is crash-aware linearizable with respect to } \tau\}$$

Corollary B.6. *$v' : A$ is crash-aware linearizable with respect to $v : A$ if and only if $v' \subseteq K_{\lightning}\ v$.*

B.2 Observational Refinement and Locality

We use the general result in Oliveira Vale et al. [31] to obtain locality and observational refinement. The requirements to obtain these properties are the following.

Lemma B.7.

- *For any $\sigma : \mathbf{1} \multimap A \in \underline{\textbf{Crash}}$ it holds that* $\qquad \mathsf{crashcopy}_1; \sigma = \sigma.$
- *For $\sigma, \tau : A \multimap B$ and $\sigma', \tau' : A' \multimap B'$ we have* $\qquad \sigma \otimes \sigma' \subseteq \tau \otimes \tau' \implies \sigma \subseteq \sigma' \wedge \tau \subseteq \tau'$

This gives as corollaries locality and observational refinement, which we write explicitly now. The proof of these results, under the conditions of our construction and Lemma B.7.

Corollary B.8 (Observational Refinement). *$v'_A : A \in \textbf{Crash}$ is crash-aware linearizable w.r.t $v_A : A \in \underline{\textbf{Crash}}$ if and only if for all $\sigma : A \multimap B$,* $\qquad v'_A; \sigma \subseteq v_A; \sigma$

Corollary B.9 (Locality). *For $v'_A : A$, $v'_B : B \in \textbf{Crash}$ and $v_A : A$, $v_B : B \in \underline{\textbf{Crash}}$:*

$$v'_A \overset{\lightning}{\rightsquigarrow} v_A \textit{ and } v'_B \overset{\lightning}{\rightsquigarrow} v_B \textit{ if and only if } v'_A \otimes v'_B \overset{\lightning}{\rightsquigarrow} v_A \otimes v_B$$

C Crash Abstraction

Many specification methodologies for crash-aware objects, including durable linearizability, use specifications without crashes. In our framework, this means that while the concrete crash-aware object ν' lives in Vol, the abstract specification ν lives in Conc. In this section we develop conversions between $\underline{\mathbf{Crash}}$ and $\underline{\mathbf{Conc}}$ that serve as a building block for strict and durable linearizability.

The main difficulty in removing crashes from a play s with crashes is that the removal may generate traces not satisfying sequential consistency. This happens when the same agent has a pending invocation in one epoch and also moves in a later epoch. So, in the definition of the operation $-^\flat$ (read *de-crash*), the projections $\pi_\Upsilon(s)$ are required to be well-formed plays.

Definition C.1. Given a game $A = (M_A, \lambda_A, P_A) \in \underline{\mathbf{Crash}}$ we define the game $A^\flat \in \underline{\mathbf{Conc}}$, by:

$$M_{A^\flat} := M_A^\Upsilon \qquad\qquad \lambda_{A^\flat}(m) := \lambda_A(m) \qquad\qquad P_{A^\flat} := \{\pi_\Upsilon(s) \in \mathbb{P}^{\mathsf{conc}}_{A^\flat} \mid s \in \|_{\alpha\in\Upsilon}\ (P_A^\alpha)^*\}$$

Given a strategy $\sigma : A \in \underline{\mathbf{Crash}}$ we similarly define $\sigma^\flat : A^\flat \in \underline{\mathbf{Conc}}$ by

$$\sigma^\flat := \{\pi_\Upsilon(s) \in \mathbb{P}^{\mathsf{conc}}_{A^\flat} \mid s \in \sigma\}$$

A result we note in passing is that $(A \multimap B)^\flat \cong A^\flat \multimap B^\flat$. This permits us to show that $-^\flat$ defines an oplax semifunctor $-^\flat : \underline{\mathbf{Crash}} \to \underline{\mathbf{Conc}}$, that is:

Proposition C.2. *For all $\sigma : A \multimap B$ and $\tau : B \multimap C$ in $\underline{\mathbf{Crash}}$,* $\qquad (\sigma;\tau)^\flat \subseteq \sigma^\flat;\tau^\flat.$
In addition, for all $A \in \underline{\mathbf{Crash}}$, $\mathsf{crashcopy}_A^\flat = \mathsf{ccopy}_A$, and $-^\flat$ is monotonic and join-preserving.

It is also useful to provide a reverse operation $-^\sharp$, read *re-crash*, that lifts, in a persistent way, a strategy $\sigma : A^\flat \multimap B^\flat$ into a strategy $\sigma^\sharp : A \multimap B$, a situation we depict diagrammatically below.

$$\begin{tikzcd}
A \arrow[r, dashed, "\sigma^\sharp", -o] \arrow[d, mapsto] & B \arrow[d, mapsto] \\
A^\flat \arrow[r, "\sigma"', -o] & B^\flat
\end{tikzcd}$$

So, suppose given $A, B \in \underline{\mathbf{Crash}}$ and $\sigma : A^\flat \multimap B^\flat \in \underline{\mathbf{Conc}}$. Then, the strategy $\sigma^\sharp : A \multimap B \in \underline{\mathbf{Crash}}$ is given by:

$$\sigma^\sharp := \{s \in \mathbb{P}^{\lightning}_A \mid \pi_\Upsilon(s) \in \sigma\}$$

Re-crash also behaves *like* an enriched semifunctor.

Proposition C.3. *For $\sigma : A^\flat \multimap B^\flat, \tau : B^\flat \multimap C^\flat \in \underline{\mathbf{Conc}}$,* $\qquad \sigma^\sharp;\tau^\sharp \subseteq (\sigma;\tau)^\sharp.$
In addition, for all $A \in \underline{\mathbf{Crash}}$, $\mathsf{ccopy}^\sharp_{A^\flat} \subseteq \mathsf{crashcopy}_A$, and $-^\sharp$ is monotonic and join-preserving.

D Strict Linearizability

Strict linearizability [2], an important linearizability criterion in the presence of crashes often used to specify objects with robust recovery routines, postulates that a pending operation must linearize within the same epoch it was issued. It was originally formulated in a system with individual crashes instead of full-system crashes, so we modify it for our setting. We do not assume atomicity or that all pending operations are removed.

Similarly to how Oliveira Vale et al. [31] characterizes linearizability by lifting a non-saturated strategy to a saturated strategy, we formalize strict linearizability by lifting a strategy without crashes into a strategy with crashes. Naively, one might think it is enough to use the lift $\nu^\sharp : A \in \underline{\mathbf{Crash}}$. Unfortunately, this does not make a crash-aware object, mainly because it does not satisfy O-receptivity anymore. Specifically, $\nu^\sharp$ never has plays such as

$$\boldsymbol{\alpha}{:}q \cdot \lightning \cdot \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}a$$

because $\boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}q \cdot \boldsymbol{\alpha}{:}a$ is not well-formed as a play of $A^\flat$. We can fix this issue by saturating the resulting strategy using $K_{\lightning}\ -$.

Definition D.1. Given games $A, B \in \underline{\mathbf{Crash}}$, we define the strict lift $\mathsf{str}(\sigma) : A \multimap B$ of a strategy $\sigma : A^\flat \multimap B^\flat \in \underline{\mathbf{Conc}}$ as the strategy

$$\mathsf{str}(\sigma) := K_{\lightning}\ \sigma^\sharp$$

We now define our variation of strict linearizability, which differs only in that it generalizes the original strict-linearizability by not requiring the linearized specification to be atomic and complete, and specializes it to full-system crashes.

Definition D.2. We say $\nu' : A \in \mathbf{Crash}$ is strictly linearizable to $\nu : A^\flat \in \underline{\mathbf{Conc}}$ when $\nu' \subseteq \mathsf{str}(\nu)$.

It is not hard to see that when ν is an atomic strategy this is equivalent to strict linearizability. Indeed, the application of $K_{\lightning}\ -$ within $\mathsf{str}(-)$ corresponds to constructing a strict completion [3], and $-^\sharp$ to removing the crashes. Finally, by first noting that

Proposition D.3. *For all* $\sigma : A^\flat \multimap B^\flat, \tau : B^\flat \multimap C^\flat \in \mathrm{Conc}, \qquad \mathsf{str}(\sigma); \mathsf{str}(\tau) \subseteq \mathsf{str}(\sigma; \tau).$

we are able to prove the following refinemenet property for strict linearizability.

Proposition D.4. *Suppose* $\nu'_A : A$ *is strictly linearizable to* $\nu_A : A^\flat$ *and that* $\sigma : A^\flat \multimap B^\flat$ *implements an object linearizable to* $\nu_B : B^\flat$ *using* ν_A*, i.e.*

$$\nu_A; \sigma \subseteq \nu_B$$

Then, $\mathsf{str}(\sigma)$ *implements an object strictly linearizable to* $\mathsf{str}(\nu_B)$ *using* ν'_A*, i.e.*

$$\nu'_A; \mathsf{str}(\sigma) \subseteq \mathsf{str}(\nu_B)$$

The reverse direction, unfortunately does not hold as $\mathsf{str}(\mathsf{ccopy}_{A^\flat}) \neq \mathsf{crashcopy}_A$. By similar reasoning as the locality for crash-aware linearizability we also obtain that

Proposition D.5 (Locality). *For* $\nu'_A : A, \nu'_B : B \in \mathbf{Crash}$ *and* $\nu_A : A, \nu_B : B \in \underline{\mathbf{Conc}}$*:*

$$\nu'_A \subseteq \mathsf{str}(\nu_A) \text{ and } \nu'_B \subseteq \mathsf{str}(\nu_B) \text{ if and only if } \nu'_A \otimes \nu'_B \subseteq \mathsf{str}(\nu_A \otimes \nu_B)$$

E Durable Linearizability

A frequently used linearizability criterion for specifying persistent objects is durable linearizability [22], which appears as an important development in linearizability with crashes [3]. Durable linearizability makes a core assumption, which we call the *durability assumption*: that the agents appearing in each epoch are disjoint across epochs. This means that there is no agent re-use between epochs. The assumption makes the definition of durable linearizability significantly simpler than previous criteria, including strict linearizability. Moreover, under the durability assumption, persistent atomicity and recoverable linearizability are equivalent. This assumption, however, is quite intrusive and must be enforced throughout, requiring us to define a new model Dur. Interestingly, this can be smoothly done by a different choice of copycat for $\underline{\mathbf{Crash}}$. We finish by showing locality and observational refinement for durable linearizability.

E.1 Durable Linearizability

We start by formalizing the durability assumption, and from now on assume Υ is countably infinite.

Definition E.1. For a move set $(M_A, \lambda_A : M_A \to \mathsf{Pol}^{\lightning})$ we say a play $s \in \mathbb{P}_A$ is *durable* when the set of agents appearing in each epoch of s are pairwise disjoint across epochs, i.e.

$$\forall i. \forall j. i \neq j \implies \Upsilon(\mathsf{epo}_i(s)) \cap \Upsilon(\mathsf{epo}_j(s)) = \varnothing$$

and write $\mathbb{P}^{\mathsf{dur}}_A$ for the set of all durable plays over M_A. We write P^{dur}_A for $P_A \cap \mathbb{P}^{\mathsf{dur}}_A$.

We say a strategy is *durable* when all of its plays are durable.

Durable plays s have the important property that $\pi_\Upsilon(s) \in P_A$ as

$$\pi_\Upsilon(s) = \mathsf{epo}_1(s) \cdot \mathsf{epo}_2(s) \cdot \ldots \cdot \mathsf{epo}_n(s) \cdot \ldots$$

writing $\mathsf{ops}(s)$ for the right-hand side above we obtain that when σ is durable $\sigma^\flat = \mathsf{ops}(\sigma)$. An important result is that durable strategies compose:

Proposition E.2. *If $\sigma : A \multimap B$ and $\tau : B \multimap C$ are durable strategies then $\sigma; \tau$ is a durable strategy.*

This means that the restriction of $\underline{\mathbf{Crash}}$ to durable strategies defines a semicategory, which we call $\underline{\mathbf{Dur}}$. This motivates defining a durable copycat strategy.

Definition E.3. For $A \in \underline{\mathbf{Crash}}$, the strategy $\mathsf{durcopy}_A : A \multimap A$ is the strategy:

$$\mathsf{durcopy}_A := \mathsf{crashcopy}_A \cap \mathbb{P}^{\mathsf{dur}}_{A \multimap A}$$

Proposition E.4. $\mathsf{durcopy}$ *is idempotent.*

This let's us define a model for durable objects and their implementations.

Definition E.5. We define the category **Dur** as the restriction of $\underline{\mathbf{Crash}}$ to strategies saturated with respect to durcopy.

The construction, like for $\underline{\mathbf{Conc}}$ and $\underline{\mathbf{Crash}}$, comes with its own operation $K_{\mathsf{dur}} : \underline{\mathbf{Dur}} \to \mathbf{Dur}$ defined as $K_{\mathsf{dur}}\ \sigma := \mathsf{durcopy}_A; \sigma; \mathsf{durcopy}_B$, as expected. Its associated notion of linearizability $\nu' \subseteq K_{\mathsf{dur}}\ \nu$ is the same as crash-aware linearizability restricted to durable strategies. An important fact that we mention in passing is that restriction to durable plays $- \cap \mathbb{P}_{A \multimap B}$ defines a semifunctor from $\underline{\mathbf{Crash}}$ to $\underline{\mathbf{Dur}}$. Composing this semifunctor with $-^\sharp$ therefore preserves its functoriality properties. For simplicity, we denote this composition simply as $-^\sharp$, as the context should make it clear when the durable assumption is in place. We are finally ready to define durable linearizability.

Definition E.6. We say a play $s \in P^{\mathsf{dur}}_A$ is durably linearizable to a play $t \in P_{A^\flat}$, written $s \overset{\mathsf{dur}}{\rightsquigarrow} t$, when $\mathsf{ops}(s) \rightsquigarrow t$. We extend the notation to strategies as we did for linearizability (see Def. B.2).

Our definition differs from the traditional one only in that: we do not assume the linearized trace is atomic, and we do require that all pending invocations be removed in the linearization.

Now, for our refinement-based formulation, we define a durable lift $\mathsf{dur}(-)$, which assigns to a strategy $\nu : A^\flat \in \underline{\mathbf{Conc}}$ the strategy $\mathsf{dur}(\nu) : A \in \mathbf{Dur}$ defined by

$$\mathsf{dur}(\nu) : A \in \underline{\mathbf{Crash}} := (K_{\mathsf{Conc}}\ \nu)^\sharp \cap P^{\mathsf{dur}}_A$$

And, indeed, $\mathsf{dur}(-)$ does provide an appropriate lifting operation for durable linearizability.

Proposition E.7. *$\nu' : A \in \underline{\mathbf{Crash}}$ is durable linearizable to $\nu : A^\flat \in \underline{\mathbf{Conc}}$ if and only if $\nu' \subseteq \mathsf{dur}(\nu)$.*

E.2 Observational Refinement and Locality

E.2.1 Observational Refinement. We now show an observational refinement property for durable linearizability. For this, we first note that $\mathsf{dur}(-)$ behaves *like* a lax semifunctor, that is:

Proposition E.8. *For all $\sigma : A^\flat \multimap B^\flat, \tau : B^\flat \multimap C^\flat \in$ Conc,* $\qquad \mathsf{dur}(\sigma); \mathsf{dur}(\tau) \subseteq \mathsf{dur}(\sigma; \tau)$.

It also satisfies the following property, which $\mathsf{str}(-)$ does not satisfy:

Proposition E.9. *For all $A \in$* **Dur**, $\qquad \mathsf{dur}(\mathsf{ccopy}_{A^\flat}) = \mathsf{durcopy}_A$

Each of these two results play an important role in each of the two directions of the following equivalence with observational refinement.

Proposition E.10. *Let $A, B \in \underline{\mathbf{Crash}}$. Then $v'_A : A$ is durably linearizable to $v_A : A^\flat$ if and only if whenever $\sigma : A^\flat \multimap B^\flat \in \mathbf{Conc}$ implements a concurrent object linearizable to v_B using v_A, then* $\mathsf{dur}(\sigma) : A \multimap B$ *implements an object durably linearizable to v_B using v'_A.*

E.2.2 Locality. For locality, we start by defining a tensor product.

Definition E.11. For strategies $\sigma : A, \tau : B \in \underline{\mathbf{Dur}}$ we define their tensor product $\sigma \boxtimes \tau : A \boxtimes B$ as

$$A \boxtimes B := A \otimes B \qquad\qquad \sigma \boxtimes \tau := (\sigma \otimes \tau) \cap P^{\mathsf{dur}}_{A \otimes B}$$

Proposition E.12. $(\mathbf{Dur}, - \boxtimes -, \mathbf{1})$ *defines a symmetric monoidal category.*

With that established, establishing locality follows the same structure as for **Conc** and **Crash**.

Proposition E.13. *For any durable $\sigma : A, \tau : B \in \underline{\mathbf{Conc}}$,* $\qquad \mathsf{dur}(\sigma \otimes \tau) = \mathsf{dur}(\sigma) \boxtimes \mathsf{dur}(\tau)$.

Proposition E.14. *For $\sigma, \tau : A \multimap B \in \mathbf{Dur}$ and $\sigma', \tau' : A' \multimap B' \in \mathbf{Dur}$ we have*

$$\sigma \boxtimes \sigma' \subseteq \tau \boxtimes \tau' \implies \sigma \subseteq \sigma' \wedge \tau \subseteq \tau'$$

Corollary E.15 (Locality). *For $v'_A : A, v'_B : B \in \mathbf{Dur}$ and $v_A : A, v_B : B \in \underline{\mathbf{Conc}}$:*

$$v'_A \overset{\mathsf{dur}}{\rightsquigarrow} v_A \text{ and } v'_B \overset{\mathsf{dur}}{\rightsquigarrow} v_B \text{ if and only if } v'_A \boxtimes v'_B \overset{\mathsf{dur}}{\rightsquigarrow} v_A \otimes v_B$$

E.3 FLiT Correctness Theorem

For the FLiT correctness theorem, we must assume that v'_{Cell} is strongly linearizable to v_{FLiT}, in the sense of [31]. This means that we assume that $v_{\mathsf{FLiT}} \subseteq v'_{\mathsf{Cell}}$ in addition to $v'_{\mathsf{Cell}} \rightsquigarrow v_{\mathsf{FLiT}}$. Note as well that the FLiT correctness theorem also assumes that a durably linearizable FLiT implementation is available, that is that an object $v'_{\mathsf{FLiT}} : \dagger\mathsf{FLiT}^{\lightning}$ durably linearizable to v_{FLiT} has been established. We do this for the implementation displayed in §1 in §I.

Proposition E.16 (FLiT Correctness). *For any object signature E, writing $v'_{\mathsf{Mem}} := \otimes_{i \in I} v'_{\mathsf{Cell}}$, if $v'_{\mathsf{Mem}}; M$ is an object linearizable to v_E then, writing $v'_{\mathsf{FLiTMem}} := \boxtimes_{i \in I} v'_{\mathsf{FLiT}}$, it follows that $v'_{\mathsf{FLiTMem}}; \mathsf{dur}(M)$ is durably linearizable to v_E.*

Proof. Note that since v'_{Cell} is linearizable to v_{FLiT}, from locality for compositional linearizability it follows that $v'_{\mathsf{Mem}} \rightsquigarrow v_{\mathsf{Mem}}$, where we write $v_{\mathsf{Mem}} := \otimes_{i \in I} v_{\mathsf{FLiT}}$. Then, by observational refinement for compositional linearizability and the assumption, we have:

$$v_{\mathsf{Mem}}; M = v'_{\mathsf{Mem}}; M \subseteq v_E$$

Now, by locality for durable linearizability we have that $v'_{\mathsf{FLiTMem}} \subseteq \mathsf{dur}(v_{\mathsf{Mem}})$. But then, the result follows from observational refinement for durable linearizability. □

F Imperative Programs

So far we have developed the theory of crash-aware, strict and durable linearizability in a rather general setting. In practice, it proves useful to focus attention to strategies that specifically represent imperative code. Specifically, these are strategies that arise from parallel compositions of sequential imperative strategies. Object specifications $v_E : \dagger E^{\lightning}$ are now assumed to have effect signatures as types, but otherwise are just any strategy in the appropriate domain. Strategies $M : \dagger E^{\lightning} \multimap \dagger F^{\lightning}$ that are used to implement new objects $v_E; M$ will be specialized to these imperative strategies. For this, we use the theory of object-based semantics proposed by Reddy [36] and further developed in Oliveira Vale et al. [30, 31]. For brevity, we do this with respect to $\underline{\mathbf{Crash}}$ and **Crash**, but the corresponding variation for durable strategies is easily obtained by enforcing durability throughout.

F.1 Parallel Strategies

Recall that in §A.1 we defined an operation Conc − for constructing concurrent games from α-indexed collections of sequential games. This operation has a suitable counterpart for strategies, which makes it into a functor.

Definition F.1. Given a collection of strategies $\sigma[\Upsilon] = (\sigma[\alpha])_{\alpha\in\Upsilon}$, where for each α, $\sigma[\alpha] : A[\alpha] \multimap B[\alpha] \in \mathbf{Seq}$, define the strategy Conc $\sigma[\Upsilon]$: Conc $A \multimap$ Conc B as

$$\sigma[\Upsilon] := \|_{\alpha\in\Upsilon}\sigma[\alpha]$$

We say a strategy in the image of Conc − is a *parallel strategy.*

Now, we define parallel strategies in $\underline{\mathbf{Crash}}$.

Definition F.2. We denote by Par − the composition of semifunctors:

$$\mathbf{Par}\ - : \underline{\mathbf{Seq}}^{\Upsilon} \to \underline{\mathbf{Crash}} := \mathsf{vol}(\mathrm{Conc}\ -)$$

We say a strategy in the image of Par is a crash-aware parallel strategy.

Since both Conc − and Vol − restrict to functors, we define the subsemicategory $\underline{\mathbf{Parallel}}$ of $\underline{\mathbf{Crash}}$ of parallel strategies, which, when restricted to saturated strategies, forms a subcategory **Parallel** of **Crash**.

We note that

Proposition F.3.

$$\mathsf{Par}\ \sigma[\Upsilon] \otimes \mathsf{Par}\ \sigma'[\Upsilon] = \mathsf{Par}\ (\sigma \otimes \sigma')[\Upsilon]$$

where $(\sigma \otimes \sigma')[\alpha] = \sigma[\alpha] \otimes \sigma'[\alpha]$.

Since, moreover, all the structural morphisms on **Crash** are parallel strategies, it follows that **Parallel** inherits the symmetric monoidal structure of **Crash**.

F.2 The Crash-Aware Replay Modality

Parallel strategies make for a nice domain for us to fully define the structure of the replay modality † −, which we have being using for our examples.

Definition F.4. Given a sequential game $A = (M_A, \lambda_A, P_A)$ we define the sequential game $\dagger A$ by the following data:

$$M_{\dagger A} := \sum_{i\in\mathbb{N}} M_A \qquad \lambda_{\dagger A} = \sum_{i\in\mathbb{N}} \lambda_A \qquad P_{\dagger A} := \{s_1 \cdot \ldots \cdot s_n \in \mathbb{P}^{\mathsf{seq}}_{\dagger A} \mid \forall i.s_i \in \boldsymbol{i{:}}P_A\}$$

Given a sequential strategy $\sigma : A \multimap B$ we define $\dagger\sigma : \dagger A \multimap \dagger B$ by

$$\dagger\sigma := \{\mathbf{1{:}}s_1 \cdot \ldots \cdot \boldsymbol{n{:}}s_n \in P_{\dagger A} \mid \forall i.s_i \in \sigma\}$$

Then, given a crash-aware game $A = (M_A, \lambda_A, P_A) \in \underline{\mathbf{Crash}}$ we define the game $\dagger A$ by the following data.

$$M_{\dagger A} := \left(\sum_{i\in\mathbb{N}} M_A^{\Upsilon}\right) + \lightning \qquad \lambda_{\dagger A} := \sum_{i\in\mathbb{N}} M_A \qquad P_{\dagger A} := ((\|_{\alpha\in\Upsilon}P_{\dagger A^{\alpha}}) \cdot \lightning) \cdot (\|_{\alpha\in\Upsilon}P_{\dagger A^{\alpha}})$$

Given a parallel strategy $\sigma = \mathsf{Par}\ \sigma[\Upsilon]$ we define $\dagger\sigma := \mathsf{Par}\ (\dagger\sigma[\alpha])_{\alpha\in\Upsilon}$.

A crucial result to appropriately define an object-based semantics model is that † − does define a modality, that is

Proposition F.5.

$$\dagger - : \underline{\mathbf{Parallel}} \to \underline{\mathbf{Parallel}}$$

defines a semifunctor restrictiting to a comonad

$$\dagger - : \mathbf{Parallel} \to \mathbf{Parallel}$$

Object-based semantics then postulates that co-algebras of $\dagger -$ capture various flavors of the semantics of imperative code. It turns out that in fact, Par $-$ transports co-algebras in

Proposition F.6. *Given a collection of strategies $(M[\alpha] : A[\alpha] \multimap B[\alpha])_{\alpha \in \Upsilon}$ such that, for every $\alpha \in \Upsilon$, $A[\alpha]$ and $B[\alpha]$ are co-algebras of the sequential $\dagger -$, and $M[\alpha]$ is a co-algebra morphism, then* Par $(M[\Upsilon], R)$ *is a co-algebra morphism with respect to the crash-aware $\dagger -$.*

This rather technical result means that we may use the same notion of imperative code used in Oliveira Vale et al. [30, 31] in our framework, and, therefore, we are able to inherit many of their techniques for modeling imperative programming to our setting.

F.3 Imperative Strategies

We are finally ready to define our model of imperative strategies.

Definition F.7. We define the category **Imp** to be the subcategory of **Parallel** defined by the following data:

Objects: games of the form $E^{\lightning} \in \mathbf{Crash}$, where E is a concurrent effect signature.

Morphisms: parallel strategies of the form Par $(\widehat{M}[\Upsilon], \Delta_{\lightning}) : \dagger E^{\lightning} \multimap \dagger F^{\lightning}$, where, for each α, $M[\alpha] : \dagger E[\alpha] \multimap F[\alpha] \in \mathbf{Seq}$, and $\widehat{M}[\alpha]$ is the co-Kleisli extension $\widehat{M[\alpha]} : \dagger E \multimap \dagger F$ of $M[\alpha]$.

This dense definition encapsulates a recent approach for the semantics of systems. The maps $M[\alpha] : \dagger E[\alpha] \multimap F[\alpha]$ are exactly the regular maps of Oliveira Vale et al. [30], which they show are effective at describing sequential imperative code. Our parallel strategies, in each epoch, play as parallel compositions of regular maps, which is the same notion of imperative concurrent code used in Oliveira Vale et al. [31]. Both of these trace back to the foundational work by Reddy [35, 36].

Two crucial results for the semantics of imperative programs are that:

Proposition F.8. *For concurrent effect signatures E and F, the game $E^{\lightning}$ & $F^{\lightning}$ is generated by the concurrent effect signature $(E[\alpha] + F[\alpha])_{\alpha \in \Upsilon}$.*

Proposition F.9 (Imperative Seely Isomorphism). *There is a natural isomorphism in* **Imp***:*

$$\dagger(E^{\lightning} \mathbin{\&} F^{\lightning}) \cong \dagger E^{\lightning} \otimes \dagger F^{\lightning}$$

These two results together mean that **Imp** inherits the symmetric monoidal structure of Vol. Essentially, a tensor product $\dagger E^{\lightning} \otimes \dagger F^{\lightning}$ is essentially the same thing as the game $\dagger(E^{\lightning} \mathbin{\&} F^{\lightning})$, which, since $E^{\lightning}$ & $F^{\lightning}$ is generated by a concurrent effect signature, itself belongs to **Imp**.

G A Program Logic for Durable Overlay Objects

G.1 Programming Language

In this section, we define a general programming language. Compared to the main text, we define in detail the state transformer $[\![B]\!]_\alpha$ and the local operational semantics that we lift the transformer into.

G.1.1 Syntax. We start by defining a language Com for commands over some effect signature $E \in \mathsf{Eff}$, where Eff is the set of effect signatures:

$$\mathsf{Prim} := x \leftarrow e(a) \mid \mathsf{assume}(\phi) \mid \mathsf{ret}\ v \qquad \mathsf{Com} := \mathsf{Prim} \mid \mathsf{Com}; \mathsf{Com} \mid \mathsf{Com} + \mathsf{Com} \mid \mathsf{Com}^* \mid \mathsf{skip}$$

Prim stands for primitive commands. The assignment command, $x \leftarrow e(a)$, executes the effect $e \in E$ with argument a and stores the response to variable x in a local environment $\Delta \in \mathsf{Env}$. The assert command, $\mathsf{assume}(\phi)$, takes a boolean function ϕ over the local environment and terminates the computation if it evaluates to False. We implement while loops and if-statements using $\mathsf{assume}(-)$ in the usual way. The return command, ret v, stores the value v into a reserved variable res, and may only be invoked once in any procedure's execution. Com is the grammar of commands defined as usual in a Kleene algebra.

An implementation $M[\alpha]$ of type $E \to F \circlearrowleft R_F$ implements the overlay's regular procedures F and recovery procedures R_F, using the underlay with the signature E. For simplicity, we require that there is only one recovery program in R_F, i.e. $R_F = \{r : \mathbf{1} \to \mathbf{1}\}$, and use r to denote the overlay's recovery method. The local implementation consists of a collection $M[\alpha] = (M[\alpha]^f)_{f \in F \cup R_F}$ of commands $M[\alpha]^f \in \mathsf{Com}$ indexed by $f \in F \cup R_F$. A concurrent module $M[\Upsilon] \in \mathsf{CMod}$ is given by a collection of local implementations $M[\Upsilon] = (M[\alpha])_{\alpha \in \Upsilon}$.

G.1.2 Semantics. Each primitive command B receives an interpretation as a state transformer $[\![B]\!]_\alpha : \mathsf{UndState} \to \mathcal{P}(\mathsf{UndState})$ over a set of states $\mathsf{UndState} := \mathsf{Env} \times P_{\dagger(E)}$ and returning a new set of states. A state $(\Delta, s) \in \mathsf{UndState}$ contains a local environment $\Delta \in \mathsf{Env}$ and a history represented as a play $s \in P_{\dagger(E)}$. The transformer $[\![B]\!]_\alpha$ depends on α only in that it tags each event it adds to the history with an agent identifier α. We define the transformer $[\![B]\!]_\alpha$ as follows.

- A special primitive is **id**, which is generated when reducing structural commands. Therefore, it has no effect on the state.

$$[\![\mathbf{id}]\!]_\alpha(\Delta, s) = \{(\Delta, s)\}$$

- The return instruction has the interpretation below.

$$[\![\mathsf{ret}\ v]\!]_\alpha(\Delta, s) = \begin{cases} \{(\Delta[\mathsf{res} \mapsto v], s)\} & \Delta(res) = \bot \\ \varnothing & \Delta(res) \neq \bot \end{cases}$$

- For the assignment, we interpret it differently according to the underlay's trace.
  - If $\mathsf{even}(\pi_\alpha(s))$, then

$$[\![x \leftarrow e(a)]\!]_\alpha(\Delta, s) = \begin{cases} \{(\Delta, s \cdot \boldsymbol{\alpha}{:}e(a))\} & a \in \mathsf{par}(e) \\ \{(\Delta, s \cdot \boldsymbol{\alpha}{:}e(\Delta(a)))\} & a \in \mathsf{Var} \wedge \Delta(a) \in \mathsf{par}(e) \\ \varnothing & \text{otherwise} \end{cases}$$

  - If $\pi_\alpha(s) = p \cdot e(a')$, where either $a' = a$ or $a' = \Delta(a)$, then

$$[\![x \leftarrow e(a)]\!]_\alpha(\Delta, s) = \{(\Delta[x \mapsto v], s \cdot \boldsymbol{\alpha}{:}v \mid v \in \mathsf{ar}(e))\}$$

  - Otherwise, $[\![x \leftarrow e(a)]\!]_\alpha(\Delta, s) = \varnothing$.
- The assert instruction only makes progress when the boolean expression evaluates to true.

$$[\![\mathsf{assume}(\phi)]\!]_\alpha(\Delta, s) = \begin{cases} \{(\Delta, s)\} & \phi(\Delta) = \mathsf{True} \\ \phi & \text{otherwise} \end{cases}$$

We lift the interpretation function to a thread local operational semantics $\langle C, \Delta, s\rangle \longrightarrow_\alpha \langle C', \Delta', s'\rangle$. It encodes how α steps on commands in a mostly standard way following the Kleene algebra structure of commands. We define the local operational semantics in Fig. 10.

$$\rightarrowtail \subseteq \mathsf{Com} \times \mathsf{Prim} \times \{O, P\} \times \mathsf{Com}$$

$$\frac{}{B \rightarrowtail^{O}_{B} B} \qquad \frac{}{B \rightarrowtail^{P}_{B} \mathsf{skip}} \qquad \frac{C_1 \rightarrowtail^{X}_{B} C_1'}{C_1;C_2 \rightarrowtail_{B} C_1';C_2} \qquad \frac{}{\mathsf{skip};C \rightarrowtail^{X}_{\mathbf{id}} C} \qquad \frac{}{C^* \rightarrowtail^{X}_{\mathbf{id}} C;C^*}$$

$$\frac{}{C^* \rightarrowtail^{X}_{\mathbf{id}} \mathsf{skip}} \qquad \frac{}{C_1 + C_2 \rightarrowtail^{X}_{\mathbf{id}} C_1} \qquad \frac{}{C_1 + C_2 \rightarrowtail^{X}_{\mathbf{id}} C_2}$$

$$\longrightarrow \subseteq (\mathsf{Com} \times \mathsf{UndState}) \times \Upsilon \times (\mathsf{Com} \times \mathsf{UndState})$$

$$\frac{(\Delta', s') \in [\![B]\!]^{X}_{\alpha}(\Delta, s) \qquad C \rightarrowtail^{X}_{B} C'}{\langle C, \Delta, s\rangle \longrightarrow_{\alpha} \langle C', \Delta', s'\rangle}$$

Fig. 10. Local Operational Semantics ($\longrightarrow$)

In Fig. 5, we lift this local operational semantics to a concurrent module operational semantics $\langle c, \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c', \Delta', s'\rangle$, which takes a continuation $c \in \mathsf{Cont} := \Upsilon \to \{\mathsf{idle}, \mathsf{skip}, \mathsf{dead}, \mathsf{halt}\} + \mathsf{Com}$ and a module state $(\Delta, s) \in \mathsf{ModState} := (\Upsilon \to \mathsf{Env}) \times P_{\dagger(E \circlearrowleft R_E) \multimap \dagger(F \circlearrowleft R_F)}$ containing local environments for all agents and the global trace of the system. It adds three highlighted rules to handle crashes, compared with the semantics in Oliveira Vale et al. [31]. These rules are:

- **Crash** Allows for crashes to happen at any time, resetting local environments for all agents, marking all the previously ran agents as dead and all remaining ones as halt.
- **StartRec** Starts the recovery phase by putting a recovery code C as the continuation, which is a sequential composition of one permutation of underlay recoveries followed by the overlay recovery $M[\alpha]^r$. Chajed et al. [8] uses a similar recovery scheme.
- **EndRed** When the recovery finishes, any thread that is not dead becomes idle, that the system can now run normally. It ensures the durable assumption since threads in previous epochs are no longer available.

We define the denotation of a module to be the set of traces it can generate under the module operational semantics from the initial configuration by the formula below, where c_0 is the initial continuation and Δ_0 is the initial environment where every agent has an empty local environment.

$$[\![M]\!]_{R_E} := \{s \mid \exists c \in \mathsf{Cont}, \Delta \in (\Upsilon \to \mathsf{Env}).\langle c_0, \Delta_0, \epsilon\rangle \twoheadrightarrow^{M}_{R_E} \langle c, \Delta, s\rangle\} \subseteq P_{\dagger(E \circlearrowleft R_E) \multimap \dagger(F \circlearrowleft R_F)}$$

G.2 Object Interfaces

The interface of a crash-aware linearizable object E is a tuple

$$(\nu_E' : \dagger(E \circlearrowleft R_E) \in \mathbf{Dur}, \nu_E : \dagger E^{\lightning} \in \underline{\mathbf{Crash}}) \qquad \text{s.t.} \qquad \nu_E' \upharpoonright_{E^{\lightning}} \subseteq K_{\lightning}\, \nu_E$$

where ν_E' is the concrete specification that contains all possible traces the object can produce, which will include concurrent ones and will also contain crash events and recovery signatures, and ν_E is the linearized specification. The interface is valid if and only if after recovery refining the concrete specification (the projection onto $E^{\lightning}$), $\nu_E' \upharpoonright_{E^{\lightning}}$ is crash-aware linearizable to ν_E, i.e., $\nu_E' \upharpoonright_{E^{\lightning}} \subseteq K_{\lightning}\, \nu_E$.

Similarly, we define the interface of a durable linearizable object E as a tuple

$$\langle \nu_E' : \dagger(E \circlearrowleft R_E) \in \mathsf{Dur}, \nu_E : \dagger E \in \underline{\mathsf{Conc}}\rangle \qquad \text{s.t.} \qquad \nu_E' \upharpoonright_{E^{\lightning}} \subseteq \mathsf{dur}(\nu_E)$$

A major difference from the crash-aware interface is that the durable interface's linearized specification ν_E does not have crashes, because durable objects can be used as if there is no crash. The interface is valid if and only if $\nu_E' \upharpoonright_{E^{\lightning}}$ is durable linearizable to ν_E, i.e., $\nu_E' \upharpoonright_{E^{\lightning}} \subseteq \mathsf{dur}(\nu_E)$.

$$\longrightarrow \subseteq (\mathsf{Com} \times \mathsf{UndState}) \times \Upsilon \times (\mathsf{Com} \times \mathsf{UndState})$$

$$\twoheadrightarrow_{R_E} \subseteq (\mathsf{Cont} \times \mathsf{ModState}) \times \mathsf{CMod} \times (\mathsf{Cont} \times \mathsf{ModState})$$

$$\frac{f \in F \qquad a \in \mathsf{par}(f) \qquad \Delta' = \Delta[\alpha \mapsto [\mathsf{arg} \mapsto a]]}{\langle c[\alpha \mapsto \mathsf{idle}], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c[\alpha \mapsto M[\alpha]^f], \Delta', s \cdot \boldsymbol{\alpha}{:}f\rangle}\ \textsc{Inv}$$

$$\frac{\langle C, \Delta, s\restriction_E\rangle \longrightarrow_\alpha \langle C', \Delta', s'\restriction_E\rangle}{\langle c[\alpha \mapsto C], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c[\alpha \mapsto C'], \Delta', s'\rangle}\ \textsc{Step}$$

$$\frac{\pi_\alpha(s\restriction_F) = p \cdot f \qquad \Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(f) \qquad \Delta' = \Delta[\alpha \mapsto \varnothing]}{\langle c[\alpha \mapsto \mathsf{skip}], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c[\alpha \mapsto \mathsf{idle}], \Delta', s \cdot \boldsymbol{\alpha}{:}v\rangle}\ \textsc{Ret}$$

$$\frac{\begin{array}{c}\forall \alpha \in s.c'[\alpha] = \mathsf{dead} \\ \forall \alpha \in \Upsilon.\alpha \notin s \Rightarrow c'[\alpha] = \mathsf{halt}\end{array}}{\langle c, \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c', \Delta_0, s \cdot \lightning\rangle}\ \textsc{Crash}$$

$$\frac{\begin{array}{c}s = s' \cdot \lightning \qquad \vec{r} = \mathsf{perm}(R_E) \\ C = \mathsf{sequence}(\vec{r}, M[\alpha]^r)\end{array}}{\langle c[\alpha \mapsto \mathsf{halt}], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c[\alpha \mapsto C], \Delta, s \cdot \boldsymbol{\alpha}{:}r\rangle}\ \textsc{StartRec}$$

$$\frac{\begin{array}{c}\pi_\alpha(s\restriction_{F \cup R_F}) = s' \cdot r \qquad \Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(r) \qquad \Delta' = \Delta[\alpha \mapsto \varnothing] \\ \forall \alpha \in \Upsilon.c[\alpha] = \mathsf{dead} \Rightarrow c'[\alpha] = \mathsf{dead} \qquad \forall \alpha \in \Upsilon.c[\alpha] \neq \mathsf{dead} \Rightarrow c'[\alpha] = \mathsf{idle}\end{array}}{\langle c[\alpha \mapsto \mathsf{skip}], \Delta, s\rangle \twoheadrightarrow^{M}_{R_E} \langle c', \Delta', s \cdot \boldsymbol{\alpha}{:}v\rangle}\ \textsc{EndRec}$$

$$\text{where}\ \ \mathsf{sequence}(\vec{r}, C) = \begin{cases} C & \vec{r} = \epsilon \\ (x_r \leftarrow r(a)); \mathsf{sequence}(\vec{r}', C) & \vec{r} = r \cdot \vec{r}' \wedge a \in \mathsf{par}(r) \wedge \mathsf{reserved}(x_r) \end{cases}$$

Fig. 11. Module Operational Semantics ($\twoheadrightarrow_{R_E}$)

Usually, the client of the overlay object F will follow certain constraints when using the it. For example, when using a lock, one is supposed to invoke the lock acquire and the lock release in an alternating fashion. These constraints on clients often help us to prove a stronger and more useful specification ν_F. We use a strategy $\mu_F : \dagger(F \circlearrowleft R_F)$ to encode these client specifications. In the main text, we do not consider the client specification due to space limit, and we consider it in the program logic in this appendix. Most of the proof rules are the same except the Prim rule.

The objective of our program logic is to establish the judgement

$$\mu_F \vdash M : (\nu'_E, \nu_E) \to (\nu'_F, \nu_F) \qquad \text{or} \qquad \mu_F \vdash M : (\nu'_E, \nu_E) \to \langle \nu'_F, \nu_F\rangle$$

which means under the assumption that the client will use the overlay F according to strategy μ_F, the implementation M implements F with either a crash-aware interface (ν'_F, ν_F) or a durable interface $\langle \nu'_F, \nu_F\rangle$, using the crash-aware underlay E with a valid interface (ν'_E, ν_E). Concrete specification ν'_F is defined by running M above ν'_E of the underlay, i.e., $\nu'_F = (\nu'_E; [\![M]\!]_{R_E} \cap \mu_F)\restriction_{(F \circlearrowleft R_F)}$. The program logic's soundness guarantees the validity of the crash-aware/durable overlay interface. With the validity, we may use the object F and its interface to implement and verify another layer of objects above it.

