The Government of the Russian Federation
Federal State Autonomous Institution for Higher Professional Education
National Research University Higher School of Economics
St. Petersburg Branch
St. Petersburg School of Physics, Mathematics, and Computer Science

Ilya Kaysin

Persistency semantics of the ext4 filesystem

Master dissertation

Area of studies 01.04.02 «Applied Mathematics and Informatics»
Master Program «Software Development and Data Analysis»

Research supervisor:
Associate Professor, PhD
Anton Podkopaev

Reviewer:
Research Associate, PhD
Christopher Pulte

Saint Petersburg
2021

Contents

1 Introduction

Modern applications substantially depend on file I/O. It is used both auxiliarly to persist data between sessions (e.g., in browsers) and as a primary functionality (e.g., in databases and text editors). Nevertheless, numerous bug reports [8, 10] indicate that programmers hardly understand modern filesystems. Because of sophisticated algorithms and optimizations filesystems use under the hood, programs can exhibit behaviors that are problematic to predict. For example, suppose that an application writes `'A'` to `a.txt`, then `'B'` to `b.txt`, and afterwards the operating system crashes. One could have assumed that after the restart, it would not be possible to observe ’`B`’ written in `b.txt` without `'A'` having also been written in `a.txt`. However, this assumption about ordering is violated by the vast majority of modern filesystems, e.g., `ext4`, NTFS, ZFS, Btrfs. This is because if the sectors of `a.txt` are *preceded* by the sectors of `b.txt` on the disk, a filesystem will save an extra disk rotation and persist the files out of order: `b.txt` before `a.txt`.

As shown by this example, the actual state of the filesystem upon a crash is not determined solely by the code. The same program crashed at the same time can leave the files with different content. The exact guarantees a filesystem provides about the content observable upon a crash are called *persistency* semantics.

However, in a *concurrent* environment, filesystems have another degree of freedom—*consistency*. Because multiple threads can simultaneously access the same shared resources—files—the behavior depends on scheduling, buffering, compiler and hardware optimizations. For example, if a thread reads a file updated by another thread, it is possible to observe the content non-atomically (i.e., having been *half*-updated). Weak consistency semantics, also known as a weak memory model [3, 19, 27], is a mathematical abstraction encompassing this kind of behaviors. It formally defines what values a thread can observe while working simultaneously with other threads. Although consistency semantics is an established research area, which advent in the early 2000s had a significant impact on formal methods and language standards (see, e.g., [40]), there is almost no work on *filesystem* weak consistency.

The combination of consistency effects of the memory and persistency effects of the disk makes it challenging to use file I/O correctly. To ensure wanted atomicity and ordering, programmers must synchronize both: stores on disk (using, for example, `sync`/`fsync` system calls) and interaction between threads (via locks, atomic variables and other primitives). The exact semantics of the synchronization and other operations is distributed across documentation and standards (e.g., POSIX [30]), which are often written in an ambiguous and even self-contradictory manner [26].

As bugs related to file I/O are hard to detect *manually*, one would suggest using formal methods to verify programs *automatically*. The development of such tool has become a goal of the PerSeVerE project [28]. PerSeVerE is a stateless model checker developed on top of GenMC [18]. It employs an effective Dynamic Partial Order Reduction [1] approach to verify consistency and persistency. However, like any formal method, PerSeVerE is based on a mathematical representation of the semantics, i.e., it requires a formal model of a filesystem.

To be used in model checking, the filesystem formal model must follow specific requirements. The first challenge is efficiency: the semantics must be implementable in adequate time and space complexity. Second, the model must account for the language semantics in a *concurrent* environment when multiple threads run in parallel. A well-known *declarative* semantics approach [3] (hereafter will be referred to as *axiomatic* semantics, not to be confused with Hoare logic) tackles both of these challenges, and thus, it is the primary candidate to base the filesystem semantics on. Additionally, modeling the semantics in an axiomatic way allows seamless adapting the *existing* model checker GenMC verifying programs interacting with the filesystem.

