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), 29 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 [39].

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:

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

What this signature expresses is that BCell provides three operations: load(), which takes unit $() \in \mathbf{1}$ as argument and returns some value in Val; store(v), which takes a value $v \in \mathsf{Val}$ as argument, and returns the unit $\mathsf{ok} \in \mathbf{1}$; and 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 flush() invocation [34, 35]. 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):

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

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

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

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:

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

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 $\lightning$ 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. 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.

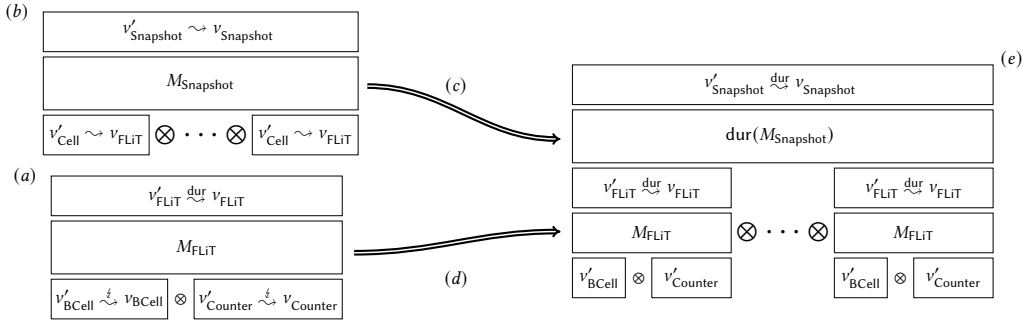


Fig. 2. (a) Using our program logic for durable linearizability, we verify the FLiT implementation; (b) Using the program logic for compositional linearizability we verify the crash-less snapshot object; (c) Using the FLiT correctness theorem, we lift the crash-less snapshot object into durably linearizable 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 ν'_{Cell} (which is Herlihy-Wing linearizable to ν_{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 TR [33].

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 $\mathbb{P}_M^{\mathsf{conc}}$.

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 $s_i \in \mathbb{P}_M^{\mathsf{conc}}$. 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 $\mathsf{epo}_i(s) := s_i$. We denote the set of all well-formed crash-aware traces over M by $\mathbb{P}_M^{\lightning}$.

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. $\nu \subseteq \mathbb{P}_M^{\mathsf{conc}}$, we call it a *crash-less specification*, and if it has crash-aware traces, i.e. $\nu \subseteq \mathbb{P}_M^{\lightning}$, 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 $s \in \mathbb{P}_M^{\mathsf{conc}}$ is linearizable to a crash-less trace $t \in \mathbb{P}_M^{\mathsf{conc}}$ 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 $s \rightsquigarrow t$ when s is linearizable to t. We say a crash-less specification ν' linearizes to another one ν, written $\nu' \rightsquigarrow \nu$, 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 $s \in \mathbb{P}_M^{\lightning}$ is *crash-aware linearizable* to a trace $t \in \mathbb{P}_M^{\lightning}$ when

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

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

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

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

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 $s \in \mathbb{P}_M^{\lightning}$ is *durable* when:

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

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

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

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:

$$\boldsymbol{\alpha_1}\text{:}\mathsf{store}(1) \cdot \boldsymbol{\alpha_2}\text{:}\mathsf{load}() \cdot \boldsymbol{\alpha_1}\text{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}\text{:}0 \cdot \boldsymbol{\alpha_2}\text{:}\mathsf{flush}() \cdot \boldsymbol{\alpha_1}\text{:}\mathsf{store}(2) \cdot \boldsymbol{\alpha_1}\text{:}\mathsf{ok} \cdot \lightning \cdot \boldsymbol{\alpha_3}\text{:}\mathsf{load}() \cdot \boldsymbol{\alpha_3}\text{:}1$$