G.3 The Rely-Guarantee Crash Linearizability Hoare Logic (CLHL) for Durable Linearizability

We have been using a simplified rely-guarantee crash Hoare logic in the main text, while we develop a more expressive one in this appendix. Their main difference is that the CLHL in this appendix uses a binary relation between the pre-state and post-state of a program as the post-condition. This gives the logic more expressiveness and allows us to verify more complicated programs. We prove the CLHL with binary post-conditions to be sound, which implies the one with unary post-conditions in the main text to be sound, because it is strictly less expressive than the former one.

The program logic uses as proof configurations triples $(\Delta, s, \rho) \in \mathsf{Config} := \mathsf{ModState} \times \mathsf{Poss}$, where Poss is a set of possibilities and is of type $\dagger F$. We define a configuration triple to be valid if and only if $s{\upharpoonright}_F$ is linearizable to ρ and ρ is linearizable to ν_F. This is exactly the definition of the durable linearizability: after removing recoveries and crashes, the trace is linearizable to its specification. We maintain this as an invariant in proofs to ensure that the concrete trace s is always durably linearizable to ν_F after the recovery refinement. Pre-conditions P and crash post-conditions $Q_{\lightning}$ are given by sets of configurations, while post-conditions Q, rely conditions $\mathcal{R}$, and guarantee conditions $\mathcal{G}$ are specified as relations over the configurations. As usual, we define the stability requirements:

$$\mathsf{stable}(\mathcal{R}, P) \iff \mathcal{R} \circ P \subseteq P \qquad \mathsf{stable}(\mathcal{R}, Q) \iff \mathcal{R} \circ Q \subseteq Q \land Q \circ \mathcal{R} \subseteq Q$$

G.3.1 *Top Level Rules.*

The top level rule Object Impl proves M implements the overlay $\langle \nu'_F, \nu_F \rangle$ using the underlay (ν'_E, ν_E).

$$\frac{\begin{array}{c}\forall \alpha, \alpha' \in \Upsilon. \alpha \neq \alpha' \Rightarrow \mathcal{G}[\alpha] \cup \mathsf{invoke}_\alpha(-) \cup \mathsf{return}_\alpha(-) \subseteq \mathcal{R}[\alpha'] \\ \forall \alpha \in \Upsilon. \mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash^F_\alpha M[\alpha] \qquad \forall \alpha \in \Upsilon. I \vDash^R_\alpha M[\alpha]\end{array}}{\mu_F \vdash M : (\nu'_E, \nu_E) \longrightarrow \langle \nu'_F, \nu_F \rangle} \text{ Object Impl}$$

It requires the prover to find an object invariant $I : \Upsilon \to \mathtt{Config} \to \mathtt{Prop}$ for the implementation and then prove the correctness of regular procedures and the recovery separately:

- *Verifying Regular Procedures.* To verify a concurrent object, the Object Impl rule requires finding the rely $\mathcal{R}$ and guarantee $\mathcal{G}$ of the object. The rely $\mathcal{R}[\alpha']$ of an agent needs to consider the effect of any other thread's executions, invocations, and returns, each represented by actions in $\mathcal{G}[\alpha]$, $\mathsf{invoke}_\alpha(-)$, and $\mathsf{return}_\alpha(-)$. And provers need to show $\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash^F_\alpha M[\alpha]$, which asserts that running regular procedures in F on the thread α, its environment will restrict its behavior in $\mathcal{R}[\alpha]$ while itself will only have behaviors in $\mathcal{G}[\alpha]$, and $I[\alpha]$ is satisfied when the thread α is idle.

Auxilliary Relations. We define some useful auxiliary relations here.

$$(\Delta, s, \rho)\mathsf{invoke}_\alpha(f)(\Delta', s', \rho') \iff \begin{pmatrix}(\Delta, s, \rho) \in \mathsf{idle}_\alpha \land s'{\upharpoonright}_F \in \mu_F \land \exists a. \Delta'(\alpha) = [\mathrm{arg} \mapsto a] \land \\ \forall \alpha' \neq \alpha. \Delta'(\alpha') = \Delta(\alpha) \land s' = s \cdot \boldsymbol{\alpha}{:}f \land \rho' = \rho \cdot \boldsymbol{\alpha}{:}f\end{pmatrix}$$

$$(\Delta, s, \rho)\mathsf{returned}_\alpha(f)(\Delta', s', \rho') \iff \begin{pmatrix}(\Delta', s', \rho') = (\Delta, s, \rho) \land \\ \exists v \in \mathsf{ar}(f). \Delta(\alpha)(\mathrm{ret}) = v \land \mathsf{last}(\pi_\alpha(\rho)) = \boldsymbol{\alpha}{:}v\end{pmatrix}$$

$$(\Delta, s, \rho)\mathsf{return}_\alpha(f(a))(\Delta', s', \rho') \iff \begin{pmatrix}\exists v \in \mathsf{ar}(f). \Delta(\alpha)(\mathrm{ret}) = v \land \Delta' = \varnothing \land \\ \rho' = \rho \land \mathsf{last}(\pi_\alpha(\rho)) = \boldsymbol{\alpha}{:}v \land s' = s \cdot \boldsymbol{\alpha}{:}v\end{pmatrix}$$

- The invoke relation requires the current thread α is idle in the pre-state, i.e., there is no pending invocation by α. The invoke relation then initialize the local environment according to method arguments and append the invocation event to both concrete trace s and possibility ρ. It also requires that after this invocation, s' satisfies the client specification. By composing this relation to pre-conditions, $\mathsf{invoke}_\alpha(f) \circ P$, the result ignores traces where client specification is violated, since both programmers and provers have no obligations to ensure the correctness in that case.
- The returned relation requires that the post-state's local environment already has the reserved variable ret assigned some value and the possibility contains the response to the latest invocation. By composing it to some post-condition, $\mathsf{returned}(f) \circ Q$, it requires provers to show that the latest invocation has returned and gets linearized.
- The return relation is a subsequent operation of the returned. It finishes the latest invocation by clearing its local environment and append the repose to the concrete trace s. The invoke and return are necessary because the program itself cannot manipulate the concrete trace by appending events of the overlay object. They are added by another client. And the invoke and return play the role of these clients by appending invocations to form a correct pre-condition and appending response to truly end a method execution.

Provers need to show $\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash_\alpha M_F[\alpha]$, which asserts that running the local implementation on thread α by invoking its methods, its environment will restrict their behaviors in $\mathcal{R}[\alpha]$ while itself will only have behaviors in $\mathcal{G}[\alpha]$, and $I[\alpha]$ is satisfied when the thread α is idle.

$$\forall \alpha, \alpha' \in \Upsilon. \alpha \neq \alpha' \Rightarrow \mathcal{G}[\alpha] \cup \mathsf{invoke}_\alpha(-) \cup \mathsf{return}_\alpha(-) \subseteq \mathcal{R}[\alpha'] \qquad \forall \alpha \in \Upsilon. \mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash_\alpha M_F[\alpha]$$

The Local Impl rule proves this judgement by splitting $I[\alpha]$ into conjunctions of $P[\alpha]^f$, each specifying the pre-condition of a method invocation, and then proving a series of objectives.

$$\frac{\begin{array}{c} I[\alpha] = \cap_{f \in F} P[\alpha]^f \qquad \forall f \in F. (\Delta_0, \epsilon, \epsilon) \in P[\alpha]^f \qquad \forall f \in F. \mathsf{stable}(\mathcal{R}[\alpha], P[\alpha]^f) \\ \forall f \in F. \mathcal{R}[\alpha], \mathcal{G}[\alpha] \vDash_\alpha^f \{P[\alpha]^f\} M[\alpha]^f \{Q[\alpha]^f\}\{\top\} \qquad \forall f \in F. \mathsf{return}_\alpha(f) \circ Q[\alpha]^f \subseteq I[\alpha] \end{array}}{\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash_\alpha M_F[\alpha]} \textsc{Local Impl}$$

- First of all, each pre-condition $P[\alpha]^f$ needs to include the initial configuration. Each pre-condition must be stable under interferences (the rely $\mathcal{R}[\alpha]$) of the environment, and therefore the invariant $I[\alpha]$ is also stable w.r.t. environment interfaces.
- Then, provers need to show that each method f satisfies

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \vDash_\alpha^f \{P[\alpha]^f\} M[\alpha]^f \{Q[\alpha]^f\}\{\top\}$$

which is a shorthand for the CLHL hexad below by hiding auxiliary relations applied to pre-/post-conditions.

$$\mathcal{R}[\alpha], \mathcal{G}[\alpha] \vDash_\alpha \{\mathsf{invoke}_\alpha(f) \circ P[\alpha]^f\} M[\alpha]^f \{\mathsf{returned}_\alpha(f) \circ Q[\alpha]^f\}\{\top\}$$

A hexad of the form $\mathcal{R}, \mathcal{G} \vDash_\alpha \{P\}C\{Q\}\{Q_\lightning\}$ means that given states satisfying P, running the program C in an environment with interference in $\mathcal{R}$ and on thread α will produce actions in $\mathcal{G}$, and if it terminates normally, the state will satisfy Q, and if it crashes, the state will satisfy $Q_\lightning$. A hexad is provable with proof rules introduced later. It is worth mentioning that *there is no need to explicitly specify and prove a crash post-condition for each regular method*, and we can simply put $\top$ as the crash post-condition. This is true because:

(1) The guarantee $\mathcal{G}[\alpha]$ of the current thread is included in any other thread's rely $\mathcal{R}[\alpha']$, and therefore any step during the execution of any method in thread α is captured in $\mathcal{R}[\alpha']$.
(2) For any other thread α', its invariant $I[\alpha']$ is stable w.r.t. $\mathcal{R}[\alpha']$, which means any state after any execution step of any method in thread α (captured in $\mathcal{R}[\alpha']$) is in $I[\alpha']$.

(3) Therefore, the state of thread α will satisfy any other thread's invariant $I[\alpha']$ at any time (including the point of crash), and the crash post-condition in α can be derived from $I[\alpha']$.

- Lastly, after finished the execution of a method and returned from it, the program state need to satisfy the invariant so that the current thread can still access the object and invoke its procedures.

$$\forall f \in F.\mathsf{return}_\alpha(f) \circ Q[\alpha]^f \subseteq I[\alpha]$$

• *Verifying the Recovery.* Then, to ensure the durability of the object, provers need to show $I \vDash^R_\alpha M[\alpha]$, which means whenever crash happens, the execution of the recovery on any thread α can restore the program state to satisfy the object invariant I. It can be verified via the Recover rule.

$$\dfrac{\begin{array}{c}\mathsf{ID}, \top \vDash^r_\alpha \{P_r[\alpha]\}M[\alpha]^r\{Q_r[\alpha]\}\{Q_\lightning[\alpha]\} \qquad Q_\lightning[\alpha] \subseteq P_r[\alpha] \\ \cup_{\alpha' \in \Upsilon} I[\alpha'] \Rightarrow_\lightning Q_\lightning[\alpha] \qquad \mathsf{return}_\alpha(r) \circ Q_r[\alpha] \subseteq \cap_{\alpha' \in \Upsilon} I[\alpha']\end{array}}{I \vDash^R_\alpha M[\alpha]}\ \textsc{Recover Impl}$$

First of all, provers need to find the pre-condition P_r, the post-condition Q_r, and the crash post-condition $Q_\lightning$ of the recovery program, and prove the following hexad [3]

$$\mathsf{ID}, \top \vDash^r_\alpha \{P_r[\alpha]\}M[\alpha]^r\{Q_r[\alpha]\}\{Q_\lightning[\alpha]\}$$

which means running the recovery program $M[\alpha]^r$ on arbitrary thread α from states in $P_r[\alpha]$ will either recover the system into states in $Q_r[\alpha]$ or crash into states in $Q_\lightning[\alpha]$. Since the recovery program always starts execution after a crash, the crash post-condition $Q_\lightning$ needs to imply the recovery's pre-condition P_r.

As mentioned before, $I[\alpha']$ will serve as the crash post-condition of other threads. Therefore, we require that all $I[\alpha']$ crash into the crash post-condition $Q_\lightning$ of the recovery program to ensure that all possible crashes are considered. The crash-into relation ($\Rightarrow_\lightning$) transforms the assertion I into $Q_\lightning$, which means that any state satisfying P will satisfy $Q_\lightning$ immediately after a crash.

$$I \Rightarrow_\lightning Q_\lightning \iff \forall(\Delta, s, \rho) \in I.(\Delta_0, s \cdot \lightning, \rho) \in Q_\lightning$$

Lastly, after the execution of the recovery, the system is restored and ready to run and therefore, the program state after the recovery's return needs to imply the invariant $I[\alpha']$ of any thread α'.

G.3.2 CLHL Proof Rules for the Hexad. According to these top level rules, proofs of both the implementation and the recovery boil down to proofs of hexads like $\mathcal{R}, \mathcal{G} \vDash_\alpha \{P\}C\{Q\}\{Q_\lightning\}$. Figure 12 shows CLHL's proof rules for the hexad.

Among CLHL proof rules for the hexad, the core proof rule for proving the durable linearizability is the Prim rule. Firstly, we need to show that the pre-/post-condition and the crash post-condition can crash into ($\Rightarrow_\lightning$) the crash post-condition, because the crash can happen at any time, even after another crash.

$$P \Rightarrow_\lightning Q_\lightning \qquad Q \circ P \Rightarrow_\lightning Q_\lightning \qquad Q_\lightning \Rightarrow_\lightning Q_\lightning$$

Then, as any usual rely-guarantee logic, the pre-/post-condition needs to be stable w.r.t. the rely.

$$\mathsf{stable}(\mathcal{R}, P) \qquad \mathsf{stable}(\mathcal{R}, Q)$$

[3] We reuse the concurrent CLHL for the sequential recovery program, so the rely and guarantee for it are ID and $\top$.

$$\frac{\begin{array}{ccc} P \Rightarrow_{\lightning} Q_{\lightning} & Q \circ P \Rightarrow_{\lightning} Q_{\lightning} & Q_{\lightning} \Rightarrow_{\lightning} Q_{\lightning} \\ \text{stable}(\mathcal{R}, P) & \text{stable}(\mathcal{R}, Q) & \mathcal{G} \vdash_{\alpha} \{P\}B\{Q\} \end{array}}{\mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}B\{Q\}\{Q_{\lightning}\}} \textsc{Prim}$$

$$\frac{\mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}C_1\{Q_1\}\{Q_{\lightning}\} \qquad \mathcal{R}, \mathcal{G} \vDash_{\alpha} \{Q_1 \circ P\}C_2\{Q_2\}\{Q_{\lightning}\}}{\mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}C_1; C_2\{Q_2 \circ Q_1\}\{Q_{\lightning}\}} \textsc{Seq}$$

$$\frac{\text{stable}(\mathcal{R}, P) \qquad P \Rightarrow_{\lightning} Q_{\lightning}}{\mathcal{R}, \mathsf{ID} \vDash_{\alpha} \{P\}\mathsf{skip}\{\mathsf{ID}\}\{Q_{\lightning}\}} \textsc{Skip} \qquad \frac{\mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}C\{Q\}\{Q_{\lightning}\} \qquad Q \circ P \subseteq P}{\mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}C^{*}\{Q\}\{Q_{\lightning}\}} \textsc{Iter}$$

$$\frac{\mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}C_1\{Q\}\{Q_{\lightning}\} \qquad \mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}C_2\{Q\}\{Q_{\lightning}\}}{\mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}C_1 + C_2\{Q\}\{Q_{\lightning}\}} \textsc{Choice}$$

$$\frac{\begin{array}{ccccc} \text{stable}(\mathcal{R}', P') & \text{stable}(\mathcal{R}', Q') & Q_{\lightning} \subseteq Q'_{\lightning} & Q'_{\lightning} \Rightarrow_{\lightning} Q'_{\lightning} & Q' \circ P' \Rightarrow_{\lightning} Q'_{\lightning} \\ P' \subseteq P & Q \subseteq Q' & \mathcal{R}' \subseteq \mathcal{R} & \mathcal{G} \subseteq \mathcal{G}' & \mathcal{R}, \mathcal{G} \vDash_{\alpha} \{P\}C\{Q\}\{Q_{\lightning}\} \end{array}}{\mathcal{R}', \mathcal{G}' \vDash_{\alpha} \{P'\}C\{Q'\}\{Q'_{\lightning}\}} \textsc{Conseq}$$

Fig. 12. Proof Rules for the CLHL Hexad

Lastly, we need to prove the commit rule $\mathcal{G} \vdash_{\alpha} \{P\}B\{Q\}$ for the primitive command B, which is directly defined by and provable by the semantics:

$$\mathcal{G} \vdash_{\alpha} \{P\}B\{Q\} \iff \begin{pmatrix} \forall(\Delta, s, \rho). s\restriction_{F \circlearrowleft R_F} \in \mu_F \land (\Delta, s, \rho) \in P \land \\ \forall(\Delta', s') \in [\![B]\!]_{\alpha}(\Delta, s) \cap \nu_E \Rightarrow s'\restriction_{F \circlearrowleft R_F} \in \mu_F \land \\ \exists\rho'.(\Delta, s, \rho)Q(\Delta', s', \rho') \land (\Delta, s, \rho)\mathcal{G}(\Delta', s', \rho') \land \rho \dashrightarrow \rho' \end{pmatrix}$$
$$\text{where } \rho \dashrightarrow \rho' \iff \exists t_P \in (M_F^P)^{*}.\rho \cdot t_P \rightsquigarrow_{\dagger F} \rho'$$

The commit rule states that command B will update configurations in P by appending the corresponding event to the concrete trace, which may be the commitment point of some pending operations. To maintain the invariant that s is durably linearizable to ρ, the commit rule allows a ghost update, $\rho \dashrightarrow \rho'$, where provers can append several response events to ρ and rewrite it according to $\rightsquigarrow_{\dagger F}$ to obtain ρ', a new possibility that s linearizes into. After the update made by the command B and the angelic update by the prover, the new configuration needs to satisfy the post-condition Q, and both updates need to be recorded in the guarantee.

To summarize, there are following steps to prove the durable linearizability of a concurrent object using CLHL:

(1) Firstly, find the rely condition $\mathcal{R}$, the guarantee condition $\mathcal{G}$, and the invariant I, and use Object Impl rule to generate separate proof goals for verifying regular procedures and the recovery program.
(2) The second step is to find pre-conditions and normal post-conditions of regular procedures and apply Local Impl to generate Hoare hexads for verifying regular procedures.
(3) The third step is to find the pre-condition, normal post-condition, and the crash post-condition (crash invariant) of the recovery program and prove the Hoare hexad for it and show that the object invariant will crash into the crash invariant.

(4) Lastly, prove hexads of each procedure and other side conditions generated by top level rules using CLHL's proof rules for hexads.

Remark. Notice that for the regular procedure verification, we do not use the crash post-condition (and instead always set it to $\top$), and for the recovery verification, we do not use the rely and guarantee condition because it is a sequential program. The another design choice is to use two different logics for them, each without the unnecessary part. But we find that their logics will have almost the same set of proof rules, and by unifying them into one rely-guarantee crash Hoare logic not only benefits our presentation but also simplifies the soundness proof.

Another benefit is that we can easily extend the CLHL for the recovery verification to concurrent recovery programs, where multiple recovery programs are running concurrently on multiple thread. The rely-guarantee condition in the Hoare hexad makes the program logic ready for the concurrent recovery verification. Moreover, since the regular procedure verification and the recovery verification are decoupled (with only an object invariant linking them together), we do not need to change the Local Impl rule when we extend the Recover Impl rule to concurrent recovery programs.

G.4 Soundness

The program logic is justified by the following soundness theorem.

Proposition G.1 (Soundness). *If $\mu_F \vdash M : (\nu'_E, \nu_E) \rightarrow \langle \nu'_F, \nu_F \rangle$ is provable, and (ν'_E, ν_E) is a valid underlay interface, and $\nu'_F = \nu'_E; [\![M]\!]_{R_E} \cap \mu_F$, then $\langle \nu'_F, \nu_F \rangle$ is a valid overlay interface with*

$$\nu'_F \upharpoonleft_{F^{\lightning}} \subseteq \mathsf{dur}(\nu_F).$$

To prove the soundness (proposition G.1), we extend methods in [23, 31] and establish the overlay interface's validity in four steps depicted in the formula below. (1) We first use the recovery refinement in §5.1 to remove the underlay's refinement. (2) Then, by the validity of the underlay interface, and observational refinement, we can use their specification in the execution of the overlay instead of using their concrete traces. (3) We use the linking lemma G.2 to integrate underlay's specification in overlay's denotation, which makes the next step easier. (4) The key step is the auxiliary soundness (lemma G.6), which establish the linearization from overlay's concrete traces to its specification.

$$\begin{aligned}
\nu'_F \upharpoonleft_{F^{\lightning}} &= (\nu'_E; [\![M]\!]_{R_E} \cap \mu_F) \upharpoonleft_{F^{\lightning}} && \text{By Definition of } \nu'_F \\
&= ((\nu'_E; [\![M]\!]_{R_E}) \upharpoonleft_{E^{\lightning} \multimap F \circlearrowleft R_F} \cap \mu_F) \upharpoonleft_{F^{\lightning}} && \\
&\subseteq (\nu'_E \upharpoonleft_{E^{\lightning}}; [\![M]\!]_{\varnothing} \cap \mu_F) \upharpoonleft_{F^{\lightning}} && \text{(1) Recovery Refinement} \\
&\subseteq (\nu_E; [\![M]\!]_{\varnothing} \cap \mu_F) \upharpoonleft_{F^{\lightning}} && \text{(2) Validity of } (\nu'_E, \nu_E) \text{ + Obs. Ref.} \\
&= ([\![\mathsf{Link}\ \nu_E; M]\!] \cap \mu_F) \upharpoonleft_{F^{\lightning}} && \text{(3) Linking (lemma G.2)} \\
&\subseteq \mathsf{dur}(\nu_F) && \text{(4) Auxiliary Soundness (lemma G.6)}
\end{aligned}$$

To make the auxiliary soundness proof easier, we first embed the underlay's specification into an auxiliary module semantics by $- \twoheadrightarrow^{M}_{\nu_E} -$.

$$\twoheadrightarrow \subseteq (\mathsf{Cont} \times \mathsf{ModState}) \times (\mathsf{CMod} \times \mathbf{Crash}) \times (\mathsf{Cont} \times \mathsf{ModState})$$

$$\frac{f \in F \qquad a \in \mathsf{par}(f) \qquad \Delta' = \Delta[\alpha \mapsto [\mathsf{arg} \mapsto a]]}{\langle c[\alpha \mapsto \mathsf{idle}], \Delta, s\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c[\alpha \mapsto M[\alpha]^{f}], \Delta', s \cdot \boldsymbol{\alpha}\text{:}f\rangle}$$

$$\frac{\langle C, \Delta, s{\upharpoonright_E}\rangle \longrightarrow_{\alpha} \langle C', \Delta', s'{\upharpoonright_E}\rangle \qquad s'{\upharpoonright_E} \in \nu_E}{\langle c[\alpha \mapsto C], \Delta, s\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c[\alpha \mapsto C'], \Delta', s'\rangle}$$

$$\frac{\pi_{\alpha}(s{\upharpoonright_F}) = p \cdot f \qquad \Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(f) \qquad \Delta' = \Delta[\alpha \mapsto \varnothing]}{\langle c[\alpha \mapsto \mathsf{skip}], \Delta, s\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c[\alpha \mapsto \mathsf{idle}], \Delta', s \cdot \boldsymbol{\alpha}\text{:}v\rangle}$$

$$\frac{\begin{array}{c}\forall \alpha \in s.c'[\alpha] = \mathsf{dead} \\ \forall \alpha \in \Upsilon.\alpha \notin s \Rightarrow c'[\alpha] = \mathsf{halt}\end{array}}{\langle c, \Delta, s\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c', \Delta_0, s \cdot \lightning\rangle} \qquad \frac{s = s' \cdot \lightning}{\langle c[\alpha \mapsto \mathsf{halt}], \Delta, s\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c[\alpha \mapsto M[\alpha]^{r}], \Delta, s \cdot \alpha : r\rangle}$$

$$\frac{\begin{array}{c}\pi_{\alpha}(s{\upharpoonright_{F \cup R_F}}) = s' \cdot r \qquad \Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(r) \qquad \Delta' = \Delta[\alpha \mapsto \varnothing] \\ \forall \alpha \in \Upsilon.c[\alpha] = \mathsf{dead} \Rightarrow c'[\alpha] = \mathsf{dead} \qquad \forall \alpha \in \Upsilon.c[\alpha] \neq \mathsf{dead} \Rightarrow c'[\alpha] = \mathsf{idle}\end{array}}{\langle c[\alpha \mapsto \mathsf{skip}], \Delta, s\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c', \Delta', s \cdot \alpha : v\rangle}$$

The only difference between $- \twoheadrightarrow^{M}_{\nu_E} -$ and $- \twoheadrightarrow^{M}_{\varnothing} -$ is the rule that lifts thread local reductions, where we ask steps by the underlay satisfies its specification ν_E. We define the linking denotation $[\![\mathsf{Link}\nu_E; M]\!] : \dagger E^{\lightning} \multimap \dagger(F \circlearrowleft R_F)$ by the formula

$$[\![\mathsf{Link}\nu_E; M]\!] := \{s \mid \exists c \in \mathsf{Cont}, \Delta \in (\Upsilon \to \mathsf{Env}).\langle c_0, \Delta_0, \epsilon\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c, \Delta, s\rangle\} \subseteq P_{\dagger E^{\lightning} \multimap \dagger(F \circlearrowleft R_F)}.$$

Lemma G.2 allows the transformation between the module denotation and the auxiliary linking denotation. Its proof is similar to the one in Oliveira Vale et al. [31].

Lemma G.2 (Linking). *For any $M \in \mathsf{CMod}$ and given $\nu_E : \dagger E^{\lightning}$, we have*

$$\nu_E; [\![M]\!]_{\varnothing} = [\![\mathsf{Link}\nu_E; M]\!]$$

Lemma G.3. *For any c, Δ, s, M, ν_E, if $\langle c_0, \Delta_0, \epsilon\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c, \Delta, s\rangle$, then*

$$\mathsf{last}(s) = \lightning \iff \forall \alpha.c(\alpha) \in \{\mathsf{dead}, \mathsf{halt}\}.$$

Proof. By discussing the last reduction step in $\langle c_0, \Delta_0, \epsilon\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c, \Delta, s\rangle$. □

Definition G.4 (Safety Judgement). We define the judgement $\mathsf{safe}_{\alpha}(\mathcal{R}, \mathcal{G}, P_0, P, s, Q, Q_{\lightning})$ inductively as follows:

$$\frac{\mathsf{rely}(\mathcal{R}, P) \subseteq Q \circ P_0}{\mathsf{safe}_{\alpha}(\mathcal{R}, \mathcal{G}, P_0, P, \mathsf{skip}, Q, Q_{\lightning})}\ \textsc{Done}$$

$$\frac{\begin{array}{c}\forall C'.C \rightarrowtail^{X}_{B} C' \Rightarrow \exists Q'.(\mathcal{G} \vdash_{\alpha} \{\mathsf{rely}(\mathcal{R}, P)\}B\{Q'\} \wedge \mathsf{stable}(\mathcal{R}, Q' \circ \mathsf{rely}(\mathcal{R}, P)) \wedge \\ \mathsf{safe}_{\alpha}(\mathcal{R}, \mathcal{G}, P_0, Q' \circ \mathsf{rely}(\mathcal{R}, P), C', Q, Q_{\lightning}) \wedge Q' \circ \mathsf{rely}(\mathcal{R}, P) \Rightarrow_{\lightning} Q_{\lightning})\end{array}}{\mathsf{safe}_{\alpha}(\mathcal{R}, \mathcal{G}, P_0, P, C, Q, Q_{\lightning})}\ \textsc{Step}$$

where $\mathsf{rely}(\mathcal{R}, P) \triangleq P \cup \mathcal{R} \circ P$.

Lemma G.5. *For any $\mathcal{R}, \mathcal{G}, P, s, Q, Q_{\lightning}$, if the quadruple $\mathcal{R}, \mathcal{G} \vDash_\alpha \{P\}s\{Q\}\{Q_{\lightning}\}$ is provable, then the followings are true.*

$$\mathsf{stable}(\mathcal{R}, P) \qquad P \Rightarrow_{\lightning} Q_{\lightning} \qquad Q_{\lightning} \Rightarrow_{\lightning} Q_{\lightning} \qquad \mathsf{safe}_\alpha(\mathcal{R}, \mathcal{G}, P, P, s, Q, Q_{\lightning})$$

Proof. By induction over the derivation tree of $\mathcal{R}, \mathcal{G} \vDash_\alpha \{P\}s\{Q\}\{Q_{\lightning}\}$. □

Lemma G.6 (Auxiliary Soundness). *If the judgement $\mu_F \vdash M : (\nu'_E, \nu_E) \to \langle \nu'_F, \nu_F \rangle$ is provable and (ν'_E, ν_E) is a valid crash-aware underlay interface, then*

$$(\llbracket \mathsf{Link}\, \nu'_E; M \rrbracket \cap \mu_F) \restriction_{F\lightning} \subseteq \mathsf{dur}(\nu_F).$$

Proof. Since $\mu_F \vdash M : (\nu'_E, \nu_E) \to \langle \nu'_F, \nu_F \rangle$ is provable, by the Object Impl rule, there exists $\mathcal{R}, \mathcal{G}, I, P_r, Q_r, Q_{\lightning}$ with the following conclusions.

$$\forall \alpha, \alpha' \in A. \alpha \neq \alpha' \Rightarrow \mathcal{G}[\alpha] \cup \mathsf{invoke}_\alpha(-) \cup \mathsf{return}_\alpha(-) \subseteq \mathcal{R}[\alpha] \tag{H-RG}$$

$$\forall \alpha \in \Upsilon. \mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash_\alpha M_F[\alpha] \tag{H-TLQ}$$

$$\forall \alpha, \alpha' \in A. Q_{\lightning}[\alpha] \subseteq P_r[\alpha'] \tag{H-RPre}$$

$$\forall \alpha. I[\alpha] \Rightarrow_{\lightning} Q_{\lightning}[\alpha] \tag{H-PC}$$

$$\forall \alpha, \alpha' \in A. \mathsf{return}_\alpha(r) \circ \mathsf{returned}_\alpha(r) \circ Q_r[\alpha] \circ \mathsf{invoke}_\alpha(r) \circ P_r[\alpha] \subseteq I[\alpha'] \tag{H-RPost}$$

$$\forall \alpha \in A. \mathsf{ID}, \top \vDash_\alpha \{\mathsf{invoke}_\alpha(r) \circ P_r[\alpha]\} M[\alpha]^r \{\mathsf{returned}_\alpha(r) \circ Q_r[\alpha]\}\{Q_{\lightning}[\alpha]\} \tag{H-RQ}$$

By the Local Impl rule and (H-TLQ), there exists $P[\alpha]^f, Q[\alpha]^f$ with the following conclusion about thread local executions of each regular function.

$$\cap_{f \in F} P[\alpha]^f = I[\alpha] \tag{H-Asrt}$$

$$\forall \alpha \in A, f \in F. (\Delta_0, \epsilon, \epsilon) \in P[\alpha]^f \tag{H-FInit}$$

$$\forall \alpha \in A, f \in F. \mathcal{R}[\alpha], \mathcal{G}[\alpha] \vDash_\alpha \{\mathsf{invoke}_\alpha(f) \circ P[\alpha]^f\} M[\alpha]^f \{\mathsf{returned}_\alpha(f) \circ Q[\alpha]^f\}\{\top\} \tag{H-FQ}$$

$$\forall \alpha \in A, f, f' \in F. \mathsf{return}_\alpha(f) \circ \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f \circ \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f \subseteq P[\alpha]^{f'} \tag{H-FPP}$$

$$\forall f \in F. \mathsf{stable}(\mathcal{R}[\alpha], P[\alpha]^f) \tag{H-Stb}$$

To prove $(\llbracket \mathsf{Link} \nu_E; M \rrbracket \cap \mu_F) \restriction_{F\lightning} \subseteq \mathsf{dur}(\nu_F)$, we only need to prove

$$\forall s \in \llbracket \mathsf{Link} \nu_E; M \rrbracket \cap \mu_F. s \restriction_{F\lightning} \in \mathsf{dur}(\nu_F)$$

which is equivalent to the following by definitions of auxiliary denotation and durable linearizability.

$$\forall c, \Delta, s. \langle c_0, \Delta_0, \epsilon \rangle \twoheadrightarrow^{M}_{\nu_E} \langle c, \Delta, s \rangle \wedge s \restriction_{F\lightning \& R_F} \in \mu_F \Rightarrow \exists \rho_F \in \nu_F. s \restriction_F \dashrightarrow \rho_F$$

We generalize this proposition to the following one.

Proposition G.7. *For any c, Δ, s, if $\langle c_0, \Delta_0, \epsilon \rangle \twoheadrightarrow^{M}_{\nu_E} \langle c, \Delta, s \rangle$, then when $s\upharpoonright_{F^{\lightning}\&R_F} \in \mu_F$ there exists a current linearization $\rho_F \in \nu_F$ and the followings hold.*

$$s\upharpoonright_F \dashrightarrow \rho_F \tag{G-Lin}$$

$$\wedge \forall \alpha . c[\alpha] \neq \mathsf{dead} \Rightarrow$$

$$\exists P_\alpha . (\Delta, s, \rho_F) \in P_\alpha \wedge (\mathsf{halt} \notin c \Rightarrow \mathsf{stable}(\mathcal{R}[\alpha], P_\alpha)) \tag{G-Pa}$$

$$\wedge\ (c[\alpha] = \mathsf{idle} \Rightarrow P_\alpha \subseteq I[\alpha]) \tag{G-Idle}$$

$$\wedge\ \big((\forall \alpha' . c[\alpha'] \in \{\mathsf{dead}, \mathsf{halt}\}) \Rightarrow \exists \alpha' . P_\alpha \subseteq Q_{\lightning}[\alpha']\big) \tag{G-Crs}$$

$$\wedge\ \big(c[\alpha] = \mathsf{halt} \wedge (\exists \alpha', C \in \mathsf{Com}. c[\alpha'] = C) \Rightarrow \exists \alpha' . P_\alpha \Rightarrow_{\lightning} Q_{\lightning}[\alpha']\big) \tag{G-Rec}$$

$$\wedge \left(\forall C \in \mathsf{Com}. c[\alpha] = C \Rightarrow \exists f \in F \cup R_F . \begin{pmatrix} \mathsf{safe}_\alpha \begin{pmatrix} \mathcal{R}[\alpha]^f, \mathcal{G}[\alpha]^f, \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, \\ P_\alpha, C, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f, Q_{\lightning}[\alpha]^f \end{pmatrix} \\ \wedge (f \in R_F \Rightarrow \forall \alpha' \neq \alpha . c(\alpha') \in \{\mathsf{dead}, \mathsf{halt}\}) \\ \wedge P_\alpha \Rightarrow_{\lightning} Q_{\lightning}[\alpha]^f \wedge f = \mathsf{last}(\pi_\alpha(s\upharpoonright_{F^{\lightning}\&R_F})) \end{pmatrix} \right) \tag{G-Exe}$$

For conciseness, We define $(P[\alpha]^r, Q[\alpha]^r, Q_{\lightning}[\alpha]^r)$ as $(P_r[\alpha], Q_r[\alpha], Q_{\lightning}[\alpha])$, and $Q_{\lightning}[\alpha]^f = \top$ for $f \in F$, and $(\mathcal{R}[\alpha]^f, \mathcal{G}[\alpha]^f)$ to be $(\mathcal{R}[\alpha], \mathcal{G}[\alpha])$ for $f \in F$, and $(\mathcal{R}[\alpha]^r, \mathcal{G}[\alpha]^r)$ to be $(\mathsf{ID}, \top)$ for the recovery r. For the right conjunct of (G-Pa), we would usually have $\mathsf{halt} \notin c$ and use/prove it simply as $\mathsf{stable}(\mathcal{R}[\alpha], P_\alpha)$ at most of the time. We only consider the LHS when necessary.

To prove the auxiliary soundness, it suffice to prove proposition G.7 under hypotheses introduced so far. We prove it by induction on the length of the reduction $\langle c_0, \Delta_0, \epsilon \rangle \twoheadrightarrow^{M}_{\nu_E} \langle c, \Delta, s \rangle$.

- ***Base Case***. In the base case, we have $(c_0, \Delta_0, \epsilon) = (c, \Delta, s)$. And we take the current linearization $\rho_F = \epsilon$, which satisfies (G-Lin). For each α, we take $P_\alpha = I[\alpha] = \cap_{f \in F} P[\alpha]^f$ ((H-Asrt) and (G-Pa)) is apparent by (H-FInit) and (G-Idle). Other conditions (G-Exe), (G-Rec), and (G-Crs) because the LHS of the implication is false.

- ***Inductive Step***. In the inductive step, we have

$$\langle c_0, \Delta_0, \epsilon \rangle \twoheadrightarrow^{M*}_{\nu_E} \langle c, \Delta, s \rangle \text{ and } \langle c, \Delta, s \rangle \twoheadrightarrow^{M}_{\nu_E} \langle c', \Delta', s' \rangle$$

and the induction hypothesis that if $s\upharpoonright_{F^{\lightning}\&R_F} \in \mu_F$, then there exists $\rho_F \in \nu_F$ with the followings.

$$s\upharpoonright_F \dashrightarrow \rho_F \tag{IH-Lin}$$

$$\wedge \forall \alpha . c[\alpha] \neq \mathsf{dead} \Rightarrow$$

$$\exists P_\alpha . (\Delta, s, \rho_F) \in P_\alpha \wedge (\mathsf{halt} \notin c \Rightarrow \mathsf{stable}(\mathcal{R}[\alpha], P_\alpha)) \tag{IH-Pa}$$

$$\wedge\ (c[\alpha] = \mathsf{idle} \Rightarrow P_\alpha \subseteq I[\alpha]) \tag{IH-Idle}$$

$$\wedge\ \big((\forall \alpha' . c[\alpha'] \in \{\mathsf{dead}, \mathsf{halt}\}) \Rightarrow \exists \alpha' . P_\alpha \subseteq Q_{\lightning}[\alpha']\big) \tag{IH-Crs}$$

$$\wedge\ \big(c[\alpha] = \mathsf{halt} \wedge (\exists \alpha', C \in \mathsf{Com}. c[\alpha'] = C) \Rightarrow \exists \alpha' . P_\alpha \Rightarrow_{\lightning} Q_{\lightning}[\alpha']\big) \tag{IH-Rec}$$

$$\wedge \left(\forall C \in \mathsf{Com}. c[\alpha] = C \Rightarrow \exists f \in F \cup R_F . \begin{pmatrix} \mathsf{safe}_\alpha \begin{pmatrix} \mathcal{R}[\alpha]^f, \mathcal{G}[\alpha]^f, \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, \\ P_\alpha, C, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f, Q_{\lightning}[\alpha]^f \end{pmatrix} \\ \wedge (f \in R_F \Rightarrow \forall \alpha' \neq \alpha . c(\alpha') \in \{\mathsf{dead}, \mathsf{halt}\}) \\ \wedge P_\alpha \Rightarrow_{\lightning} Q_{\lightning}[\alpha]^f \wedge f = \mathsf{last}(\pi_\alpha(s\upharpoonright_{F^{\lightning}\&R_F})) \end{pmatrix} \right) \tag{IH-Exe}$$

And we want to find a $\rho'_F \in \nu_F$ and for each agent α that is not dead in c', find a P'_α and prove (G-Lin), (G-Pa), (G-Idle), (G-Exe), (G-Rec), and (G-Crs) for (c', Δ', s'), under the condition that $s' \restriction_{F^{\lightning}\,\&R_F} \in \mu_F$. By definition of the semantics, it is easy to observe that $s \sqsubseteq s'$, which means $s \restriction_{F^{\lightning}\,\&R_F} \sqsubseteq s' \restriction_{F^{\lightning}\,\&R_F}$. Moreover, since μ_F is prefix-closed, $s \restriction_{F^{\lightning}\,\&R_F} \in \mu_F$ is true and we can freely use above induction hypotheses.

We perform case analysis on the reduction $\langle c, \Delta, s\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c', \Delta', s'\rangle$ to prove the inductive step.

∗ *Invocation*. If the reduction is an invocation to a regular procedure, then there exists $\alpha, f \in F$, and $a \in \mathsf{par}(f)$ such that

$$c[\alpha] = \mathsf{idle} \qquad c' = c[\alpha \mapsto M[\alpha]^f] \qquad \Delta' = \Delta[\alpha \mapsto [\mathrm{arg} \mapsto a]] \qquad s' = s \cdot \boldsymbol{\alpha}{:}f.$$

We take $\rho'_F = \rho_F \cdot \boldsymbol{\alpha}{:}f$. By (IH-Lin), we know there exists t_P such that

$$s\restriction_F \cdot\, t_P \rightsquigarrow_{\dagger F} \rho_F$$

and we prove (G-Lin) by

$$s' \restriction_F = s\restriction_F \cdot \boldsymbol{\alpha}{:}f \cdot t_P \rightsquigarrow_{\dagger F} s\restriction_F \cdot\, t_P \cdot \boldsymbol{\alpha}{:}f \rightsquigarrow_{\dagger F} \rho_F \cdot \boldsymbol{\alpha}{:}f = \rho'_F.$$

We first prove the remaining goals for the α that invokes f. By (IH-Lin), (IH-Idle), and (IH-Pa), there exists a stable P_α such that

$$(c, \Delta, s) \in P_\alpha \subseteq I[\alpha] \subseteq P[\alpha]^f.$$

By definition, we have $(\Delta, s, \rho_F)\mathsf{invoke}_\alpha(f)(\Delta', s', \rho'_F)$, and therefore we define P'_α and prove the left conjunct of (G-Pa) by

$$(\Delta', s', \rho'_F) \in \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f \triangleq P'_\alpha.$$

By (H-FQ) and lemma G.5, we have

$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha], \mathcal{G}[\alpha], P'_\alpha, P'_\alpha, M[\alpha]^f, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f, Q_{\lightning}[\alpha]^f)$$

$$\mathsf{stable}(\mathcal{R}[\alpha], P'_\alpha) \qquad P'_\alpha \Rightarrow_{\lightning} Q_{\lightning}[\alpha]^f$$

which proves (G-Exe) and the right conjunct of (G-Pa). The remaining (G-Idle), (G-Rec), and (G-Crs) are apparent because the LHS of the implication is false.