Ext4 [23] is the default filesystem for many Linux distributions, e.g., Ubuntu, Debian, Fedora, Arch Linux. Optimizations and heuristics, while making `ext4` fast, also make it bug-vulnerable as it admits many counter-intuitive (so-called *weak*) program behaviors. Despite its ubiquity, the `ext4` filesystem has not been formalized yet, and thus, there are no tools that enable checking persistency violations under `ext4`.

In this work, we present a formal model of the `ext4` filesystem integrated with the weak memory consistency semantics of C/C++. In order to build this model, we first developed a general persistency model framework based on the axiomatic approach for weak memory semantics. Next, we specialized this framework with the `ext4` features acquired from reading manuals, discussion with filesystem developers, consulting the `ext4` implementation, and litmus testing. As a basis for PerSeVerE model checker [28], it allowed us to uncover several crash-safety bugs in popular text editors (`emacs`, `vim`, and `nano`). The great diversity of weak behaviors `ext4` exhibits required our formal approach to be general to a high degree. It makes us believe that the developed framework can be extended for the formalization of other filesystems and non-volatile memory models.

2 Problem statement

The purpose of this work is to develop a formal model of the `ext4` semantics. For this objective it is required to solve the problems listed below:

(i) Introduce a general model framework that enables covering the persistency and consistency aspects of `ext4`. In order to be independent of the language/hardware platform, the framework must be compatible with different underlying weak *memory* models.

(ii) Extract the particular guarantees provided by the `ext4` filesystem. Fit them into the developed framework.

(iii) Ensure the capability of the obtained model to be *efficiently* used in formal verification algorithms.

3 Background

In this section, we first overview the Linux I/O stack—the context in which any filesystem exists. Then we introduce types of filesystem guarantees that programmers rely on and describe how they are implemented in `ext4`. Next, we formally present the notion of axiomatic semantics—the modeling approach that we use in formalization in §4. Finally, we overview model checking technique by the example of PerSeVerE—the algorithm in which the formal semantics was applied (see §7).

3.1 Linux File I/O

VFS and Page Cache. In Linux, data goes through several layers before reaching the disk, as shown in Fig. 1. The first two layers are the *Virtual File System* (VFS) [22] and the kernel *page cache*. VFS is an abstract software layer in the kernel that provides a common API to different filesystem implementations; the page cache sits between VFS and the filesystem implementation (in our case `ext4`), and its purpose is to cache disk data in memory. The page cache comprises physical RAM pages, which in turn contain a number of disk *blocks* (see below).

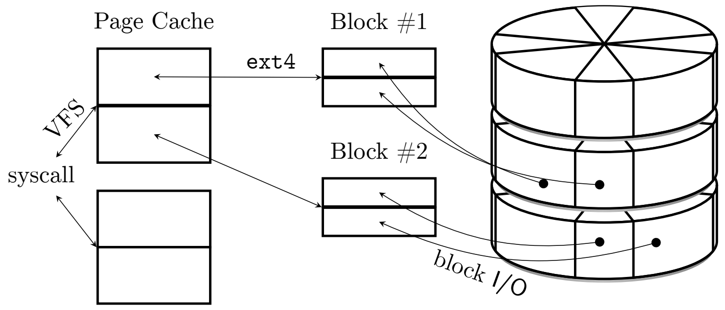


Figure 1: Different I/O layers until data reaches disk

In most cases, interacting with VFS/page cache does not imply interacting with the disk. For instance, a call to write does not guarantee that the written data is persisted to disk prior to returning, and a call to read does not necessarily fetch the desired data directly from the disk. Instead, both I/O operations simply manipulate the page cache; in fact, with the page cache, the VFS has to barely touch filesystem-specific code, if all the desired data is in the page cache already.