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

$$\boldsymbol{\alpha_2}\text{:}\mathsf{load}() \cdot \boldsymbol{\alpha_2}\text{:}0 \cdot \boldsymbol{\alpha_1}\text{:}\mathsf{store}(1) \cdot \boldsymbol{\alpha_1}\text{:}\mathsf{ok} \cdot \boldsymbol{\alpha_2}\text{:}\mathsf{flush}() \cdot \boldsymbol{\alpha_2}\text{:}\mathsf{ok} \cdot \boldsymbol{\alpha_1}\text{:}\mathsf{store}(2) \cdot \boldsymbol{\alpha_1}\text{:}\mathsf{ok} \cdot \lightning \cdot \boldsymbol{\alpha_3}\text{:}\mathsf{load}() \cdot \boldsymbol{\alpha_3}\text{:}1$$

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:

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

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):

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

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

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

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

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 ν, we can construct a crash-aware specification $\mathsf{vol}(\nu)$ of a volatile version of that object by the Kleene algebra formula $\mathsf{vol}(\nu) := (\nu \cdot \lightning)^* \cdot \nu$.

For example, given the usual atomic counter specification ν_{Counter}, the following trace is allowed by $\mathsf{vol}(\nu_{\mathsf{Counter}})$ (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 ν' correctly implements a volatile version of a crash-less object ν when $\nu' \overset{\lightning}{\rightsquigarrow} \mathsf{vol}(\nu)$. With our methods, it is easy to show that

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

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

$$\nu_A; M \rightsquigarrow \nu_B \implies \mathsf{vol}(\nu_A); \mathsf{vol}(M) \overset{\lightning}{\rightsquigarrow} \mathsf{vol}(\nu_B)$$

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 $\mathrm{par}(-), \mathrm{ar}(-) : E \to \mathbf{Set}$ of a set of parameters $\mathrm{par}(e)$ and a set of return values $\mathrm{ar}(e)$ for each operation $e \in E$. This is conveniently described by the following notation.

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

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:

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

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

in other words, moves in $M_{\dagger E}$ are either $\boldsymbol{\alpha}{:}e(a)$ for $e \in E[\alpha]$ and $a \in \mathrm{par}(e)$, in which case $\lambda_{\dagger E}(\boldsymbol{\alpha}{:}e(a)) = \boldsymbol{\alpha}{:}O$, or $\boldsymbol{\alpha}{:}v$ with $v \in \mathrm{ar}(e)$, 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 $P_A \subseteq \mathbb{P}^{\mathrm{conc}}_{M_A}$ 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:

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

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 $s\restriction_{-,B,C}$.

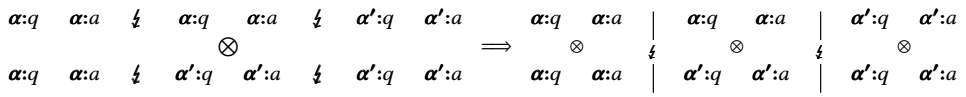
We now define horizontal composition of games, and the affine arrow.

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

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

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

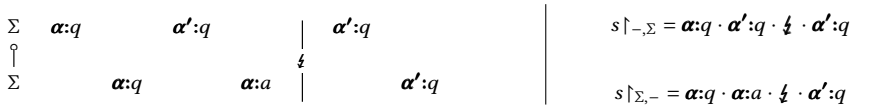
It is implicit in this definition that by composing in parallel the two crash-aware plays, the resulting set of traces synchronizes the crash events, merging them into a single crash event and then producing any (locally sequential) parallel composition of the subtraces appearing in each epoch. Consider,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



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):



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

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

$\lightning$-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 **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, **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.

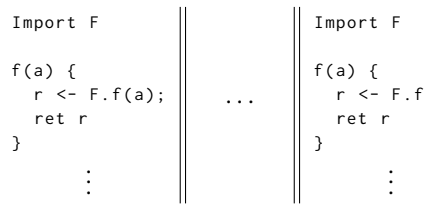


Fig. 3. Code corresponding to the copycat strategy $\mathsf{crashcopy}_{\dagger F \multimap \dagger F} : \dagger F \multimap \dagger F$

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 **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:

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

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 TR). 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 $A \otimes B \in \underline{\textbf{Crash}}$. 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* $\underline{\textbf{Crash}}$ *to strategies saturated with respect to* crashcopy. *Then,* $(\textbf{Crash}, - \otimes -, \mathbf{1})$ *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* ($\overset{\lightning}{\rightsquigarrow}$). 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, $\underline{\textbf{Crash}}$.