For other α' that are not α, we take $P'_{\alpha'} = P_{\alpha'}$ and only need to prove (G-Lin). The remaining (G-Idle), (G-Exe), (G-Rec), and (G-Crs) are apparent because truth values of both the LHS and the RHS of implications are not changed compared to those in induction hypotheses. We only need to show

$$(\Delta', s', \rho'_F) \in P_{\alpha'}$$

which is true because $P_{\alpha'}$ is stable (by the left conjunct of (IH-Pa)) w.r.t. $\mathcal{R}[\alpha']$, which contains $\mathsf{invoke}_\alpha(f)$ (by (H-RG)) that changes (Δ, s, ρ_F) into (Δ', s', ρ'_F), and $(c, \Delta, s) \in P_{\alpha'}$ is true by (IH-Pa).

∗ *Return*. If the reduction is a return of a regular procedure, then there exists α, f, v such that

$$c[\alpha] = \mathsf{skip} \qquad c' = c[\alpha \mapsto \mathsf{idle}] \qquad \Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(f) \qquad \Delta' = \Delta[\alpha \mapsto \varnothing]$$

$$\mathsf{last}(\pi_\alpha(s\restriction_{F^{\lightning}\,\&R_F})) = f \in F \qquad s' = s \cdot \boldsymbol{\alpha}{:}v.$$

By (IH-Exe), there exists P_α such that

$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha], \mathcal{G}[\alpha], \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, P_\alpha, \mathsf{skip}, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f, Q_{\lightning}[\alpha]^f)$$

By the definition of the safety judgement and (G-Pa), we can further have

$$(\Delta, s, \rho_F) \in P_\alpha \subseteq \mathsf{rely}(\mathcal{R}[\alpha], P_\alpha) \subseteq \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f \circ \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f$$

which means ρ_F already has some $\boldsymbol{\alpha}{:}v'$ linearized to the end of $\pi_\alpha(\rho_F)$, where $v' = \Delta(\alpha)(\mathsf{res})$ by the definition of returned. Therefore, $\boldsymbol{\alpha}{:}v$ is linearized since $v' = \Delta(\alpha)(\mathsf{res}) = v$. We take $\rho'_F = \rho_F$ and have

$$(\Delta, s, \rho_F)\mathsf{return}_\alpha(f)(\Delta', s', \rho'_F).$$

By (IH-Lin), there exists t_P such that

$$s\restriction_F \cdot\, t_P \rightsquigarrow_{\dagger F} \rho_F$$

and $\boldsymbol{\alpha}{:}v \in t_P$ since $\mathsf{last}(\pi_\alpha(s\restriction_F)) = f$. We take $t'_P = t_P \backslash \boldsymbol{\alpha}{:}v$ and prove (G-Lin) by

$$s'\restriction_F = s\restriction_F \cdot \boldsymbol{\alpha}{:}v \cdot t'_P \rightsquigarrow_{\dagger F} s\restriction_F \cdot t_P \rightsquigarrow_{\dagger F} \rho_F = \rho'_F.$$

We first prove remaining goals for the α that returns from f. We define $P'_\alpha = I[\alpha]$ and by our previous argument and (H-FPP), (H-Asrt), we have

$$(\Delta', s', \rho'_F) \in \mathsf{return}_\alpha(f) \circ \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f \circ \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f \subseteq I[\alpha] = P'_\alpha$$

which proves the left conjunct of (G-Pa). By (H-FQ) and lemma G.5, we have

$$\forall f \in F.\mathsf{stable}(\mathcal{R}[\alpha], P[\alpha]^f)$$

which implies $\mathsf{stable}(\mathcal{R}[\alpha], \cap_{f\in F} P[\alpha]^f)$, i.e., $\mathsf{stable}(\mathcal{R}[\alpha], I[\alpha])$. And this proves the right conjunct of (G-Pa).

By (H-Asrt) and reflexivity, we prove (G-Idle). Other branches (G-Crs), (G-Rec), and (G-Exe) are apparent because their LHS of the implication is false.

For other α' that are not α, we take $P'_{\alpha'} = P_{\alpha'}$ and prove (G-Lin), (G-Idle), (G-Exe), (G-Rec), and (G-Crs) with similar argument as the invocation case.

∗ *Execution*. If the reduction is one step of execution, then there exists α and $C, C' \in \mathsf{Com}$ such that

$$c[\alpha] = C \qquad c' = c[\alpha \mapsto C'] \qquad \langle C, \Delta, s\rangle \longrightarrow_\alpha \langle C', \Delta', s'\rangle \qquad s'\restriction_E \in \nu_E.$$

By unfolding the definition of $- \longrightarrow_\alpha -$, we have

$$C \rightarrowtail^X_B C' \wedge (\Delta', s') \in [\![B]\!]^X_\alpha(\Delta, s) \tag{H-Cmd}$$

By the definition of $C \rightarrowtail^X_B C'$, we know $C \neq \mathsf{skip}$. By (IH-Exe), we know there exists some P_α and $f \in F \cup R_F$ such that

$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha]^f, \mathcal{G}[\alpha]^f, \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, P_\alpha, C, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f, Q_{\lightning}[\alpha]^f).$$

By unfolding the safety judgement's definition and by (H-Cmd), there exists Q' with the followings.

$$\mathcal{G}[\alpha]^f \vdash_\alpha \{\mathsf{rely}(\mathcal{R}[\alpha]^f, P_\alpha)\}B\{Q'\} \tag{H-BT}$$

$$\mathsf{stable}(\mathcal{R}[\alpha]^f, Q' \circ \mathsf{rely}(\mathcal{R}[\alpha]^f, P_\alpha)) \tag{H-MStb}$$

$$Q' \circ \mathsf{rely}(\mathcal{R}[\alpha]^f, P_\alpha) \Rightarrow_{\lightning} Q_{\lightning}[\alpha]^f \tag{H-CInto}$$

$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha]^f, \mathcal{G}[\alpha]^f, \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, Q' \circ \mathsf{rely}(\mathcal{R}[\alpha]^f, P_\alpha), C', \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f, Q_{\lightning}[\alpha]^f) \tag{H-C'Safe}$$

By (IH-Pa), we have $(\Delta, s, \rho_F) \in P_\alpha \subseteq \mathsf{rely}(\mathcal{R}[\alpha]^f, P_\alpha)$. By $s'\restriction_E \in \nu_E$ and (H-Cmd), we have $(\Delta', s') \in [\![B]\!]^X_\alpha(\Delta, s) \cap \nu_E$. And by $s\restriction_{F^{\lightning}\&R_F} \in \mu_F$, (H-BT), and the definition of single instruction's judgement, we know there exists ρ' such that

$$(\Delta, s, \rho_F)Q'(\Delta', s', \rho') \qquad (\Delta, s, \rho_F)\mathcal{G}[\alpha]^f(\Delta', s', \rho') \qquad \rho_F \dashrightarrow \rho'$$

And by (IH-Lin) and transitivity of $- \dashrightarrow -$, we have $s{\upharpoonright_F} \dashrightarrow \rho_F \dashrightarrow \rho'$, i.e., $s{\upharpoonright_F} \dashrightarrow \rho'$. By taking $\rho'_F = \rho'$, we prove (G-Lin).

We take $P'_\alpha = Q' \circ \mathsf{rely}(\mathcal{R}[\alpha]^f, P_\alpha)$. Since $(\Delta, s, \rho_F) \in \mathsf{rely}(\mathcal{R}[\alpha]^f, P_\alpha)$ and $(\Delta, s, \rho_F) Q' (\Delta', s', \rho')$, we know $(\Delta', s', \rho'_F) \in P'_\alpha$. Combined with the stability from (H-MStb), we prove (G-Pa) when $f \in F$.

We prove The first and third conjuncts of (G-Exe) directly by (H-C'Safe) and (H-CInto). The last conjunct of (G-Exe) is true by (IH-Exe) because the current reduction does not modify $\pi_\alpha(s{\upharpoonright_F})$. If $f \in F$, then the second conjunct is true. If $f \in R_F$, because the current reduction does not change other locations in the continuation c, the second conjunct is true by (IH-Exe). This also proves (G-Pa) under the condition $f \in R_F$ since the LHS of the implication is false now.

Other branches (G-Idle), (G-Rec), and (G-Crs) are apparent because the LHS of the implication is false.

If $f \in F$, for other α' that is not α, we take $P'_{\alpha'} = P_{\alpha'}$ and prove (G-Lin), (G-Idle), (G-Exe), (G-Rec), and (G-Crs) with similar argument as previous cases.

If $f \in R_F$, we have shown that any other thread α' are either dead or halt in both c and c' and we need to maintain (G-Rec) for them. We take $P'_{\alpha'} = P'_\alpha = Q' \circ \mathsf{rely}(\mathcal{R}[\alpha]^f, P_\alpha)$ and (G-Pa) is already proved and branches other than (G-Rec) are trivially true. We prove (G-Rec) by (H-CInto).

∗ <u>*Crash*</u>. If the reduction is a crash, then

$$\forall \alpha \in s. c'[\alpha] = \mathsf{dead} \qquad \forall \alpha \in \Upsilon. \alpha \notin s \Rightarrow c'[\alpha] = \mathsf{halt} \qquad \Delta' = \Delta_0 \qquad s' = s \cdot \lightning.$$

We take $\rho'_F = \rho_F$. Since $s{\upharpoonright_F} = s'{\upharpoonright_F}$ and by (IH-Lin), we know $s'{\upharpoonright_F} \dashrightarrow \rho'_F$ and prove (G-Lin).

For any α, if $c'[\alpha] = \mathsf{dead}$, then we do not need to prove anything for it. If $c'[\alpha] = \mathsf{halt}$, then we know $\alpha \notin s$, which implies $c[\alpha] \in \{\mathsf{idle}, \mathsf{halt}\}$. For these two cases, we only need to prove (G-Pa) and (G-Crs). The branches (G-Idle), (G-Rec), and (G-Exe) are trivially true because their LHS of the implication is false.

+ $c[\alpha] = \mathsf{idle}$. By (IH-Pa) (IH-Idle),

  $$(\Delta, s, \rho_F) \in P_\alpha \subseteq I[\alpha].$$

  By (H-PC), we have

  $$(\Delta, s, \rho_F) \in I[\alpha] \Rightarrow_{\lightning} Q_{\lightning}[\alpha].$$

  We take $P'_\alpha = Q_{\lightning}[\alpha]$ and by definition, we have $(\Delta', s', \rho'_F) = (\Delta_0, s \cdot \lightning, \rho'_F) \in P'_\alpha$ and proves (G-Pa), since the right conjunct of it is trivially true.
  By reflexivity, $P'_\alpha = Q_{\lightning}[\alpha] \subseteq Q_{\lightning}[\alpha]$, (G-Crs) is true.

+ $c[\alpha] = \mathsf{halt}$. It can be further divided into following cases.
  - $\forall \alpha'. c[\alpha'] \in \{\mathsf{dead}, \mathsf{halt}\}$, i.e., recovery has not started yet. We take $P'_\alpha = Q_{\lightning}[\alpha]$. By (H-RQ) and lemma G.5, we have

    $$\forall \alpha. Q_{\lightning}[\alpha] \Rightarrow_{\lightning} Q_{\lightning}[\alpha]$$

    Then by (IH-Pa), (IH-Crs), we have

    $$(\Delta, s, \rho_F) \in P_\alpha \subseteq Q_{\lightning}[\alpha] \Rightarrow_{\lightning} Q_{\lightning}[\alpha] = P'_\alpha$$

    which indicates $(\Delta', s', \rho'_F) = (\Delta_0, s \cdot \lightning, \rho'_F) \in P'_\alpha$ by definition and proves (G-Pa).
    By reflexivity, $P'_\alpha = Q_{\lightning}[\alpha] \subseteq Q_{\lightning}[\alpha]$, (G-Crs) is true.

- $\exists \alpha', C \in \mathsf{Com}.c[\alpha'] = C$, i.e., recovery has started. By (IH-Pa) and (IH-Rec), we know
$$(\Delta, s, \rho_F) \in P_\alpha \Rightarrow_{\lightning} Q_{\lightning}[\alpha']$$
for some α'. Therefore, we take $P'_\alpha = Q_{\lightning}[\alpha']$. And we have $(\Delta', s', \rho'_F) = (\Delta_0, s \cdot \lightning, \rho'_F) \in P'_\alpha$ and prove (G-Pa).
By reflexivity, $P'_\alpha = Q_{\lightning}[\alpha'] \subseteq Q_{\lightning}[\alpha']$, (G-Crs) is true.

- $\exists \alpha'.c[\alpha'] = \mathsf{idle}$. This case is impossible, because idle and halt will not exist in c at the same time by the definition of the semantics.

$*$ *Recovery Invocation.* If the reduction is an invocation to the recovery, then there exists α such that
$$c[\alpha] = \mathsf{halt} \qquad c' = c[\alpha \mapsto M[\alpha]^r] \qquad \Delta' = \Delta \qquad \mathsf{last}(s) = \lightning \qquad s' = s \cdot \boldsymbol{\alpha}{:}r.$$
We take $\rho'_F = \rho_F$ and prove (G-Lin) by the same argument as the crash case.

For any α' other than α that are not dead, by lemma G.3, they are in halt state. Since the current reduction does not change $c[\alpha']$, $c'[\alpha']$ is still halt. And any thread dead are still dead, which showcase the second conjunct in (G-Exe) for α. While $c'[\alpha] = M[\alpha]^r$, the LHS of (G-Idle), (G-Crs), and (G-Exe) are all false for α' in the new configuration, which means they are proven. And when we found P'_α, we will take $P'_{\alpha'} = P'_\alpha$ and (G-Pa) and (G-Rec) will be proved for α' when we prove the invariant for α.

We take $P'_\alpha = \mathsf{invoke}_\alpha(r) \circ P_r[\alpha]$. By (IH-Pa), (IH-Crs), (H-RPre), we have
$$(\Delta, s, \rho_F) \in P_\alpha \subseteq Q_{\lightning}[\alpha] \subseteq P_r[\alpha].$$
And by definition, we have
$$(\Delta, s, \rho_F)\mathsf{invoke}_\alpha(r)(\Delta', s', \rho'_F)$$
which implies $(\Delta', s', \rho'_F) \in \mathsf{invoke}_\alpha(r) \circ P_r[\alpha] = P'_\alpha$ and proves (G-Pa) for both α and α'.

By (H-RQ) and lemma G.5, we have
$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha]^r, \mathcal{G}[\alpha]^r, P'_\alpha, P'_\alpha, M[\alpha]^r, \mathsf{returned}_\alpha(r) \circ Q[\alpha]^r, Q_{\lightning}[\alpha]^r)$$
$$\mathsf{stable}(\mathcal{R}[\alpha]^r, P'_\alpha) \qquad P'_\alpha \Rightarrow_{\lightning} Q_{\lightning}[\alpha]^r$$
which proves (G-Rec) for α', and proves (G-Exe) for α (the second conjunct is proved in last paragraph). Other branches (G-Idle) and (G-Crs) are trivially true.

$*$ *Recovery Return.* If the reduction is a return from the recovery, then there exists α and v such that
$$c[\alpha] = \mathsf{skip} \qquad \forall \alpha \in \Upsilon.c[\alpha] = \mathsf{dead} \Rightarrow c'[\alpha] = \mathsf{dead} \qquad \forall \alpha \in \Upsilon.c[\alpha] \neq \mathsf{dead} \Rightarrow c'[\alpha] = \mathsf{idle}$$
$$\Delta(\alpha)(\mathsf{res}) = v \in \mathsf{ar}(r) \qquad \Delta' = \Delta[\alpha \mapsto \varnothing] \qquad \mathsf{last}(\pi_\alpha(s{\upharpoonright_{F \cup R_F}})) = r \qquad s' = s \cdot \boldsymbol{\alpha}{:}v.$$
We take $\rho'_F = \rho_F$ and prove (G-Lin) by the same argument as the crash case.

We take $P'_{\alpha'} = I[\alpha']$ for any α' that are not dead.

By (IH-Exe), there exists P_α such that
$$\mathsf{safe}_\alpha(\mathcal{R}[\alpha]^r, \mathcal{G}[\alpha]^r, \mathsf{invoke}_\alpha(r) \circ P[\alpha]^r, P_\alpha, \mathsf{skip}, \mathsf{returned}_\alpha(r) \circ Q[\alpha]^r, Q_{\lightning}[\alpha]^r)$$
By the definition of the safety judgement and (IH-Pa), we can further have
$$(\Delta, s, \rho_F) \in P_\alpha \subseteq \mathsf{rely}(\mathcal{R}[\alpha]^r, P_\alpha) \subseteq \mathsf{returned}_\alpha(r) \circ Q[\alpha]^r \circ \mathsf{invoke}_\alpha(r) \circ P[\alpha]^r$$

By definition, we have

$$(\Delta, s, \rho_F)\mathsf{return}_\alpha(f)(\Delta', s', \rho'_F).$$

By (H-RPost), we have the following for any α'.

$$(\Delta', s', \rho'_F) \in \mathsf{return}_\alpha(f) \circ \mathsf{returned}_\alpha(r) \circ Q[\alpha]^r \circ \mathsf{invoke}_\alpha(r) \circ P[\alpha]^r \subseteq I[\alpha'] = P'_\alpha$$

which proves (G-Pa) for any α'.

Moreover, since any non-dead α' is in idle state, we only need to prove (G-Idle), which is trivially true by reflexivity, $P'_\alpha = I[\alpha'] \subseteq I[\alpha']$.

This concludes the proof of proposition G.7, which proves the auxiliary soundness.

□

Theorem G.8 (Soundness). *Proposition G.1 is true, i.e., the program logic is sound.*

Proof. We prove the soundness by the following derivation.

$$\begin{aligned}
\nu'_F \upharpoonright_{F^\lightning} &= (\nu'_E; [\![M]\!]_{R_E} \cap \mu_F) \upharpoonright_{F^\lightning} && \text{By Definition of } \nu'_F \\
&= ((\nu'_E; [\![M]\!]_{R_E}) \upharpoonright_{E^\lightning \multimap F \circlearrowleft R_F} \cap \mu_F) \upharpoonright_{F^\lightning} && \\
&\subseteq (\nu'_E \upharpoonright_{E^\lightning}; [\![M]\!]_\varnothing \cap \mu_F) \upharpoonright_{F^\lightning} && \text{(1) Recovery Refinement} \\
&\subseteq (\nu_E; [\![M]\!]_\varnothing \cap \mu_F) \upharpoonright_{F^\lightning} && \text{(2) Validity of } (\nu'_E, \nu_E) + \text{Obs. Ref.} \\
&= ([\![\mathsf{Link}\, \nu_E; M]\!] \cap \mu_F) \upharpoonright_{F^\lightning} && \text{(3) Linking (lemma G.2)} \\
&\subseteq \mathsf{dur}(\nu_F) && \text{(4) Auxiliary Soundness (lemma G.6)}
\end{aligned}$$

□

H A Program Logic for Crash-Aware Overlay Objects

In the main text, we mainly present the program logic for durable overlay objects. However, it is often necessary to verify crash-aware objects, e.g., volatile objects and buffered objects, which are crucial for implementations of many objects. Our file system example is a two layer crash-aware object implementation. Therefore, we propose a program logic for crash-aware linearizability and crash-aware overlay objects in this section.

We use the same programming language and semantics from G for crash-aware objects. For simplicity, we still enforce the durable assumption here so that we can reuse some previous definitions and conclusions.

H.1 Interfaces

We use the same interface definitions for crash-aware objects in G, which we repeat here. The interface of a crash-aware linearizable object E is a tuple

$$(\nu'_E : \dagger(E \circlearrowleft R_E) \in \mathbf{Dur}, \nu_E : \dagger E^\lightning \in \underline{\mathbf{Crash}}) \qquad \text{s.t.} \qquad \nu'_E \upharpoonright_{E^\lightning} \subseteq K_\lightning\, \nu_E$$

where ν'_E is the concrete specification that contains all possible traces the object can produce, which will include concurrent ones and will also contain crash events and recovery signatures, and ν_E is the linearized specification. The interface is valid if and only if after recovery refining the concrete specification (the projection onto $E^\lightning$), $\nu'_E \upharpoonright_{E^\lightning}$ is crash-aware linearizable to ν_E, i.e., $\nu'_E \upharpoonright_{E^\lightning} \subseteq K_\lightning\, \nu_E$.

The objective for the program logic is to establish the judgement:

$$\mu_F \vdash^{\mathsf{ca}} M : (\nu'_E, \nu_E) \longrightarrow (\nu'_F, \nu_F)$$

where μ_F is the client guarantee. The soundness will ensure (ν'_F, ν_F) to be valid given a valid underlay (ν'_E, ν_E).

H.2 The CLHL for Crash-Aware Linearizability

The program logic uses as proof configurations triples $(\Delta, s, \rho) \in \mathsf{Config} := \mathsf{ModState} \times \mathsf{Poss}$, where Poss is a set of possibilities and is of type $\dagger F^{\lightning}$. The possibility is different from the one in the program logic for durable objects, where we need to consider crash events in the linearization. We define assertions and rely-guarantees as sets and relations just like before.

The program logic is almost the same as the one for the program logic for durable objects, except one crucial difference: **we are using a different rewrite system now**.

Definition H.1. Let $A = (M_A, P_A)$ be a crash-aware game. We define a string rewrite system $(P_A, \rightsquigarrow^{\lightning}_A)$ with rewrite rules:

- $\forall m, m' \in M_A . \forall \alpha, \alpha' \in \Upsilon . \forall X \in \{O, P\} . \alpha \neq \alpha' \wedge \lambda_A(m) = \boldsymbol{\alpha}{:}X \wedge \lambda_A(m') = \boldsymbol{\alpha'}{:}X \implies m{\cdot}m' \rightsquigarrow^{\lightning}_A m' \cdot m$
- $\forall o, p \in M_A . \forall \alpha, \alpha' \in \Upsilon . \alpha \neq \alpha' \wedge \lambda_A(o) = \boldsymbol{\alpha}{:}O \wedge \lambda_A(p) = \boldsymbol{\alpha'}{:}P \implies o \cdot p \rightsquigarrow^{\lightning}_A p \cdot o$
- $\forall m \in M^{\Upsilon}_A . \forall \Upsilon \in \Upsilon . \lambda_A(m) = \boldsymbol{\alpha}{:}P \Rightarrow \lightning \cdot m \rightsquigarrow^{\lightning}_A m \cdot \lightning$

It differs from the rewrite system for concurrent games mainly in the third rule, which allows insertion of a proponent action before a crash, if this insertion is valid. We define the crash-aware ghost update as

$$s \overset{\lightning}{\dashrightarrow} t \iff \exists s_P \in (M^P_F)^* . s \cdot s_P \rightsquigarrow^{\lightning}_{\dagger F^{\lightning}} t$$

and lemma H.2 establishes the connection between the crash-aware update and crash-aware linearizability.

Lemma H.2. *For any crash-aware play s and t in $\dagger F^{\lightning}$, $s \overset{\lightning}{\dashrightarrow} t \Rightarrow s \overset{\lightning}{\leadsto} t$.*

For the configuration triple (Δ, s, ρ), we maintain the invariant that $s \overset{\lightning}{\dashrightarrow} \rho$, i.e., $s \overset{\lightning}{\leadsto} \rho$, and $\rho \in \nu_F$.

There are two differences in the program logic's proof rules. One is the primitive judgement $\mathcal{G} \vdash^{\mathsf{ca}}_{\alpha} \{P\}B\{Q\}$, where $B \in \mathsf{Prim}$. We now use the crash-aware ghost update to linearize the possibility.

$$\mathcal{G} \vdash^{\mathsf{ca}}_{\alpha} \{P\}B\{Q\} \iff \begin{pmatrix} \forall(\Delta, s, \rho) . s\upharpoonright_{F \circlearrowleft R_F} \in \mu_F \wedge (\Delta, s, \rho) \in P \wedge \\ \forall(\Delta', s') \in [\![B]\!]_{\alpha}(\Delta, s) \cap \nu_E \Rightarrow s'\upharpoonright_{F \circlearrowleft R_F} \in \mu_F \wedge \\ \exists \rho' . (\Delta, s, \rho) Q (\Delta', s', \rho') \wedge (\Delta, s, \rho) \mathcal{G} (\Delta', s', \rho') \wedge \rho \overset{\lightning}{\dashrightarrow} \rho' \end{pmatrix}$$

Another one is the crash-into relation $(- \Rightarrow^{\mathsf{ca}}_{\lightning} -)$, where we also add the crash event to the possibility as well. And we replace with this crash-into relation in Prim rule, Skip rule, Conseq rule, and Object Impl rule, obtaining the program logic for crash-aware linearizability.

$$P \Rightarrow^{\mathsf{ca}}_{\lightning} Q_{\lightning} \iff \forall(\Delta, s, \rho) \in P . (\Delta_0, s \cdot \lightning, \rho \cdot \lightning) \in Q_{\lightning}$$

Essential, the usage of this program logic is the same as the one for durable objects, except we use different possibility update operation when establishing the primitive judgement and need to prove with a slightly different crash-into relation. Fig. 13 shows all proof rules in the program logic for crash-aware objects.

Crash Hoare Logic Rules:

$$\frac{\begin{array}{ccc} P \Rightarrow^{\mathrm{ca}}_{\lightning} Q_{\lightning} & Q \circ P \Rightarrow^{\mathrm{ca}}_{\lightning} Q_{\lightning} & Q_{\lightning} \Rightarrow^{\mathrm{ca}}_{\lightning} Q_{\lightning} \\ \mathsf{stable}(\mathcal{R}, P) & \mathsf{stable}(\mathcal{R}, Q) & \mathcal{G} \vdash^{\mathrm{ca}}_{\alpha} \{P\}B\{Q\} \end{array}}{\mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}B\{Q\}\{Q_{\lightning}\}}\ \textsc{Prim}$$

$$\frac{\mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}C_1\{Q_1\}\{Q_{\lightning}\} \qquad \mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{Q_1 \circ P\}C_2\{Q_2\}\{Q_{\lightning}\}}{\mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}C_1; C_2\{Q_2 \circ Q_1\}\{Q_{\lightning}\}}\ \textsc{Seq}$$

$$\frac{\mathsf{stable}(\mathcal{R}, P) \qquad P \Rightarrow^{\mathrm{ca}}_{\lightning} Q_{\lightning}}{\mathcal{R}, \mathsf{ID} \vDash^{\mathrm{ca}}_{\alpha} \{P\}\mathsf{skip}\{\mathsf{ID}\}\{Q_{\lightning}\}}\ \textsc{Skip} \qquad \frac{\mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}C\{Q\}\{Q_{\lightning}\} \qquad Q \circ P \subseteq P}{\mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}C^{*}\{Q\}\{Q_{\lightning}\}}\ \textsc{Iter}$$

$$\frac{\mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}C_1\{Q\}\{Q_{\lightning}\} \qquad \mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}C_2\{Q\}\{Q_{\lightning}\}}{\mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}C_1 + C_2\{Q\}\{Q_{\lightning}\}}\ \textsc{Choice}$$

$$\frac{\begin{array}{c} \mathsf{stable}(\mathcal{R}', P') \qquad \mathsf{stable}(\mathcal{R}', Q') \qquad Q_{\lightning} \subseteq Q'_{\lightning} \qquad Q'_{\lightning} \Rightarrow^{\mathrm{ca}}_{\lightning} Q'_{\lightning} \qquad Q' \circ P' \Rightarrow^{\mathrm{ca}}_{\lightning} Q'_{\lightning} \\ P' \subseteq P \qquad Q \subseteq Q' \qquad \mathcal{R}' \subseteq \mathcal{R} \qquad \mathcal{G} \subseteq \mathcal{G}' \qquad \mathcal{R}, \mathcal{G} \vDash^{\mathrm{ca}}_{\alpha} \{P\}C\{Q\}\{Q_{\lightning}\} \end{array}}{\mathcal{R}', \mathcal{G}' \vDash^{\mathrm{ca}}_{\alpha} \{P'\}C\{Q'\}\{Q'_{\lightning}\}}\ \textsc{Conseq}$$

Top Level Rules:

$$\frac{\begin{array}{c} \forall f \in F.(\Delta_0, \epsilon, \epsilon) \in P[\alpha]^f \qquad \forall f \in F.P[\alpha]^f \subseteq \mathsf{idle}_{\alpha} \qquad \forall f \in F.\mathsf{stable}(\mathcal{R}[\alpha], P[\alpha]^f) \\ \forall f \in F.\mathcal{R}[\alpha], \mathcal{G}[\alpha] \vDash^{\mathrm{ca}}_{\alpha} \{\mathsf{invoke}_{\alpha}(f) \circ P[\alpha]^f\}M[\alpha]^f\{\mathsf{returned}_{\alpha}(f) \circ Q[\alpha]^f\}\{\top\} \\ \forall f, f' \in F.\mathsf{return}_{\alpha}(f) \circ \mathsf{returned}_{\alpha}(f) \circ Q[\alpha]^f \circ \mathsf{invoke}_{\alpha}(f) \circ P[\alpha]^f \subseteq P[\alpha]^{f'} \end{array}}{\mathcal{R}[\alpha], \mathcal{G}[\alpha], (\cap_{f \in F} P[\alpha]^f) \vDash^{\mathrm{ca}}_{\alpha} M_F[\alpha]}\ \textsc{Local Impl}$$

$$\frac{\begin{array}{c} \forall \alpha \in \Upsilon.\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash^{\mathrm{ca}}_{\alpha} M_F[\alpha] \\ \forall \alpha, \alpha' \in \Upsilon.\alpha \neq \alpha' \Rightarrow \mathcal{G}[\alpha] \cup \mathsf{invoke}_{\alpha}(-) \cup \mathsf{return}_{\alpha}(-) \subseteq \mathcal{R}[\alpha] \\ \forall \alpha.I[\alpha] \Rightarrow^{\mathrm{ca}}_{\lightning} Q_{\lightning}[\alpha] \qquad \forall \alpha, \alpha' \in \Upsilon.Q_{\lightning}[\alpha] \subseteq P_r[\alpha'] \\ \forall \alpha \in \Upsilon.\mathsf{ID}, \top \vDash^{\mathrm{ca}}_{\alpha} \{\mathsf{invoke}_{\alpha}(r) \circ P_r[\alpha]\}M[\alpha]^r\{\mathsf{returned}_{\alpha}(r) \circ Q_r[\alpha]\}\{Q_{\lightning}[\alpha]\} \\ \forall \alpha, \alpha' \in \Upsilon.\mathsf{return}_{\alpha}(r) \circ \mathsf{returned}_{\alpha}(r) \circ Q_r[\alpha] \circ \mathsf{invoke}_{\alpha}(r) \circ P_r[\alpha] \subseteq I[\alpha'] \end{array}}{\mu_F \vdash^{\mathrm{ca}} M : (\nu'_E, \nu_d, \nu_c) \longrightarrow (\nu'_F, \nu_F)}\ \textsc{Object Impl}$$

Fig. 13. Proof Rules in the Program Logic for Crash-Aware Objects

H.3 Soundness

The program logic is justified by the following soundness theorem.

Proposition H.3 (Soundness). *If $\mu_F \vdash^{\mathrm{ca}} M : (\nu'_E, \nu_E) \longrightarrow (\nu'_F, \nu_F)$ is provable, and (ν'_E, ν_E) is a valid underlay interface, and $\nu'_F = \nu'_E; [\![M]\!]_{R_E} \cap \mu_F$, then (ν'_F, ν_F) is a valid overlay interface with*

$$\nu'_F\restriction_{F\lightning} \subseteq K_{\lightning}\,\nu_F.$$

The soundness proof of the crash-aware program logic is also almost identical to the one of the durable program logic in G. We use the same auxiliary definitions for judgement of the crash-aware version and the only difference is the auxiliary soundness (lemma H.4), because it is the only one mentions the only thing different between the two logics, the linearization (i.e., the possibility update).

$$
\begin{aligned}
\nu'_F\upharpoonright_{F^{\lightning}} &= (\nu'_E; [\![M]\!]_{R_E} \cap \mu_F)\upharpoonright_{F^{\lightning}} && \text{By Definition of } \nu'_F \\
&= ((\nu'_E; [\![M]\!]_{R_E})\upharpoonright_{E^{\lightning} \multimap F \circlearrowleft R_F} \cap \mu_F)\upharpoonright_{F^{\lightning}} \\
&\subseteq (\nu'_E\upharpoonright_{E^{\lightning}}; [\![M]\!]_{\varnothing} \cap \mu_F)\upharpoonright_{F^{\lightning}} && \text{(1) Recovery Refinement} \\
&\subseteq (\nu_E; [\![M]\!]_{\varnothing} \cap \mu_F)\upharpoonright_{F^{\lightning}} && \text{(2) Validity of } (\nu'_E, \nu_E) + \text{Obs. Ref.} \\
&= ([\![\mathsf{Link}\, \nu_E; M]\!] \cap \mu_F)\upharpoonright_{F^{\lightning}} && \text{(3) Linking (lemma G.2)} \\
&\subseteq K_{\lightning}\, \nu_F && \text{(4) Auxiliary Soundness (lemma H.4)}
\end{aligned}
$$

Lemma H.4 (Auxiliary Soundness). *If the judgement $\mu_F \vdash^{\mathrm{ca}} M : (\nu'_E, \nu_E) \longrightarrow (\nu'_F, \nu_F)$ is provable and (ν'_E, ν_E) is a valid underlay interface, then*

$$([\![\mathsf{Link}\, \nu_E; M]\!] \cap \mu_F)\upharpoonright_{F^{\lightning}} \subseteq K_{\lightning}\, \nu_F.$$

Proof. The proof of this auxiliary soundness is almost the same as the proof of lemma G.6 for the durable program logic. We maintain the invariant that for any c, Δ, s, if $\langle c_0, \Delta_0, \epsilon\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c, \Delta, s\rangle$, then when $s\upharpoonright_{F^{\lightning}\,\&R_F} \in \mu_F$ there exists a current linearization $\rho_F \in \nu_F$ and the followings hold.

$$
\begin{aligned}
&s\upharpoonright_{F^{\lightning}} \overset{\lightning}{\dashrightarrow} \rho_F && \text{(G-Lin)} \\
&\wedge \forall \alpha. c[\alpha] \neq \mathsf{dead} \Rightarrow \\
&\quad \exists P_\alpha. (\Delta, s, \rho_F) \in P_\alpha \wedge (\mathsf{halt} \notin c \Rightarrow \mathsf{stable}(\mathcal{R}[\alpha], P_\alpha)) && \text{(G-Pa)} \\
&\quad\quad \wedge (c[\alpha] = \mathsf{idle} \Rightarrow P_\alpha \subseteq P[\alpha]) && \text{(G-Idle)} \\
&\quad\quad \wedge \big((\forall \alpha'. c[\alpha'] \in \{\mathsf{dead}, \mathsf{halt}\}) \Rightarrow \exists \alpha'. P_\alpha \subseteq Q_{\lightning}[\alpha']\big) && \text{(G-Crs)} \\
&\quad\quad \wedge \Big(c[\alpha] = \mathsf{halt} \wedge (\exists \alpha', C \in \mathsf{Com}. c[\alpha'] = C) \Rightarrow \exists \alpha'. P_\alpha \Rightarrow^{\mathrm{ca}}_{\lightning} Q_{\lightning}[\alpha']\Big) && \text{(G-Rec)} \\
&\quad\quad \wedge \left(\forall C \in \mathsf{Com}. c[\alpha] = C \Rightarrow \exists f \in F \cup R_F. \begin{pmatrix} \mathsf{safe}_\alpha \begin{pmatrix} \mathcal{R}[\alpha]^f, \mathcal{G}[\alpha]^f, \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, \\ P_\alpha, C, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f, Q_{\lightning}[\alpha]^f \end{pmatrix} \\ \wedge (f \in R_F \Rightarrow \forall \alpha' \neq \alpha. c(\alpha') \in \{\mathsf{dead}, \mathsf{halt}\}) \\ \wedge P_\alpha \Rightarrow^{\mathrm{ca}}_{\lightning} Q_{\lightning}[\alpha]^f \wedge f = \mathsf{last}(\pi_\alpha(s\upharpoonright_{F^{\lightning}\,\&R_F})) \end{pmatrix} \right) \\
&&& \text{(G-Exe)}
\end{aligned}
$$

Combined with lemma H.2 and the crash-aware linearizability's definition in §B, this invariant proves the auxiliary soundness by showing

$$\forall s \in ([\![\mathsf{Link}\, \nu_E; M]\!] \cap \mu_F). s\upharpoonright_{F^{\lightning}} \in K_{\lightning}\, \nu_F.$$

We prove the invariant by first inducting over the reduction $\langle c_0, \Delta_0, \epsilon\rangle \twoheadrightarrow^{M*}_{\nu_E} \langle c, \Delta, s\rangle$ and in the inductive step where we have

$$\langle c_0, \Delta_0, \epsilon\rangle \twoheadrightarrow^{M*}_{\nu_E} \langle c, \Delta, s\rangle \text{ and } \langle c, \Delta, s\rangle \twoheadrightarrow^{M}_{\nu_E} \langle c', \Delta', s'\rangle.$$

We then prove by a case analysis of the last reduction step. The major difference between this proof and the one in G is the case where the last reduction step is a crash. We only demonstrate the proof for the crash case here.

We have the induction hypothesis that there exists a current linearization $\rho_F \in \nu_F$ and the followings hold.

$$
\begin{aligned}
& s\restriction_{F\lightning} \overset{\lightning}{\dashrightarrow} \rho_F && \text{(IH-Lin)}\\
& \land \forall \alpha . c[\alpha] \neq \mathsf{dead} \Rightarrow \\
& \quad \exists P_\alpha . (\Delta, s, \rho_F) \in P_\alpha \land (\mathsf{halt} \notin c \Rightarrow \mathsf{stable}(\mathcal{R}[\alpha], P_\alpha)) && \text{(IH-Pa)}\\
& \qquad \land (c[\alpha] = \mathsf{idle} \Rightarrow P_\alpha \subseteq P[\alpha]) && \text{(IH-Idle)}\\
& \qquad \land \big((\forall \alpha' . c[\alpha'] \in \{\mathsf{dead}, \mathsf{halt}\}) \Rightarrow \exists \alpha' . P_\alpha \subseteq Q_{\lightning}[\alpha']\big) && \text{(IH-Crs)}\\
& \qquad \land \Big(c[\alpha] = \mathsf{halt} \land (\exists \alpha', C \in \mathsf{Com} . c[\alpha'] = C) \Rightarrow \exists \alpha' . P_\alpha \Rightarrow_{\lightning}^{\mathsf{ca}} Q_{\lightning}[\alpha']\Big) && \text{(IH-Rec)}\\
& \qquad \land \left(\forall C \in \mathsf{Com} . c[\alpha] = C \Rightarrow \exists f \in F \cup R_F . \left(\begin{array}{l} \mathsf{safe}_\alpha \left(\begin{array}{l} \mathcal{R}[\alpha]^f, \mathcal{G}[\alpha]^f, \mathsf{invoke}_\alpha(f) \circ P[\alpha]^f, \\ P_\alpha, C, \mathsf{returned}_\alpha(f) \circ Q[\alpha]^f, Q_{\lightning}[\alpha]^f \end{array} \right) \\ \land (f \in R_F \Rightarrow \forall \alpha' \neq \alpha . c(\alpha') \in \{\mathsf{dead}, \mathsf{halt}\}) \\ \land P_\alpha \Rightarrow_{\lightning}^{\mathsf{ca}} Q_{\lightning}[\alpha]^f \land f = \mathsf{last}(\pi_\alpha(s\restriction_{F\lightning\&R_F})) \end{array} \right) \right)\\
& && \text{(IH-Exe)}
\end{aligned}
$$

We also list some premise extracted from the judgement $\mu_F \vdash^{\mathsf{ca}} M : (\nu_E', \nu_E) \longrightarrow (\nu_F', \nu_F)$ that are used in the crash case's proof.

$$\forall \alpha \in A.\mathsf{ID}, \top \vDash_\alpha^{\mathsf{ca}} \{\mathsf{invoke}_\alpha(r) \circ P_r[\alpha]\} M[\alpha]^r \{\mathsf{returned}_\alpha(r) \circ Q_r[\alpha]\} \{Q_{\lightning}[\alpha]\} \qquad \text{(H-RQ)}$$

$$\forall \alpha . I[\alpha] \Rightarrow_{\lightning}^{\mathsf{ca}} Q_{\lightning}[\alpha] \qquad \text{(H-PC)}$$

We also have the crash-aware version of lemma G.5 as lemma H.5

Lemma H.5. *For any $\mathcal{R}, \mathcal{G}, P, s, Q, Q_{\lightning}$, if the quadruple $\mathcal{R}, \mathcal{G} \vDash_\alpha^{\mathsf{ca}} \{P\} s \{Q\} \{Q_{\lightning}\}$ is provable, then the followings are true.*

$$\mathsf{stable}(\mathcal{R}, P) \qquad P \Rightarrow_{\lightning}^{\mathsf{ca}} Q_{\lightning} \qquad Q_{\lightning} \Rightarrow_{\lightning}^{\mathsf{ca}} Q_{\lightning} \qquad \mathsf{safe}_\alpha^{\mathsf{ca}}(\mathcal{R}, \mathcal{G}, P, P, s, Q, Q_{\lightning})$$

∗ *Crash.* If the reduction is a crash, then

$$\forall \alpha \in s . c'[\alpha] = \mathsf{dead} \qquad \forall \alpha \in \Upsilon . \alpha \notin s \Rightarrow c'[\alpha] = \mathsf{halt} \qquad \Delta' = \Delta_0 \qquad s' = s \cdot \lightning .$$

We have the induction hypothesis that there exists $\rho_F \in \nu_F$, such that

$$s\restriction_{F\lightning} \overset{\lightning}{\dashrightarrow} \rho_F$$

We take $\rho_F' = \rho_F \cdot \lightning$. By definition, the induction hypothesis is equivalent to

$$\exists t_P \in (M_F^P)^* . s\restriction_{F\lightning} \cdot t_P \rightsquigarrow_{\dagger F\lightning}^{\lightning} \rho_F .$$

By the third rewrite rule of $- \rightsquigarrow_{\dagger F\lightning}^{\lightning} -$, we insert t_P from previous linearization before the crash and have

$$s'\restriction_{F\lightning} = s\restriction_{F\lightning} \cdot \lightning \rightsquigarrow_{\dagger F\lightning}^{\lightning} s\restriction_{F\lightning} \cdot t_P \cdot \lightning \rightsquigarrow_{\dagger F\lightning}^{\lightning} \rho_F \cdot \lightning = \rho_F'$$

which shows $s'\restriction_{F\lightning} \overset{\lightning}{\dashrightarrow} \rho_F'$ by definition and proves (G-Lin).