Files in memory. The VFS maintains the in-memory infrastructure of the filesystem. There, each file is represented by three tables: per-process File Descriptor Table, system-wide Open File Table, and Inode Table (Fig. 2).

The File Descriptor Table maps file descriptors (special non-negative integer indices, used to access the files) into Open File Table. Each entry in Open File Table (a.k.a. file description) stores the file lock, file flags, the current *offset* within the file (when reading/writing the file), and the index in the Inode Table, containing file metadata, along with a separate inode lock.

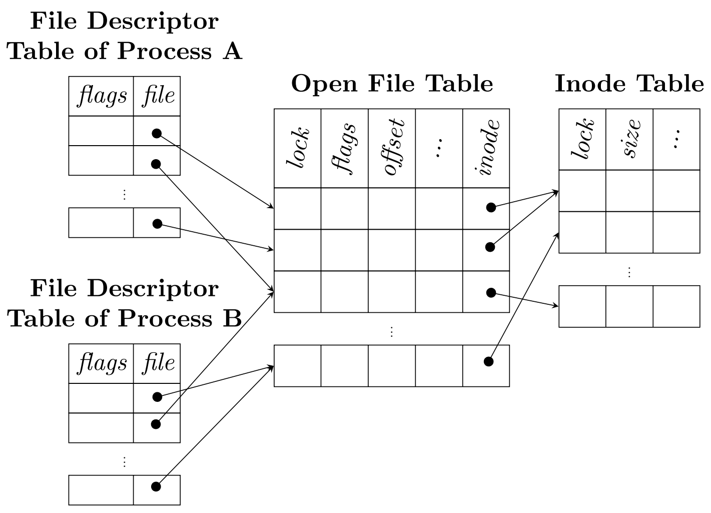


Figure 2: File descriptors, descriptions and inodes

It must be noted that there is one-to-one correspondence between inodes and actual file content in the filesystem. Contrarily, every inode can be referenced by *multiple* Open File Table entries if the file is open multiple times (either by one process or by different processes); and, in turn, each entry in the Open File Table can be referenced by several entries in the File Descriptor Tables. The latter is because of fork and dup system calls, which we, however, do not cover in our model.

Files on disk. The file representation on disk is filesystem-specific. Usually, file metadata is represented by the *on-disk* Inode Table (similar to the aforementioned *in-memory* Inode Table). The content of the file is stored separately.

Interaction with the disk must account for the physical properties of the underlying hardware. These properties are important as they affect how data is transferred to disk.

The smallest addressable unit on a disk is a *sector*. It holds `4KiB` in modern disks [2] and `512B` in older disks.

However, the filesystem communicates with the disk in logical units called *blocks*, denoting contiguous disk sectors (typically `4KiB` in Linux). For instance, to write a page from the page cache to disk, the OS issues a sequence of I/O requests over a number of blocks.

3.1.1 POSIX file operations

Applications interact with the filesystem via invoking system calls. The concrete interface Linux filesystems provide is specified by POSIX standard and includes the following operations, which we cover in details below: open, read, pread, write, pwrite, lseek, close, link, unlink, rename, fsync, sync. For convenience, hereafter we do not use the original POSIX system call syntax leaving out unimportant arguments.

Opening a file. Before accessing the file, one must open it using the open system call. As arguments, it takes the file name and flags describing the options, such as whether the file is to be created (`O_CREAT`) or truncated (`O_TRUNC`). The open system call returns a *file descriptor*, a non-negative integer, which is used by other system calls to access this file.

When open is called, the *virtual filesystem* (*VFS*) layer of the operating system updates its internal data structures (Fig. 1). First, it creates new entries in a per-process File Descriptor Table and in a system-wide Open File Table (to which entries the File Descriptor Table refers). Second, the operating system creates a new entry in Inode Table, but only if the opened file does not exist and `O_CREAT` flag is specified.