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

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

for $\tau : A \multimap B \in \underline{\textbf{Crash}}$. 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 $\nu' \subseteq K_\lightning\ \nu$. Indeed, we are able to show the following characterization of $K_\lightning\ -$, which provides a concrete characterization of $K_\lightning\ \nu$ as the set of all plays that are crash-aware linearizable w.r.t. ν.

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

It follows immediately from this characterization that

Proposition 4.2. *v' is crash-aware linearizable w.r.t. v if and only if $v' \subseteq K_\natural\ v$.*

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).

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

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 TR). 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:

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

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 $v'_{\mathsf{Disk}} : \dagger\mathsf{Disk}$ is crash-aware linearizable to a specification $v_{\mathsf{Disk}} : \dagger\mathsf{Disk}$ which guarantees read and writes are persistent and atomic. The disk array specification $v'_{\mathsf{Disk}[N]}$ is required to be crash-aware linearizable to the horizontal composition of N atomic disk specifications $v_{\mathsf{Disk}[N]} := \otimes_{i \in [N]} v_{\mathsf{Disk}}$. The concrete object v'_{File} is required to be crash-aware linearizable to a specification v_{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.

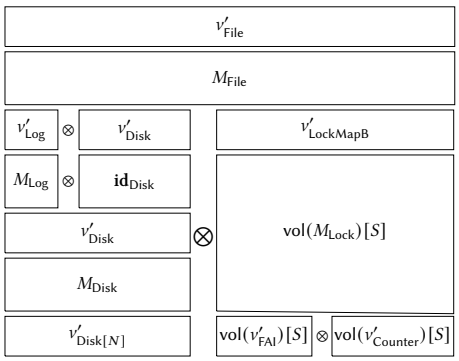


Fig. 4. The structure of our File example.

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 ν'_{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 ν'_{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 $\sigma^\sharp : A$.

$$\sigma^\flat := \{\mathsf{ops}(s) \in \mathbb{P}^{\mathsf{conc}}_{M_A} \mid s \in \sigma\} \qquad \sigma^\sharp := \{s \in \mathbb{P}^{\not\downarrow}_{M_A} \mid \mathsf{ops}(s) \in \sigma\}$$

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: $\mathsf{str}(\sigma) := K_{\not\downarrow}\ \sigma^\sharp$

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

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

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

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 $v'_A : A, v'_B : B$ and crash-less strategies $v_A : A, v_B : B$: $v'_A \subseteq \mathsf{str}(v_A)$ and $v'_B \subseteq \mathsf{str}(v_B)$ if and only if $v'_A \otimes v'_B \subseteq \mathsf{str}(v_A \otimes v_B)$*

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 $P_A^{\not\downarrow}$ 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 $v : A^\flat$ the durable strategy $\mathsf{dur}(v) : A$ defined by $\mathsf{dur}(v) : A := (K_{\mathsf{Conc}}\, v)^\sharp \cap P_A^{\mathsf{dur}}$.

The operation $K_{\mathsf{Conc}}\, -$ in the formula is defined by Oliveira Vale et al. [31] similarly to $K_{\not\downarrow}\, -$, but in the crash-less setting. It may be more intuitively understood through their result that:

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

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

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

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

Proposition 4.11 (Observational Refinement and Locality).

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

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 TR 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 $v' : \dagger(E \circlearrowleft R)$ recovery-refines to $v : \dagger E$ when $v'\restriction_E \subseteq v$.

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

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

5.2 Programming Language

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

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

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 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. $R_F = \{r : \mathbf{1} \to \mathbf{1}\}$. 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 $x \leftarrow B.\mathsf{load}(); B.\mathsf{flush}()$ 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 $p \in v_{\mathsf{FLiT}}$ of one FLiT memory cell, we can define the evaluation function $\mathsf{fstate} : v_{\mathsf{FLiT}} \to \mathsf{Val}$ 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.