For any α, if $c'[\alpha] = \mathsf{dead}$, then we do not need to prove anything for it. If $c'[\alpha] = \mathsf{halt}$, then we know $\alpha \notin s$, which implies $c[\alpha] \in \{\mathsf{idle}, \mathsf{halt}\}$. For these two cases, we only need to prove (G-Pa) and (G-Crs). The branches (G-Idle), (G-Rec), and (G-Exe) are trivially true because their LHS of the implication is false.

+ $c[\alpha] = \mathsf{idle}$. By (IH-Pa) (IH-Idle),

$$(\Delta, s, \rho_F) \in P_\alpha \subseteq I[\alpha].$$

By (H-PC), we have

$$(\Delta, s, \rho_F) \in I[\alpha] \Rightarrow^{\mathsf{ca}}_{\lightning} Q_{\lightning}[\alpha].$$

We take $P'_\alpha = Q_{\lightning}[\alpha]$ and by definition, we have $(\Delta', s', \rho'_F) = (\Delta_0, s \cdot \lightning, \rho_F \cdot \lightning) \in P'_\alpha$ and proves (G-Pa), since the right conjunct of it is trivially true.
By reflexivity, $P'_\alpha = Q_{\lightning}[\alpha] \subseteq Q_{\lightning}[\alpha]$, (G-Crs) is true.

+ $c[\alpha] = \mathsf{halt}$. It can be further divided into following cases.
  - $\forall\alpha'.c[\alpha'] \in \{\mathsf{dead}, \mathsf{halt}\}$, i.e., recovery has not started yet. We take $P'_\alpha = Q_{\lightning}[\alpha]$. By (H-RQ) and lemma H.5, we have

$$\forall\alpha.Q_{\lightning}[\alpha] \Rightarrow^{\mathsf{ca}}_{\lightning} Q_{\lightning}[\alpha]$$

  Then by (IH-Pa), (IH-Crs), we have

$$(\Delta, s, \rho_F) \in P_\alpha \subseteq Q_{\lightning}[\alpha] \Rightarrow^{\mathsf{ca}}_{\lightning} Q_{\lightning}[\alpha] = P'_\alpha$$

  which indicates $(\Delta', s', \rho'_F) = (\Delta_0, s \cdot \lightning, \rho_F \cdot \lightning) \in P'_\alpha$ by definition and proves (G-Pa).
  By reflexivity, $P'_\alpha = Q_{\lightning}[\alpha] \subseteq Q_{\lightning}[\alpha]$, (G-Crs) is true.

  - $\exists\alpha', C \in \mathsf{Com}.c[\alpha'] = C$, i.e., recovery has started. By (IH-Pa) and (IH-Rec), we know

$$(\Delta, s, \rho_F) \in P_\alpha \Rightarrow^{\mathsf{ca}}_{\lightning} Q_{\lightning}[\alpha']$$

  for some α'. Therefore, we take $P'_\alpha = Q_{\lightning}[\alpha']$. And we have

$$(\Delta', s', \rho'_F) = (\Delta_0, s \cdot \lightning, \rho_F \cdot \lightning) \in P'_\alpha$$

  and prove (G-Pa). By reflexivity, $P'_\alpha = Q_{\lightning}[\alpha'] \subseteq Q_{\lightning}[\alpha']$, (G-Crs) is true.

  - $\exists\alpha'.c[\alpha'] = \mathsf{idle}$. This case is impossible, because idle and halt will not exist in c at the same time by the definition of the semantics.

□

I Applications of the Program Logic

In this section, we demonstrate the ability of our program logic from appendix G by presenting detailed proofs of the FLiT example and the file system example in §4.2. In I.1, we formalize and verify the FLiT memory cell implementation. And in I.3, we formalize and verify the file system example.

I.1 The FLiT Memory Cell

I.1.1 Implementation. The FLiT memory cell implementation has the signature:

$$\mathsf{FLiT} := \{\mathsf{load} : \mathsf{Val}, \mathsf{store} : \mathsf{Val} \to \mathbf{1}\}$$

It is durably linearizable to its specification ν_{flit}, which is the largest set satisfying the following property.

$$p \in \nu_{\mathsf{flit}} \iff \left(\begin{array}{c} p = \epsilon \lor \\ (p \sqsubseteq p' \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land p' \in \nu_{\mathsf{flit}}) \lor \\ (p \sqsubseteq p' \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \land p' \in \nu_{\mathsf{flit}} \land v = \mathsf{fstate}(p')) \end{array}\right)$$

$$\text{where } \mathsf{fstate}(p) = \begin{cases} v_0 & p = \epsilon \\ v & p = p' \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land \pi_\alpha(p') = p'' \cdot \mathsf{store}(v) \\ \mathsf{fstate}(p') & \text{otherwise, } p = p' \cdot \_ \end{cases}$$

It builds on a volatile and atomic counter Counter and a buffered and atomic memory cell BCell with the following signatures:

$$\mathsf{Counter} := \{\mathsf{inc} : \mathbf{1}, \mathsf{dec} : \mathbf{1}, \mathsf{get} : \mathbb{Z}\}$$
$$\mathsf{BCell} := \{\mathsf{load} : \mathsf{Val}, \mathsf{store} : \mathsf{Val} \rightarrow \mathbf{1}, \mathsf{flush} : \mathbf{1}\}$$

The counter has the specification ν_{counter} as the largest set satisfying the following property.

$$s \in \nu_{\mathsf{counter}} \iff \left(\begin{array}{c} s = \epsilon \lor (s = s' \cdot \lightning \land s' \in \nu_{\mathsf{counter}}) \lor \\ (s \sqsubseteq s' \cdot \boldsymbol{\alpha}{:}\mathsf{inc} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land s' \in \nu_{\mathsf{counter}}) \lor \\ (s \sqsubseteq s' \cdot \boldsymbol{\alpha}{:}\mathsf{dec} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land s' \in \nu_{\mathsf{counter}}) \lor \\ (s \sqsubseteq s' \cdot \boldsymbol{\alpha}{:}\mathsf{get} \cdot \boldsymbol{\alpha}{:}n \land s' \in \nu_{\mathsf{counter}} \land n = \mathsf{cstate}(s')) \end{array}\right)$$

$$\text{where } \mathsf{cstate}(s) = \begin{cases} 0 & s = \epsilon \lor s = s' \cdot \lightning \\ \mathsf{cstate}(s') + 1 & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land \pi_\alpha(s') = s'' \cdot \mathsf{inc} \\ \mathsf{cstate}(s') - 1 & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land \pi_\alpha(s') = s'' \cdot \mathsf{dec} \\ \mathsf{cstate}(s') & \text{otherwise, } s = s' \cdot \_ \end{cases}$$

The buffered memory cell has the specification ν_{bcell} as the largest set satisfying the following property.

$$s \in \nu_{\mathsf{bcell}} \iff \left(\begin{array}{c} s = \epsilon \lor (s = s' \cdot \lightning \land s' \in \nu_{\mathsf{bcell}}) \lor \\ (s \sqsubseteq s' \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land s' \in \nu_{\mathsf{bcell}}) \lor \\ (s \sqsubseteq s' \cdot \boldsymbol{\alpha}{:}\mathsf{flush} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land s' \in \nu_{\mathsf{bcell}}) \lor \\ (s \sqsubseteq s' \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \land s' \in \nu_{\mathsf{bcell}} \land v \in \mathsf{mstate}(s') \upharpoonright_2) \end{array}\right)$$

$$\text{where } (v_1, v_2) \in \mathsf{mstate}(s) \iff \left(\begin{array}{c} (v_1 = v_2 = v_0 \land s = \epsilon) \lor \\ (v_1 = v_2 = v \land s = s' \cdot \lightning \land (v, v') \in \mathsf{mstate}(s')) \lor \\ \left(\begin{array}{c} v_1 = v' \land v_2 = v \land s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land \\ \pi_\alpha(s') = s'' \cdot \mathsf{store}(v) \land (v', v'') \in \mathsf{mstate}(s') \end{array}\right) \lor \\ (v_1 = v_2 = v \land s = s' \cdot \mathsf{ok} \land \pi_\alpha(s') = s'' \cdot \mathsf{store}(v)) \lor \\ \left(\begin{array}{c} v_1 = v_2 = v \land s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land \\ \pi_\alpha(s') = s'' \cdot \mathsf{flush} \land (v', v) \in \mathsf{mstate}(s') \end{array}\right) \lor \\ (s = s' \cdot \boldsymbol{\alpha}{:}v_2 \land \pi_\alpha(s') = s'' \cdot \mathsf{load} \land (v_1, v_2) \in \mathsf{mstate}(s')) \lor \\ (s = s' \cdot e \land e \in \mathsf{inv} \land (v_1, v_2) \in \mathsf{mstate}(s')) \end{array}\right)$$

$$\text{and } \mathsf{mstate}(s) \upharpoonright_i = \{v_i \mid (v_1, v_2) \in \mathsf{mstate}(s)\}$$

Basically, the state function mstate computes all possibilities of the persisted content v_1 and the buffered content v_2.

- When crash happens, only the persisted content is preserved and is loaded into the buffered content.
- When storing a value to the cell, the value may be written to the buffered content only or be written to both the persisted and the buffered one.
- When flush happens, the persisted content gets synchronized with the buffered content.
- Any load to the cell gets the buffered content. Moreover, after a load returns, the buffered content is determined, which will eliminate any non-determinism brought up by unflushed stores before previous crashes.

Figure 14 shows the implementation of the FLiT memory cell. To make implementations and proofs more readable, for structure commands like if-statements and loop-statements, we do not unfold them into the encoding using $C_1 + C_2$ and C^* and instead write the code and do the proof in the high-level syntax.

```
M_flit :
Import M:BCell
Import C:Counter

load() {                          store(v) {
    v ← M.load();                     C.inc();
    if (C.get() != 0) {               M.store(v);
        M.flush();                    M.flush();
    }                                 C.dec();
    return v;                         return;
}                                 }
```

Fig. 14. FLiT Memory Cell Implementation

I.1.2 Proof. To prove this example, we need to add more structures to the possibility's definition. We use the possibility with the form of $\rho = (p, s_O)$, where $p \in P_{!\mathsf{flit}}$ and is accessed by $\mathsf{lin}(\rho)$, and s_O is the set of pending invocations. The set $P_{!\mathsf{flit}}$ is the largest set of atomically linearized traces.

$$p \in P_{!\mathsf{flit}} \iff \begin{pmatrix} p = \epsilon \vee \\ (p = p' \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge p' \in v_{\mathsf{flit}}) \vee \\ (p = p' \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \wedge p' \in v_{\mathsf{flit}} \wedge v = \mathsf{fstate}(p')) \end{pmatrix}$$

For this definition, we maintain two invariants that: $s\upharpoonright_{\mathsf{FLiT}}$ is linearizable to $p \cdot \langle s_O \rangle$ and $p \cdot \langle s_O \rangle \in v_{\mathsf{flit}}$, where $\langle s_O \rangle$ is any sequence of pending invocations in s_O. We use a ghost variable $B = \mathsf{List}\, s_O$ to store all buffered pending invocations in order, which implicitly implies that $\forall e \in B.e \in s_O$. As a result, the program state is now a quadruple (Δ, s, ρ, B). Using this definition of the configuration will not change the soundness of the logic.

There are three states of the system:

- The Flushed state. It means every stores have persisted in the NVM.

$$\mathsf{Flushed} \iff B = \epsilon$$

- The Unflushed state. It means there are buffered stores but the determinism of the buffered content has not been broken by crashes. However, there must be different possibilities in the persisted content. Buffered stores that are consistent with the current persisted content will directly get linearized.

$$\mathsf{Unflushed} \iff B \neq \epsilon \wedge \begin{pmatrix} \forall v_1, v_2 \in \mathsf{mstate}(s{\upharpoonright}_{M^{\lightning}}){\upharpoonright}_2 . v_1 = v_2 \wedge \\ \exists v_1, v_2 \in \mathsf{mstate}(s{\upharpoonright}_{M^{\lightning}}){\upharpoonright}_1 . v_1 \neq v_2 \end{pmatrix}$$

- The Unsynced state. It means there are buffered stores and we do not know which one is persisted and loaded into the volatile memory due to some crashes.

$$\mathsf{Unsynced} \iff B \neq \epsilon \wedge \exists v_1, v_2 \in \mathsf{mstate}(s{\upharpoonright}_{M^{\lightning}}){\upharpoonright}_2 . v_1 \neq v_2$$

The object invariant $I(s, \rho, B)$ is a conjunction of the following conditions. We maintain the invariant at any point of the program.

- The system is always in one of the three states.

$$\mathsf{Flushed} \vee \mathsf{Unflushed} \vee \mathsf{Unsynced}$$

- If the system is in Flushed state, then the memory cell's state is persistent and is the same as the overlay's state.

$$\mathsf{Flushed} \Rightarrow \forall (v_1, v_2) \in \mathsf{mstate}(s{\upharpoonright}_{M^{\lightning}}) . v_1 = v_2 = \mathsf{fstate}(\rho)$$

- If the system is in Unflushed state, then the memory cell's buffered value is the value of the latest buffered store and any persisted content corresponds to the current overlay's state or a buffered store.

$$\mathsf{Unflushed} \Rightarrow \begin{pmatrix} (\forall v \in \mathsf{mstate}(s{\upharpoonright}_{M^{\lightning}}){\upharpoonright}_2 . \mathsf{last}(B{\upharpoonright}_{\mathsf{store}}) = \mathsf{store}(v)) \wedge \\ (\forall v . (\mathsf{store}(v) \in B \vee \mathsf{fstate}(\rho) = v) \Leftrightarrow v \in \mathsf{mstate}(s{\upharpoonright}_{M^{\lightning}}){\upharpoonright}_1) \end{pmatrix}$$

- Any Unsynced state is caused by crashes from a Unflushed state. If the system is in Unsynced state, then any content in the memory cell corresponds to the current overlay's state or a buffered store.

$$\mathsf{Unsynced} \Rightarrow (\forall v . (\mathsf{store}(v) \in B \vee \mathsf{fstate}(\rho) = v) \Leftrightarrow (v, v) \in \mathsf{mstate}(s{\upharpoonright}_{M^{\lightning}}))$$

- The counter increment by any agent is always non-negative, which implies the counter value to be non-negative, and when the the system is in the Unflushed state, the counter value is non-zero.

$$\forall \alpha . \mathsf{cstate}(\pi_\alpha(s{\upharpoonright}_{C^{\lightning}})) \geq 0 \wedge (\mathsf{Unflushed} \Rightarrow \mathsf{cstate}(s{\upharpoonright}_{C^{\lightning}}) \neq 0)$$

Informal explanation of the load proof. Upon invocation of the load to the FLiT memory cell, it invokes the **load to the underlay** M, which can perform three different transitions according to the state.

load-f When the system is in the Flushed state, we directly linearize the load with the underlay's return value, because according to the invariant, we know the underlay's persisted value and buffered value is the overlay's linearized value. Therefore, the linearization of the current load is consistent with the linearized part.

load-uf When the system is in the Unflushed state, we know that v is the underlay's buffered value and according to the invariant, the last buffered store has the argument v. We then append the current load to B, waiting someone to help linearize the load and corresponding store before it to ensure consistency.

load-s When the system is in the Unsynced state, the current load will determine how do buffered operations from previous epochs linearize.

```
{$\mathsf{invoke}_\alpha(\mathsf{load}) \circ I$}
load(){
  {$I \land \boldsymbol{\alpha}{:}\mathsf{load} \in s_O \land (\mathsf{Flushed} \lor \mathsf{Unflushed} \lor \mathsf{Unsynced})$}
  $v \leftarrow M.\mathsf{load}();$   // load-f/load-uf/load-us
  $\left\{\begin{array}{c} I(s,\rho,B) \land \boldsymbol{\alpha}{:}\mathsf{load} \in s_O \land \\ \left(\left(\begin{array}{c}(v,v) \in \mathsf{mstate}(s\restriction_{M^{\lightning}}) \land v = \mathsf{fstate}(\rho) \land \mathsf{lin}(\rho') = \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \\ \land B' = B = \epsilon \land \mathsf{mstate}(s'\restriction_{M^{\lightning}}) = \mathsf{mstate}(s\restriction_{M^{\lightning}})\end{array}\right) \lor \\ \left(\begin{array}{c}\mathsf{last}(B\restriction_{\mathsf{store}}) = \mathsf{store}(v) \land \rho' = \rho \land \\ B' = B \cdot \boldsymbol{\alpha}{:}\mathsf{load} \land \mathsf{mstate}(s'\restriction_{M^{\lightning}}) = \mathsf{mstate}(s\restriction_{M^{\lightning}})\end{array}\right) \lor \\ \left(\begin{array}{c}\left(\begin{array}{c}(B = B_1 \cdot B_2 \land \mathsf{last}(B_1) = \mathsf{store}(v)) \lor \\ (\mathsf{fstate}(\rho) = v \land B_1 = \epsilon)\end{array}\right) \land (v,v) \in \mathsf{mstate}(s\restriction_{M^{\lightning}}) \land \\ \mathsf{lin}(\rho') = \mathsf{merge}(\mathsf{lin}(\rho), B_1) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \land \\ B' = \epsilon \land \mathsf{mstate}(s'\restriction_{M^{\lightning}}) = \{(v,v)\}\end{array}\right)\right)\end{array}\right\}$
  $\left\{I \land \left(\begin{array}{c}(\mathsf{Flushed} \land \mathsf{last}(\pi_\alpha(\rho)) = v) \lor \\ (\mathsf{Unflushed} \land (\exists B'.B' \cdot \boldsymbol{\alpha'}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \lor \mathsf{last}(\pi_\alpha(\rho)) = v))\end{array}\right)\right\}$
  $n \leftarrow C.\mathsf{get}();$
  $\left\{I \land \left(\begin{array}{c}(\mathsf{Flushed} \land (n \neq 0 \lor n = 0) \land \mathsf{last}(\pi_\alpha(\rho)) = v) \lor \\ (\mathsf{Unflushed} \land n \neq 0 \land (\exists B'.B' \cdot \boldsymbol{\alpha'}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \lor \mathsf{last}(\pi_\alpha(\rho)) = v))\end{array}\right)\right\}$
  $\left\{I \land \left(\begin{array}{c}(n = 0 \land \mathsf{last}(\pi_\alpha(\rho)) = v) \lor \\ (n \neq 0 \land (\exists B'.B' \cdot \boldsymbol{\alpha'}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \lor \mathsf{last}(\pi_\alpha(\rho)) = v))\end{array}\right) \land (\mathsf{Flushed} \lor \mathsf{Unflushed})\right\}$
  if($n \neq 0$){
    {$I \land (\mathsf{Flushed} \lor \mathsf{Unflushed}) \land (\exists B'.B' \cdot \boldsymbol{\alpha'}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \lor \mathsf{last}(\pi_\alpha(\rho)) = v)$}
    $M.\mathsf{flush}();$  // flush
    $\left\{I(s,\rho,B) \land \left(\begin{array}{c}(\exists B'.B' \cdot \boldsymbol{\alpha'}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \lor \mathsf{last}(\pi_\alpha(\rho)) = v) \land \\ \mathsf{lin}(\rho') = \mathsf{merge}(\mathsf{lin}(\rho), B) \land B' = \epsilon \land \mathsf{last}(\pi_\alpha(\rho')) = v \land \\ s' = s \cdot \boldsymbol{\alpha}{:}\mathsf{flush} \cdot \boldsymbol{\alpha}{:}\mathsf{ok}\end{array}\right)\right\}$
    {$I \land \mathsf{last}(\pi_\alpha(\rho)) = v$}
  }
  {$I \land \mathsf{last}(\pi_\alpha(\rho)) = v$}
  ret $v$
}
{$\mathsf{returned}_\alpha(\mathsf{load}) \circ I$}{$\top$}
```

Fig. 15. Proof of the FLiT Memory Cell: load

- The underlay's load may return the value from some buffered store, and we take all buffered operation no-later than this store as B_1. In this case, all operations in B_1 are persisted in order in the underlay and are visible to all future operations. Therefore, we can safely linearized operations in B_1 by the merge function, followed by the current overlay's load. The merge function, merge : $P_{!\mathsf{flit}} \times s_O \longrightarrow P_{!\mathsf{flit}}$, linearizes buffered invocations in B in order after the existing linearized possibility.

$$\mathsf{merge}(p, B) = \begin{cases} p & B = \epsilon \\ \mathsf{merge}(p', B') & B = \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot B' \land p' = p \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \\ \mathsf{merge}(p', B') & B = \boldsymbol{\alpha}{:}\mathsf{load} \cdot B' \land v = \mathsf{fstate}(p) \land p' = p \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \end{cases}$$

We forget all remaining buffered operation by setting B to empty, because they are no longer visible.

- Or the underlay's load may return the persisted value before all buffered operations, i.e., the same value as the linearized value. Then we forget all buffered operations and linearize the current load after the linearized part.

The post-condition is stabilized to

$$I \wedge \left(\begin{array}{c} (\mathsf{Flushed} \wedge \mathsf{last}(\pi_\alpha(\rho)) = v) \vee \\ \left(\mathsf{Unflushed} \wedge \left(\exists B'.B' \cdot \boldsymbol{\alpha'}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \vee \mathsf{last}(\pi_\alpha(\rho)) = v\right)\right) \end{array} \right).$$

The transition load-f and load-us results in the Flushed branch, and load-uf results in the Unflushed branch, where someone may help the current thread linearize its buffered load in B. For simplicity, we use variable v in assertions as the program variable v's value.

Then, it **gets the current counter value**. In the Flushed branch, the returned counter value n can be any non-negative number. In the Unflushed branch, by invariant I, the counter value must be non-negative. We then stabilize this post-condition into

$$I \wedge \left(\begin{array}{c} (n = 0 \wedge \mathsf{last}(\pi_\alpha(\rho)) = v) \vee \\ \left(n \neq 0 \wedge \left(\exists B'.B' \cdot \boldsymbol{\alpha'}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \sqsubseteq B \vee \mathsf{last}(\pi_\alpha(\rho)) = v\right)\right) \end{array} \right) \wedge (\mathsf{Flushed} \vee \mathsf{Unflushed})$$

by extracting Flushed ∨ Unflushed outside, because they can change into each other under the rely. We keep the information that when $n = 0$, the current load is linearized, and otherwise, the current load may remains in B or is linearized.

Into the if branch, we know $n \neq 0$ and therefore, the current load may either be in the buffered list B or already linearized. When it **flushes**, we linearize every thing in the buffered list (which may be empty in the Flushed case), because after the flush, the underlay's persisted value will be consistent with the overlay's value after linearizing operations in B according to the invariant. As a result, after this flush and linearizations, the current load is definitely linearized.

Informal explanation of the store proof. In the load proof, we only use the invariant to get information about the counter, but in the store proof, it is a major challenge to maintain the invariant for the counter. When **increasing the counter** at the beginning of the store, we know from the invariant that the counter increment ($\mathrm{cstate}(\pi_\alpha(s\restriction_{C^{\lightning}}))$) by the current thread α is non-negative and after the increment, it will be strictly larger than 0 because no one else can change $\mathrm{cstate}(\pi_\alpha(s\restriction_{C^{\lightning}}))$. This step preserves the invariant, because it does not decrease the counter value.

Then it **performs a store**, which can have three different transitions under four different cases.

store-ff If we are in the Flushed state and we are storing a value that is exactly the same as the current linearized value. Then we can directly linearize this store because the underlay is the same no matter this store is persisted or not. Another choice is to put it in the buffered list and let someone else linearize it, but this makes defining the Unflushed states difficult. This step preserves the invariant because we are still in the Flushed state.

store-fu If we are in the Flushed state and we are storing a value that is different from the current linearized value. Then we need to put it in the buffered list, because it may not be persisted and is unsafe to be linearized. This step preserves the invariant, because although we changed into the Unflushed state, we have ensured the local increment to the counter is positive, which means the counter value is positive.

store-uu (unflushed) If we are in the Unflushed state, we append the store to the buffered list B and it is still in the Unflushed state. This step preserves the invariant for the same reason as the store-fu transition.

```
$\{\mathsf{invoke}_\alpha(\mathsf{store}(v)) \circ I\}$
$\mathsf{store}(v)\{$
    $\{I \wedge \boldsymbol{\alpha}{:}\mathsf{store}(v) \in s_O \wedge 0 \leq \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}}))\}$
    $C.\mathsf{inc}();$
    $\{I \wedge \boldsymbol{\alpha}{:}\mathsf{store}(v) \in s_O \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \wedge (\mathsf{Flushed} \vee \mathsf{Unflushed} \vee \mathsf{Unsynced})\}$
    $M.\mathsf{store}(v);$   // store-ff/store-fu/store-uu
    $\left\{\begin{array}{c} I \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \wedge s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge \\ \left(\begin{array}{c} (\mathsf{Flushed}(s, B) \wedge \mathsf{fstate}(\rho) = v \wedge B' = \epsilon \wedge \mathsf{lin}(\rho') = \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok}) \vee \\ (\mathsf{Flushed}(s, B) \wedge \mathsf{fstate}(\rho) \neq v \wedge B' = \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \epsilon \wedge \rho' = \rho) \vee \\ ((\mathsf{Unflushed}(s, B) \vee \mathsf{Unsynced}(s, B)) \wedge B' = B \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \wedge \rho' = \rho) \end{array}\right) \end{array}\right\}$
    $\left\{I \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \wedge \left(\begin{array}{c} (\mathsf{Flushed} \wedge \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok}) \vee \\ (\mathsf{Unflushed} \wedge ((\boldsymbol{\alpha}{:}\mathsf{store}(v) \in B \vee \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok})) \end{array}\right)\right\}$
    $M.\mathsf{flush}();$   // flush
    $\left\{\begin{array}{c} I \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \leq \mathsf{cstate}(s\upharpoonright_{C^{\lightning}}) \wedge \\ \left(\begin{array}{c} (\mathsf{Flushed}(s, B) \wedge \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok} \wedge (\rho', B') = (\rho, B)) \vee \\ \left(\begin{array}{l} \mathsf{Unflushed}(s, B) \wedge \mathsf{lin}(\rho') = \mathsf{merge}(\mathsf{lin}(\rho), B) \wedge \\ B' = \epsilon \wedge ((\boldsymbol{\alpha}{:}\mathsf{store}(v) \in B \vee \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok}) \end{array}\right) \end{array}\right) \end{array}\right\}$
    $\left\{I \wedge \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok} \wedge \left(\begin{array}{c} (\mathsf{Flushed} \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \leq \mathsf{cstate}(s\upharpoonright_{C^{\lightning}})) \vee \\ (\mathsf{Unflushed} \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) < \mathsf{cstate}(s\upharpoonright_{C^{\lightning}})) \end{array}\right)\right\}$
    $C.\mathsf{dec}();$
    $\left\{I \wedge \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok} \wedge \left(\begin{array}{c} (\mathsf{Flushed} \wedge 0 \leq \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \leq \mathsf{cstate}(s\upharpoonright_{C^{\lightning}})) \vee \\ (\mathsf{Unflushed} \wedge 0 \leq \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) < \mathsf{cstate}(s\upharpoonright_{C^{\lightning}})) \end{array}\right)\right\}$
    $\mathsf{ret\ ok}$
$\}$
$\{\mathsf{returned}_\alpha(\mathsf{store}(v)) \circ I\}\{\top\}$
```

Fig. 16. Proof of the FLiT Memory Cell: store

store-uu (unsynced) If we are in the Unsynced state, we append the store to the buffered list B and step into the Unflushed state. This transition is valid because an underlay store synchronizes all possible buffered values to be one value after the crash breaks such synchronization. This step preserves the invariant for the same reason as the store-fu transition.

We stabilize the post-condition into

$$I \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \wedge \left(\begin{array}{c} (\mathsf{Flushed} \wedge \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok}) \vee \\ (\mathsf{Unflushed} \wedge ((\boldsymbol{\alpha}{:}\mathsf{store}(v) \in B \vee \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok})) \end{array}\right)$$

The transition store-ff results into the Flushed branch and other two transitions results into the Unflushed branch. Other threads may change Flushed into Unflushed but the current store will remain linearized or change Unflushed into Flushed and guarantee the current store will be linearized since it is in B.

Then it **flushes**. By definition, the counter value ($\mathsf{cstate}(s\upharpoonright_{C^{\lightning}})$) is no less than the local increment ($\mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}}))$). In the Flushed case, the flush operation has no effect. In the Unflushed case, the flush operation will linearize all buffered operations in B, which includes the current store if it has not been linearized. Therefore, it ensures the current store is linearized and results in the

Flushed in the stabilized post-condition:

$$I \wedge \mathsf{last}(\pi_\alpha(\rho)) = \mathsf{ok} \wedge \begin{pmatrix} (\mathsf{Flushed} \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\restriction_{C^{\lightning}})) \le \mathsf{cstate}(s\restriction_{C^{\lightning}}))\vee \\ (\mathsf{Unflushed} \wedge 0 < \mathsf{cstate}(\pi_\alpha(s\restriction_{C^{\lightning}})) < \mathsf{cstate}(s\restriction_{C^{\lightning}})) \end{pmatrix}$$

This step preserves the invariant because it changes into the Flushed state. However, other threads may change it into the Unflushed state. Luckily, when they make the store-fu or store-uu transition, they will guarantee that their local counter increment is non-zero, which means the counter value is strictly larger than the local increment of the current thread. This fact is important when proving the next step.

Then it **decreases the counter**. This step does not change the system state, but we need to verify that it preserves the invariant. In either branch, the local increment ($\mathsf{cstate}(\pi_\alpha(s\restriction_{C^{\lightning}}))$) now is non-negative. If the system is still Flushed, the invariant is preserved. If the system is Unflushed, since we have maintained that $\mathsf{cstate}(\pi_\alpha(s\restriction_{C^{\lightning}})) < \mathsf{cstate}(s\restriction_{C^{\lightning}})$, we know the counter value is still non-zero and the invariant is preserved.

I.1.3 Summary & More Proof Details. To summarize, we define the invariant I of the object as the following, which is satisfied at any point of the program and is the crash-invariant as well.

$$I(s,\rho,B) \iff \begin{pmatrix} (\mathsf{Flushed} \vee \mathsf{Unflushed} \vee \mathsf{Unsynced})\wedge \\ (\mathsf{Flushed} \Rightarrow \forall (v_1,v_2) \in \mathsf{mstate}(s\restriction_{M^{\lightning}}).v_1 = v_2 = \mathsf{fstate}(\rho))\wedge \\ \left(\mathsf{Unflushed} \Rightarrow \begin{pmatrix} (\forall v \in \mathsf{mstate}(s\restriction_{M^{\lightning}})\restriction_2.\mathsf{last}(B\restriction_{\mathsf{store}}) = \mathsf{store}(v))\wedge \\ (\forall v.(\mathsf{store}(v) \in B \vee \mathsf{fstate}(\rho) = v) \Leftrightarrow v \in \mathsf{mstate}(s\restriction_{M^{\lightning}})\restriction_1) \end{pmatrix}\right) \wedge \\ (\mathsf{Unsynced} \Rightarrow (\forall v.(\mathsf{store}(v) \in B \vee \mathsf{fstate}(\rho) = v) \Leftrightarrow (v,v) \in \mathsf{mstate}(s\restriction_{M^{\lightning}})))\wedge \\ \forall \alpha.\mathsf{cstate}(\pi_\alpha(s\restriction_{C^{\lightning}})) \ge 0 \wedge (\mathsf{Unflushed} \Rightarrow \mathsf{cstate}(s\restriction_{C^{\lightning}}) \ne 0) \end{pmatrix}$$

And there are three states of the object with transitions illustrated in Fig. 17.

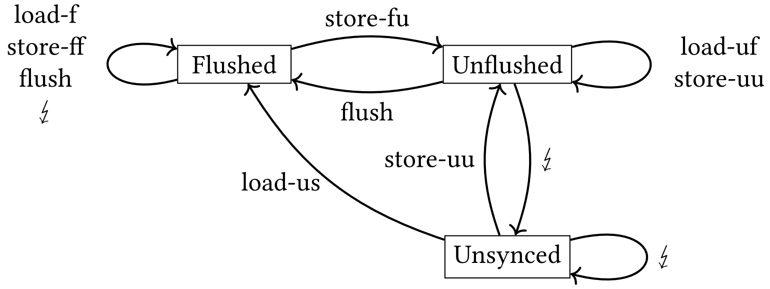


Fig. 17. STS for FLiT Memory Cell

Notice that this invariant I is idempotent under the crash-into relation, i.e., $I \Rightarrow_{\lightning} I$.

- Firstly, as the state transition system in Fig. 17 and the invariant definition indicates, the crash will change the system into the Flushed or Unsynced state, which is still in the invariant.
- Secondly, a crash does not invalidate any conjuncts in the invariant. It is worth mentioning that when a crash changes Unflushed in Unsynced, the second branch of the Unflushed branch, which is about the persisted value of M, is necessary for establishing the invariant in the Unsynced branch.

This object does not need a recovery program, so we set the body of r to be ret ok, which does nothing and returns immediately. And it is obvious that the following quadruple holds.

$$\mathsf{ID}, \top \vDash_\alpha \{\mathsf{invoke}_\alpha(r) \circ I\}\mathsf{ret\ ok}\{\mathsf{returned}_\alpha(r) \circ I\}\{I\}$$

The pre-/post-conditions, I, of this recovery program obviously connect to the crash invariant and the object invariant, which are also I.

We have proved $\mathcal{R}[\alpha], \mathcal{G}[\alpha], I[\alpha] \vDash_\alpha M_F[A]$ in the previous section. And it is easy to check that we have established all conditions (except the stability) in Object Impl for the judgement

$$P_{\dagger(\mathsf{FLiT} \circlearrowleft R_\varnothing)} \vdash M_{\mathsf{FLiT}} : (\nu'_{\mathsf{BCell}} \boxtimes \nu'_{\mathsf{Counter}}, \nu_{\mathsf{BCell}} \otimes \nu_{\mathsf{Counter}}) \to \langle \nu'_{\mathsf{FLiT}}, \nu_{\mathsf{FLiT}} \rangle$$

We then define the guarantee relations of each transitions and they form the rely relation. It is easy to check that all stabilized assertions are stable w.r.t. the rely relation. And this finishes the proof of the FLiT memory cell.

$$(s, \rho, B)\mathcal{G}_{\text{load-f}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v.\mathsf{Flushed}(s, B) \land (v, v) \in \mathsf{mstate}(s\upharpoonright_{M^{\lightning}}) \land v = \mathsf{fstate}(\rho) \land \\ \mathsf{lin}(\rho') = \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \land B' = \epsilon \land \\ s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{load-uf}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v.\mathsf{Unflushed}(s, B) \land \mathsf{last}(B\upharpoonright_{\mathsf{store}}) = \mathsf{store}(v) \land \\ B' = B \cdot \boldsymbol{\alpha}{:}\mathsf{load} \land \rho' = \rho \land \\ s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{load-us}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v, B_1, B_2.\mathsf{Unsynced}(s, B) \land (v, v) \in \mathsf{mstate}(s\upharpoonright_{M^{\lightning}}) \land \\ \begin{pmatrix} (B = B_1 \cdot B_2 \land \mathsf{last}(B_1) = \mathsf{store}(v)) \\ \lor (\mathsf{fstate}(\rho) = v \land B_1 = \epsilon) \end{pmatrix} \land \\ \mathsf{lin}(\rho') = \mathsf{merge}(\mathsf{lin}(\rho), B_1) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \land B' = \epsilon \land \\ s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{store-ff}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v.\mathsf{Flushed}(s, B) \land 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \land \mathsf{fstate}(\rho) = v \land \\ \mathsf{lin}(\rho') = \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land B' = \epsilon \\ s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{store-fu}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v.\mathsf{Flushed}(s, B) \land 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \land \mathsf{fstate}(\rho) \neq v \land \\ B' = \boldsymbol{\alpha}{:}\mathsf{store}(v) \cdot \epsilon \land \rho' = \rho \\ s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{store-uu}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v.(\mathsf{Unflushed}(s, B) \lor \mathsf{Unsynced}(s, B)) \land \\ 0 < \mathsf{cstate}(\pi_\alpha(s\upharpoonright_{C^{\lightning}})) \land B' = B \cdot \boldsymbol{\alpha}{:}\mathsf{store}(v) \land \\ \rho' = \rho \land s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{store}(v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{flush}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \mathsf{lin}(\rho') = \mathsf{merge}(\mathsf{lin}(\rho), B) \land B' = \epsilon \land \\ s' = s \cdot \boldsymbol{\alpha}{:}\mathsf{flush} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{inc}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \rho' = \rho \land B' = B \land s' = s \cdot \boldsymbol{\alpha}{:}\mathsf{inc} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land \\ \mathsf{cstate}(\pi_\alpha(s' \upharpoonleft_{C^{\lightning}})) = \mathsf{cstate}(\pi_\alpha(s \upharpoonleft_{C^{\lightning}})) + 1 \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{dec}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \rho' = \rho \land B' = B \land s' = s \cdot \boldsymbol{\alpha}{:}\mathsf{dec} \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \land \\ \mathsf{cstate}(\pi_\alpha(s' \upharpoonleft_{C^{\lightning}})) = \mathsf{cstate}(\pi_\alpha(s \upharpoonleft_{C^{\lightning}})) - 1 \end{pmatrix}$$

$$(s, \rho, B)\mathcal{G}_{\text{id}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \rho' = \rho \land B' = B \land \\ \mathsf{mstate}(\pi_\alpha(s' \upharpoonleft_{M^{\lightning}})) = \mathsf{mstate}(\pi_\alpha(s \upharpoonleft_{M^{\lightning}})) \land \\ \mathsf{cstate}(\pi_\alpha(s' \upharpoonleft_{C^{\lightning}})) = \mathsf{cstate}(\pi_\alpha(s \upharpoonleft_{C^{\lightning}})) \end{pmatrix}$$

$$\mathcal{R}[\alpha] \triangleq \bigcup_{\alpha' \in \Upsilon, \alpha' \neq \alpha} \begin{pmatrix} \mathcal{G}_{\text{load-f}}[\alpha'] \cup \mathcal{G}_{\text{load-uf}}[\alpha'] \cup \mathcal{G}_{\text{load-us}}[\alpha'] \cup \mathcal{G}_{\text{store-ff}}[\alpha'] \cup \mathcal{G}_{\text{store-fu}}[\alpha'] \cup \\ \mathcal{G}_{\text{store-uu}}[\alpha'] \cup \mathcal{G}_{\text{flush}}[\alpha'] \cup \mathcal{G}_{\text{inc}}[\alpha'] \cup \mathcal{G}_{\text{dec}}[\alpha'] \cup \mathcal{G}_{\text{id}}[\alpha'] \cup \\ \mathsf{invoke}_{\alpha'}(-) \cup \mathsf{return}_{\alpha'}(-) \end{pmatrix}$$

I.2 The Interval-Sequential Write-Snapshot Object

With the FLiT memory cell's correctness established in appendix I.1 and the FLiT correctness theorem (proposition 1.1), we can prove any object implement to be durably linearizable as long as we can prove it satisfying linearizability defined in [31]. For example, the interval-sequential write-snapshot object in [7] can be implemented as a durably linearizable object using the FLiT cell's read and write instead of the usual atomic memory cell. In this section, we prove the interval-linearizability of the one-shot write-snapshot object in [7] using the program logic in [31].

I.2.1 Specification & Implementation. The write-snapshot object has the signature below.

$$\mathsf{Snapshot} := \{\mathsf{write\_snapshot} : \mathsf{Val} \to 2^{\mathsf{Val}}\}$$

The operation write_snapshot writes the current value to the memory and returns a set of values that have been written to the object before.

Its interval-sequential specification $\nu_{\text{write\_snapshot}}$ is the largest set of traces satisfying the following

$$s \in \nu_{\text{write\_snapshot}} \iff \begin{pmatrix} s = \epsilon \\ \lor \left(s = s' \cdot \boldsymbol{\alpha}{:}V \land V = \mathsf{snpstate}(s') \land s' \in \nu_{\text{write\_snapshot}}\right) \\ \lor \left(s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{write\_snapshot}(v) \land \alpha \notin s' \land s' \in \nu_{\text{write\_snapshot}}\right) \end{pmatrix}$$

where the state of the write-snap shot object snpstate(s) is defined as the set of values given by all invocations before a point.

$$\mathsf{snpstate}(s) := \begin{cases} \{\,\} & \text{if } s = \epsilon \\ \mathsf{snpstate}(s') \cup \{v\} & \text{if } s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{write\_snapshot}(v) \\ \mathsf{snpstate}(s') & \text{otherwise, } s = s' \cdot \alpha : \_ \end{cases}$$

The specification clarifies two key points of the object:

- Firstly, the object is not atomic-sequential, which means the linearized specification does not guarantee the response happens immediately after the corresponding invocation. The response will take into account all invocations linearized before itself.

- Secondly, the object is one-shot. Each agent (thread) can only invoke $\nu_{\text{write\_snapshot}}$ once. We use the client specification $\mu_{\text{write\_snapshot}}$ to pose this one-shot requirement on clients.

$$s \in \mu_{\text{write\_snapshot}} \iff \begin{pmatrix} \forall s', \alpha, v. s' \cdot \boldsymbol{\alpha}{:}\text{write\_snapshot}(v) \sqsubseteq s \Rightarrow \\ (\forall v. \boldsymbol{\alpha}{:}\text{write\_snapshot}(v') \notin s' \wedge \alpha \in S) \end{pmatrix}$$

As [7], we consider the one-shot write-snapshot algorithm for solving problems with only finite participating agents. Therefore, we require the client to take a finite subset S of all agents Υ, and use the client specification to require all invocations be made in these threads.