Reading/Writing a File. Once a file is opened, its contents can be read and written using read and write, respectively. These calls expect three arguments: a file descriptor referring to an open file, a buffer to store the data read/written, and the number of bytes to read/write. Given a file descriptor d_f, we write $r = \mathsf{read}\,(d_f, \texttt{count})$ to read `count` bytes from d_f into buffer r, and $\mathsf{write}\,(d_f, \texttt{buf})$ to write the whole buffer `buf` to d_f. For readability, we omit the error handling code.

The read/write calls access the file at the *offset* specified in the file description. This offset is initialized to 0 when the file is first opened, and is increased by the number of bytes read/written.

$$\begin{aligned} d_f &= \mathsf{open}\,(\text{“}\texttt{foo.txt}\text{”}, \texttt{O\_RDWR}); \\ r_0 &= \mathsf{read}\,(d_f, 1); \quad \textit{// reads “}\texttt{f}\textit{”} \\ r_1 &= \mathsf{read}\,(d_f, 2); \quad \textit{// reads “}\texttt{oo}\textit{”} \end{aligned}$$

Figure 3: Reading a file

For example, given the “`foo.txt`” file containing the string “`foo`”, the snippet in Fig. 3 reads the strings “`f`” and “`oo`” into r_0 and r_1, and sets the offset to the end of the file (`EOF`).

POSIX also supports accessing a file at a given offset using pread and pwrite. These calls take the absolute offset as an additional argument, and do not change the offset stored in the file description. Attempting to read beyond `EOF` results in reading 0 bytes, while writing beyond `EOF` extends the file size and fills the gap between `EOF` and the offset where the new data is to be written with zeros.

The Linux kernel, however, is not fully POSIX-compliant. For example, suppose that “`foo.txt`” above were opened with the `O_APPEND` flag, which sets the initial offset in the file description to `EOF`. The $\mathsf{pwrite}\,(d_f, \text{“}\texttt{bar}\text{”}, 0)$ call (at absolute offset 0) would then update “`foo.txt`” to contain the string “`foobar`” rather than the expected string “`bar`”. This is only one example of non-POSIX-compliant behavior exhibited by the Linux kernel. In general, the Linux kernel exhibits multiple other non-POSIX-compliant behaviors, which we strive to model precisely.

Seeking in a File. Given a file descriptor d_f, the lseek $(d_f, ...)$ system call updates the offset associated with d_f according to the given arguments: the offset may be set to an absolute value or to a value relative to predefined locations in the file (e.g., the file size or the current location). Indeed, lseek allows the offset to be set *beyond* EOF. Such a call does not alter the file size, but subsequent writes to the file will write at the offset specified by lseek and therefore increase the file size.

Note that if a file is opened with O_APPEND, lseek is of little use since (as per the POSIX standard) subsequent calls to write will reposition the offset to EOF before writing the data.

Closing a File. A file associated with a file descriptor d_f may be closed by calling close (d_f), which removes d_f from the file descriptor table of the calling process. If d_f is the only file descriptor associated with an open file description, then its resources are freed; otherwise, the file description is preserved so long as there are other file descriptors (in any process) associated with it.

Synchronization operations. As mentioned above, every file is stored simultaneously on disk and in memory. To enforce the synchronization between these two representations, developers can use special disk-flushing instructions or flags. The sync and fsync system calls can be used to flush persist-pending (data and metadata) writes to disk. Concretely, sync flushes all persist-pending writes across the *entire filesystem synchronously*: it waits for I/O to complete before returning. Analogously, given a file descriptor d_f, a call to fsync (d_f) synchronously flushes all persist-pending writes on d_f.