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

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

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

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

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

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

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 $(\Delta, s) \in \mathsf{ModState} := (\Upsilon \to \mathsf{Env}) \times P^\dagger_{(E \cup R_E) \multimap \dagger(F \cup R_F)}$ 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 ($\vec{r} = \mathrm{perm}(R_E)$) and then the overlay recovery $M[\alpha]^r$. This is achieved by using sequence to sequence a list of commands (note that $\mathrm{reserved}(x_r)$ 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.

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

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.

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

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.

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

The objective of our program logic is to establish the judgment

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

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 TR) 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 $I : \Upsilon \to \mathcal{P}(\mathsf{Config})$ for the implementation and then verify regular procedures and the recovery separately.

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

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).

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

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.

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

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 TR for other proof rules, which are standard.

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

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 $\mathcal{R}$.

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

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* $\vdash M : (\nu'_E, \nu_E) \to \langle \nu'_F, \nu_F \rangle$ *is provable, and* (ν'_E, ν_E) *is a valid crash-aware interface, and* $\nu'_F = \nu'_E; [\![M]\!]_{R_E}$, *then* $\langle \nu'_F, \nu_F \rangle$ *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 ν'_{BCell} are crash-aware linearizable to its specification ν_{BCell}, which we can define through an interpretation function, $\mathsf{mstate} : \nu_{\mathsf{BCell}} \to \mathcal{P}(\mathsf{Val} \times \mathsf{Val})$, 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 ν_{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 ν_{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 $\mathsf{fstate}(\rho)$ 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 $\mathsf{store}(v_b)$ 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 $\mathsf{store}(v_b)$ operation in the ghost list B, in case it has persisted, or is equal to the current overlay state $\mathsf{fstate}(\rho)$, 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 TR 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.

$$(s, \rho, B)\mathcal{G}_{\text{load-f}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v.\mathsf{Flushed}(s, B) \wedge (v, v) \in \mathsf{mstate}(s\restriction_{\mathsf{BCell}^{\not z}}) \wedge v = \mathsf{fstate}(\rho) \wedge \\ \mathsf{lin}(\rho') = \mathsf{lin}(\rho) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \wedge B' = \epsilon \wedge s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \end{pmatrix} \tag{3}$$

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

$$(s, \rho, B)\mathcal{G}_{\text{load-us}}[\alpha](s', \rho', B') \iff \begin{pmatrix} \exists v, B_1, B_2.\mathsf{Unsynced}(s, B) \wedge (v, v) \in \mathsf{mstate}(s\restriction_{\mathsf{BCell}^{\not z}}) \wedge \\ s' = s \cdot \boldsymbol{\alpha}{:}M.\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \wedge \begin{pmatrix} (B = B_1 \cdot B_2 \wedge \mathsf{last}(B_1) = \mathsf{store}(v)) \\ \vee (\mathsf{fstate}(\rho) = v \wedge B_1 = \epsilon) \end{pmatrix} \wedge \\ \mathsf{lin}(\rho') = \mathsf{merge}(\mathsf{lin}(\rho), B_1) \cdot \boldsymbol{\alpha}{:}\mathsf{load} \cdot \boldsymbol{\alpha}{:}v \wedge B' = \epsilon \end{pmatrix} \tag{5}$$