The implementation of the write-snapshot object is shown in figure 18. With a total number of $|S|$ threads where the write-snapshot object will be used, each thread is assigned a FLiT memory cell (with initial value $\perp$) to store the value it writes to the object. Since this is a one-shot object, we know that each memory cell is written at most once. After written to the cell, the algorithm repeatedly takes a snapshot of all values written to the object. It returns only when two consecutive snapshot converges, which guarantees the snapshot to be correct, meaning the snapshot did occur at a certain moment. This is non-trivial. For example, when taking a snapshot, if two writes to a visited cell and un-visited cell occurs, only the second one will be captured in the snapshot, which makes the snapshot invalid because in any possible snapshot, the second value should be recorded if the first value is not recorded.

```
$M_{\text{write\_snapshot}}$:
Import M:$\otimes_{i \in [|S|]}$FLiT$_i$

write_snapshot(int v) {
  M[$\alpha$].write(v);
  old ← {⊥}; new ← ∅; i ← 1;
  while (i ≤ $|S|$) {
    v ← M[$\alpha_i$].read();
    new ← new ∪ {v};
    i ← i+1
  }
  while (new ≠ old) {
    old ← new; new ← ∅; i ← 1;
    while (i ≤ $|S|$) {
      v ← M[$\alpha_i$].read();
      new ← new ∪ {v};
      i ← i+1
    }
  };
  return new\{⊥}
}
```

Fig. 18. One-Shot Write-Snapshot Implementation

I.2.2 Proof. Like what we did in the FLiT example, we use the possibility with the form of $\rho = (p, s_O)$, where p is the linearized trace with $p \in \nu_{\text{write\_snapshot}}$ (and is accessed through $\text{lin}(\rho)$) and s_O are pending invocations. Notice that there can exist pending invocations in p as well, since we are proving an interval-sequential object which allows an interval without the response, but these invocations will have impact on linearized responses in the future.

We do not use any other ghost variables in this proof, and the program configuration is a triple (Δ, s, ρ). We need to maintain two invariants that: $s\restriction_{\text{Snapshot}}$ is linearizable to $p \cdot \langle s_O \rangle$ and $p \cdot \langle s_O \rangle \in \nu_{\text{write\_snapshot}}$.

For this concurrent object, we maintain the following object invariant at any point of the program execution. It simply ensures the the order of writes in s is the same as the order of write_snapshot in the linearized trace.

$$I(\Delta, s, \rho) \iff s\restriction_{\mathsf{write}} \sim \mathsf{lin}(\rho)\restriction_{\mathsf{write\_snapshot}}$$

$$\text{where } s_1 \sim s_2 \iff \left(\begin{array}{c} (s_1 = s_2 = \epsilon) \vee \\ \exists \alpha, v, s_1', s_2'.\ \left(\begin{array}{c} s_1' \sim s_2' \wedge s_1 = s_1' \cdot \boldsymbol{\alpha}{:}\mathsf{write}(v) \\ \wedge s_2 = s_2' \cdot \boldsymbol{\alpha}{:}\mathsf{write\_snapshot}(v) \end{array} \right) \end{array} \right)$$

For the three while loop, we introduce the following two loop invariant.

$$I_o[i, \mathsf{old}, p](\Delta, s, \rho) \iff p \sqsubseteq \mathsf{lin}(\rho) \wedge \mathsf{old}\backslash\{\bot\} \subseteq \bigcup_{j=1}^{i-1} \mathsf{snpstate}(\pi_{\alpha_j}(p)) \wedge i \leq |S| + 1$$

$$I_n[i, \mathsf{new}, p](\Delta, s, \rho) \iff \left(\begin{array}{c} \displaystyle\bigcup_{j=1}^{i-1} \mathsf{snpstate}(\pi_{\alpha_j}(p)) \subseteq \mathsf{new}\backslash\{\bot\} \wedge i \leq |S| + 1 \\ \wedge (\forall \alpha : \mathsf{write\_snapshot}(v) \in p \Rightarrow \alpha : M[\alpha].\mathsf{write}(v) \in s) \end{array} \right)$$

- The loop invariant I_o guarantees there is always a prefix p of the linearized possibility such that each write in the old snapshot is captured by p. The invariant I_o assets that the old snapshot is a lower bound of a linearized trace's snapshot state at a certain moment.
- The loop invariant I_n guarantees that any write in the prefix p extracted from I_o is always in the new snapshot. The invariant I_n assets that the new snapshot is an upper bound of the linearized trace p's snapshot state.
- Both invariant takes a parameter i, which is the loop variable. We will only consider the first i threads operations in the trace and maintain the invariant. When the loop terminates, i becomes $|S| + 1$, and the invariant will be true under the consideration of all threads, which means it is true for the complete trace.

When the outer loop terminates, we know $\mathsf{old}\backslash\{\bot\} = \mathsf{snpstate}(p) = \mathsf{new}\backslash\{\bot\}$, and we can linearize the response at the end of p since $\mathsf{new}\backslash\{\bot\}$ is valid snapshot at that moment.

```
    $\{\mathsf{invoke}_\alpha(\mathsf{write\_snapshot}) \circ I\}$
1:  write_snapshot($v$){
      $\{I \land \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \in s_O\}$
2:    $M[\alpha]$.write($v$);   // write
      $\{I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v)\}$
3:    old $\leftarrow \{\bot\}$; new $\leftarrow \emptyset$; $i \leftarrow 1$;
      $\{I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \land I_o[|S|+1, \mathsf{old}, \epsilon] \land I_n[i, \mathsf{new}, \epsilon] \land \exists q.\, I_o[i, \mathsf{new}, q]\}$
      $\{I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \land \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[i, \mathsf{new}, p] \land \exists q.\, I_o[i, \mathsf{new}, q]\}$
4:    while($i \le |S|$){
5:      $v \leftarrow M[\alpha_i]$.read();
6:      new $\leftarrow$ new $\cup \{v\}$; $i \leftarrow i + 1$
7:    }
      $\left\{\begin{matrix} I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \land \\ \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[|S|+1, \mathsf{new}, p] \land \exists q.\, I_o[|S|+1, \mathsf{new}, q] \end{matrix}\right\}$
8:    while(new $\neq$ old){
        $\{I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \land \exists q.\, I_o[|S|+1, \mathsf{new}, q]\}$
9:      old $\leftarrow$ new; new $\leftarrow \emptyset$; $i \leftarrow 1$;
        $\{I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \land \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[i, \mathsf{new}, p] \land \exists q.\, I_o[i, \mathsf{new}, q]\}$
10:     while($i \le |S|$){
11:       $v \leftarrow M[\alpha_i]$.read();
12:       new $\leftarrow$ new $\cup \{v\}$; $i \leftarrow i + 1$
13:     }
        $\left\{\begin{matrix} I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \land \\ \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[|S|+1, \mathsf{new}, p] \land \exists q.\, I_o[|S|+1, \mathsf{new}, q] \end{matrix}\right\}$
14:   }
      $\{I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \land \exists p.\, I_o[|S|+1, \mathsf{new}, p] \land I_n[|S|+1, \mathsf{new}, p]\}$
      $\{I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v) \land \exists p \sqsubseteq \mathsf{lin}(\rho, L).\, \mathsf{snpstate}(p) = \mathsf{new} \backslash \{\bot\}\}$
      $\{I \land \mathsf{last}(\pi_\alpha(\rho)) = \boldsymbol{\alpha}\mathbf{:}\mathsf{new} \backslash \{\bot\}\}$ // snapshot
15:   ret new$\backslash\{\bot\}$
16: }
    $\{\mathsf{returned}_\alpha(\mathsf{write\_snapshot}) \circ I\}$
```

Fig. 19. Proof of Write-Snapshot

Figure 19 shows the proof outline for the write-snapshot object. At line 2, when writing to the FLiT cell, we linearize the corresponding write_snapshot invocation to the linearized trace so that the object invariant can be maintained. This operation satisfies the following guarantee $\mathcal{G}_{\mathsf{write}}$.

$$(\Delta, s, \rho)\mathcal{G}_{\mathsf{write}}[\alpha](\Delta', s', \rho') \iff \begin{pmatrix} s' = s \cdot \alpha : M[\alpha]\mathsf{write}(v) \cdot \alpha : \mathsf{ok} \\ \land s'_O = s_O \backslash \{\boldsymbol{\alpha}\mathbf{:}\mathsf{write\_snapshot}(v)\} \\ \land \mathsf{lin}(\rho') = \mathsf{lin}(\rho) \cdot \alpha : \mathsf{write\_snapshot}(v) \end{pmatrix}$$

Then, we initial the old snapshot old, new snapshot new, and the loop variable i. Since old contains only the initial value, it is a lower bound of any trace's snapshot state, and therefore we can take $p = \epsilon$ and $I[|S| + 1, \mathsf{old}, \epsilon]$ is true. Moreover, since ϵ produce empty snapshot state, any set is an upper bound of it and thus $I_n[i, \mathsf{new}, \epsilon]$ is true for any i.

The loop from line 4 to line 6 is the process of taking the snapshot. We may split the proof for this loop into two parts,

- one for constructing I_o, the lower bound invariant for the next snapshot loop;
- one for constructing I_n, the upper bound invariant for the current snapshot loop.

Their proof are independent, and since there are no control flow can break out of the loop, we may present their proofs separately for a clearer demonstration.

Construct Lower Bound. In figure 20, we use $I \wedge \exists p.\, I_o[i, \mathsf{new}, p]$ as the loop invariant. In each iteration, the read operation does not change the state of the memory cell and the write-snapshot object and there are two cases for the read value.

- If the value is $\bot$, meaning thread α_i has not written to the cell and new is still the lower bound, i.e., $I_o[i+1, \mathsf{new} \cup \{\bot\}, p]$ is true.
- If the value is not $\bot$, then α_i has written to its cell, and by the object invariant I, its write_snapshot(v) invocation has been linearized. Therefore, there exists $q \sqsubseteq \mathsf{lin}(\rho)$ which includes this invocation and we can take the longest one of p and q. Since this object is one-shot, values written to any cell cannot be overwrite when we consider more future events by extending p to $\max\{p, q\}$, and any value in new still appears in $\bigcup_{j=1}^{i-1} \mathsf{snpstate}(\pi_{\alpha_j}(\max\{p, q\}))$. As a result, $\mathsf{new} \cup \{v\}$ is a lower bound considering only the first $i-1$ threads, i.e., $I_o[i+1, \mathsf{new} \cup \{v\}, \max\{p, q\}]$ is true.

Then we can safely include v in new and increase the loop counter while maintaining the invariant.

```
$\{I \wedge \exists p.\, I_o[i, \mathsf{new}, p]\}$
while($i \le |S|$){
    $\{I \wedge \exists p.\, I_o[i, \mathsf{new}, p] \wedge i \le |S|\}$
    $v \leftarrow M[\alpha_i]$.read();
    $\left\{\begin{array}{c} I \wedge \exists p.\, I_o[i, \mathsf{new}, p] \wedge i \le |S| \\ \wedge \left(\begin{array}{c} (v = \bot \wedge I_o[i+1, \mathsf{new} \cup \{\bot\}, p]) \\ \vee \left(\begin{array}{c} v \neq \bot \wedge \boldsymbol{\alpha_i}{:}M[\alpha_i].\mathsf{write}(v) \in s \\ \wedge \boldsymbol{\alpha_i}{:}\mathsf{write\_snapshot}(v) \in \mathsf{lin}(\rho) \\ \wedge \exists q \sqsubseteq \mathsf{lin}(\rho).\, \boldsymbol{\alpha_i}{:}\mathsf{write\_snapshot}(v) \in q \\ \wedge I_o[i+1, \mathsf{new} \cup \{v\}, \max\{p, q\}] \end{array}\right) \end{array}\right) \end{array}\right\}$
    new $\leftarrow$ new $\cup \{v\}$; $i \leftarrow i + 1$
    $\{I \wedge \exists p.\, I_o[i, \mathsf{new}, p]\}$
}
$\{I \wedge \exists p.\, I_o[|S| + 1, \mathsf{new}, p]\}$
```

Fig. 20. Lower Bound Construction Proof

Construct Upper Bound. In figure 21, we use $I \wedge \exists p.\, I_o[|S| + 1, \mathsf{old}, p] \wedge I_n[i, \mathsf{new}, p]$ as the loop invariant. According to the read value, there will be four different cases.

- If the read value is $\bot$, then thread α_i has not written to its cell in $\mathsf{lin}(\rho)$. Obviously, its write will no appear in the prefix p and $\mathsf{new} \cup \{\bot\}$ is an upper bound, i.e., $I_n[i+1, \mathsf{new} \cup \{\bot\}, p]$ is true.
- If the read value is not $\bot$ and $\boldsymbol{\alpha_i}$:write_snapshot(v) already appears in the prefix p, then it is safe to include v in the upper bound, i.e., $I_n[i+1, \mathsf{new} \cup \{v\}, p]$ is true.
- If the read value is not $\bot$ and $\boldsymbol{\alpha_i}$:write_snapshot(w) does not appear in the prefix p for any w, then we can safely add v to the upper bound for the same reason as the first case, and $I_n[i+1, \mathsf{new} \cup \{v\}, p]$ is true.
- If the read value is not $\bot$ and $\boldsymbol{\alpha_i}$:write_snapshot(w) already appears in the prefix p for a different w from v, then adding v to the upper bound new may produce a snapshot inconsistent with the trace. However, since we read a different v from the memory cell, $\boldsymbol{\alpha_i}$:write(v)

must appear in s, and by I, we know $\boldsymbol{\alpha_i}$:write_snapshot(v) must appear in $\mathsf{lin}(\rho)$. With two different write_snapshot invocations for the same thread α_i in $\mathsf{lin}(\rho)$, the client specification $\mu_{\text{write\_snapshot}}$ is violated, which leads to a contradiction with $\mu_{\text{write\_snapshot}}$. As a result, under the restriction of $\mu_{\text{write\_snapshot}}$, this case will not happen.

Then we can safely include v in new and increase the loop counter while maintaining the invariant.

```
{$I \land \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[i, \mathsf{new}, p]$}
while($i \leq |S|$){
  {$I \land \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[i, \mathsf{new}, p] \land i \leq |S|$}
  $v \leftarrow M[\alpha_i]$.read();
  $\left\{\begin{array}{c} I \land \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[i, \mathsf{new}, p] \land i \leq |S| \\ \land \left(\begin{array}{c} (v = \bot \land I_n[i+1, \mathsf{new} \cup \{\bot\}, p]) \\ \lor \left(v \neq \bot \land \boldsymbol{\alpha_i}{:}\mathsf{write\_snapshot}(v) \in p \land I_n[i+1, \mathsf{new} \cup \{v\}, p]\right) \\ \lor \left(v \neq \bot \land \forall w.\, \boldsymbol{\alpha_i}{:}\mathsf{write\_snapshot}(w) \notin p \land I_n[i+1, \mathsf{new} \cup \{v\}, p]\right) \\ \lor \left(\begin{array}{c} v \neq \bot \land \exists w \neq v.\, \boldsymbol{\alpha_i}{:}\mathsf{write\_snapshot}(w) \in p \land \boldsymbol{\alpha_i}{:}M[\alpha_i].\mathsf{write}(v) \in s \\ \land \boldsymbol{\alpha_i}{:}\mathsf{write\_snapshot}(v) \in \mathsf{lin}(\rho) \land \boldsymbol{\alpha_i}{:}\mathsf{write\_snapshot}(w) \in \mathsf{lin}(\rho) \end{array}\right) \end{array}\right) \end{array}\right\}$
  new $\leftarrow$ new $\cup \{v\}$; $i \leftarrow i + 1$
  {$I \land \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[i, \mathsf{new}, p]$}
}
{$I \land \exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[|S|+1, \mathsf{new}, p]$}
```

Fig. 21. Upper Bound Construction Proof

Then, we reach the loop at line 8 in figure 19. Here, we use the loop invariant

$$I \land \mathsf{last}(\pi_\alpha(\mathsf{lin}(\rho))) = \boldsymbol{\alpha}{:}\mathsf{write\_snapshot}(v) \land$$
$$\exists p.\, I_o[|S|+1, \mathsf{old}, p] \land I_n[|S|+1, \mathsf{new}, p] \land \exists q.\, I_o[|S|+1, \mathsf{new}, q]$$

which is the asserts that old and new are lower and upper bounds of the snapshot state for some prefix p of the linearized trace, and new is the lower bound of another linearized prefix q.

- Inside the loop, we only keep the second branch and by assigning new to old, the new old is still the lower bound. Then, a loop identical to the one at line 4 takes a new snapshot into new and we reuse the previous proof for it. The loop establishes the new as the new upper bound for the new prefix p and we re-establish the loop invariant.
- If the loop terminates, we only keep the first branch. With the snapshot state bounded by new on both side, we can derive $\mathsf{snpstate}(p) = \mathsf{new} \backslash \{\bot\}$, which means $\mathsf{new} \backslash \{\bot\}$ is the correct snapshot at the time where p is the complete linearized trace. We can then linearize the response to the current operation after p and $\mathsf{last}(\pi_\alpha(\rho)) = \boldsymbol{\alpha}{:}\mathsf{new} \backslash \{\bot\}$ is true after the linearization, which finishes the proof. This linearization step satisfies the following guarantee.

$$(\Delta, s, \rho)\mathcal{G}_{\text{snapshot}}[\alpha](\Delta', s', \rho') \iff \left(\begin{array}{c} s' = s \land \exists p_1, p_2, V.\, \mathsf{lin}(\rho) = p_1 \cdot p_2 \\ \land \mathsf{lin}(\rho') = p_1 \cdot \boldsymbol{\alpha}{:}V \cdot p_2 \land V = \mathsf{snpstate}(p_1) \end{array}\right)$$

The rely rely is defined as

$$\mathcal{R}[\alpha] \triangleq \bigcup_{\alpha' \in \Upsilon, \alpha' \neq \alpha} \mathcal{G}_{\text{write}}[\alpha'] \cup \mathcal{G}_{\text{snapshot}}[\alpha'] \cup \mathcal{G}_{\text{id}}[\alpha'] \cup \mathsf{invoke}_{\alpha'}(-) \cup \mathsf{return}_{\alpha'}(-)$$

and one may easily check that all assertions in the proof is stable w.r.t. it. Now, we have proved that the write-snapshot object is linearizable w.r.t. its specification $\nu_{\mathsf{write\_snapshot}}$ and by the FLiT correctness theorem, this object is also durably linearizable.

I.3 Swap Operation in File System through Write-Ahead Logs

In this four-layered example, we demonstrate that our framework can handle sophisticated file system patterns compositionally. In the upper layer, we present a file system capable of swapping files atomically between directories. The file system depends on a write-ahead log objects and an array of replicated disk cells, both of which are implemented in the lower layer. The file system is presented in I.3.1, while the replicated disks and write-ahead log is described in I.3.3 and I.3.5.

I.3.1 File System with Read, Write, and Swap. We first present and verify a file system that supports file read and write operation, as well as file swapping between directories. The file system exposes a two level structure. At the first level lies a set of folders, each occupies a single disk block as their inode. For simplicity, the API uses block ids instead of strings to uniquely identify folders. Each folder contains a set of files, identified by their file id (unique within each folder). A swap operation will swap the pointer in respective folders' inodes, which can be considered as a symmetric move operation seen in actual file systems. For simplicity, we restrict each file to contain a single block for file content, and assumes all files and directories are pre-allocated on the disk. Allowing for transactions involving several blocks is straight-forward given that we show our techniques work in this simplified setting.

While file read and write operations are mostly straightforward, the swap operation requires special treatment for its atomicity. As swap operations need to update two different folders (and thus two different disk blocks), the possibilty that a crash happens in between can never be ruled out. To ensure persistent linearizability, we record the operations in write-ahead logs so that the recovery routine can finish incomplete operations. Figure 22 showcases the pseudocode for this file system.

The underlying disk object is modeled as a map from block_id to block. block_id can be considered as an integer type whereas block can be considered as a constant-sized byte array. In the case a block actually contains folder inode information, we byte-cast them into the correct type folder_inode.

$$\mathsf{Disk} := \{\mathsf{write} : \mathsf{block\_id} \times \mathsf{block} \longrightarrow 1, \mathsf{read} : \mathsf{block\_id} \longrightarrow \mathsf{block}\}$$

$$\mathsf{eval}_{\mathsf{Disk}^{\lightning}} : P_{\dagger\mathsf{Disk}^{\lightning}} \longrightarrow \{\bot\} + (\mathsf{file\_id} \overset{\mathsf{fin}}{\rightharpoonup} \mathsf{block})$$

$$\mathsf{eval}_{\mathsf{Disk}^{\lightning}}(s) := \begin{cases} \bigcup_{b \in \mathsf{block\_id}} \{b \leftarrow 0\} & \epsilon \\ M & s = s' \cdot \lightning \wedge \mathsf{eval}_{\mathsf{Disk}^{\lightning}}(s') = M \\ M & s = s' \cdot m \cdot \lightning \wedge \mathsf{eval}_{\mathsf{Disk}^{\lightning}}(s' \cdot \lightning) = M \wedge \lambda_{\mathsf{Disk}}(m) = O \\ M[b \mapsto v] & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{write}(b, v) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge \mathsf{eval}_{\mathsf{Disk}^{\lightning}}(s') = M \\ M & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{read}(b) \cdot \boldsymbol{\alpha}{:}v \wedge \mathsf{eval}_{\mathsf{Disk}^{\lightning}}(s') = M \wedge M[b] = v \\ \bot & \text{otherwise} \end{cases}$$

$$\nu_{\mathsf{Disk}^{\lightning}} := \{s \mid \exists s'. s \sqsubseteq s' \wedge \mathsf{eval}_{\mathsf{Disk}^{\lightning}}(s') \neq \bot\}$$

```
M_FS:
Import disk:Disk
Import lockmap:LockMapB
Import log:Log

void write(block_id dir, file_id fid, data_block data) {
  lock[dir].acquire();
  dir_inode ← (folder_inode) disk.read(dir);
  file ← dir_inode[fid];
  disk.write(file, data);
  lock[dir].release();
}

data_block read(block_id dir, file_id fid) {
  lock[dir].acquire();
  dir_inode ← (folder_inode) disk.read(dir);
  file ← dir_inode[fid];
  data ← disk.read(file);
  lock[dir].release();
  return data;
}

void swap(block_id src, block_id tgt, file_id src_f, file_id tgt_f) {
  if (src < tgt) {
    lock[src].acquire();
    lock[tgt].acquire();
  } else {
    lock[tgt].acquire();
    lock[src].acquire();
  }
  src_inode ← (folder_inode) disk.read(src);
  tgt_inode ← (folder_inode) disk.read(tgt);
  src_file ← src_inode[src_f];
  tgt_file ← tgt_inode[tgt_f];
  log.insert(src, tgt, src_f, tgt_f, src_file, tgt_file);

  disk.write(tgt, tgt_inode[tgt_f -> src_file]);
  disk.write(src, src_inode[src_f -> tgt_file]);
  log.remove(α);
  lock[src].release();
  lock[tgt].release();
}

void recovery() {
  for (i = 0; i < |agents|; i ++) {
    entry ← log.get(agents[i]);
    if (entry = None)
      continue;
    (src, tgt, src_f, tgt_f, src_file, tgt_file) ← entry;

    src_inode ← (folder_inode) disk.read(src);
    tgt_inode ← (folder_inode) disk.read(tgt);

    disk.write(tgt, tgt_inode[tgt_f -> src_file]);
    disk.write(src, src_inode[src_f -> tgt_file]);
    log.remove(agents[i]);
  }
}
```

Fig. 22. Implementation of the File System

The LockMapB object is specified as a collection of individual locks, indexed by block_id type.

$$\mathbf{LockMapB} \coloneqq \{\mathbf{acq} : \mathbf{block\_id} \to \mathbf{1}, \mathsf{rel} : \mathsf{block\_id} \to \mathbf{1}\}$$

$$\mathsf{eval}_{\mathsf{LockMapB}} : P_{\dagger\mathsf{LockMapB}} \to \{\bot\} + (\mathsf{block\_id} \overset{\mathsf{fin}}{\rightharpoonup} \mathcal{P}(\Upsilon))$$

$$\mathsf{eval}_{\mathsf{LockMapB}}(s) \coloneqq \begin{cases} \bigcup_{b \in \mathsf{block\_id}}\{f \leftarrow \varnothing\} & \epsilon \\ m[b \mapsto \{\alpha\}] & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{acq}(b) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge \mathsf{eval}_{\mathsf{LockMapB}}(s') = m \wedge m[b] = \varnothing \\ m[b \mapsto \varnothing] & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{rel}(b) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge \mathsf{eval}_{\mathsf{LockMapB}}(s') = m \wedge m[b] = \{\alpha\} \\ \bot & \text{otherwise} \end{cases}$$

$$\nu_{\mathsf{LockMapB}^{\lightning}} \coloneqq \mathsf{vol}(\{s \mid \exists s'.s \sqsubseteq s' \wedge \mathsf{eval}_{\mathsf{LockMapB}}(s') \neq \bot\})$$

The lock map is in fact equivalent to a horizontal composition of volatile locks:

$$\nu_{\mathsf{LockMapB}^{\lightning}} \coloneqq \otimes_{b \in \mathsf{block\_id}} \mathsf{vol}(\nu_{\mathsf{Lock}})$$

Since Oliveira Vale et al. [31] have verified the linearizability of a lock (in fact, the ticket lock implementation M_{Lock}) we may lift their proof to our setting using Prop. 2.8, as explained in the main paper.

The last object Log is a write ahead log used for crash atomicity of swap operations. It stores at most one log entry per thread, and each log entry contains all the information about a single swap operation: the source and target folder block id, the source and target file id, and the block id of the swapped files. While a thread may only insert entry for itself, it can remove entry for any thread due to the recovery routine. Otherwise, the specification of Log is simply another map, with the following formal specification,

$$\mathsf{entry} \coloneqq \mathsf{block\_id} \times \mathsf{block\_id} \times \mathsf{file\_id} \times \mathsf{file\_id} \times \mathsf{block\_id} \times \mathsf{block\_id}$$

$$\mathsf{Log} \coloneqq \{\mathsf{insert} : \mathsf{entry} \to 1, \mathsf{get} : \Upsilon \to \mathsf{option\ entry}, \mathsf{remove} : \Upsilon \to 1\}$$

$$\mathsf{eval}_{\mathsf{Log}^{\lightning}} : P_{\dagger\mathsf{Log}^{\lightning}} \to \{\bot\} + (\Upsilon \overset{\mathsf{fin}}{\rightharpoonup} \mathsf{entry})$$

$$\mathsf{eval}_{\mathsf{Log}^{\lightning}}(s) \coloneqq \begin{cases} \cup_{\alpha \in \Upsilon}[\alpha \mapsto \mathsf{None}] & \epsilon \\ l & s = s' \cdot \lightning \wedge \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s') = l \\ l & s = s' \cdot m \cdot \lightning \wedge \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s' \cdot \lightning) = l \wedge \lambda_{\mathsf{Log}}(m) = O \\ l[\alpha \mapsto e] & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{insert}(e) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s') = l \\ l & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{get}(\alpha') \cdot \boldsymbol{\alpha}{:}l[\alpha'] \wedge \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s') = l \\ l[\alpha' \mapsto \mathsf{None}] & s = s' \cdot \boldsymbol{\alpha}{:}\mathsf{remove}(\alpha') \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \wedge \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s') = l \\ \bot & \text{otherwise} \end{cases}$$

$$\nu_{\mathsf{Log}^{\lightning}} \coloneqq \{s \mid \exists s'.s \sqsubseteq s' \wedge \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s') \neq \bot\}$$

Finally, the specification of the file system FS is a nested map from block ids (of folders) into a map from file ids to file contents, and the formal definition is given below,

$$\mathbf{FS} := \left\{ \begin{matrix} \text{write} : \text{block\_id} \times \text{file\_id} \times \text{block} \to \mathbf{1}, \\ \text{read} : \text{block\_id} \times \text{file\_id} \to \text{block}, \\ \textbf{swap} : \textbf{block\_id} \times \textbf{block\_id} \times \textbf{file\_id} \times \textbf{file\_id} \to \mathbf{1} \end{matrix} \right\}$$

$$f[a \mapsto b \mapsto c] := f[a \mapsto [f[a][b \mapsto c]]]$$

$$\mathsf{eval}_{\mathsf{FS}^{\lightning}} : P_{\dagger \mathsf{FS}^{\lightning}} \to \{\bot\} + (\text{block\_id} \overset{\text{fin}}{\rightharpoonup} \text{file\_id} \overset{\text{fin}}{\rightharpoonup} \text{block})$$

$$\mathsf{eval}_{\mathsf{FS}^{\lightning}}(s) := \begin{cases} \cup_{\alpha \in \Upsilon}[\alpha \mapsto \text{None}] & \epsilon \\ f & s = s' \cdot \lightning \wedge \mathsf{eval}_{\mathsf{FS}^{\lightning}}(s') = f \\ f & s = s' \cdot m \cdot \lightning \wedge \mathsf{eval}_{\mathsf{FS}^{\lightning}}(s' \cdot \lightning) = fs \wedge \lambda_{\mathsf{FS}}(m) = O \\ f[a \mapsto b \mapsto c] & s = s' \cdot \boldsymbol{\alpha}{:}\text{write}(a, b, c) \cdot \boldsymbol{\alpha}{:}\text{ok} \wedge \mathsf{eval}_{\mathsf{FS}^{\lightning}}(s') = f \\ f & s = s' \cdot \boldsymbol{\alpha}{:}\text{read}(a, b) \cdot \boldsymbol{\alpha}{:}f[a][b] \wedge \mathsf{eval}_{\mathsf{FS}^{\lightning}}(s') = f \\ f[a \mapsto b \mapsto f[c][d]][c \mapsto d \mapsto f[a][b]] & s = s' \cdot \boldsymbol{\alpha}{:}\text{swap}(a, b, c, d) \cdot \boldsymbol{\alpha}{:}\text{ok} \wedge \mathsf{eval}_{\mathsf{FS}^{\lightning}}(s') = f \\ \bot & \text{otherwise} \end{cases}$$

$$\nu_{\mathsf{FS}^{\lightning}} := \{s \mid \exists s'. s \sqsubseteq s' \wedge \mathsf{eval}_{\mathsf{FS}^{\lightning}}(s') \neq \bot\}$$

I.3.2 Proof of the File System. The line-by-line proof is presented in Figure 23, Figure 24, Figure 25, Figure 26, and we highlight the important steps here.

First, we define the maximal linearized prefix of current possibility ρ,

$$\mathsf{lin} : P_{\dagger \mathsf{FS}^{\lightning}} \to P_{\dagger \mathsf{FS}^{\lightning}}$$

$$\mathsf{lin}(\rho) = p_{\lightning} \cdot p \iff \left(\begin{matrix} p_{\lightning} \cdot p \sqsubseteq \rho \wedge \\ (p_{\lightning} = \epsilon \vee \exists p'. p' \cdot \lightning = p_{\lightning}) \wedge \\ p \in P_{!\mathsf{FS}} \wedge \forall p'. p \sqsubseteq p' \wedge p_{\lightning} \cdot p' \sqsubseteq \rho \implies p' \notin P_{!\mathsf{FS}} \end{matrix} \right)$$

which helps us derive the current state of the object according to the possibility as well as the concrete play,

$$\mathsf{state}^{\rho} : P_{\dagger \mathsf{FS}^{\lightning}} \to \{\bot\} + (\text{block\_id} \overset{\text{fin}}{\rightharpoonup} \text{file\_id} \overset{\text{fin}}{\rightharpoonup} \text{block})$$
$$\mathsf{state}^{\rho}(\rho) := \mathsf{eval}_{\mathsf{FS}^{\lightning}}(\mathsf{lin}(\rho))$$
$$\mathsf{state}^{s} : P_{\dagger (\mathsf{Disk\&Log\&LockMapB})^{\lightning}} \to \{\bot\} + (\text{file\_id} \overset{\text{fin}}{\rightharpoonup} \text{block})$$
$$\mathsf{state}^{s}(s) := \mathsf{eval}_{\mathsf{Disk}^{\lightning}}(s \restriction_{\mathsf{Disk}^{\lightning}})$$

Since the directory inodes are never moved around, we take the liberty to use notation $\mathsf{state}^{s}(s)[d][f]$ when it's clear that $d \in$ block_id is a folder block and f is a file id. Next, we are interested in the current status of lock ownership,

$$\mathsf{owned} : \mathcal{P}(P_{\dagger (\mathsf{Disk\&Log\&LockMapB})^{\lightning}} \times \text{block\_id})$$
$$\mathsf{owned}(s, b) \iff \mathsf{eval}_{\mathsf{LockMapB}^{\lightning}}(s \restriction_{\mathsf{LockMapB}^{\lightning}})[b] \neq \varnothing$$
$$\mathsf{ownedby} : P_{\dagger (\mathsf{Disk\&Log\&LockMapB})^{\lightning}} \times \Upsilon \to \mathcal{P}(\text{block\_id})$$
$$\mathsf{ownedby}(s, \alpha) := \{b \mid \mathsf{eval}_{\mathsf{LockMapB}^{\lightning}}(s \restriction_{\mathsf{LockMapB}^{\lightning}})[b] = \{\alpha\}\}$$

Finally, we care about whether certain blocks are currently mentioned by some entry in WAL,

$$\mathsf{logged} : \mathcal{P}(P_{\dagger(\mathsf{Disk\&Log\&LockMapB})^{\lightning}} \times \mathsf{block\_id} \times \mathsf{file\_id})$$

$$\mathsf{logged}(s, d, f) \iff \exists \alpha, d_1, d_2, f_1, f_2, b_1, b_2. \begin{pmatrix} \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s \upharpoonright_{\mathsf{Log}^{\lightning}})[\alpha] = (d_1, d_2, f_1, f_2, b_1, b_2) \wedge \\ ((d, f) = (d_1, f_1) \vee (d, f) = (d_2, f_2)) \end{pmatrix}$$

$$\mathsf{logged}^2 : \mathcal{P}(P_{\dagger(\mathsf{Disk\&Log\&LockMapB})^{\lightning}} \times \mathsf{block\_id} \times \mathsf{block\_id} \times \mathsf{file\_id} \times \mathsf{file\_id})$$

$$\mathsf{logged}^2(s, d_1, d_2, f_1, f_2) \iff \exists \alpha, b_1, b_2. \begin{pmatrix} \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s \upharpoonright_{\mathsf{Log}^{\lightning}})[\alpha] = (d_1, d_2, f_1, f_2, b_1, b_2) \vee \\ \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s \upharpoonright_{\mathsf{Log}^{\lightning}})[\alpha] = (d_2, d_1, f_2, f_1, b_1, b_2) \end{pmatrix}$$

The runtime invariant of the program is then a collection of observations. Firstly, the file content we compute from the two different states matches with each other except for those currently in a WAL entry,

$$I_1(s, \rho) \iff \forall d, f. \neg \mathsf{logged}(d, f) \implies \mathsf{state}^s(s)[\mathsf{state}^s(s)[d][f]] = \mathsf{state}^{\rho}(\rho)[d][f]$$

Secondly, two file blocks can only be duplicates of each other only if both of them are in the same WAL entry,

$$I_2(s, \rho) \iff \forall d_1, d_2, f_1, f_2. \mathsf{state}^s(s)[d_1][f_1] = \mathsf{state}^s(s)[d_2][f_2] \implies \mathsf{logged}^2(s, d_1, d_2, f_1, f_2)$$

Thirdly, all entries in WAL corresponds to a pending invocation (potentially before some crashes) in the possibility, and the recorded file block id matches the current overlay state (before swap),

$$I_3(s, \rho) \iff \forall \alpha, d_1, d_2, f_1, f_2, b_1, b_2. \begin{pmatrix} \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s \upharpoonright_{\mathsf{Log}^{\lightning}})[\alpha] = (d_1, d_2, f_1, f_2, b_1, b_2) \implies \\ \mathsf{state}^{\rho}(\rho)[d_1][f_1] = b_1 \wedge \mathsf{state}^{\rho}(\rho)[d_2][f_2] = b_2 \wedge \\ \exists p. \pi_{\alpha}(\rho) \upharpoonright_{\mathsf{FS}} = p \cdot \mathsf{swap}(d_1, d_2, f_1, f_2) \end{pmatrix}$$

Finally, membership in WAL implies lock ownership by the same thread,

$$I_4(s, \rho) \iff \forall \alpha, d_1, d_2, f_1, f_2, b_1, b_2. \begin{pmatrix} \mathsf{eval}_{\mathsf{Log}^{\lightning}}(s \upharpoonright_{\mathsf{Log}^{\lightning}})[\alpha] = (d_1, d_2, f_1, f_2, b_1, b_2) \implies \\ \{d_1, d_2\} = \mathsf{ownedby}(s, \alpha) \end{pmatrix}$$

The runtime invariant (both the precondition and postcondition of all FS methods) is then the conjunction of all above,

$$I := I_1 \cap I_2 \cap I_3 \cap I_4$$

However, due to the fact that the locks are not persistent, the last conjunction is not stable with respect to crashes. Thus the crash invariant is the conjunction of the first three,

$$I_{\lightning} := I_1 \cap I_2 \cap I_3$$

$$(s, \rho)\ Q_{\lightning}\ (s', \rho') \iff I_{\lightning}(s', \rho')$$

The precondition and postcondition of any method (except for recovery) assumes the invariant as well as empty ownership of current thread,

$$P^f(\Delta, s, \rho) \iff I(s, \rho) \wedge \mathsf{ownedby}(s, \alpha) = \varnothing$$

$$(-)\ Q^f\ (\Delta, s', \rho') \iff I(s', \rho') \wedge \mathsf{ownedby}(s', \alpha) = \varnothing$$

The guarantee condition, in addition to preservation of invariants, further specifies that one thread may only update folders or files they currently owns, physically or abstractly. Rely condition is then the union of guarantee and method invocations and returns,

$$(s, \rho)\ \mathcal{G}[\alpha]\ (s', \rho') \iff \begin{pmatrix} \forall d.d \notin \mathsf{ownedby}(s, \alpha) \implies \mathsf{state}^s(s)[b] = \mathsf{state}^s(s')[b]\ \land \\ \forall d.d \notin \mathsf{ownedby}(s, \alpha) \implies \mathsf{state}^\rho(s)[d] = \mathsf{state}^\rho(s')[d]\ \land \\ I(s', \rho') \end{pmatrix}$$

$$\mathcal{R}[\alpha] \triangleq \bigcup_{\alpha' \in \Upsilon, \alpha' \neq \alpha} \mathcal{G}[\alpha'] \cup \mathsf{invoke}_{\alpha'}(-) \cup \mathsf{return}_{\alpha'}(-)$$

The linearization point of swap normally happens at the time when log entry is removed, which by itself is a straightforward proof thanks to mutual exclusion. However, if a swap method crashes after inserting the WAL entry but before removing the entry, it will instead be retroactively linearized during the recovery procedure, also at the point when the log is removed from WAL. While the same operation may be performed multiple times and some are even partially performed, it is safe nonetheless thanks to the idempotent nature of log entry application.

The linearization point of write happens at the point when the underlay actually writes to the disk and read linearizes at the disk read operation, as one would expect. The safety is provided by the mutual exclusion of locks, the same as the swap operation.

```
{$I(s, \rho) \land \mathsf{ownedby}(s, \alpha) = \varnothing$}
write(dir, fid, data){
    lock[dir].acquire();
    {$I(s, \rho) \land \mathsf{ownedby}(s, \alpha) = \{\mathsf{dir}\}$}
    dir_inode ← (folder_inode)disk.read(dir);
    file ← dir_inode[fid];
    {$I(s, \rho) \land \mathsf{ownedby}(s, \alpha) = \{\mathsf{dir}\} \land \mathsf{state}^s(s)[\mathsf{dir}][\mathsf{fid}] = \mathsf{file}$}
    disk.write(file, data);
    {$I(s, \rho) \land \mathsf{ownedby}(s, \alpha) = \{\mathsf{dir}\} \land \mathsf{state}^s(s)[\mathsf{dir}][\mathsf{fid}] = \mathsf{file} \land \mathsf{state}^s(s)[\mathsf{file}] = \mathsf{data}$}

    $\left\{ I(s', \rho') \land \begin{pmatrix} \mathsf{ownedby}(s', \alpha) = \{\mathsf{dir}\} \land \mathsf{state}^s(s')[\mathsf{dir}][\mathsf{fid}] = \mathsf{file} \land \mathsf{state}^s(s')[\mathsf{file}] = \mathsf{data}\ \land \\ \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{write}(\mathsf{dir}, \mathsf{fid}, \mathsf{data}) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \sqsubseteq \rho' \end{pmatrix} \right\}$
    lock[dir].release();
    {$I(s, \rho) \land \mathsf{ownedby}(s, \alpha) = \varnothing \land \exists p.\pi_\alpha(\rho) = p \cdot \mathsf{ok}$}

}
{$I(s, \rho) \land \mathsf{ownedby}(s, \alpha) = \varnothing \land \mathsf{returned}[\mathsf{write}](\Delta, s, \rho)$}
```

Fig. 23. Proof of file system - write

I.3.3 Replicated Disk.