The effects of fsync can be emulated by opening a file with the O_SYNC *flag*. Using O_SYNC, all blocks written to the file by a write/pwrite call are flushed immediately after the call. In practice, the guarantees of O_SYNC are slightly stronger in that *all* earlier writes to the file are flushed, including those of the write request. For instance, consider a scenario where a process A opens a file "f.txt" without O_SYNC, while a concurrent process B opens "f.txt" with O_SYNC. If first A and then B each write several blocks to "f.txt", then all persist-pending blocks written to "f.txt" up to and including those of process B are flushed to disk, including those of A.

Directory Operations. Several system calls, referred to as *directory operations* in our model, can be used to manipulate the file inode and the directory containing the file. Examples of such operations in our model are: (i) creat (nl), which creates a new file named nl (and is equivalent to open with O_CREAT|O_WRONLY|O_TRUNC flags); (ii) link (nl_{old}, nl_{new}), which creates a new directory entry with name nl_{new} (if such entry does not already exist) referring to nl_{old}'s inode; (iii) unlink (nl), which deletes the entry named nl; and (iv) rename (nl_{old}, nl_{new}), which renames the entry named nl_{old} as nl_{new}. That is, rename (nl_{old}, nl_{new}) is similar to link (nl_{old}, nl_{new}) followed by unlink (nl_{old}).

Such operations exhibit interesting behaviors when interacting with open file descriptions. To see this, consider the following program, where "`bar.txt`" is unlinked immediately after creation:

$$d_f = \mathsf{creat}\,(\text{“}\texttt{bar.txt}\text{”}); \mathsf{unlink}\,(\text{“}\texttt{bar.txt}\text{”}); \mathsf{write}\,(d_f, \text{“}\texttt{bar}\text{”});$$

The question is whether the subsequent write is valid. When a file is unlinked, if the removed entry is the last entry on the file, then the file must be deleted with its allocated space made available for reuse. However, if the file is still open, then unlink does not delete it immediately; instead, it returns an error, and the file is eventually deleted once all its associated file descriptions are closed. As such, close and unlink may execute in either order with respect to one another, and processes with open file descriptions on the file can read/write the file until they close their file descriptions.

Similar observations hold of the interaction of other directory operations with the I/O operations described thus far. In general, directory operations on a file with open file descriptions do not hinder subsequent I/O calls that use these descriptions. They may, however, affect the outcome of subsequent I/O calls: e.g., if we call creat on an open file, creat will truncate the file size to 0.

3.2 Filesystem guarantees

In presence of multiple threads, filesystems (akin to memory) exhibit non-intuitive, so-called *weak* behaviors. For example, consider the following program comprising two threads:

$$\mathsf{pwrite}\,(d_f, \text{“}\texttt{xyz}\text{”}, 0); \;\Big\|\; r = \mathsf{pread}\,(d_f, 3, 0);$$

Having the file initially empty, one may assume that pread can only read either an empty string or "`xyz`". However, `ext4` does not guarantee *atomicity* of pwrite; as such, reading "`xy`" is also possible.

The guarantees filesystems provide for file operations can be described by two axes: consistency/persistency, and ordering/atomicity.

Consistency and Persistency. The *consistency* semantics of a file operation describes the way its effects become visible to *concurrent threads*. The *persistency* semantics comes to play only if a system crash occurs. It defines how the effects of file operations persist *on disk* thus determining the observable disk states upon recovery from a crash (e.g., power loss or software crash).

Ordering and Atomicity. A filesystem may or may not preserve ordering between certain operations. In the former case, it is guaranteed that the effects of the operations are performed (or, more precisely, are visible) in the same order as they proceed in the program. If the ordering guarantee is not provided, a filesystem is entitled to rearrange the instructions (e.g., for optimization purposes).

In the case of a non-atomic operation, it is possible to observe its interim state. For instance, reading only part of a file updated by `write`, as in the example above. On the contrary, an *atomic* operation has “all-or-nothing” semantics: its effects are either invisible or visible entirely.