$\{\mathsf{invoke}_\alpha(\mathsf{load}) \circ I\}$
1: load(){
$\{I \wedge \boldsymbol{\alpha}\text{:load} \in s_O \wedge (\text{Flushed} \vee \text{Unflushed} \vee \text{Unsynced})\}$ // P_{load}
2: $v \leftarrow M.\text{load}()$; // load-f/load-uf/load-us
$\left\{I \wedge \begin{pmatrix} (\text{Flushed} \wedge \text{last}(\pi_\alpha(\rho)) = v) \vee \\ (\text{Unflushed} \wedge (\exists B'.B' \cdot \boldsymbol{\alpha'}\text{:store}(v) \cdot \boldsymbol{\alpha}\text{:load} \sqsubseteq B \vee \text{last}(\pi_\alpha(\rho)) = v)) \end{pmatrix}\right\}$ // Q_{load}
3: $n \leftarrow C.\text{get}()$;
$\left\{I \wedge \begin{pmatrix} (n = 0 \wedge \text{last}(\pi_\alpha(\rho)) = v) \vee \\ (n \neq 0 \wedge (\exists B'.B' \cdot \boldsymbol{\alpha'}\text{:store}(v) \cdot \boldsymbol{\alpha}\text{:load} \sqsubseteq B \vee \text{last}(\pi_\alpha(\rho)) = v)) \end{pmatrix} \wedge (\text{Flushed} \vee \text{Unflushed})\right\}$
4: if($n \neq 0$){
$\{I \wedge (\text{Flushed} \vee \text{Unflushed}) \wedge (\exists B'.B' \cdot \boldsymbol{\alpha'}\text{:store}(v) \cdot \boldsymbol{\alpha}\text{:load} \sqsubseteq B \vee \text{last}(\pi_\alpha(\rho)) = v)\}$
5: $M.\text{flush}()$; // flush
$\{I \wedge \text{last}(\pi_\alpha(\rho)) = v\}$
6: }
$\{I \wedge \text{last}(\pi_\alpha(\rho)) = v\}$
7: ret v
8: }
$\{\mathsf{returned}_\alpha(\mathsf{load}) \circ I\}\{\top\}$

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 $\text{fstate}(\rho)$ is exactly the same as the content in the underlay memory cell v. Therefore, we can simply extend the linearized prefix $\text{lin}(\rho)$ in ρ with $\boldsymbol{\alpha}$: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 $\boldsymbol{\alpha}$:load $\cdot$ $\boldsymbol{\alpha}$:v to $\text{lin}(\rho)$. 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 $\text{lin}(\rho)$. Based on the return value v of the underlay load, we decide how to handle buffered operations

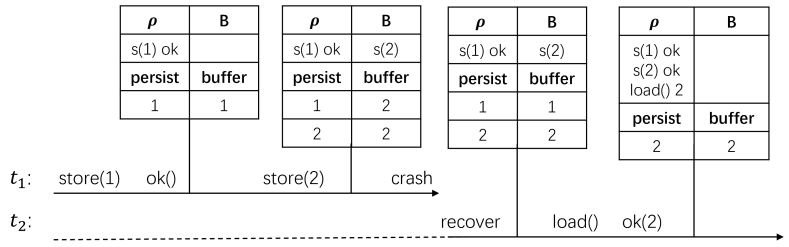


Fig. 7. External Linearization Point and Crash: the tables above the timeline show the content of the linearized trace ρ and the ghost list B in the first two rows and the mstate content in the remaining rows. $s(-)$ is a shorthand for the store$(-)$ operation.

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 TR 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, $M[\alpha]^{r_\varnothing} := \mathsf{r()\ \{\ ret\ ok\ \}}$. According to the Recover Impl rule, we need to prove the hexad $\mathtt{ID}, \top \vDash_\alpha \{I\} M[\alpha]^{r_\varnothing} \{I\}\{I\}$ 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 $\vdash M_{\mathsf{FLiT}} : (\nu'_{\mathsf{BCell}} \otimes \nu'_{\mathsf{Counter}}, \nu_{\mathsf{BCell}} \otimes \nu_{\mathsf{Counter}}) \rightarrow \langle \nu'_{\mathsf{FLiT}}, \nu_{\mathsf{FLiT}} \rangle$, 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 [36, 37]. 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 **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 $\mathsf{dur}(-)$, 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 [39]. 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, 38].

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] Arthur Oliveira Vale, Zhongye Wang, Yixuan Chen, Peixin You, and Zhong Shao. 2024. *Compositionality and Observational Refinement for Linearizability with Crashes.* Technical Report YALEU/DCS/TR-1570. Yale Univ. https://flint.cs.yale.edu/publications/crashlin.html
[34] 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
[35] 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
[36] Uday S. Reddy. 1993. *A Linear Logic Model of State.* Technical Report. Dept. of Computer Science, UIUC, Urbana, IL.
[37] 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
[38] 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
[39] 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

Received 2024-04-06; accepted 2024-08-18