To demonstrate vertical composition, we implement the aforementioned disk interface on top of several disks. Thanks to the locality property of tensor operator, we only need to verify a single disk block and safely compose into the whole disk with little effort. The

```
$\{I(s,\rho) \land \text{ownedby}(s,\alpha) = \emptyset\}$
write(dir, fid){
    lock[dir].acquire();
    $\{I(s,\rho) \land \text{ownedby}(s,\alpha) = \{\text{dir}\}\}$
    dir_inode ⟵ (folder_inode)disk.read(dir);
    file ⟵ dir_inode[fid];
    $\{I(s,\rho) \land \text{ownedby}(s,\alpha) = \{\text{dir}\} \land \text{state}^s(s)[\text{dir}][\text{fid}] = \text{file}\}$
    data ⟵ disk.read(file);
    $\{I(s,\rho) \land \text{ownedby}(s,\alpha) = \{\text{dir}\} \land \text{state}^s(s)[\text{dir}][\text{fid}] = \text{file} \land \text{state}^s(s)[\text{file}] = \text{data}\}$

    $\left\{I(s',\rho') \land \begin{pmatrix} \text{ownedby}(s',\alpha) = \{\text{dir}\} \land \text{state}^s(s')[\text{dir}][\text{fid}] = \text{file} \land \text{state}^s(s')[\text{file}] = \text{data} \land \\ \text{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\text{read}(\text{data}) \cdot \boldsymbol{\alpha}{:}\text{data} \sqsubseteq \rho' \end{pmatrix}\right\}$
    lock[dir].release();
    $\{I(s,\rho) \land \text{ownedby}(s,\alpha) = \emptyset \land \exists p.\pi_\alpha(\rho) = p \cdot \text{data}\}$

}
$\{I(s,\rho) \land \text{ownedby}(s,\alpha) = \emptyset \land \text{returned}[\text{write}](\Delta, s, \rho)\}$
```

Fig. 24. Proof of file system - read

single block specification, which both the underlay and overlay follows, is provided below,

$$\text{DiskBlock} := \{\text{write} : \text{block} \longrightarrow 1, \text{read} : \mathbf{1} \longrightarrow \text{block}\}$$

$$\text{eval}_{\text{DiskBlock}^{\lightning}} : P_{\dagger\text{DiskBlock}^{\lightning}} \longrightarrow \{\bot\} + \text{block}$$

$$\text{eval}_{\text{DiskBlock}^{\lightning}}(s) := \begin{cases} 0 & \epsilon \\ v & s = s' \cdot \lightning \land \text{eval}_{\text{Disk}^{\lightning}}(s') = v \\ v & s = s' \cdot m \cdot \lightning \land \text{eval}_{\text{Disk}^{\lightning}}(s' \cdot \lightning) = v \land \lambda_{\text{Disk}}(m) = O \\ v & s = s' \cdot \boldsymbol{\alpha}{:}\text{write}(v) \cdot \boldsymbol{\alpha}{:}\text{ok} \land \text{eval}_{\text{Disk}^{\lightning}}(s') \neq \bot \\ v & s = s' \cdot \boldsymbol{\alpha}{:}\text{read}() \cdot \boldsymbol{\alpha}{:}v \land \text{eval}_{\text{Disk}^{\lightning}}(s') = v \\ \bot & \text{otherwise} \end{cases}$$

$$\nu_{\text{DiskBlock}^{\lightning}} := \{s \mid \exists s'.s \sqsubseteq s' \land \text{eval}_{\text{DiskBlock}^{\lightning}}(s') \neq \bot\}$$

The implementation is given in Figure 27. In the implementaion, we do not acquire locks when writing or reading disk blocks. This is sound since the file system always guarantees mutual exclusion when calling disk operations on the same block. We express this fact through a client policy that requires the client to only access the disk atomically:

$$\xi_{\text{DiskBlock}} := P_{!\text{DiskBlock}}$$

where $P_{!\text{DiskBlock}}$ is the set of well-formed atomic plays over DiskBlock.

I.3.4 Verification of Replicated Disks. For the purpose of verification, we assume the lin function and the eval function are defined in the same way as in the file interface verification. Following a similar approach, we define state^ρ and state^s according to the possibility and the the underlay play respectively. We use $\text{DiskBlock}[N]_i$ to denote the i-th disk of the N disk array of the underlay.

$$\text{state}^\rho : P_{\dagger\text{DiskBlock}^{\lightning}} \longrightarrow \{\bot\} + \text{block}$$

$$\text{state}^\rho(\rho) := \text{eval}_{\text{DiskBlock}^{\lightning}}(\text{lin}(\rho))$$

$$\text{state}^s : P_{\dagger\text{DiskBlock}[N]^{\lightning}} \longrightarrow \{\bot\} + \text{block}$$

$$\text{state}^s(\rho) := \text{eval}_{\text{DiskBlock}^{\lightning}}(s\restriction_{\text{DiskBlock}[N]_0^{\lightning}})$$

```
$\{I(s, \rho) \wedge \text{ownedby}(s, \alpha) = \emptyset\}$
swap(src, tgt, src_f, tgt_f){
  if(src < tgt){
    lock[src].acquire();
    lock[tgt].acquire();
  }else{
    lock[tgt].acquire();
    lock[src].acquire();
  }
  $\{I(s, \rho) \wedge \text{ownedby}(s, \alpha) = \{\text{src}, \text{tgt}\}\}$
  src_inode ← (folder_inode)disk.read(src);
  tgt_inode ← (folder_inode)disk.read(tgt);
  src_file ← src_inode[src_f];
  tgt_file ← tgt_inode[tgt_f];
  $\left\{I(s, \rho) \wedge \begin{pmatrix} \text{ownedby}(s, \alpha) = \{\text{src}, \text{tgt}\} \wedge \\ \text{state}^s(s)[\text{src}] = \text{src\_inode} \wedge \text{state}^s(s)[\text{tgt}] = \text{tgt\_inode} \end{pmatrix}\right\}$
  log.insert(src, tgt, src_f, tgt_f, src_file, tgt_file);
  $\left\{I(s, \rho) \wedge \begin{pmatrix} \text{ownedby}(s, \alpha) = \{\text{src}, \text{tgt}\} \wedge \\ \text{logged}^2(s, \text{src}, \text{tgt}, \text{src\_f}, \text{tgt\_f}) \wedge \\ \text{state}^s(s)[\text{src}] = \text{src\_inode} \wedge \text{state}^s(s)[\text{tgt}] = \text{tgt\_inode} \end{pmatrix}\right\}$
  disk.write(tgt, tgt_inode[tgt_f ↦ src_file]);
  $\left\{I(s, \rho) \wedge \begin{pmatrix} \text{ownedby}(s, \alpha) = \{\text{src}, \text{tgt}\} \wedge \\ \text{logged}^2(s, \text{src}, \text{tgt}, \text{src\_f}, \text{tgt\_f}) \wedge \\ \text{state}^s(s)[\text{src}] = \text{src\_inode} \wedge \text{state}^s(s)[\text{tgt}] = \text{tgt\_inode}[\text{tgt\_f} \mapsto \text{src\_file}] \end{pmatrix}\right\}$
  disk.write(src, src_inode[src_f ↦ tgt_file]);
  $\left\{I(s, \rho) \wedge \begin{pmatrix} \text{ownedby}(s, \alpha) = \{\text{src}, \text{tgt}\} \wedge \\ \text{logged}^2(s, \text{src}, \text{tgt}, \text{src\_f}, \text{tgt\_f}) \wedge \\ \text{state}^s(s)[\text{src}] = \text{src\_inode}[\text{src\_f} \mapsto \text{tgt\_file}] \wedge \text{state}^s(s)[\text{tgt}] = \text{tgt\_inode}[\text{tgt\_f} \mapsto \text{src\_file}] \end{pmatrix}\right\}$
  log.remove($\alpha$);
  $\left\{I(s, \rho) \wedge \begin{pmatrix} \text{ownedby}(s, \alpha) = \{\text{src}, \text{tgt}\} \wedge \\ \text{state}^s(s)[\text{src}] = \text{src\_inode}[\text{src\_f} \mapsto \text{tgt\_file}] \wedge \text{state}^s(s)[\text{tgt}] = \text{tgt\_inode}[\text{tgt\_f} \mapsto \text{src\_file}] \end{pmatrix}\right\}$

  $\left\{I(s', \rho') \wedge \begin{pmatrix} \text{lin}(\rho) \cdot \boldsymbol{\alpha}\text{:swap}(\text{src}, \text{tgt}, \text{src\_f}, \text{tgt\_f}) \cdot \boldsymbol{\alpha}\text{:ok} \sqsubseteq \rho' \wedge \\ \text{ownedby}(s', \alpha) = \{\text{src}, \text{tgt}\} \end{pmatrix}\right\}$
  $\{I(s, \rho) \wedge \text{ownedby}(s, \alpha) = \{\text{src}, \text{tgt}\} \wedge \exists p.\pi_\alpha(\rho) = p \cdot \text{ok}\}$
  lock[src].release();
  lock[tgt].release();
  $\{I(s, \rho) \wedge \text{ownedby}(s, \alpha) = \emptyset \wedge \exists p.\pi_\alpha(\rho) = p \cdot \text{ok}\}$
}
$\{I(s, \rho) \wedge \text{ownedby}(s, \alpha) = \emptyset \wedge \text{returned}[\text{swap}](\Delta, s, \rho)\}$
```

Fig. 25. Proof of file system - swap

We first give the invariant of this object,

$$I(s, \rho) \iff \text{state}^\rho(\rho) = \text{state}^s(s)$$

We further require that at the boundary of overlay methods, all disk blocks match the content of the possibility,

$$P^f(\Delta, s, \rho) \iff I(s, \rho) \wedge \forall i.0 \leq i < N \implies \text{eval}_{\text{DiskBlock}^\lightning}(s \restriction_{\text{DiskBlock}[N]_i^\lightning}) = \text{state}^\rho(\rho)$$

$$(-)\ Q^f\ (\Delta, s', \rho') \iff I(s', \rho') \wedge \forall i.0 \leq i < N \implies \text{eval}_{\text{DiskBlock}^\lightning}(s' \restriction_{\text{DiskBlock}[N]_i^\lightning}) = \text{state}^\rho(\rho')$$

```
$\{I_{\lightning}(s,\rho)\}$
recovery(){
  for(i = 0; i < |agents|; i++){
    $\left\{I_{\lightning}(s,\rho) \land \forall j. 0 \le j < i \implies \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[j]] = \mathsf{None}\right\}$
    entry ← log.get(agents[i]);
    if(entry = None)
      continue;
    (src, tgt, src_f, tgt_f, src_file, tgt_file) ← entry;
    $\left\{I_{\lightning}(s,\rho) \land \left(\begin{array}{c} \forall j. 0 \le j < i \implies \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[j]] = \mathsf{None} \land \\ (\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}, \mathsf{src\_file}, \mathsf{tgt\_file}) = \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[i]] \land \\ \exists p. \pi_{\alpha}(\rho)\restriction_{\mathsf{Log}} = p \cdot \mathsf{swap}(\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}) \end{array}\right)\right\}$
    src_inode ← (folder_inode)disk.read(src);
    tgt_inode ← (folder_inode)disk.read(tgt);
    $\left\{I_{\lightning}(s,\rho) \land \left(\begin{array}{c} \forall j. 0 \le j < i \implies \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[j]] = \mathsf{None} \land \\ (\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}, \mathsf{src\_file}, \mathsf{tgt\_file}) = \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[i]] \land \\ \exists p. \pi_{\alpha}(\rho)\restriction_{\mathsf{Log}} = p \cdot \mathsf{swap}(\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}) \land \\ \mathsf{state}^{s}(s)[\mathsf{src}] = \mathsf{src\_inode} \land \mathsf{state}^{s}(s)[\mathsf{tgt}] = \mathsf{tgt\_inode} \end{array}\right)\right\}$
    disk.write(tgt, tgt_inode[tgt_f ↦ src_file]);
    $\left\{I_{\lightning}(s,\rho) \land \left(\begin{array}{c} \forall j. 0 \le j < i \implies \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[j]] = \mathsf{None} \land \\ (\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}, \mathsf{src\_file}, \mathsf{tgt\_file}) = \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[i]] \land \\ \exists p. \pi_{\alpha}(\rho)\restriction_{\mathsf{Log}} = p \cdot \mathsf{swap}(\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}) \land \\ \mathsf{state}^{s}(s)[\mathsf{src}] = \mathsf{src\_inode} \land \mathsf{state}^{s}(s)[\mathsf{tgt}] = \mathsf{tgt\_inode}[\mathsf{tgt\_f} \mapsto \mathsf{src\_file}] \end{array}\right)\right\}$
    disk.write(src, src_inode[src_f ↦ tgt_file]);
    $\left\{I_{\lightning}(s,\rho) \land \left(\begin{array}{c} \forall j. 0 \le j < i \implies \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[j]] = \mathsf{None} \land \\ (\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}, \mathsf{src\_file}, \mathsf{tgt\_file}) = \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[i]] \land \\ \exists p. \pi_{\alpha}(\rho)\restriction_{\mathsf{Log}} = p \cdot \mathsf{swap}(\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}) \land \\ \mathsf{state}^{s}(s)[\mathsf{src}] = \mathsf{src\_inode}[\mathsf{src\_f} \mapsto \mathsf{tgt\_file}] \land \mathsf{state}^{s}(s)[\mathsf{tgt}] = \mathsf{tgt\_inode}[\mathsf{tgt\_f} \mapsto \mathsf{src\_file}] \end{array}\right)\right\}$
    log.remove(agents[i]);
    $\left\{I_{\lightning}(s,\rho) \land \left(\begin{array}{c} \forall j. 0 \le j \le i \implies \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[j]] = \mathsf{None} \land \\ \exists p. \pi_{\alpha}(\rho)\restriction_{\mathsf{Log}} = p \cdot \mathsf{swap}(\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}) \land \\ \mathsf{state}^{s}(s)[\mathsf{src}] = \mathsf{src\_inode}[\mathsf{src\_f} \mapsto \mathsf{tgt\_file}] \land \mathsf{state}^{s}(s)[\mathsf{tgt}] = \mathsf{tgt\_inode}[\mathsf{tgt\_f} \mapsto \mathsf{src\_file}] \end{array}\right)\right\}$

    $\left\{I_{\lightning}(s',\rho') \land \left(\begin{array}{c} \forall j. 0 \le j \le i \implies \mathsf{eval}_{\mathsf{Log}\lightning}(s'\restriction_{\mathsf{Log}\lightning})[\mathsf{agents}[j]] = \mathsf{None} \land \\ \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{swap}(\mathsf{src}, \mathsf{tgt}, \mathsf{src\_f}, \mathsf{tgt\_f}) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \sqsubseteq \rho' \end{array}\right)\right\}$
  }
  $\left\{I_{\lightning}(s,\rho) \land \forall \alpha \in \mathsf{agents}. \mathsf{eval}_{\mathsf{Log}\lightning}(s\restriction_{\mathsf{Log}\lightning})[\alpha] = \mathsf{None}\right\}$
}
$\{I(s,\rho)\}$
```

Fig. 26. Proof of file system - recovery

We can now define the crash-postcondition as preservation of invariant,

$$(s,\rho)\ Q_{\lightning}\ (s',\rho') \iff I(s',\rho')$$

since the invariant is preserved, the sole purpose of the recovery method is to re-establish the universal precondition of normal routines.

```
M_DiskBlock :
Import blocks: DiskBlock[N]

void write(block_data data) {
  for (i ← 0; i < N; i ← i + 1) {
    blocks[i].write(data);
  }
}

block_data read() {
  i ← random();
  block_data data ← blocks[i].read();
  return data;
}

void recover() {
  data ← disks[0].read();
  for (i ← 0; i < N; i ← i + 1) {
    disks[i].write(data);
  }
}
```

Fig. 27. Implementation for Replicated Disks

The rely and guarantee condition is trivial since the client policy effectively disallow any type of interleaving,

$$(s, \rho)\ \mathcal{G}[\alpha]\ (s', \rho') \iff \mathbf{id}$$

$$\mathcal{R}[\alpha] \triangleq \bigcup_{\alpha' \in \Upsilon, \alpha' \neq \alpha} \mathcal{G}[\alpha'] \cup \mathsf{invoke}_{\alpha'}(-) \cup \mathsf{return}_{\alpha'}(-)$$

With everything defined, the step through proofs are given in Figure 28, Figure 29, and Figure 30. For brevity, we use $\mathsf{eval}_{\mathsf{DiskBlock}[N]_i^{\lightning}}(s)$ as a abbreviation for $\mathsf{eval}_{\mathsf{Disk}^{\lightning}}(s \restriction_{\mathsf{DiskBlock}[N]_i})$.

```
{$I(s, \rho)$}
write(data){
  disks[0].write(data);
  {$I(s, \rho) \wedge \mathsf{state}^s(s) = \mathsf{state}^\rho(\rho) = \mathsf{data}$}

  $\left\{ I(s', \rho') \wedge \begin{pmatrix} \mathsf{state}^s(s') = \mathsf{state}^\rho(\rho') = \mathsf{data} \wedge \\ \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{write}(\mathsf{data}) \cdot \boldsymbol{\alpha}{:}\mathsf{ok} \sqsubseteq \rho' \end{pmatrix} \right\}$
  {$I(s, \rho) \wedge (\exists p. \pi_\alpha(\rho) = p \cdot \mathsf{ok}) \wedge \mathsf{state}^s(s) = \mathsf{state}^\rho(\rho) = \mathsf{data}$}
  for($i \leftarrow 1; i < N; i \leftarrow i + 1$)
    {$I(s, \rho) \wedge (\exists p. \pi_\alpha(\rho) = p \cdot \mathsf{ok}) \wedge \forall i'. 0 \leq i' < i \implies \mathsf{eval}_{\mathsf{DiskBlock}[N]_{i'}^{\lightning}}(s) = \mathsf{state}^\rho(\rho) = \mathsf{data}$}
    disks[$i$].write(data);
    {$I(s, \rho) \wedge (\exists p. \pi_\alpha(\rho) = p \cdot \mathsf{ok}) \wedge \forall i'. 0 \leq i' \leq i \implies \mathsf{eval}_{\mathsf{DiskBlock}[N]_{i'}^{\lightning}}(s) = \mathsf{state}^\rho(\rho) = \mathsf{data}$}
  {$I(s, \rho) \wedge (\exists p. \pi_\alpha(\rho) = p \cdot \mathsf{ok}) \wedge \forall i'. 0 \leq i' \leq N \implies \mathsf{eval}_{\mathsf{DiskBlock}[N]_{i'}^{\lightning}}(s) = \mathsf{state}^\rho(\rho) = \mathsf{data}$}
}
{$I(s, \rho) \wedge \mathsf{returned}[\mathsf{write}](\Delta, s, \rho)$}
```

Fig. 28. Proof of Replicated Disks - write

```
$\{I(s, \rho)\}$
read() {
   $i \leftarrow \mathsf{random}(0, N-1)$;
   $\{I(s, \rho) \wedge 0 \leq i < N\}$
   data ← disks[$i$].read();
   $\{I(s, \rho) \wedge 0 \leq i < N \wedge \mathsf{state}^{\rho}(\rho) = \mathsf{data}\}$

   $\{I(s', \rho') \wedge \mathsf{state}^{\rho}(\rho')[b] = \mathsf{data} \wedge \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{read} \cdot \boldsymbol{\alpha}{:}\mathsf{data} \sqsubseteq \rho'\}$
   $\{I(s, \rho) \wedge (\exists p. \pi_{\alpha}(\rho) = p \cdot \mathsf{data})\}$
   return data;
}
$\{I(s, \rho) \wedge \mathsf{returned}[\mathsf{read}](\Delta, s, \rho)\}$
```

Fig. 29. Proof of Replicated Disks - read

```
$\{I(s, \rho)\}$
recover() {
   data ← disks[0].read();
   $\{I(s, \rho) \wedge \mathsf{data} = \mathsf{state}^{\rho}(\rho)\}$
   for($i \leftarrow 1; i < N; i \leftarrow i + 1$)
      $\left\{I(s, \rho) \wedge \forall i'. 0 \leq i' < i \implies \mathsf{eval}_{\mathsf{DiskBlock}[N]^{\lightning}_{i'}}(s) = \mathsf{state}^{\rho}(\rho) = \mathsf{data}\right\}$
      disks[$i$].write($b$, data);
      $\left\{I(s, \rho) \wedge \forall i'. 0 \leq i' \leq i \implies \mathsf{eval}_{\mathsf{DiskBlock}[N]^{\lightning}_{i'}}(s) = \mathsf{state}^{\rho}(\rho) = \mathsf{data}\right\}$
   $\left\{I(s, \rho) \wedge \forall i'. 0 \leq i' < N \implies \mathsf{eval}_{\mathsf{DiskBlock}[N]^{\lightning}_{i'}}(s) = \mathsf{state}^{\rho}(\rho) = \mathsf{data}\right\}$
}
$\left\{I(s, \rho) \wedge \forall i'. 0 \leq i' < N \implies \mathsf{eval}_{\mathsf{DiskBlock}[N]^{\lightning}_{i'}}(s) = \mathsf{state}^{\rho}(\rho) = \mathsf{data}\right\}$
```

Fig. 30. Proof of Replicated Disks - recovery

I.3.5 Write-Ahead Log Implementation. The implementation of the write-ahead log used in the file system is presented in Figure 31. We omit the verification as it is straightforward in terms of crash linearizability: all operations are immediately persisted and there is no in-between states for the disk block.

Because the disk is equivalent to the horizontal composition of its blocks:

$$\nu_{\mathsf{Disk}^{\lightning}} \cong \otimes_{b \in \mathsf{block\_id}} \nu_{\mathsf{DiskBlock}^{\lightning}}$$

It follows that we may separate one location from the disk for the log, as follows:

$$\begin{aligned}
&(\mathsf{vol}(\nu_{\mathsf{Buffer}}) \otimes \mathsf{vol}(\nu_{\mathsf{Lock}})) \otimes \nu_{\mathsf{Disk}^{\lightning}} \\
&\qquad \cong (\mathsf{vol}(\nu_{\mathsf{Buffer}}) \otimes \mathsf{vol}(\nu_{\mathsf{Lock}})) \otimes (\otimes_{b \in \mathsf{block\_id}} \nu_{\mathsf{DiskBlock}^{\lightning}}) \\
&\qquad \cong (\nu_{\mathsf{DiskBlock}^{\lightning}} \otimes \mathsf{vol}(\nu_{\mathsf{Buffer}}) \otimes \mathsf{vol}(\nu_{\mathsf{Lock}})) \otimes \otimes_{b \in \mathsf{block\_id} \backslash \mathsf{logblk}} \nu_{\mathsf{DiskBlock}^{\lightning}}
\end{aligned}$$

so that

$$(\mathsf{vol}(\nu_{\mathsf{Buffer}}) \otimes \mathsf{vol}(\nu_{\mathsf{Lock}}) \otimes \nu_{\mathsf{Disk}^{\lightning}}); (M_{\mathsf{Log}} \otimes \mathsf{crashcopy})$$

is linearizable to $\nu_{\mathsf{Log}^{\lightning}} \otimes (\otimes_{b \in \mathsf{block\_id} \backslash \mathsf{log_b lk}} \nu_{\mathsf{DiskBlock}^{\lightning}})$ where logblk is the block id where the log is located. That is to say, we obtain a log together with a disk with size one block less than before, such that in the underlay the log lives in the same disk.

```
M_Log :
Import block:DiskBlock
Import buffer:Buffer
Import lock:Lock

void insert(block_id a, block_id b, file_id c, file_id d, block_id e, block_id f) {
  lock.acquire();
  buffer[α] = (a, b, c, d, e, f);
  block.write(buffer);
  lock.release();
}

void remove(agent_id agent) {
  lock.acquire();
  buffer[agent] = None;
  block.write(buffer);
  lock.release();
}

void get(agent_id agent) {
  return buffer[agent];
}

void recovery() {
  buffer = (log) block.read();
}
```

Fig. 31. Implementation of the Write Ahead Log

J Proofs

J.1 Basic Semicategorical Structure

In the following we assume that the operation $-;-$ is defined arbitrary sets of plays, instead of just on strategies. The definition is exactly the same. Given $s \in P_{A \multimap B}$ and $t \in P_{B \multimap C}$ we write $s;t$ for $\{s\};\{t\}$. We also take the convention of writing, for $s \in P_A$ with $A \in \underline{\mathbf{Crash}}$,

$$s = s_1 \cdot \lightning \cdot s_2 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where for each i, $s_i = \mathrm{epo}_i(s)$.

The following important lemma characterizes composition in $\underline{\mathbf{Crash}}$ using composition in $\underline{\mathbf{Conc}}$.

Lemma J.1. *Given strategies $\sigma : A \multimap B$, $\tau : B \multimap C \in \underline{\mathbf{Crash}}$, for any play $u \in \sigma;\tau$, there exists $s \in \sigma, t \in \tau$ such that*

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \qquad t = t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1}$$

and, for all i,

$$s_i \in \mathbb{P}^{\mathsf{conc}}_{A^{\Upsilon} \multimap B^{\Upsilon}} \qquad \text{and} \qquad t_i \in \mathbb{P}^{\mathsf{conc}}_{B^{\Upsilon} \multimap C^{\Upsilon}}$$

and such that u decomposes as

$$u = u_1 \cdot \lightning \cdot u_2 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1}$$

with $u_i \in s_i;t_i$ for all i.

Proof. Suppose $u \in \sigma;\tau$. By definition there exists $u' \in \mathrm{int}(\sigma,\tau)$ such that $u' \upharpoonright_{A,B,-} \in \sigma$, $u' \upharpoonright_{-,B,C} \in \tau$ and $u' \upharpoonright_{A,-,C} = u$, we claim that assigning $s := u' \upharpoonright_{A,B,-}$ and $t := u' \upharpoonright_{-,B,C}$ the claim is proven.

Since all three of u', s, and t are well-formed it follows that they can be written as (it will soon be clear why all plays feature the same number of epochs):

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \qquad t = t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \qquad u' = u'_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u'_{n+1}$$

So let $u_i = u'_i \upharpoonright_{A,-,C}$. By definition we know $u' \upharpoonright_{-,B,-} = s \upharpoonright_{-,B} = t \upharpoonright_{B,-}$, so it follows that the i-th crash signal in s matches t with the i-th crash signal in u', and that they have the same number of crash signals. Furthermore, since $s = u' \upharpoonright_{A,B,-}$, the i-th epoch of u', i.e. u'_i, projects to the i−th epoch of s, i.e. $u'_i \upharpoonright_{A,B,-} = s_i$. Similarly, $u'_i \upharpoonright_{-,B,C} = t_i$. Hence, it follows that $u'_i \upharpoonright_{A,-,C} = u_i \in s_i; t_i$. □

Proposition J.2. *strategy composition is well-defined and associative.*

Proof. **Well-defined** Suppose $\sigma : A \multimap B$ and $\tau : B \multimap C$, then we have $\epsilon \in \sigma$ and $\epsilon \in \tau$. Then taking

$$\epsilon \in \mathsf{int}(A, B, C)$$

we have $\epsilon \upharpoonright_{A,-,C} = \epsilon \in \mathbb{P}^{\lightning}_{A \multimap C}$ which implies $\epsilon \in \sigma; \tau$. So $\sigma; \tau$ is non-empty

Now suppose $s \in \sigma; \tau$ and that $p \sqsubseteq s$, then there exists $s' \in \mathsf{int}(\sigma, \tau)$ such that $s' \upharpoonright_{A,-,C} = s$. In particular $p \sqsubseteq s' \upharpoonright_{A,-,C}$. Hence there exists $p' \sqsubseteq s'$ such that $p' \upharpoonright_{A,-,C} = p$. Since $s' \upharpoonright_{A,B,-} \in \sigma$ and $s' \upharpoonright_{-,B,C} \in \tau$ and both σ, τ are prefix-closed, so $p' \upharpoonright_{A,B,-} \in \sigma$ and $p' \upharpoonright_{-,B,C} \in \tau$. So $p' \in \mathsf{int}(\sigma, \tau)$. Since $\mathbb{P}^{\lightning}_{A \multimap C}$ is prefix-closed, $p' \upharpoonright_{A,-,C} \sqsubseteq s' \upharpoonright_{A,-,C} \in \mathbb{P}^{\lightning}_{A \multimap C}$ so $p' \upharpoonright_{A,-,C} \in \mathbb{P}^{\lightning}_{A \multimap C}$. Hence $p \in \sigma; \tau$.

Lastly, we show $\sigma; \tau$ is $\lightning$−receptive. Suppose $s \in \sigma; \tau$. So there exists $s' \in \mathsf{int}(\sigma, \tau)$ such that $s' \upharpoonright_{A,-,C} = s$. Note then that we have $s' \cdot \lightning \in \mathsf{int}(A, B, C)$. By $\lightning$−receptivity, it holds that $s' \cdot \lightning \upharpoonright_{A,B,-} \in \sigma$ and $s' \cdot \lightning \upharpoonright_{-,B,C} \in \tau$. So it follows that $s' \cdot \lightning \upharpoonright_{A,-,C} = s \cdot \lightning \in \sigma; \tau$.

Associative Suppose $\sigma : A \multimap B, \tau : B \multimap C$ and $\rho : C \multimap D$, fix $s \upharpoonright_{A,-,D} \in (\sigma; \tau); \rho$ where $u \in \mathsf{int}((\sigma; \tau), \rho)$. Applying J.1 between the composition of $\sigma; \tau$ and ρ, and then applying the lemma again to decompose its projection to $\sigma; \tau$, we obtain that $u \upharpoonright_{A,-,D}$ can be written as

$$u_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1}$$

and $u'_i \in (s_i; t_i); r_i$ where $s_i \in \mathbb{P}^{\mathsf{conc}}_{A^\Upsilon \multimap B^\Upsilon}$, $t_i \in \mathbb{P}^{\mathsf{conc}}_{B^\Upsilon \multimap C^\Upsilon}$ and $r_i \in \mathbb{P}^{\mathsf{conc}}_{C^\Upsilon \multimap D^\Upsilon}$, with, futhermore

$$s := s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \in \sigma$$

$$t := t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \in \tau$$

$$r := r_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot r_{n+1} \in \rho$$

From concurrent games [31] we already know that $(s_i; t_i); r_i = s_i; (t_i; r_i)$. So there exists $u'_i \in \mathsf{int}(s_i, t_i; r_i)$ such that $u'_i \upharpoonright_{A,-,D} = u_i$, $u'_i \upharpoonright_{A,B,-} = s_i$ and $u'_i \upharpoonright_{-,B,D} \in t_i; r_i$ Now let's define

$$u'' := u'_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u'_{n+1}$$

Now we have $u'' \upharpoonright_{A,B,-} = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$ and $u'' \upharpoonright_{-,B,D} \in t; r \subseteq \tau; \rho$. So $u = u'' \upharpoonright_{A,-,D} \in \sigma; (\tau; \rho)$.

The other direction is analogous.

□

We also prove that composition is monotonic and join-preserving, the main requirement to obtain an enriched semicategory.

Proposition J.3. *For strategies*

$$\sigma : A \multimap B, \qquad \tau : B \multimap C$$

the following hold:

(1) *if $\sigma \subseteq \sigma' : A \multimap B$ and $\tau \subseteq \tau' : B \multimap C$ then $\sigma; \tau \subseteq \sigma'; \tau'$*

(2) Given a family of strategies $(\sigma_i : A \multimap B)_{i \in I}$ it holds that

$$\left(\bigcup_{i \in I} \sigma_i\right); \tau = \bigcup_{i \in I} (\sigma_i; \tau)$$

(3) Given a family of strategies $(\tau_i : B \multimap C)_{i \in I}$ it holds that

$$\sigma; \left(\bigcup_{i \in I} \tau_i\right) = \bigcup_{i \in I} (\sigma; \tau_i)$$

Proof. (1) Suppose $s \upharpoonright_{A,-,C} \in \sigma; \tau$ then

$$s \upharpoonright_{A,B,-} \in \sigma \Longrightarrow s \upharpoonright_{A,B,-} \in \sigma'$$
$$s \upharpoonright_{-,B,C} \in \tau \Longrightarrow s \upharpoonright_{-,B,C} \in \tau'$$

Also since $s \in \mathsf{int}(A, B, C), s \upharpoonright_{A,-,C} \in \mathbb{P}^{\lightning}_{A \multimap C}$ so $s \upharpoonright_{A,-,C} \in \sigma'; \tau'$.

(2) For one direction, since we have $\sigma_i \subseteq \cup_{i \in I} \sigma_i$ so

$$\sigma_i; \tau \subseteq \left(\bigcup_{i \in I} \sigma_i\right); \tau$$

hence

$$\bigcup_{i \in I} (\sigma_i; \tau) \subseteq \left(\bigcup_{i \in I} \sigma_i\right); \tau$$

For the other direction, suppose $s \upharpoonright_{A,-,C} \in \left(\bigcup_{i \in I} \sigma_i\right); \tau$ which means $s \upharpoonright_{A,B,-} \in \left(\bigcup_{i \in I} \sigma_i\right)$ and $s \upharpoonright_{-,B,C} \in \tau$.
so there exists i such that $s \upharpoonright_{A,B,-} \in \sigma_i$, and therefore $s \upharpoonright_{A,-,C} \in \sigma_i; \tau \subseteq \bigcup_{i \in I} (\sigma_i; \tau)$ so that

$$s \upharpoonright_{A,-,C} \in \bigcup_{i \in I} (\sigma_i; \tau)$$

(3) For one direction we have $\tau_i \subseteq \bigcup_{i \in I} \tau_i$ so

$$\sigma; \tau_i \subseteq \sigma; \left(\bigcup_{i \in I} \tau_i\right)$$

hence

$$\bigcup_{i \in I} (\sigma; \tau_i) \subseteq \sigma; \left(\bigcup_{i \in I} \tau_i\right)$$

For the other direction, suppose $s \upharpoonright_{A,-,C} \in \sigma; \left(\bigcup_{i \in I} \tau_i\right)$ which means $s \upharpoonright_{A,B,-} \in \sigma$ and $s \upharpoonright_{-,B,C} \in \bigcup_{i \in I} \tau_i$.
so there exists i such that $s \upharpoonright_{-,B,C} \in \tau_i$, and therefore $s \upharpoonright_{A,-,C} \in \sigma; \tau_i \subseteq \bigcup_{i \in I} (\sigma; \tau_i)$ so that

$$s \upharpoonright_{A,-,C} \in \bigcup_{i \in I} (\sigma; \tau_i)$$

□

J.2 Volatile Lift and Idempotence

Proposition J.4.

$$\mathsf{vol}(-) : \underline{\mathbf{Conc}} \to \underline{\mathbf{Crash}}$$

is a semi-functor.

Proof. For given

$$\sigma : A \multimap B \in \underline{\mathbf{Conc}} \qquad\qquad \sigma' : B \multimap C \in \underline{\mathbf{Conc}}$$

we want to show

$$\mathsf{vol}(\sigma); \mathsf{vol}(\sigma') = \mathsf{vol}(\sigma; \sigma')$$

For one direction, fix $s \in \mathsf{vol}(\sigma); \mathsf{vol}(\sigma')$. By lemma J.1, we have that s can be decomposed as

$$s := s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where for each i, $s_i \in \sigma; \sigma'$. It then follows immediately by the definition of $\mathsf{vol}(-)$ that $s \in \mathsf{vol}(\sigma; \sigma')$.

For the other direction, fix $s \in \mathsf{vol}(\sigma; \sigma')$. We have that, by well-formedness, s can be decomposed as

$$s := s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where for each i $s_i \in \sigma; \sigma'$. So there exists s'_i such that $s'_i \upharpoonright_{A,B,-} \in \sigma$ and $s'_i \upharpoonright_{-,B,C} \in \sigma'$.

Let

$$t := s'_1 \upharpoonright_{A,B,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s'_{n+1} \upharpoonright_{A,B,-}$$
$$t' := s'_1 \upharpoonright_{-,B,C} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s'_{n+1} \upharpoonright_{-,B,C}$$

It is immediate from the definition of $\mathsf{vol}(-)$ that $t \in \mathsf{vol}(\sigma)$, $t' \in \mathsf{vol}(\sigma')$, and so it follows that $s \in \mathsf{vol}(\sigma); \mathsf{vol}(\sigma')$.

We also need to show $\mathsf{vol}(\sigma)$ satisfies $\lightning$-receptivity. Suppose $s \in \mathsf{vol}(\sigma)$, $\lightning \in M^{\lightning}_{A^{\lightning} \multimap B^{\lightning}}$, $s \cdot \lightning \in P_{A^{\lightning} \multimap B^{\lightning}}$, by definition we know s may be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where for each i, $s_i \in \sigma$ and $r_i \in R$. Since $\epsilon \in \sigma$ so we could set

$$s' := s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \cdot \lightning \cdot \epsilon$$

we obtain by definition that $s' \in \mathsf{vol}(\sigma)$. □

Proposition J.5. *The copycat strategy* $\mathsf{crashcopy}_A$ *is idempotent, i.e.*

$$\mathsf{crashcopy}_A; \mathsf{crashcopy}_A = \mathsf{crashcopy}_A$$

Proof. Note first that it is immediate from the definition of $\mathsf{crashcopy}_{A^{\lightning}}$ that

$$\mathsf{crashcopy}_{A^{\lightning}} = \mathsf{vol}(\mathsf{ccopy}_A)$$

Then, observe that

$$\begin{aligned}
\mathsf{crashcopy}_A; \mathsf{crashcopy}_A &= \mathsf{vol}(\mathsf{ccopy}_{A^{\Upsilon}}); \mathsf{vol}(\mathsf{ccopy}_{A^{\Upsilon}}) && \text{(Def. of crashcopy)} \\
&= \mathsf{vol}(\mathsf{ccopy}_{A^{\Upsilon}}; \mathsf{ccopy}_{A^{\Upsilon}}) && \text{(Prop. J.4)} \\
&= \mathsf{vol}(\mathsf{ccopy}_{A^{\Upsilon}}) && \text{(Prop. ccopy is idempotent)} \\
&= \mathsf{crashcopy}_A && \text{(Def. of crashcopy)}
\end{aligned}$$

□

Proposition J.6. *The restriction*

$$\mathsf{vol}(-) : \mathbf{Conc} \to \mathbf{Crash}$$

of $\mathsf{vol}(-)$ *defines a functor.*

Proof. We already argued that

$$\mathsf{crashcopy}_{A^{\lightning}} = \mathsf{vol}(\mathsf{ccopy}_A)$$

in the proof of Prop. J.5. It remains to show that whenever σ is saturated with respect to ccopy i.e. $\mathsf{ccopy}_A; \sigma; \mathsf{ccopy}_B = \sigma$ then $\mathsf{vol}(\sigma)$ is saturated with respect to crashcopy. For that, just note that:

$$\begin{aligned}\mathsf{crashcopy}_{A^{\lightning}}; \mathsf{vol}(\sigma); \mathsf{crashcopy}_{B^{\lightning}} &= \mathsf{vol}(\mathsf{ccopy}_A); \mathsf{vol}(\sigma); \mathsf{vol}(\mathsf{ccopy}_B)\\ &= \mathsf{vol}(\mathsf{ccopy}_A; \sigma; \mathsf{ccopy}_B)\\ &= \mathsf{vol}(\sigma)\end{aligned}$$

□

J.3 Concrete Saturation for Crash-Aware Games

We start by providing the statement of the main result of this section, which concretely characterizes what a saturated strategy in **Crash** looks like.

Proposition J.7. *A strategy* $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$ *is saturated with respect to* crashcopy *if and only if it is*

O-**receptive:** $\forall s \in \sigma. \forall \alpha \in \Upsilon. \forall m \in M^{\boldsymbol{\alpha}:O}_{A \multimap B}. \exists 1 \leq i \leq \|s\|. \mathsf{epo}_i(s) \cdot m \in P_{A^{\Upsilon} \multimap B^{\Upsilon}} \implies$

$$\mathsf{epo}_1(s) \cdot \lightning \cdot \ldots \cdot \lightning \cdot \mathsf{epo}_i(s) \cdot m \cdot \lightning \cdot \ldots \cdot \lightning \cdot \mathsf{epo}_{\|s\|}(s) \in \sigma$$

$\rightsquigarrow$-**closed:** $\forall s \in \sigma. \forall t \in P_{A \multimap B}. t \rightsquigarrow_{A \multimap B} s \implies t \in \sigma$
P-**delaying:** $\forall s \in \sigma. \forall m \in M^P_{A \multimap B}. s = p \cdot m \cdot \lightning \cdot t \Rightarrow p \cdot \lightning \cdot t \in \sigma$

One might think that it is possible to directly use the proofs of concrete saturation for **Conc** from Oliveira Vale et al. [31] across each epoch of a crash-aware strategy to obtain the corresponding concrete saturation result for **Crash**. It turns out that those theorems made a key use of prefix-closure of **Conc** strategies at specific points that make the proofs not translate to our setting, as strategies in **Crash** do not have per-epoch prefix-closure. Because of this, we must reprove some of the results appearing in their appendix, including the Synchronization Lemma.

In the following, we refer to O-receptive, P-delaying closure of a set of plays $S \subseteq P_A$ with $A \in \underline{\mathbf{Conc}}$ by $\mathsf{dr}\,(S)$. That is, $\mathsf{dr}\,(S)$ is the smallest set of plays of P_A such that

$$\forall s \in \mathsf{dr}\,(S)\,. \forall m \in M^O_A. s \cdot m \in P_A \implies s \cdot m \in \mathsf{dr}\,(S)$$

$$\forall m \in M^P_A. \forall s \cdot t \in P_A. s \cdot m \cdot t \in \mathsf{dr}\,(S) \implies s \cdot t \in \mathsf{dr}\,(S)$$

The following few re-statements of propositions from Oliveira Vale et al. [31] admit essentially the same proofs as might be found there.

Proposition J.8 (Synchronization Lemma). *Let* $s = p \cdot \boldsymbol{\alpha}{:}m \cdot \boldsymbol{\alpha'}{:}m' \cdot p'$ *be a play of* $A \multimap B \in \mathbf{Conc}$. *Let* $S = \mathsf{dr}\,(p \cdot m \cdot m' \cdot p')$. *Then,*

$$p \cdot m' \cdot m \cdot p' \in \mathsf{ccopy}_A; S; \mathsf{ccopy}_B \iff m' \cdot m \rightsquigarrow_{A \multimap B} m \cdot m'$$

Corollary J.9. *Let* $s \in P_{A \multimap B}$ *with* $A \multimap B \in \mathbf{Conc}$ *and that* t *is a play such that*

$$\forall \alpha \in \Upsilon. \pi_\alpha(t) = \pi_\alpha(s)$$

and moreover

$$t \in \mathsf{ccopy}_A; \mathsf{dr}\,(s)\,; \mathsf{ccopy}_B$$

then,

$$t \rightsquigarrow_{A \multimap B} s$$

Lemma J.10. *For every set S of plays of $P_{A \multimap B}$:*

$$S \subseteq \mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B$$

The first lemma we need that does require a novel proof is the following.

Lemma J.11. *For any O-receptive and P-delaying set S of plays of $P_{A \multimap B}$ with $A \multimap B \in$ **Conc** it holds that for all $t \in \mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B$ there exists $s \in S$ such that $t \in \mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B$ and $\forall \alpha \in \Upsilon . \pi_\alpha(t) = \pi_\alpha(s)$.*

Proof. Fix $t \in \mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B$ there exists t' such that $t' \restriction_{A,A,-,-} \in \mathsf{ccopy}_A$, $t' \restriction_{-,A,B,-} \in S$ and $t' \restriction_{-,-,B,B} \in \mathsf{ccopy}_B$.