This way, file operations are described by the following aspects: c-ordering, p-ordering, c-atomicity, and p-atomicity (where ‘c-’ stands for consistency and ‘p-’ stands for persistency) defined as follows:

- *c-ordering* defines the order in which the effects of the operations become *visible to the other threads*;
- *p-ordering* defines the order in which the effects of the operations *persist on disk*;
- *c-atomicity* holds for an operation iff its effects are *instantaneous for the other threads*;
- *p-atomicity* holds for an operation iff it is *atomic at the level of persistency*, i.e., its effects become durable instantaneously, and it its interim state is not observable upon a crash.

3.3 The `ext4` filesystem

Ext4 is a modern journaling filesystem, default for many widespread Linux distributions, e.g., Ubuntu, Debian, Fedora, Arch Linux. In most cases, file operations in `ext4` are implemented as described above. However, there are certain important peculiarities, that we would like to discuss.

3.3.1 Journaling

As a crash can occur at any time during the program execution, including during a system call, a filesystem must guarantee data integrity. Such guarantees do not pre-empt data loss, but merely ensure that the filesystem can be restored to a consistent state after a crash. For example, when appending to a file, the file size update must not persist before the appended data: if a crash occurs right after the size update persists, then invalid data can be read upon recovery.

To ensure data integrity, `ext4` employs *write-ahead logging* or *journaling* [39], which uses a transaction to record the intended changes in a *journal* (a designated disk area) before carrying out the changes. Once the transaction *commits*, the intended changes can be carried out in place. This way, if a crash occurs while enacting the changes, upon recovery one can simply replay the journal to bring the filesystem to a consistent state.

By default, `ext4` journals only metadata, e.g., the on-disk file inode (`data=ordered` journaling). As `ext4` stores the file metadata in a different place on disk than its data [9], this introduces dependencies between file data and metadata, leading to interesting persistency behaviors, as will be discussed in §5.

3.3.2 Delayed allocation

In most cases, `ext4` reading and writing operations provide p-atomicity on the level of blocks. However, there is an exception: when a file has preallocated blocks that are partially filled. For example, suppose that file "`f`" has its last block b_l already allocated on disk but not completely filled, and a crash occurs while a process performs an append to "`f`". Upon recovery, it is then possible to observe a disk state where the size of the file is partially increased (either to the end of b_l or to the size dictated by the append, whichever is smaller), but with '`0`'s appended to the end instead of the data written, as shown in Fig. 4.

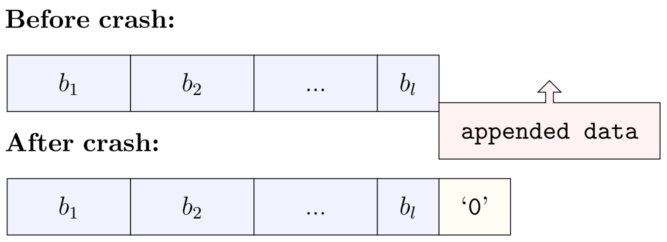


Figure 4: `ext4` p-atomicity violation: delayed allocation

This is because, as part of an optimization, `ext4` does *not* bind the data persist of the *preallocated blocks* to the transaction commit. Before elaborating on this optimization, let us briefly discuss block allocation under `ext4`.

Recall that writes happen asynchronously under `ext4`. This is to ensure that writes to the same page are merged, and that write blocks can be allocated more efficiently (a process called *delayed allocation*). Block allocation for an inode, however, requires that the inode be journaled to ensure filesystem consistency. Conversely, if a block is already preallocated when a write is issued, then the inode need not be journaled.

Thus, when extending files with preallocated blocks, `ext4` optimizes the number of times an inode is journaled. Since the inode is already included in the transaction when a write to a preallocated block is issued (as the inode contains other metadata that are journaled), `ext4` journals the inode with the updated size in the transaction, thus not journaling the same inode twice. However, it does so without binding the transaction commit to the data persist to avoid stalling the transaction. As such, this optimization decouples the on-disk size update from the data persist, thus allowing the transaction to commit before the data persists, leading to the behavior shown in Fig. 4.