Now, notice that for any $\alpha \in \Upsilon$ there are four possibilities for the lengths of $t' \restriction_{A,A,-,-}$ and $t' \restriction_{-,-,B,B}$.

Both are even-length: It is immediate from the definition of ccopy that $\pi_\alpha(t) = \pi_\alpha(t' \restriction_{-,A,B,-})$.

$t' \restriction_{A,A,-,-}$ **is even-length and** $t' \restriction_{-,-,B,B}$ **is odd-length:** Then either $\pi_\alpha(t' \restriction_{-,A,B,-})$ differs from $\pi_\alpha(t)$ by having an extra O-move in the end or by missing a P-move. Either way, we can find a new $t'' \in S$ that either adds the required O-move or removes the missing P-move by O-receptivity or P-delaying respectively.

$t' \restriction_{A,A,-,-}$ **is odd-length and** $t' \restriction_{-,-,B,B}$ **is even-length:** This case is similar to the previous one.

Both length are odd-length: This case is impossible by the switching conditions.

□

Proposition J.12. *An O-receptive and P-delaying set S of plays $P_{A \multimap B}$, for $A \multimap B \in$ **Conc** is saturated with respect to* ccopy *if and only if*

$$\forall s \in \sigma . \forall t \in P_{A \multimap B} . t \rightsquigarrow_{A \multimap B} s \Rightarrow t \in \sigma$$

Proof. Suppose S is saturated. It follows that if $s \in S = \mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B$ and $t \rightsquigarrow_{A \multimap B} s$ then there is a sequence of single steps:

$$t = t_0 \rightsquigarrow_{A \multimap B} t_1 \rightsquigarrow_{A \multimap B} \ldots \rightsquigarrow_{A \multimap B} t_n = s$$

then by applying the Sychronization Lemma (Prop. J.8) starting with

$$t_{n-1} \rightsquigarrow_{A \multimap B} s$$

to conclude that

$$t_{n-1} \in \mathsf{ccopy}_A ; \mathsf{dr}\,(s)\,; \mathsf{ccopy}_B \subseteq \sigma$$

in a finite number of applications we obtain that

$$t = t_0 \in \mathsf{ccopy}_A ; \mathsf{dr}\,(t_1)\,; \mathsf{ccopy}_A \subseteq \mathsf{ccopy}_A ; \mathsf{dr}\,(s)\,; \mathsf{ccopy}_B \subseteq S$$

as desired.

Note that for every set of plays S that satisfies O-receptivity and P-delaying it holds that:

$$S = \bigcup_{s \in S} \mathsf{dr}\,(s)$$

But

$$\mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B = \bigcup_{s \in S} \mathsf{ccopy}_A ; \mathsf{dr}\,(s)\,; \mathsf{ccopy}_B$$

by the fact that composition is join-preserving. Hence,

$$t \in \mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B \iff \exists s \in S . t \in \mathsf{ccopy}_A ; \mathsf{dr}\,(s)\,; \mathsf{ccopy}_B$$

moreover, by J.11, s can be chosen so that

$$\forall \alpha \in \Upsilon . \pi_\alpha(t) = \pi_\alpha(s)$$

by corollary to the Synchronization Lemma (Prop. J.8) it follows that

$$t \in \mathsf{ccopy}_A ; \mathsf{dr}\,(s)\,; \mathsf{ccopy}_B \iff t \rightsquigarrow_{A \multimap B} s$$

And hence

$$t \in \mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B \iff \exists s \in S . t \rightsquigarrow_{A \multimap B} s$$

So, suppose $t \in \mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B$. Then, there is some $s \in \sigma$ such that $t \rightsquigarrow_{A \multimap B} s$ and hence by assumption $t \in \sigma$. Hence,

$$\mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B \subseteq S$$

the reverse containment is exactly lemma J.10 so that it follows that

$$\mathsf{ccopy}_A ; S ; \mathsf{ccopy}_B = S$$

and hence S is saturated. □

Finally, toward the concrete saturation theorem.

Lemma J.13. *For every strategy $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$ we have*

$$\sigma \subseteq \mathsf{crashcopy}_A ; \sigma ; \mathsf{crashcopy}_B$$

Proof. Suppose $s \in \sigma$ then we know s can be decomposed as

$$s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where $s_i \in \mathbb{P}^{\mathsf{conc}}_{A^\Upsilon \multimap B^\Upsilon}$. Note then that we have $\mathsf{dr}\,(s_i) \subseteq \mathsf{ccopy}_{A^\Upsilon} ; \mathsf{dr}\,(s_i)\,; \mathsf{ccopy}_{B^\Upsilon}$. So there exists $t_i \upharpoonright_{A,-,-,B} \in \mathbb{P}^{\mathsf{conc}}_{A^\Upsilon \multimap B^\Upsilon}$ such that $t_i \upharpoonright_{A,A,-,-} \in \mathsf{ccopy}_{A^\Upsilon}$, $t_i \upharpoonright_{-,A,B,-} = s_i$ and $t_i \upharpoonright_{-,-,B,B} \in \mathsf{ccopy}_{B^\Upsilon}$.

So set

$$\begin{aligned} s' &:= t_1 \upharpoonright_{A,A,-,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \upharpoonright_{A,A,-,-} \\ s'' &:= t_1 \upharpoonright_{-,A,B,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \upharpoonright_{-,A,B,-} \\ s''' &:= t_1 \upharpoonright_{-,-,B,B} \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

By definition we have $s' \in \mathsf{crashcopy}_A$, $s'' \in \sigma$ and $s''' \in \mathsf{crashcopy}_B$. Hence, $s \in K_\lightning\ \sigma$ □

Proposition J.14. *A strategy $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$ is saturated with respect to* crashcopy *if and only if it is*

O-**receptive:** $\forall s \in \sigma . \forall \alpha \in \Upsilon . \forall m \in M^{\boldsymbol{\alpha}:O}_{A \multimap B} . \exists 1 \leq i \leq \|s\| . \mathsf{epo}_i(s) \cdot m \in P_{A^\Upsilon \multimap B^\Upsilon} \implies$

$$\mathsf{epo}_1(s) \cdot \lightning \cdot \ldots \cdot \lightning \cdot \mathsf{epo}_i(s) \cdot m \cdot \lightning \cdot \ldots \cdot \lightning \cdot \mathsf{epo}_{\|s\|}(s) \in \sigma$$

$\rightsquigarrow$-**closed:** $\forall s \in \sigma . \forall t \in P_{A \multimap B} . t \rightsquigarrow_{A \multimap B} s \implies t \in \sigma$

P-**delaying:** $\forall s \in \sigma . \forall m \in M^P_{A \multimap B} . s = p \cdot m \cdot \lightning \cdot t \Rightarrow p \cdot \lightning \cdot t \in \sigma$

Proof.

($\Longrightarrow$): suppose $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$ is saturated i.e. $\mathsf{crashcopy}_A ; \sigma ; \mathsf{crashcopy}_B = \sigma$. Let's first show that σ is O-receptive. Fix $s \in \sigma$ by definition s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

Note that for $m \in M^{\boldsymbol{\alpha}:O}_{A \multimap B}$, if $s_i \cdot m = \mathsf{epo}_i(s) \cdot m \in P_{A^\Upsilon \multimap B^\Upsilon}$ then $s_i \cdot m \in \mathbb{P}^{\mathsf{conc}}_{A^\Upsilon \multimap B^\Upsilon}$. Thanks to Oliveira Vale et al. [31] we know there exists u_j for each j such that $u_j \upharpoonright_{A,-,-,B} = s_j \cdot m$

if $j = i$, and otherwise $u_j \upharpoonright_{A,-,-,B} = s_j$, and $u_j \upharpoonright_{A,A,-,-} \in \mathsf{ccopy}_A$, $u_j \upharpoonright_{-,-,B,B} \in \mathsf{ccopy}_B$ and $u_i \upharpoonright_{-,A,B,-} \in \mathsf{dr}\,(s_i)$.
Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

Then, note that $u \in \mathsf{crashcopy}_A$, $u' = s$, $u'' \in \mathsf{crashcopy}_B$ and

$$s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_i \cdot m \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \in u; u'; u''$$

so that

$$s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_i \cdot m \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \in \mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B = \sigma$$

as σ is saturated.
Now let's show σ is $\rightsquigarrow$-closed. Fix $s \in \sigma$, $t \in P_{A \multimap B}$ and suppose $t \rightsquigarrow_{A \multimap B} s$. More precisely we can decompose s and t as:

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \qquad t = t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1}$$

And we have for all i, $t_i \rightsquigarrow_{A \multimap B} s_i$. Now we want to show $t \in \sigma$. Since for all i we have $t_i \rightsquigarrow_{A \multimap B} s_i$, it follows that $t_i \in \mathsf{ccopy}_A; \mathsf{dr}\,(s_i)\,; \mathsf{ccopy}_B$ which means there exists u_i such that $u_i \upharpoonright_{A,A,-,-} \in \mathsf{ccopy}_A$, $u_i \upharpoonright_{-,A,B,-} = s_i$, $u_i \upharpoonright_{-,-,B,B} \in \mathsf{ccopy}_B$ and $u_i \upharpoonright_{A,-,-,B} = t_i$. Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

we have $t \in u; u'; u''$ and $u \in \mathsf{crashcopy}_A$, $u' \in \mathsf{strat}\,(s)$ and $u'' \in \mathsf{crashcopy}_B$. So $t \in \mathsf{crashcopy}_A; \mathsf{strat}\,(s)\,; \mathsf{crashcopy}_B \subseteq \mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B = \sigma$.
Finally, we want to show σ is P-delaying. Fix $s \in \sigma$ such that moreover there is an epoch i and a P-move $m \in M^{\alpha:P}$ such that $s_i = p \cdot m$. By definition s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

From J.12 we know there exists u_j for each j such that $u_j \upharpoonright_{A,-,-,B} = p$ if $i = j$ otherwise $u_j \upharpoonright_{A,-,-,B} = s_j$, and $u_j \upharpoonright_{A,A,-,-} \in \mathsf{ccopy}_A$, $u_j \upharpoonright_{-,-,B,B} \in \mathsf{ccopy}_B$ and $u_j \upharpoonright_{-,A,B,-} \in \mathsf{dr}\left(s_j\right)$
Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

Then, note that $u \in \mathsf{crashcopy}_A$, $u' = s$, $u'' \in \mathsf{crashcopy}_B$ and

$$s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot p \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \in u; u'; u''$$

so that

$$s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot p \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \in K_{\lightning}\ \sigma = \sigma$$

as σ is saturated.

($\Longleftarrow$): Suppose $\sigma : A \multimap B$ satisfies the O-receptive, $P-$delaying and $\rightsquigarrow$-closed conditions. We want to show $\mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B = \sigma$.

By lemma J.1 we know that for any $u \in \mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B$, there is

$$t = t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1}$$

such that u can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where all $s_i \in \mathsf{ccopy}_A; \mathsf{dr}\,(t_i)\,; \mathsf{ccopy}_B$. Since s satisfies the O-receptive, $P-$delaying and $\rightsquigarrow$-closed conditions, whcih means there exists S_i for each i such that $s_i \in S_i$ for all i, S_i satisfy the $O-$receptive, $P-$delaying and $\rightsquigarrow$-closed. Thanks to J.12 we obtain that $\mathsf{ccopy}_A; \mathsf{dr}\,(t_i)\,; \mathsf{ccopy}_B \subseteq S_i$. So we have $s \in \sigma$ so $\mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B \subseteq \sigma$

The other direction follows from lemma J.13.

□

J.4 Symmetric Monoidal Structure of Crash

Proposition J.15. *For any $\sigma : A \multimap B \in \underline{\mathbf{Conc}}$ and $\sigma' : A' \multimap B' \in \underline{\mathbf{Conc}}$ it holds that*

$$\mathsf{vol}(\sigma \otimes \sigma') = \mathsf{vol}(\sigma) \otimes \mathsf{vol}(\sigma')$$

Proof. For one direction, fix $s \in \mathsf{vol}(\sigma) \otimes \mathsf{vol}(\sigma')$. By definition of the tensor s can be decomposed as

$$s := s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where for each i, $s_i \in \sigma \otimes \sigma'$. It then follows immediately by the definition of $\mathsf{vol}(-)$ that $s \in \mathsf{vol}(\sigma \otimes \sigma')$.

For the other direction, fix $s \in \mathsf{vol}(\sigma \otimes \sigma')$. We have that, by well-formedness, s can be decomposed as

$$s := s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where for each i, $s_i \in \sigma \otimes \sigma'$. So $s_i \upharpoonright_{A \multimap B} \in \sigma$ and $s_i \upharpoonright_{A' \multimap B'} \in \sigma'$.

Let

$$t := s_1 \upharpoonright_{A \multimap B} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \upharpoonright_{A \multimap B}$$
$$t' := s_1 \upharpoonright_{A' \multimap B'} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \upharpoonright_{A' \multimap B'}$$

It is immediate from the definition of $\mathsf{vol}(-)$ that $t \in \mathsf{vol}(\sigma)$, $t' \in \mathsf{vol}(\sigma')$, and so it follows that $s \in \mathsf{vol}(\sigma) \otimes \mathsf{vol}(\sigma')$. □

Lemma J.16.

$$\mathsf{crashcopy}_{A \otimes B} = \mathsf{crashcopy}_A \otimes \mathsf{crashcopy}_B$$

Proof. Note that

$$\begin{aligned} \mathsf{crashcopy}_{A \otimes B} &= \mathsf{vol}(\mathsf{ccopy}_{(A \otimes B)^\Upsilon}) && \text{(Def. of crashcopy)} \\ &= \mathsf{vol}(\mathsf{ccopy}_{A^\Upsilon} \otimes \mathsf{ccopy}_{B^\Upsilon}) && \text{(Symm. Mon. Cat. } \mathbf{Conc}\text{)} \\ &= \mathsf{vol}(\mathsf{ccopy}_{A^\Upsilon}) \otimes \mathsf{vol}(\mathsf{ccopy}_{B^\Upsilon}) && \text{(Prop. J.15)} \\ &= \mathsf{crashcopy}_A \otimes \mathsf{crashcopy}_B && \text{(Def. of crashcopy)} \end{aligned}$$

□

Lemma J.17.

$$- \otimes - : \underline{\mathbf{Crash}} \otimes \underline{\mathbf{Crash}} \to \underline{\mathbf{Crash}}$$

is a bi-semifunctor

Proof. For given

$$\sigma_1 : A_1 \multimap A_2, \qquad \sigma_2 : A_2 \multimap A_3$$
$$\tau_1 : B_1 \multimap B_2, \qquad \tau_2 : B_2 \multimap B_3$$

and fix $s \in (\sigma_1;\sigma_2) \otimes (\tau_1;\tau_2)$ From lemma J.1 we know s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

each s_i satisfies

$$s_i \restriction_{A_1 \multimap A_3} \in \mathsf{dr}(t_i);\mathsf{dr}(t'_i)$$
$$s_i \restriction_{B_1 \multimap B_3} \in \mathsf{dr}(u_i);\mathsf{dr}(u'_i)$$

where

$$t := t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \in \sigma_1$$
$$t' := t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t'_{n+1} \in \sigma_2$$
$$u := t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \in \tau_1$$
$$u' := t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u'_{n+1} \in \tau_2$$

futhermore we have for each s_i

$$\begin{aligned} s_i &\in (\mathsf{dr}(t_i);\mathsf{dr}(t'_i)) \otimes (\mathsf{dr}(u_i);\mathsf{dr}(u'_i)) \\ &= (\mathsf{dr}(t_i) \otimes \mathsf{dr}(u_i)); (\mathsf{dr}(t'_i) \otimes \mathsf{dr}(u'_i)) \end{aligned}$$

so there exists s'_i for each i such that

$$s'_i \restriction_{A_1 \otimes B_1 \multimap A_3 \otimes B_3} = s_i$$
$$s'_i \restriction_{A_1 \otimes B_1 \multimap A_2 \otimes B_2} \in \mathsf{dr}(t_i) \otimes \mathsf{dr}(u_i)$$
$$s'_i \restriction_{A_2 \otimes B_2 \multimap A_3 \otimes B_3} \in \mathsf{dr}(t'_i) \otimes \mathsf{dr}(u'_i)$$

let

$$r := s'_1 \restriction_{A_1 \otimes B_1 \multimap A_2 \otimes B_2} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s'_{n+1} \restriction_{A_1 \otimes B_1 \multimap A_2 \otimes B_2}$$
$$r' := s'_1 \restriction_{A_2 \otimes B_2 \multimap A_3 \otimes B_3} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s'_{n+1} \restriction_{A_2 \otimes B_2 \multimap A_3 \otimes B_3}$$

we have $r \in \sigma_1 \otimes \tau_1, r' \in \sigma_2 \otimes \tau_2$ futhermore we know $s \in \mathrm{strat}(r);\mathrm{strat}(r') \subseteq (\sigma_1 \otimes \tau_1);(\sigma_2 \otimes \tau_2)$. The other direction is similar.

The enrichment is obvious. First, if $\sigma \subseteq \sigma'$ and $\tau \subseteq \tau'$ it follows immediately from the definition that

$$\sigma \otimes \tau \subseteq \sigma' \otimes \tau'$$

Unions are handled in the same way. □

Proposition J.18. *$(\mathbf{Crash}, - \otimes -, \mathbf{1})$ defines an enriched symmetric monoidal category.*

Proof. We start by showing that the structural morphisms assemble into natural isomorphisms:

$$\begin{tikzcd} A \otimes (B \otimes C) \arrow[r, "\alpha_{A,B,C}"] \arrow[d, "\sigma_A \otimes (\sigma_B \otimes \sigma_C)"'] \arrow[dr, phantom, "\cong"] & (A \otimes B) \otimes C \arrow[d, "(\sigma_A \otimes \sigma_B) \otimes \sigma_C"] \\ A' \otimes (B' \otimes C') \arrow[r, "\alpha_{A',B',C'}"'] & (A' \otimes B') \otimes C' \end{tikzcd} \quad \begin{tikzcd} \mathbf{1} \otimes A \arrow[r, "\lambda_A"] \arrow[d, "\mathbf{1} \otimes \sigma"'] \arrow[dr, phantom, "\cong"] & A \arrow[d, "\sigma"] \\ \mathbf{1} \otimes B \arrow[r, "\lambda_B"'] & B \end{tikzcd} \quad \begin{tikzcd} A \otimes \mathbf{1} \arrow[r, "\rho_A"] \arrow[d, "\sigma \otimes \mathbf{1}"'] \arrow[dr, phantom, "\cong"] & A \arrow[d, "\sigma"] \\ B \otimes \mathbf{1} \arrow[r, "\rho_B"'] & B \end{tikzcd}$$

The left and right unital are straight-forward. Indeed, they are given by

$$\lambda''_A := \mathsf{vol}(\lambda_{A^\Upsilon}) \qquad\qquad \rho''_A := \mathsf{vol}(\rho_{A^\Upsilon})$$

where λ and ρ are the left and right unitals in Conc. It is easy to note that, up to renaming, this makes both unitals the same as crashcopy. Because of this, again, up to renaming, we essentially have:

$$1 \otimes \sigma = \{s' \in P_{(1\multimap 1)\otimes(A\multimap B)} \mid \exists s \in \sigma.(\{\epsilon\} \otimes \mathsf{epo}_1(s)) \cdot \lightning \cdot \ldots \cdot \lightning \cdot \{(\{\epsilon\} \otimes \mathsf{epo}_{\|s\|}(s))\} = \sigma$$

which differs from σ only in the shape of the crashes.Therefore, we easily check that:

$$(\mathbf{1} \otimes \sigma); \lambda_B = \sigma; \mathsf{crashcopy}_B = \sigma = \mathsf{crashcopy}_A; \sigma = \lambda_A; \sigma$$

$$(\sigma \otimes \mathbf{1}); \rho_B = \sigma; \mathsf{crashcopy}_B = \sigma = \mathsf{crashcopy}_A; \sigma = \rho_A; \sigma$$

Let β be the braiding in **Conc** and α be the associator in **Conc**. We define the associator and braiding in **Crash** by

$$\beta''_{A,B} := \mathsf{vol}(\beta_{A^\Upsilon,B^\Upsilon}) \quad \text{and} \quad \alpha''_{A,B,C} := \mathsf{vol}(\alpha_{A^\Upsilon,B^\Upsilon,C^\Upsilon})$$

Now, for the associator, the equation essentially follows from the fact that:

$$\begin{aligned} p_\alpha(\sigma_A \otimes (\sigma_B \otimes \sigma_C)); \alpha'_{A',B',C'} &= (p_\alpha(\sigma_A) \otimes (p_\alpha(\sigma_B) \otimes p_\alpha(\sigma_C))); \alpha'_{A',B',C'} \\ &= \alpha'_{A,B,C}; ((p_\alpha(\sigma_A) \otimes p_\alpha(\sigma_B)) \otimes p_\alpha(\sigma_C)) \\ &= \alpha'_{A,B,C}; p_\alpha((\sigma_A \otimes \sigma_B) \otimes \sigma_C) \end{aligned}$$

where for each $s \in \sigma$

$$p_\alpha(s) := \pi'_\alpha(\mathsf{epo}_1(s)) \cdot \lightning \cdot \ldots \cdot \lightning \cdot \pi'_\alpha(\mathsf{epo}_{\|s\|}(s))$$

π'_α is the corresponding projection in **Conc** this is the key step to establish that the naturality square commutes. The reverse direction follows similarly.

For the braiding, let's first show its naturality. For a given strategy $\sigma : A \multimap C, \tau : B \multimap D$ we want to show the following diagram commutes:

$$\begin{tikzcd} A \otimes B \arrow[r, "\sigma\otimes\tau"] \arrow[d, "\beta''_{A,B}"'] & C \otimes D \arrow[d, "\beta''_{C,D}"] \\ B \otimes A \arrow[r, "\tau\otimes\sigma"'] & D \otimes C \end{tikzcd}$$

Note that since β is a natural isomorphism, we have that for any $s \in \sigma$, $t \in \tau$:

$$\begin{tikzcd} A^\Upsilon \otimes B^\Upsilon \arrow[r, "\mathsf{dr}\left(\mathsf{epo}_i(s)\right)\otimes\mathsf{dr}\left(\mathsf{epo}_i(t)\right)"] \arrow[d, "\beta_{A^\Upsilon,B^\Upsilon}"'] & C \otimes D \arrow[d, "\beta_{C^\Upsilon,D^\Upsilon}"] \\ B^\Upsilon \otimes A^\Upsilon \arrow[r, "\mathsf{dr}\left(\mathsf{epo}_i(t)\right)\otimes\mathsf{dr}\left(\mathsf{epo}_i(s)\right)"'] & D^\Upsilon \otimes C^\Upsilon \end{tikzcd}$$

commutes.

So for any given $u \in \mathsf{strat}\,(s) \otimes \mathsf{strat}\,(t)\,; \beta''_{C,D}$ we have for each i,

$$\begin{aligned} \mathsf{epo}_i(u) &\in \mathsf{dr}\left(\mathsf{epo}_i(s)\right) \otimes \mathsf{dr}\left(\mathsf{epo}_i(t)\right); \beta_{C^\Upsilon,D^\Upsilon} \\ &= \beta_{A^\Upsilon,B^\Upsilon}; \mathsf{dr}\left(\mathsf{epo}_i(t)\right) \otimes \mathsf{dr}\left(\mathsf{epo}_i(s)\right) \end{aligned}$$

And we know for each i, $(\lightning_A, \lightning_B)\ \beta'_{A,B}\ (\lightning_B, \lightning_A)$ and $(\lightning_C, \lightning_D)\ \beta'_{C,D}\ (\lightning_D, \lightning_C)$, so we have

$$s \in \beta''_{A,B}; \tau \otimes \sigma$$

so $\sigma \otimes \tau; \beta''_{C,D} \subseteq \beta''_{A,B}; \tau \otimes \sigma$, other direction is similar. which let us to conclude that

$$\begin{tikzcd}
A \otimes B \arrow[r, "\sigma\otimes\tau"] \arrow[d, "\beta''_{A,B}"'] & C \otimes D \arrow[d, "\beta''_{C,D}"] \\
B \otimes A \arrow[r, "\tau\otimes\sigma"'] & D \otimes C
\end{tikzcd}$$

commutes.

The coherence diagrams follow from functoriality of Vol − together with the fact that Vol − distributes over $- \otimes -$ (Prop. J.15), by noting that all the structural morphisms in **Crash** were defined by lifting the corresponding structural morphisms in Conc and **Rel**, which is also why they are isomorphisms. □

J.5 Crash-Aware Linearizability

Proposition J.19.

$$K_{\lightning}\ : \underline{\mathbf{Crash}} \to \mathbf{Crash}$$

is an enriched oplax semifunctor

Proof. **Oplax semifunctor:** So we want to show for any $\sigma : A \multimap B$ and $\tau : B \multimap C$ we have

$$K_{\lightning}\ (\sigma; \tau) \subseteq K_{\lightning}\ \sigma; K_{\lightning}\ \tau$$

We have the composition is associative, crashcopy is idempotent and $\sigma \subseteq K_{\lightning}\ \sigma$ so

$$\begin{aligned}
K_{\lightning}\ (\sigma;\tau) &= \mathsf{crashcopy}_A; \sigma; \tau; \mathsf{crashcopy}_C \\
&\subseteq \mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B; \tau; \mathsf{crashcopy}_C \\
&= \mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B; \mathsf{crashcopy}_B; \tau; \mathsf{crashcopy}_C \\
&= K_{\lightning}\ \sigma; K_{\lightning}\ \tau
\end{aligned}$$

Enrichment: Suppose $\sigma \subseteq \sigma'$ then

$$K_{\lightning}\ \sigma = \mathsf{crashcopy}; \sigma; \mathsf{crashcopy} \subseteq \mathsf{crashcopy}; \sigma'; \mathsf{crashcopy} = K_{\lightning}\ \sigma'$$

by monontonicity of composition. Similarly

$$K_{\lightning}\ (\cup_{i\in I}\sigma_i) = \mathsf{crashcopy}; \cup_{i\in I}\sigma_i; \mathsf{crashcopy} = \bigcup_{i\in I} \mathsf{crashcopy}; \sigma_i; \mathsf{crashcopy} = \bigcup_{i\in I} K_{\lightning}\ \sigma_i$$

□

Proposition J.20. *For $\tau : A \multimap B \in \underline{\mathbf{Crash}}$,*

$$K_{\lightning}\ \tau = \{s \in P_{A\multimap B} \mid s \text{ is crash-linearizable with respect to } \tau\}$$

Proof. For one direction, let's fix $s \in K_{\lightning}\ \tau$ then by lemma J.1 there exists $t \in \tau$ such that s and t can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \quad \text{and} \quad t = t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1}$$

where for each i we have $s_i \in \mathsf{ccopy}_A; \mathsf{strat}\,(t_i)\,; \mathsf{ccopy}_B$ which implies that $s_i \rightsquigarrow t_i$ by [31]. Since this is true for all i, we have $s \overset{\lightning}{\rightsquigarrow} \tau$.

For the other direction, fix s crash-linearizable with respect to τ. Then, there exists t in τ such that s and t can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \quad \text{and} \quad t = t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1}$$

and for all i we have $s_i \rightsquigarrow t_i$. So we have $s_i \in \mathsf{ccopy}_A ; \mathsf{dr}\,(t_i) \,; \mathsf{ccopy}_B$ which implies there exists u_i such that $u_i \upharpoonright_{A,A,-,-} \in \mathsf{ccopy}_A$, $u_i \upharpoonright_{-,A,B,-} \in \mathsf{dr}\,(t_i)$, $u_i \upharpoonright_{-,-,B,B} \in \mathsf{ccopy}_B$ and $u_i \upharpoonright_{A,-,-,B} = s_i$. Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

we have $u \in \mathsf{crashcopy}_A$, $u' \in \mathsf{strat}\,(t) \subseteq \tau$ and $u'' \in \mathsf{crashcopy}_B$ so we have $s \in K_{\lightning}\ \tau$. □

Proposition J.21. *For $v' : A \in \mathbf{Conc}$ and $v : A \in \underline{\mathbf{Conc}}$, if $v' \rightsquigarrow v$ then $\mathsf{vol}(v') \overset{\lightning}{\rightsquigarrow} \mathsf{vol}(v)$.*

Proof.

$$\begin{aligned} \mathsf{vol}(v') &= (v' \cdot \lightning)^* \cdot v' && \text{(Def.)} \\ &\subseteq ((K_{\mathrm{Conc}}\ v) \cdot \lightning)^* \cdot K_{\mathrm{Conc}}\ v && \text{(Lin. equiv. } K_{\mathrm{Conc}}) \\ &= ((v; \mathsf{ccopy}_A) \cdot \lightning)^* \cdot (v; \mathsf{ccopy}_A) && \text{(Def. of } K_{\mathrm{Conc}}) \\ &= ((v \cdot \lightning)^* \cdot v); ((\mathsf{ccopy}_A \cdot \lightning)^* \cdot \mathsf{ccopy}_A) && \text{(Def. } -;- \text{ in } \underline{\mathbf{Crash}}) \\ &= \mathsf{vol}(v); \mathsf{crashcopy}_A = K_{\lightning}\ \mathsf{vol}(v) && \text{(Def. } \mathsf{vol}(-)) \end{aligned}$$

□

Proposition J.22.

$$K_{\lightning}\ \tau = \{s \in P_{A \multimap B} \mid s \textit{ is crash-linearizable with respect to } \tau\}$$

Proof. For one direction, let's fix $s \in K_{\lightning}\ \tau$ then by lemma J.1 there exists $t \in \tau$ such that s and t can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \quad \text{and} \quad t = t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1}$$

where for each i we have $s_i \in \mathsf{ccopy}_A ; \mathsf{strat}\,(t_i) \,; \mathsf{ccopy}_B$ which implies that $s_i \rightsquigarrow t_i$ by the result from Oliveira Vale et al. [31]. Since this is true for all i, we have $s \overset{\lightning}{\rightsquigarrow} \tau$, as desired.

For the other direction, fix s crash-linearizable with respect to τ. Then, there exists t in τ such that s and t can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1} \quad \text{and} \quad t = t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1}$$

and for all i we have $s_i \rightsquigarrow t_i$. So we have $s_i \in \mathsf{ccopy}_A ; \mathsf{dr}\,(t_i) \,; \mathsf{ccopy}_B$ which implies there exists u_i such that $u_i \upharpoonright_{A,A,-,-} \in \mathsf{ccopy}_A$, $u_i \upharpoonright_{-,A,B,-} \in \mathsf{dr}\,(t_i)$, $u_i \upharpoonright_{-,-,B,B} \in \mathsf{ccopy}_B$ and $u_i \upharpoonright_{A,-,-,B} = s_i$. Let

$$\begin{aligned} u &:= u_1 \upharpoonright_{A,A,-,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{A,A,-,-} \\ u' &:= u_1 \upharpoonright_{-,A,B,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,A,B,-} \\ u'' &:= u_1 \upharpoonright_{-,-,B,B} \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \upharpoonright_{-,-,B,B} \end{aligned}$$

we have $u \in \mathsf{crashcopy}_A$, $u' \in \mathsf{strat}\,(t) \subseteq \tau$ and $u'' \in \mathsf{crashcopy}_B$ so we have $s \in K_{\lightning}\ \tau$. □

Corollary J.23. *$v' : A$ is crash-aware linearizable with respect to $v : A$ if and only if $v' \subseteq K_{\lightning}\ v$.*

Now, we move to locality and observational refinement. According to §6 of Oliveira Vale et al. [31], it is enough to prove the following lemma.

Lemma J.24.

- *For any* $\sigma : \mathbf{1} \multimap A \in \underline{\textbf{Crash}}$ *it holds that* $\mathsf{crashcopy}_{\mathbf{1}}; \sigma = \sigma$.
- *For* $\sigma, \tau : A \multimap B$ *and* $\sigma', \tau' : A' \multimap B'$ *we have*

$$\sigma \otimes \sigma' \subseteq \tau \otimes \tau' \implies \sigma \subseteq \sigma' \wedge \tau \subseteq \tau'$$

Proof.

- For one direction, fix $s \in \mathsf{crashcopy}_{\mathbf{1}}; \sigma$, by J.1 we know s can be decomposed as

  $$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

  and $s_i \in t_i; u_i$ such that

  $$t := t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \in \mathsf{crashcopy}_{\mathbf{1}}$$
  $$u := u_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \in \sigma$$

  By definition of crashcopy we know $t_i \in \mathsf{ccopy}_{\mathbf{1}}$. Therefore, $t_i = \epsilon$ for all i so $t_i; u_i = u_i$, and therefore $s_i = u_i$. It then follows that $s \in \sigma$.
  For the other direction, fix $s \in \sigma$, by definition we know s can be decomposed as

  $$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

  Now let

  $$t := \epsilon \cdot \lightning \cdot \ldots \cdot \lightning \cdot \epsilon$$

  it is easy to check $t \in \mathsf{crashcopy}_{\mathbf{1}}$. But then, $s \in t; s$, so $s \in \mathsf{crashcopy}_{\mathbf{1}}; \sigma$.
- For given $\sigma, \tau : A \multimap B$ and $\sigma', \tau' : A' \multimap B'$ suppose $\sigma \otimes \sigma' \subseteq \tau \otimes \tau'$. We wish to show that $\sigma \subseteq \tau$. Fix $s \in \sigma$, by well-formedness s can be decomposed as

  $$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

  By $\lightning$−receptivity it follows that

  $$t := \epsilon \cdot \lightning \cdot \ldots \cdot \lightning \cdot \epsilon \in \sigma'$$

  But $s \otimes t = \{s\}$ up to re-indexing, by definition. So, by monotonicity, $s \in s \otimes t \subseteq \tau \otimes \tau'$. But then it follows that $s \in \tau$, so $\sigma \subseteq \tau$.
  The proof of $\sigma' \subseteq \tau'$ is similar.

□

Since by now we have proven all the requirements on the embeddable subcategory of the Karoubi envelope that we have constructed, it follows that

Proposition J.25. *Locality and the equivalence with observational refinement both hold for crash-aware linearizability.*

J.6 Crash Abstraction

We first prove a basic property about $-^\flat$.

Lemma J.26. *For* $s \in P_{A \multimap B}$,

- *if* $\pi_\Upsilon(s \restriction_{-,B}) \in \mathbb{P}_{B^\flat}$ *then* $\pi_\Upsilon(s \restriction_{A,-}) \in \mathbb{P}_{A^\flat}$;
- $\pi_\Upsilon(s) \in \mathbb{P}_{(A \multimap B)^\flat}$ *if and only if* $\pi_\Upsilon(s \restriction_{A,-}) \in \mathbb{P}_{A^\flat}$ *and* $\pi_\Upsilon(s \restriction_{-,B}) \in \mathbb{P}_{B^\flat}$.

Proof.

- Let's prove this result by contradiction. Suppose there exists $s \in P_{A \multimap B}$ such that $\pi_\Upsilon(s \upharpoonright_{-,B}) \in \mathbb{P}_{B^\flat}$ and $\pi_\Upsilon(s \upharpoonright_{A,-}) \notin \mathbb{P}_{A^\flat}$. First of all, by definition s can be decomposed as
$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$
and for each i, $s_i \in \mathbb{P}_{A^\Upsilon \multimap B^\Upsilon}$
Since $\pi_\Upsilon(s \upharpoonright_{A,-}) \notin \mathbb{P}_{A^\flat}$, there exists a $\alpha \in \Upsilon$ such that $\pi_\alpha(s \upharpoonright_{A,-})$ is not a valid sequential play. And, since for each i, $\pi_\alpha(s_i \upharpoonright_A)$ is a valid sequential play, there must exists i such that there is a pending O-move in $\pi_\alpha(s_i \upharpoonright_{A,-})$ and there exists $j > i$ such that $\pi_\alpha(s_j \upharpoonright_{A,-})$ is non-empty. Since $\pi_\alpha(s_i)$ is also a valid sequential play, there must be a pending O-move also in $\pi_\alpha(s_i \upharpoonright_{-,B})$ by the switching condition, and $\pi_\alpha(s_j \upharpoonright_{-,B})$ is also non-empty, since plays are O-starting. But then, this means $\pi_\alpha(s \upharpoonright_{-,B})$ is not a valid sequential play. So $\pi_\Upsilon(s \upharpoonright_{-,B}) \notin \mathbb{P}_{B^\flat}$ which is a contradiction.
- The forward direction is easy to see by definition, while the backward direction follows immediately from the first bullet point.

□

Corollary J.27.
$$(A \multimap B)^\flat \cong A^\flat \multimap B^\flat$$

We now address the functoriality of $-^\flat$.

Proposition J.28. $-^\flat : \underline{\mathbf{Crash}} \longrightarrow \underline{\mathbf{Conc}}$ *defines an enriched oplax semifunctor.*

Proof. Suppose $s \in (\sigma;\tau)^\flat$. Then, there is $s' \in \mathsf{int}(\sigma, \tau)$ such that $s = \pi_\Upsilon(s' \upharpoonright_{A,-,C})$ and in particular $\pi_\Upsilon(s' \upharpoonright_{A,-,C}) \in \mathbb{P}_{A \multimap B^\flat}$. Note then that by Prop. J.26 it follows that $\pi_\Upsilon(s' \upharpoonright_{-,B,C} \upharpoonright_{-,C}) = \pi_\Upsilon(s' \upharpoonright_{A,-,C} \upharpoonright_{-,C}) \in \mathbb{P}_{C^\flat}$, so that by the same proposition it follows that $\pi_\Upsilon(s' \upharpoonright_{-,B,C} \upharpoonright_{B,-}) \in P_{B^\flat}$ and therefore $\pi_\Upsilon(s' \upharpoonright_{-,B,C}) \in \mathbb{P}_{(B \multimap C)^\flat}$. At this point we have $s' \upharpoonright_{A,B,-} \upharpoonright_{-,B} = s' \upharpoonright_{-,B,C} \upharpoonright_{B,-} \in \mathbb{P}_{B^\flat}$ so that analogous reasoning gives that $s' \upharpoonright_{A,B,-} \in \mathbb{P}_{(A \multimap B)^\flat}$. But we also have $s' \upharpoonright_{A,B,-} \in \sigma$ and $s' \upharpoonright_{-,B,C} \in \tau$, so that we have shown that $\pi_\Upsilon(s' \upharpoonright_{A,B,-}) \in \sigma^\flat$ and $\pi_\Upsilon(s' \upharpoonright_{-,B,C}) \in \tau^\flat$. We then obtain that $\pi_\Upsilon(s') \upharpoonright_{A^\flat,B^\flat,-} = \pi_\Upsilon(s' \upharpoonright_{A^\flat,B^\flat,-}) \in \sigma^\flat$ and $\pi_\Upsilon(s') \upharpoonright_{-,B^\flat,C^\flat} = \pi_\Upsilon(s' \upharpoonright_{-,B,C}) \in \tau^\flat$. Hence, $\pi_\Upsilon(s') \in \mathsf{int}(\sigma^\flat, \tau^\flat)$ and therefore $s = \pi_\Upsilon(s' \upharpoonright_{A,-,C}) = \pi_\Upsilon(s') \upharpoonright_{A^\flat,-,C^\flat} \in \sigma^\flat;\tau^\flat$.

We move on to the enrichment. Suppose $\sigma \subseteq \sigma'$, fix $s \in \sigma^\flat$, by definition there exists $s' \in \sigma$ such that $\pi_\Upsilon(s') = s$. Since $\sigma \subseteq \sigma'$ so $s' \in \sigma'$ so $s \in (\sigma')^\flat$. So we have $\sigma^\flat \subseteq (\sigma')^\flat$

Given a family of stratgies $(\sigma_i : A \multimap B)_{i \in I}$. For one direction, fix $s \in (\cup_{i \in I} \sigma_i)^\flat$ so there exists $s' \in \cup_{i \in I} \sigma_i$ such that $\pi_\Upsilon(s') = s$. So there exists $i \in I$ such that $s' \in \sigma_i$ and $\pi_\Upsilon(s') = s \in \mathbb{P}_{A^\flat}$ which means $s \in \sigma_i^\flat$. So $s \in \cup_{i \in I} \sigma_i^\flat$. For the other direction, fix $s \in \cup_{i \in I} \sigma_i^\flat$, we know there exists $i \in I$ such that $s' \in \sigma_i$ and $s' \upharpoonright_= s$. So $s' \in \cup_{i \in I} \sigma_i$ and $\pi_\Upsilon(s') = s \in \mathbb{P}_{A^\flat}$, so $s \in (\cup_{i \in I} \sigma_i)^\flat$ □

Proposition J.29. $(\mathsf{crashcopy}_A)^\flat = \mathsf{ccopy}_{A^\flat}$

Proof. By definition it is easy to see $\mathsf{ccopy}_{A^\flat} \subseteq (\mathsf{crashcopy}_A)^\flat$.

For the other direction, fix $s \in (\mathsf{crashcopy}_A)^\flat$, by definition, s can be decomposed as
$$s = s_1 \cdot s_2 \cdot \ldots \cdot s_{n+1}$$
where for each i, $s_i \in \mathsf{ccopy}_{A^\flat}$.

By the definition of copy we know for any two plays $t, t' \in$ copy, if $t \cdot t'$ is a valid play, then $t \cdot t' \in$ copy. So for any $\alpha \in \Upsilon$ we get that $\pi_\alpha(s)$ is in $\mathsf{copy}_{A^\flat}$ so $s \in \mathsf{ccopy}_{A^\flat}$ □

Corollary J.30. *For every* $\sigma : A \multimap B \in \underline{\mathbf{Crash}}$*:*
$$(K_\lightning\ \sigma)^\flat \subseteq K_{\mathsf{Conc}}\ \sigma^\flat$$

We now move to the functoriality like properties of re-crash operation $-^\sharp$.

Proposition J.31. *For strategies $\sigma : A^\flat \multimap B^\flat$, and $\tau : B^\flat \multimap C^\flat$, the following hold:*

- *if $\sigma \subseteq \sigma'$, for $\sigma' : A^\flat \multimap B^\flat$, then $\sigma^\sharp \subseteq (\sigma')^\sharp$*
- *Given a family $(\sigma_i : A^\flat \multimap B^\flat)_{i \in I}$ it holds that $(\cup_{i\in I}\sigma_i)^\sharp = \cup_{i\in I}\sigma_i^\sharp$*
- *$\sigma^\sharp; \tau^\sharp \subseteq (\sigma;\tau)^\sharp$*

Proof.

- Suppose $\sigma \subseteq \sigma'$, fix $s \in \sigma^\sharp$ by definition we know $\pi_\Upsilon(s) \in \sigma$. Since $\sigma \subseteq \sigma'$, so $\pi_\Upsilon(s) \in \sigma'$ which implies $s \in (\sigma')^\sharp$. So $\sigma^\sharp \subseteq (\sigma')^\sharp$
- For a given family $(\sigma_i)_{i\in I}$. For one direction, fix $s \in (\cup_{i\in I}\sigma_i)^\sharp$ we know $\pi_\Upsilon(s) \in \cup_{i\in I}\sigma_i$ which means there exists $i \in I$ such that $\pi_\Upsilon(s) \in \sigma_i$. By definition this means $s \in \sigma_i^\sharp$, so $s \in \cup_{i\in I}\sigma_i^\sharp$. For the other direction, let fix $s \in \cup_{i\in I}\sigma_i^\sharp$, by definition, there exists $i \in I$ such that $\pi_\Upsilon(s) \in \sigma_i \subseteq \cup_{i\in I}\sigma_i$. So $s \in \cup_{i\in I}\sigma_i^\sharp$
- fix $s \in \sigma^\sharp; \tau^\sharp$, by definition there exists $s' \in \mathsf{int}(\sigma^\sharp, \tau^\sharp)$ such that $s' \upharpoonright_{A,-,C} = s$, $s' \upharpoonright_{A,B,-} \in \sigma^\sharp$ and $s' \upharpoonright_{-,B,C} \in \tau^\sharp$.
  Set $t := \pi_\Upsilon(s' \upharpoonright_{A,B,-})$, $u := \pi_\Upsilon(s' \upharpoonright_{-,B,C})$, it is easy to check
  $$\pi_\Upsilon(s) = \pi_\Upsilon(s' \upharpoonright_{A,-,C}) \in \mathsf{strat}\,(t)\,;\mathsf{strat}\,(u) \subseteq \sigma;\tau$$
  so $s \in (\sigma;\tau)^\sharp$. Well-formedness of $\pi_\Upsilon(s)$ is guaranteed by well-formedness of $\pi_\Upsilon(s' \upharpoonright_{A,-,-})$, $\pi_\Upsilon(s' \upharpoonright_{-,-,C})$ by Prop. J.26.

□

Proposition J.32. *For all $A \in \underline{\mathbf{Crash}}$,*

$$\mathsf{ccopy}_{A^\flat}^\sharp \subseteq \mathsf{crashcopy}_A$$

Proof. Fix $s \in \mathsf{ccopy}_{A^\flat}^\sharp$, by definition we know there exists $s' \in \mathsf{ccopy}_{A^\flat}$ such that s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

where $s_1 \cdot \ldots \cdot s_{n+1} = s'$ and for each i, s_i is a play in $\mathbb{P}_{A^\Upsilon}$. Furthermore we know for each $i \neq n+1$, $\alpha \in \Upsilon$, there is no pending O moves in $\pi_\alpha(s_i)$.

Now we want to show for all i, $\alpha \in \Upsilon$, $p \sqsubseteq_{\mathsf{even}} \pi_\alpha(s_i)$ we have $p\upharpoonright_{A,-} = p\upharpoonright_{-,A}$ by contradiction. Suppose i is the smallest i such that there exists $\alpha \in \Upsilon$, $p \sqsubseteq_{\mathsf{even}} \pi_\alpha(s_i)$ such that $p\upharpoonright_{A,-} \neq p\upharpoonright_{-,A}$.

Since $s'' := s_1 \cdot \ldots \cdot s_i \sqsubseteq s'$ so we know $s'' \in \mathsf{ccopy}_{A^\flat}$ which means for every $q \sqsubseteq_{\mathsf{even}} \pi_\alpha(s'')$ we have $q\upharpoonright_{A,-} = q\upharpoonright_{-,A}$. Since there is no pending O moves in $s_1, \ldots, s_{i-1}$ so $p' := \pi_\alpha(s_1 \cdot \ldots \cdot s_{i-1})$ has to be an even prefix of $\pi_\alpha(s'')$ so is $p' \cdot p$. But it is easy to see $p' \cdot p\upharpoonright_{A,-} \neq p' \cdot p\upharpoonright_{-,A}$ which is a contradiction. So we know for each i, $s_i \in \mathsf{ccopy}_{A^\flat}$. By definition, $s \in \mathsf{crashcopy}_A$. □

We also take the opportunity to show that how $-^\sharp$ interacts with horizontal composition.

Lemma J.33. *For any $\sigma : (A_1)^\flat \multimap (B_1)^\flat$, $\tau : (A_2)^\flat \multimap (B_2)^\flat$,*

$$(\sigma \otimes \tau)^\sharp = \sigma^\sharp \otimes \tau^\sharp$$

Proof. For one direction fix $s \in (\sigma \otimes \tau)^\sharp$, we know s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

for each i we know there exists t_i, u_i such that $s_i \in t_i \otimes u_i$ such that $u_1 \cdot \ldots \cdot u_{n+1} \in \sigma$ and $t_1 \cdot \ldots \cdot t_{n+1} \in \tau$

By definition we can see

$$t := t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \in \sigma^\sharp$$

$$u := u_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \in \tau^\sharp$$

so we know $s \in \mathsf{strat}\,(t) \otimes \mathsf{strat}\,(u) \subseteq \sigma^\sharp \otimes \tau^\sharp$

For the other direction let's fix $s \in \sigma^\sharp \otimes \tau^\sharp$ then there exists $t \in \sigma^\sharp$, $u \in \tau^\sharp$ such that $s \in \mathsf{strat}\,(t) \otimes \mathsf{strat}\,(u)$. Let $s' = \pi_\Upsilon(t) \otimes \pi_\Upsilon(u)$ by definition we know $s' \in \sigma \otimes \tau$ and $\pi_\Upsilon(s) = s'$ so $s \in (\sigma \otimes \tau)^\sharp$ □

J.7 Strict Linearizability

We start by showing some auxiliary lemmas.

Lemma J.34. *For any given $\sigma, \tau \in$ **Conc** we have*

$$\sigma^\sharp; \mathsf{crashcopy}_B; \tau^\sharp = \sigma^\sharp; \mathsf{ccopy}^\sharp_{B^\flat}; \tau^\sharp$$

Proof. One direction follow from an already proved lemma.

$$\sigma^\sharp; \mathsf{ccopy}^\sharp_{B^\flat}; \tau^\sharp \subseteq \sigma^\sharp; \mathsf{crashcopy}_B; \tau^\sharp$$

Now let's try to prove the other direction by contradiction. Suppose $s \in \sigma^\sharp; \mathsf{crashcopy}_B; \tau^\sharp$, assume $s \notin \sigma^\sharp; \mathsf{ccopy}^\sharp_{B^\flat}; \tau^\sharp$. So there exists s' such that $s' \restriction_{A,B,-,-} \in \sigma^\sharp$, $s' \restriction_{-,B,B,-} \in \mathsf{crashcopy}_B$ but $s' \restriction_{-,B,B,-} \notin \mathsf{ccopy}^\sharp_{B^\flat}$, $s' \restriction_{-,-,B,C} \in \tau^\sharp$ and $s' \restriction_{A,-,-,C} = s$

Since we already know $(\mathsf{crashcopy})^\flat = \mathsf{ccopy}$, it follows that $\mathsf{ccopy}^\sharp_{B^\flat} = (\mathsf{crashcopy}^\flat)^\sharp$. So we get that $s' \restriction_{-,B,B,-} \notin (\mathsf{crashcopy}^\flat)^\sharp$. But this means that at least one of $\pi_\Upsilon(s' \restriction_{-,B,-,-})$ and $\pi_\Upsilon(s' \restriction_{-,-,B,-})$ is not well-formed. But we know both $s' \restriction_{A,B,-,-}$ and $s' \restriction_{-,-,B,C}$ are well-formed which is a contradiction. □

Lemma J.35. *For any given $\sigma, \tau \in$ **Conc** we have*

$$\sigma^\sharp; \mathsf{crashcopy}_B; \tau^\sharp = \sigma^\sharp; \tau^\sharp$$

Proof. For one direction, note that $\sigma^\sharp; \tau^\sharp \subseteq \sigma^\sharp; \mathsf{crashcopy}_B; \tau^\sharp$ follows from what we showed about $K_\lightning$ $-$.

For the other direction notice that

$$\begin{aligned}\sigma^\sharp; \mathsf{crashcopy}_B; \tau^\sharp &= \sigma^\sharp; \mathsf{ccopy}^\sharp_{B^\flat}; \tau^\sharp \\ &\subseteq \sigma^\sharp; (\mathsf{ccopy}_{B^\flat}; \tau)^\sharp \\ &= \sigma^\sharp; \tau^\sharp\end{aligned}$$

□

Proposition J.36.

$$\mathsf{crashcopy}_A; \mathsf{ccopy}^\sharp_{A^\flat} = \mathsf{ccopy}^\sharp_{A^\flat}$$

Proof. One direction $\mathsf{ccopy}^\sharp_{A^\flat} \subseteq \mathsf{crashcopy}_A; \mathsf{ccopy}^\sharp_{A^\flat}$ follows readily from what we showed about $K_\lightning$ $-$.

The other direction, let's fix $s \in \mathsf{crashcopy}_A; \mathsf{ccopy}^\sharp_{A^\flat}$, then there exists s' such that $s' \restriction_{A,A,-} \in \mathsf{crashcopy}_A$, $s' \restriction_{-,A,A} \in \mathsf{ccopy}^\sharp_{A^\flat}$, and $s' \restriction_{A,-,A} = s$.

Let's first show that $\pi_\alpha(s)$ is well-formed by contradiction. By definition we know s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

If s is not well-formed, we know there exists $\alpha \in \Upsilon$ such that there exists l and $m > l$ such that $\pi_\alpha(s_l \cdot s_m)$ is not well-formed and $\pi_\alpha(s_m) \neq \epsilon$. Let m be the first m satisfies the property.

This means either $\pi_\alpha(s_l \cdot s_m)\upharpoonright_{A,-}$ or $\pi_\alpha(s_l \cdot s_m)\upharpoonright_{-,A}$ is not well-formed. But we know $\pi_\alpha(s_l \cdot s_m)\upharpoonright_{-,A} = \pi_\alpha(s'_l \cdot s'_m)\upharpoonright_{-,-,A}$ which is well-formed by definition. So the only possibility left is $\pi_\alpha(s_l \cdot s_m)\upharpoonright_{A,-}$ is not well-formed.

This means there is a pending O move in $\pi_\alpha(s_l)\upharpoonright_{A,-}$ which will become a P move in s_l and this will force $\pi_\alpha(s_l)\upharpoonright_{-,A}$ has a pending O move. But we already know $\pi_\alpha(s_m)\upharpoonright_{-,A}$ is non-empty, so this is a contradiction.

Now we want to show $\pi_\Upsilon(s) \in \mathsf{ccopy}_{A^\flat}$. We only need to show for each $\alpha \in \Upsilon$ we have $\pi_\alpha(s) \in \mathsf{copy}_{(A^\flat)^\alpha}$. Let's also show this by contradiction, suppose there exists $\alpha \in \Upsilon$ and $p \sqsubseteq_{\mathsf{even}} \pi_\alpha(s)$ such that $\pi_\alpha(p)\upharpoonright_{A,-} \neq \pi_\alpha(p)\upharpoonright_{-,A}$

So there exists $p' \sqsubseteq \pi_\alpha(s')$ such that $p'\upharpoonright_{A,-,A} = p$. Since $p'\upharpoonright_{A,A,-}$ is in $\mathsf{copy}_{(A^\flat)^\alpha}$ so $p'\upharpoonright_{A,-,-} = p'\upharpoonright_{-,A,-}$. Same we can get $p'\upharpoonright_{-,A,-} = p'\upharpoonright_{-,-,A}$. So we know $p'\upharpoonright_{A,-,-} = p'\upharpoonright_{-,-,A}$ which means $p\upharpoonright_{A,-} = p\upharpoonright_{-,A}$ which is a contradiction. □

With all these lemmas proved, we are ready to show observational refinement.

Proposition J.37. *Let* $\sigma : A^\flat \multimap B^\flat \in \mathbf{Conc}$ *and* $\tau : B^\flat \multimap C^\flat \in \mathbf{Conc}$*, then*

$$\mathsf{str}(\sigma); \mathsf{str}(\tau) \subseteq \mathsf{str}(\sigma; \tau)$$

Proof.

$$\begin{aligned}
\mathsf{str}(\sigma); \mathsf{str}(\tau) &= K_\lightning\ \sigma^\sharp; K_\lightning\ \tau^\sharp \\
&= \mathsf{crashcopy}_A; \sigma^\sharp; \mathsf{crashcopy}_B; \tau^\sharp; \mathsf{crashcopy}_C \\
&= \mathsf{crashcopy}_A; \sigma^\sharp; \tau^\sharp; \mathsf{crashcopy}_C \\
&\subseteq \mathsf{crashcopy}_A; (\sigma; \tau)^\sharp; \mathsf{crashcopy}_C \\
&\subseteq K_\lightning\ (\sigma; \tau)^\sharp \\
&= \mathsf{str}(\sigma; \tau)
\end{aligned}$$

□

Proposition J.38. *If* $\nu'_A \subseteq \mathsf{str}(\nu_A)$ *then, for all* $\sigma : A^\flat \multimap B^\flat \in \mathbf{Conc}$ *that implements an object linearizable to* $\nu_B : B^\flat$ *using* ν_A*, i.e.*

$$\nu_A; \sigma \subseteq \nu_B$$

It holds that,

$$\nu'_A; \mathsf{str}(\sigma) \subseteq \mathsf{str}(\nu_B)$$

Proof. For the forward direction, we start by noting that since

$$\nu_A; \sigma \subseteq \nu_B$$

First, note that

$$\nu'_A \subseteq \mathsf{str}(\nu_A) \subseteq \mathsf{str}(K_{\mathsf{Conc}}\ \nu_A)$$

By the observational refinement property on Conc it follows that

$$K_{\mathsf{Conc}}\ \nu_A; \sigma \subseteq \nu_B$$

Then, by Prop. J.37

$$\nu'_A ; \mathsf{str}(\sigma) \subseteq \mathsf{str}(K_{\mathrm{Conc}}\ \nu_A) ; \mathsf{str}(\sigma) = \mathsf{str}(K_{\mathrm{Conc}}\ \nu_A ; \sigma) \subseteq \mathsf{str}(\nu_B)$$

□

For locality, note first that:

PROPOSITION J.39. *For $\sigma : A_1^\flat \multimap B_1^\flat$ and $\tau : A_2^\flat \multimap B_2^\flat$,*

$$\mathsf{str}(\sigma \otimes \tau) = \mathsf{str}(\sigma) \otimes \mathsf{str}(\tau)$$

PROOF.

$$\mathsf{str}(\sigma \otimes \tau) = K_{\lightning}\ (\sigma \otimes \tau)^\sharp = K_{\lightning}\ (\sigma^\sharp \otimes \tau^\sharp) = K_{\lightning}\ \sigma^\sharp \otimes K_{\lightning}\ \tau^\sharp = \mathsf{str}(\sigma) \otimes \mathsf{str}(\tau)$$

□

PROPOSITION J.40 (LOCALITY). *For $\nu'_A : A, \nu'_B : B \in$ **Crash** and $\nu_A : A, \nu_B : B \in \underline{\mathbf{Conc}}$:*

$$\nu'_A \subseteq \mathsf{str}(\nu_A) \text{ and } \nu'_B \subseteq \mathsf{str}(\nu_B) \text{ if and only if } \nu'_A \otimes \nu'_B \subseteq \mathsf{str}(\nu_A \otimes \nu_B)$$

PROOF. For the forward direction we have that by monotonicity:

$$\nu'_A \otimes \nu'_B \subseteq \mathsf{str}(\nu_A) \otimes \mathsf{str}(\nu_B) = \mathsf{str}(\nu_A \otimes \nu_B)$$

For the reverse direction, first note that

$$\nu'_A \otimes \nu'_B \subseteq \mathsf{str}(\nu_A \otimes \nu_B) = \mathsf{str}(\nu_A) \otimes \mathsf{str}(\nu_B)$$

since we have shown the tensor is an order-isomorphism the result follows. □

J.8 Durability

LEMMA J.41. *Let $s \in P_{A \multimap B}$ for $A, B \in \underline{\mathbf{Crash}}$.*

- *If $s\restriction_{-,B}$ is durable then $s\restriction_{A,-}$ is durable.*
- *s is durable if and only if $s\restriction_{A,-}$ and $s\restriction_{-,B}$ are both durable.*

PROOF.

- Let's prove this by contradiction. Suppose $s\restriction_{-,B}$ is durable, but $s\restriction_{A,-}$ is not. Then there exists $i \neq j$ such that $\exists \alpha \in \Upsilon. \alpha \in \Upsilon(\mathsf{epo}_i(s\restriction_{A,-})) \cap \Upsilon(\mathsf{epo}_j(s\restriction_{A,-}))$. Since both $\mathsf{epo}_i(s), \mathsf{epo}_j(s) \in \mathbb{P}_{A^\Upsilon \multimap B^\Upsilon}$, it follows that $\pi_\alpha(\mathsf{epo}_i(s)), \pi_\alpha(\mathsf{epo}_j(s)) \in \mathbb{P}_{A^\alpha \multimap B^\alpha}$. So by the switching condition, and since $\pi_\alpha(\mathsf{epo}_i(s)\restriction_{A,-})$ is non-empty we have that $\pi_\alpha(\mathsf{epo}_i(s)\restriction_{-,B})$ is also non-empty. The same applies for j. Again by the switching condition, $\alpha \in \Upsilon(\mathsf{epo}_i(s\restriction_{-,B})) \cap \Upsilon(\mathsf{epo}_j(s\restriction_{-,B}))$. But we know $s\restriction_{-,B}$ is durable which is a contradiction.
- **(⇒)** It is easy to see by definition
  (⇐) Suppose both $s\restriction_{A,-}$ and $s\restriction_{-,B}$ are durable. Let's prove s is also durable by contradiction. Suppose s is not, so there exists $i \neq j$ such that $\exists \alpha \in \Upsilon. \alpha \in \Upsilon(\mathsf{epo}_i(s)) \cap \Upsilon(\mathsf{epo}_j(s))$.
  So $\pi_\alpha(\mathsf{epo}_i(s))\restriction_{-,B}$ and $\pi_\alpha(\mathsf{epo}_j(s))\restriction_{-,B}$ both can't be empty play. So $\alpha \in \Upsilon(\mathsf{epo}_i(s)\restriction_{-,B}) \cap \Upsilon(\mathsf{epo}_j(s)\restriction_{-,B})$ by the switching condition. But we know $s\restriction_{-,B}$ is durable which is a contradiction.

□

PROPOSITION J.42. *Durable strategies compose.*

Proof. Suppose $\sigma : A \multimap B, \tau : B \multimap C$, are both durable.

We already have that $\sigma;\tau : A \multimap C$ is well-defined, it remains to show that it is also durable. By definition of composition we know for any $s \in \sigma;\tau$ there exists $t \in \sigma, u \in \tau$ such that $s\upharpoonright_{A,-} = t\upharpoonright_{A,-}$ and $s\upharpoonright_{-,C} = u\upharpoonright_{-,C}$.

Since both σ, τ are durable, by proposition J.41 $t\upharpoonright_{A,-}$ and $u\upharpoonright_{-,B}$ are both durable. So thanks to proposition J.41 again we know s is also durable. □

This means that the restriction of **Crash** to durable strategies, **Dur**, defines a semicategory.

Lemma J.43. *For sets of plays*

$$S \subseteq P_{A\multimap B} \qquad T \subseteq P_{B\multimap C}$$

and

$$(S \cap P^{\mathrm{dur}}_{A\multimap B}); (T \cap P^{\mathrm{dur}}_{B\multimap C}) = (S;T) \cap P^{\mathrm{dur}}_{A\multimap C}$$

Proof. For one direction, fix $s \in (S \cap P^{\mathrm{dur}}_{A\multimap B}); (T \cap P^{\mathrm{dur}}_{B\multimap C})$, we know there exists s' such that $s'\upharpoonright_{A,B,-} \in S \cap P^{\mathrm{dur}}_{A\multimap B}$, $s'\upharpoonright_{-,B,C} \in T \cap P^{\mathrm{dur}}_{B\multimap C}$ and $s'\upharpoonright_{A,-,C} = s$.

In particular we know $s'\upharpoonright_{A,B,-} \in S$ and $s'\upharpoonright_{-,B,C} \in T$ which implies $s \in S;T$. By applying proposition J.41 we know $s\upharpoonright_{A,-} = s'\upharpoonright_{A,-,-}$, $s\upharpoonright_{-,C} = s'\upharpoonright_{-,-,C}$ are both durable. So also by proposition J.41 we know s is durable. So $s \in (S;T) \cap P^{\mathrm{dur}}_{A\multimap C}$.

For other direction, fix $s \in (S;T) \cap P^{\mathrm{dur}}_{A\multimap C}$. We know there exists s' such that $s'\upharpoonright_{A,B,-} \in S$, $s'\upharpoonright_{-,B,C} \in T$, $s'\upharpoonright_{A,-,C} = s$. By proposition J.41 we $s\upharpoonright_{-,C}$ is durable so is $s'\upharpoonright_{-,-,C}$. By proposition J.41 we know $s'\upharpoonright_{-,B,-}$ has to be durable. Then by applying proposition J.41 again we get $s'\upharpoonright_{A,-,-}$ is durable.

So we know $s'\upharpoonright_{A,B,-} \in S \cap P^{\mathrm{dur}}_{A\multimap B}$ and $s'\upharpoonright_{-,B,C} \in T \cap P^{\mathrm{dur}}_{B\multimap C}$. □

Corollary J.44. *The assignment:*

$$A \longmapsto A \qquad\qquad \sigma : A \multimap B \longmapsto \sigma \cap P^{\mathrm{dur}}_{A\multimap B}$$

defines a semifunctor from **Crash** *to* **Dur**.

Corollary J.45. durcopy *is idempotent.*

Corollary J.46. *For any given (lax, oplax, semi) functor $F : \mathbf{C} \longrightarrow$* **Crash**, *we have*

$$F(-) \cap P^{\mathrm{dur}}_{F\,X\multimap F\,Y} : \mathbf{C} \longrightarrow \underline{\mathbf{Dur}}$$

also defines (lax, oplax, semi) functor respectively

Proposition J.47. *A strategy $\sigma : A \multimap B \in$* **Crash** *is saturated with respect to* durcopy *if and only if it is a durable strategy and*

durably O-receptive:

$$\forall s \in \sigma.\forall \alpha \in \Upsilon.\forall m \in M^{\boldsymbol{\alpha:O}}_{A\multimap B}.\exists i \leq \|s\|.$$

$$\mathrm{epo}_i(s) \cdot m \in P_{A^\Upsilon\multimap B^\Upsilon} \wedge \forall j \neq i.\Upsilon(m) \notin \Upsilon(\mathrm{epo}_j(s)) \implies$$

$$\mathrm{epo}_1(s) \cdot \lightning \cdot \ldots \cdot \lightning \cdot \mathrm{epo}_i(s) \cdot m \cdot \lightning \cdot \ldots \cdot \lightning \cdot \mathrm{epo}_{\|s\|}(s) \in \sigma$$

$\rightsquigarrow$-closed: $\forall s \in \sigma.\forall t \in P_{A\multimap B}.t \rightsquigarrow_{A\multimap B} s \implies t \in \sigma$

P-delaying: $\forall s \in \sigma.\forall m \in M^{P}_{A\multimap B}.\forall m_{\lightning} \in M^{\lightning}_{A\multimap B}.s = p \cdot m \cdot m_{\lightning} \cdot t \Rightarrow p \cdot m_{\lightning} \cdot t \in \sigma$

Proof. For one direction, notice that

$$\mathsf{durcopy}_A; \sigma; \mathsf{durcopy}_B = (\mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B) \cap P^{\mathsf{dur}}_{A \multimap B}$$

By J.7 we know $\mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B$ satisfies O-receptive, P-delaying and $\rightsquigarrow$-closed. So $(\mathsf{crashcopy}_A; \sigma; \mathsf{crashcopy}_B) \cap P^{\mathsf{dur}}_{A \multimap B}$ will satisfies durably O-receptive and $\rightsquigarrow$-closed.

For the other direction, suppose σ is durable, durably O-receptive, P-delaying and $\rightsquigarrow$-closed. We want to show $\sigma = \mathsf{durcopy}_A; \sigma; \mathsf{durcopy}_B$.

Set σ' be the smallest strategy contain σ and satisfies $O-$receptive and P-delaying. By definition we know $\sigma' \cap P^{\mathsf{dur}}_{A \multimap B} = \sigma$ and σ' satisfies $\rightsquigarrow$-closed.

Since σ' satisfies O-receptive, P-delaying and $\rightsquigarrow$-closed, $\sigma' = \mathsf{crashcopy}_A; \sigma'; \mathsf{crashcopy}_B$. So we have $(\mathsf{crashcopy}_A; \sigma'; \mathsf{crashcopy}_B) \cap P^{\mathsf{dur}}_{A \multimap B} = \sigma$.

Now we have

$$\begin{aligned}
\sigma &= \sigma' \cap P^{\mathsf{dur}}_{A \multimap B} \\
&= (\mathsf{crashcopy}_A; \sigma'; \mathsf{crashcopy}_B) \cap P^{\mathsf{dur}}_{A \multimap B} \\
&= (\mathsf{crashcopy}_A \cap P^{\mathsf{dur}}_{A \multimap A}); (\sigma' \cap P^{\mathsf{dur}}_{A \multimap B}); (\mathsf{crashcopy}_B \cap P^{\mathsf{dur}}_{B \multimap B}) \\
&= \mathsf{durcopy}_A; \sigma; \mathsf{durcopy}_B
\end{aligned}$$

□

Proposition J.48. *For any $A, B \in \underline{\mathbf{Crash}}$ and $\sigma : A^\flat \multimap B^\flat \in \underline{\mathbf{Conc}}$, $\mathsf{dur}(\sigma) \in \mathbf{Dur}$.*

Proof. We want to show $\mathsf{dur}(\sigma) = \mathsf{durcopy}_A; K_{\mathsf{Conc}}\ \sigma^\sharp; \mathsf{durcopy}_B$. For the direction $\mathsf{dur}(\sigma) \subseteq \mathsf{durcopy}_A; \mathsf{dur}(\sigma); \mathsf{durcopy}_B$ follows from the analogous fact about K_{Conc} – and monotonicity of $- \cap P^{\mathsf{dur}}_{A \multimap B}$.

Now for the other direction, fix $s \in \mathsf{durcopy}_A; \mathsf{dur}(\sigma); \mathsf{durcopy}_B$ first by definition s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

and for each i we know $s_i \in \mathsf{ccopy}_A; s_i; \mathsf{ccopy}_B$ and we know for each $i \neq j$ we have $\Upsilon(s_i) \neq \Upsilon(s_j)$

So there exists s'_i such that $s'_i \upharpoonright_{A,A,-,-} \in \mathsf{ccopy}_A$, $s'_i \upharpoonright_{-,A,B,-} = s_i$, $s'_i \upharpoonright_{-,-,B,B} \in \mathsf{ccopy}_B$. Since we already know for each $i \neq j$ we have $\Upsilon(s_i) \neq \Upsilon(s_j)$ so we know $\Upsilon(s'_i) \neq \Upsilon(s'_j)$.

So we know

$$\begin{aligned}
t &:= s'_1 \upharpoonright_{A,A,-,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s'_{n+1} \upharpoonright_{A,A,-,-} \\
t' &:= s'_1 \upharpoonright_{-,A,B,-} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s'_{n+1} \upharpoonright_{-,A,B,-} \\
t'' &:= s'_1 \upharpoonright_{-,-,B,B} \cdot \lightning \cdot \ldots \cdot \lightning \cdot s'_{n+1} \upharpoonright_{-,-,B,B}
\end{aligned}$$

such that $\pi_\Upsilon(t) \in \mathsf{ccopy}_{A^\flat}$, $\pi_\Upsilon(t'') \in \mathsf{ccopy}_{B^\flat}$ and $\pi_\Upsilon(t') \in \sigma$, so $s \in \mathsf{dur}(\sigma)$.

For the $\lightning$-receptivity, notice that suppose $s \in \mathsf{dur}(\sigma), m_\lightning \in M^{\lightning}_{A \multimap B}, s \cdot m_\lightning \in P_{A \multimap B}$, it is easy to check $\pi_\Upsilon(s \cdot m_\lightning) = \pi_\Upsilon(s) \in \sigma$. So $s \cdot m_\lightning \in \mathsf{dur}(\sigma)$ □

Proposition J.49.

$$\mathsf{dur}(\mathsf{ccopy}_{A^\flat}) = \mathsf{durcopy}_A$$

Proof.

$$\begin{aligned}
\mathsf{dur}(\mathsf{ccopy}_{A^\flat}) &= (K_{\mathsf{Conc}}\ \mathsf{ccopy}_{A^\flat})^\sharp \cap P^{\mathsf{dur}}_{A \multimap A} \\
&= \mathsf{ccopy}^\sharp_{A^\flat} \cap P^{\mathsf{dur}}_{A \multimap A}
\end{aligned}$$

Now we only need to show $\mathsf{ccopy}^{\sharp}_{A^\flat} \cap P^{\mathsf{dur}}_{A\multimap A} = \mathsf{crashcopy}_A \cap P^{\mathsf{dur}}_{A\multimap A}$. One direction just follows from J.32. Now we only need to show $\mathsf{crashcopy}_A \cap P^{\mathsf{dur}}_{A\multimap A} \subseteq \mathsf{ccopy}^{\sharp}_{A^\flat} \cap P^{\mathsf{dur}}_{A\multimap A}$.

Fix $s \in \mathsf{crashcopy}_A \cap P^{\mathsf{dur}}_{A\multimap A}$, by definition we know $\pi_\Upsilon(s)$ is well-formed. Futhermore since for each i, $\mathsf{epo}_i(s) \in \mathsf{ccopy}_{A^\flat}$ so we know $\pi_\Upsilon(s)$ is in $\mathsf{ccopy}_{A^\flat}$. So $s \in \mathsf{ccopy}^{\sharp}_{A^\flat}$. By assumption, s is durable. So $s \in \mathsf{ccopy}^{\sharp}_{A^\flat} \cap P^{\mathsf{dur}}_{A\multimap A}$. □

Proposition J.50.

$$\mathsf{dur}(-) : \underline{\mathbf{Conc}} \to \underline{\mathbf{Dur}}$$

behaves like an (enriched) lax semifunctor.

Proof. When $\sigma, \tau \in \mathbf{Conc}$ we have

$$\begin{aligned}
\mathsf{dur}(\sigma); \mathsf{dur}(\tau) &= (\sigma^\sharp \cap P^{\mathsf{dur}}_{A\multimap B}); (\tau^\sharp \cap P^{\mathsf{dur}}_{B\multimap C}) \\
&= (\sigma^\sharp; \tau^\sharp) \cap P^{\mathsf{dur}}_{A\multimap C} \\
&\subseteq (\sigma; \tau)^\sharp \cap P^{\mathsf{dur}}_{A\multimap C} \\
&\subseteq (K_{\mathrm{Conc}}\ (\sigma; \tau))^\sharp \cap P^{\mathsf{dur}}_{A\multimap C} \\
&= \mathsf{dur}(\sigma; \tau)
\end{aligned}$$

□

J.9 Symmetric Monoidal Structure of Dur

Proposition J.51.

$$\mathsf{Dur} : \mathbf{Conc} \to \mathbf{Dur}$$

defined by

$$\mathsf{Dur}\ \sigma := \mathsf{vol}(\sigma) \cap P^{\mathsf{dur}}_{A\multimap B}$$

is a functor

Proof. It follows immediately from the fact that Dur is by definition the composition of two functors. □

Lemma J.52. *For sets of plays*

$$S \subseteq P_{A\multimap B} \qquad T \subseteq P_{A'\multimap B'}$$

and

$$(S \cap P^{\mathsf{dur}}_{A\multimap B}) \otimes (T \cap P^{\mathsf{dur}}_{A'\multimap B'}) \cap P^{\mathsf{dur}}_{A\otimes A'\multimap B\otimes B'} = (S \otimes T) \cap P^{\mathsf{dur}}_{A\otimes A'\multimap B\otimes B'}$$

Proof. Since $S \cap P^{\mathsf{dur}}_{A\multimap B} \subseteq S, T \cap P^{\mathsf{dur}}_{A'\multimap B'} \subseteq T$, one direction is trivial.

For the other direction, fix $s \in (S \otimes T) \cap P^{\mathsf{dur}}_{A\otimes A'\multimap B\otimes B'}$. By definition we know s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

And for each i, we know

$$s_i \in t_i \otimes u_i$$

and

$$\begin{aligned}
t &:= t_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot t_{n+1} \in \sigma \\
u &:= u_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot u_{n+1} \in \tau
\end{aligned}$$

Since for each i, $\Upsilon(t_i) \subseteq \Upsilon(s_i)$, and $\Upsilon(u_i) \subseteq \Upsilon(s_i)$, s is durable implies that both t and u are durable.

So $u \in \sigma \cap P^{\mathsf{dur}}_{A \multimap B}$, $t \in \tau \cap P^{\mathsf{dur}}_{A' \multimap B'}$. So $s \in (S \cap P^{\mathsf{dur}}_{A \multimap B}) \otimes (T \cap P^{\mathsf{dur}}_{A' \multimap B'}) \cap P^{\mathsf{dur}}_{A \otimes A' \multimap B \otimes B'}$ □

It immediately implies that:

Proposition J.53. *For any given*

$$\sigma : A \multimap B \in \mathbf{Conc} \qquad\qquad \sigma' : A' \multimap B' \in \mathbf{Conc}$$

we have

$$\mathsf{Dur}\ (\sigma \otimes \tau) = \mathsf{Dur}\ \sigma \boxtimes \mathsf{Dur}\ \sigma'$$

Proposition J.54. $(\mathbf{Dur}, - \boxtimes -, \mathbf{1})$ *defines a symmetric monoidal category.*

Proof. Functoriality follows from the fact that $- \boxtimes -$ is the composition of two functors.

We start by defining the structural morphisms. The left and right unital are straight-forward. Indeed, they are given by

$$\lambda''_A := \mathsf{Dur}\ \lambda_A \qquad\qquad \rho''_A := \mathsf{Dur}\ \rho_A$$

where λ and ρ are the left and right unitals in **Crash**. The braiding and associator are given b

Now we have enough tools to define the braiding and associator in **Dur**

$$\beta''_{A,B} := \mathsf{Dur}\ \beta_{A,B} \quad \text{and} \quad \alpha''_{A,B,C} := \mathsf{Dur}\ \alpha_{A,B,C}$$

It immediately follows from functoriality of $- \cap P^{\mathsf{dur}}_{-}$, the definitions of the unitals, associators and braiding, and the fact the corresponding structural morphisms are natural transformations in **Crash** that the naturality squares still commute (not that for durable σ, $\sigma \cap P^{\mathsf{dur}} = \sigma$).

The coherence diagrams follow from functoriality of Dur $-$ together with the fact that Dur $-$ distributes over $- \boxtimes -$ (Prop. J.53), by noting that all the structural morphisms in **Dur** were defined by lifting the corresponding structural morphisms in **Crash**, which is also why they are isomorphisms. □

J.10 Durable Linearizability

Proposition J.55. *For a strategy $\nu : A^{\flat} \in \underline{\mathbf{Crash}}$ where A is a durable game*

$$\mathsf{dur}(\nu) = \{s \in P^{\mathsf{dur}}_A \mid s \textit{ is durably linearizable with respect to } \nu\}$$

Proof. Suppose $s \overset{\mathsf{dur}}{\rightsquigarrow} t$ with $t \in \nu$. So first, by definition, s is durable. Then, $\mathsf{ops}(s) \rightsquigarrow t$, so that $\mathsf{ops}(s) \in K_{\mathsf{Conc}}\ \nu$. But then, note that $\pi_{\Upsilon}(s) = \mathsf{ops}(s)$ as s is durable so that $s \in (K_{\mathsf{Conc}}\ \nu)^{\sharp} \cap P^{\mathsf{dur}}_A \subseteq \mathsf{dur}(\nu)$.

For the other direction, suppose $s \in \mathsf{dur}(\nu)$. Then, $s \in P^{\mathsf{dur}}_A$ and $s \in (K_{\mathsf{Conc}}\ \nu)^{\sharp}$. But then, $\mathsf{ops}(s) = \pi_{\Upsilon}(s) \in K_{\mathsf{Conc}}\ \nu$, so that $\mathsf{ops}(s) \rightsquigarrow t$ [31], and therefore $s \overset{\mathsf{dur}}{\rightsquigarrow} t$. □

Corollary J.56. *For a game $A \in \underline{\mathbf{Crash}}$, $\nu' : A \in \underline{\mathbf{Crash}}$ is durable linearizable to $\nu : A^{\flat} \in \underline{\mathbf{Conc}}$ if and only if $\nu' \subseteq \mathsf{dur}(\nu)$.*

Proposition J.57. *For any durable $\sigma : A \in \underline{\mathbf{Conc}}$ and $\tau : B \in \underline{\mathbf{Conc}}$*

$$\mathsf{dur}(\sigma \otimes \tau) = \mathsf{dur}(\sigma) \boxtimes \mathsf{dur}(\tau)$$

Proof.

$$\begin{aligned} \mathsf{dur}(\sigma \otimes \tau) &= (K_{\mathrm{Conc}}\ (\sigma \otimes \tau))^{\sharp} \cap P^{\mathsf{dur}}_{A\otimes A' \multimap B \otimes B'} \\ &= (K_{\mathrm{Conc}}\ \sigma \otimes K_{\mathrm{Conc}}\ \tau)^{\sharp} \cap P^{\mathsf{dur}}_{A\otimes A' \multimap B \otimes B'} \\ &= (K_{\mathrm{Conc}}\ \sigma^{\sharp} \otimes K_{\mathrm{Conc}}\ \tau^{\sharp}) \cap P^{\mathsf{dur}}_{A\otimes A' \multimap B \otimes B'} \\ &= (K_{\mathrm{Conc}}\ \sigma^{\sharp} \cap P^{\mathsf{dur}}_{A \multimap B}) \otimes (K_{\mathrm{Conc}}\ \tau^{\sharp} \cap P^{\mathsf{dur}}_{A' \multimap B'}) \cap P^{\mathsf{dur}}_{A\otimes A' \multimap B \otimes B'} \\ &= \mathsf{dur}(\sigma) \boxtimes \mathsf{dur}(\tau) \end{aligned}$$

□

Proposition J.58. *For $\sigma, \tau : A \multimap B \in$ **Dur** and $\sigma', \tau' : A' \multimap B' \in$ **Dur** we have*

$$\sigma \boxtimes \sigma' \subseteq \tau \boxtimes \tau' \implies \sigma \subseteq \tau \wedge \sigma' \subseteq \tau'$$

Proof. For given $\sigma, \tau : A \multimap B$ and $\sigma', \tau' : A' \multimap B'$ suppose $\sigma \boxtimes \sigma' \subseteq \tau \boxtimes \tau'$.

Now let's try to show $\sigma \subseteq \tau$. Fix $s \in \sigma$, by definition s can be decomposed as

$$s = s_1 \cdot \lightning \cdot \ldots \cdot \lightning \cdot s_{n+1}$$

By $\lightning$−receptive we know

$$t := \epsilon \cdot \lightning \cdot \ldots \cdot \lightning \cdot \epsilon \in \sigma'$$

we know that $s \boxtimes t \in \tau \boxtimes \tau'$ but by definition we know it force that $s \in \tau$. So $\sigma \subseteq \tau$

The proof of $\sigma' \subseteq \tau'$ is similar. □

Locality. For $\nu'_A : A, \nu'_B : B \in$ **Dur** and $\nu_A : A, \nu_B : B \in$ <u>**Conc**</u>:

$$\nu'_A \overset{\mathsf{dur}}{\rightsquigarrow} \nu_A \text{ and } \nu'_B \overset{\mathsf{dur}}{\rightsquigarrow} \nu_B \text{ if and only if } \nu'_A \boxtimes \nu'_B \overset{\mathsf{dur}}{\rightsquigarrow} \nu_A \otimes \nu_B$$

□

Proof.

$$\nu'_A \boxtimes \nu'_B \subseteq \mathsf{dur}(\nu_A \otimes \nu_B) = \mathsf{dur}(\nu_A) \boxtimes \mathsf{dur}(\nu_B) \iff \nu'_A \subseteq \mathsf{dur}(\nu_A) \wedge \nu'_B \subseteq \mathsf{dur}(\nu_B)$$

□

Proposition J.59. *Let $A, B \in$ <u>Crash</u>. Then $\nu'_A : A$ is durably linearizable to $\nu_A : A^{\flat}$ if and only if whenever $\sigma : (A \multimap B)^{\flat} \in$ **Conc** implements a concurrent object linearizable to ν_B using ν_A, then* $\mathsf{dur}(\sigma) : A \multimap B$ *implements an object durably linearizable to ν_B using ν'_A.*

Proof. For the forward direction, we have that by assumption

$$\nu_A; \sigma \subseteq K_{\mathrm{Conc}}\ \nu_B$$

And by lax functoriality of $\mathsf{dur}(-)$

$$\nu'_A; \mathsf{dur}(\sigma) \subseteq \mathsf{dur}(\nu_A); \mathsf{dur}(\sigma) \subseteq \mathsf{dur}(\nu_A; \sigma) \subseteq \mathsf{dur}(K_{\mathrm{Conc}}\ \nu_B) = \mathsf{dur}(\nu_B)$$

For the backward direction, note that

$$\nu_A; \mathsf{ccopy}_{A^{\flat}} = K_{\mathrm{Conc}}\ \nu_A$$

So, by assumption,

$$\nu'_A; \mathsf{dur}(\mathsf{ccopy}_{A^{\flat}}) \subseteq \mathsf{dur}(K_{\mathrm{Conc}}\ \nu_A)$$

and hence,

$$\nu'_A = \nu'_A; \mathsf{durcopy}_A = \nu'_A; \mathsf{dur}(\mathsf{ccopy}_{A^{\flat}}) \subseteq \mathsf{dur}(K_{\mathrm{Conc}}\ \nu_A) = \mathsf{dur}(\nu_A)$$

□