3.4 Axiomatic semantics

As will be present in §4 and §5, we formalize the `ext4` semantics. In other words, we provide a mathematical model which defines the `ext4`'s guarantees for any arbitrary program. As the `ext4` semantics largely depends on *memory*, we formalize it in a fashion

$$\texttt{f.txt} = \text{“}\texttt{foo}\text{”}$$

$$\begin{array}{l||l}
d_f = \mathsf{open}\,(\text{“}\texttt{f.txt}\text{”}); & d_f = \mathsf{open}\,(\text{“}\texttt{f.txt}\text{”}); \\
\mathsf{pwrite}\,(d_f, \text{“}\texttt{bar}\text{”}, 0); & r = \mathsf{pread}\,(d_f, 3, 0); \\
\mathsf{close}\,(d_f); & \mathsf{close}\,(d_f);
\end{array}$$

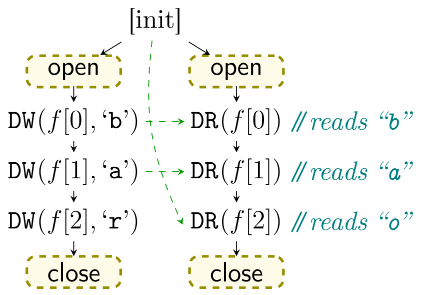


Figure 5: A concurrent program (left) and one of its possible executions (right). The green dashed edges denote the `rf` order, the black edges denote `po`.

usual for *weak memory models*, that is in an *axiomatic* way [3].

In axiomatic models, each program execution is represented by a graph. The vertices of the graph are operations such as loads (in our case, for example, disk-reads `DR`) and stores (e.g., disk-writes `DW`). The edges define different relations on the operations.

There are two relations common for all axiomatic models: `po` (*program-order*) and `rf` (*reads-from*). The `po` relation corresponds to the order in which instructions proceed in the program. `rf` defines the semantics of *reading* operations: for any *read* operation, there is an incoming `rf`-edge from the operation writing the read value.

Consider the example at Fig. 5. Assuming `f.txt` initialized with "`foo`", two threads run concurrently. The left thread overwrites the file from "`foo`" to "`bar`" and the right thread reads the file content.

In this case, `ext4` does not provide c-atomicity, thus it is possible for the right thread to read, for example, "`bao`". As such, this behavior must be allowed by the semantics. The graph justifying this behavior is shown in Fig. 5 (right). As depicted by the `rf` edges, the first two bytes of the file are read from the overwrites made by the left thread, whereas the last byte is read from the initialization operation, and thus, it is "`o`" but not "`r`".

The essence of any semantics is distinguishing possible and impossible executions. Axiomatic semantics is named "*axiomatic*" because it filters "good" executions via a set of rules called *axioms*. Such rules are usually formulated in terms of graphs. For example, one of the common axioms forbids the executions that contain cycles comprising `po` and `rf` edges. In our work, we build the `ext4` axiomatic semantics. As such, in §4, §5, and §6 we provide the axioms in the form of *Consistency* and *Persistency* predicates, that are used to filter out the behaviors unobservable on `ext4`.

3.5 Model checking

In the context of formal verification of concurrency, the crux for model checkers is the *optimal* exploration of all the executions permitted by a given model. An effective technique for verifying the consistency guarantees of concurrent programs is *Stateless Model Checking*

(SMC) [12, 13, 25] coupled with *Dynamic Partial Order Reduction* (DPOR) [1, 11, 18].

A key challenge of SMC is that a concurrent program may have a large number of executions to explore, typically exponential in the program size. To address this, existing literature includes several effective DPOR techniques that partition the executions into *consistency equivalence* classes (c-classes), aiming to explore exactly one execution per c-class. That is, all executions in a c-class have the same consistency guarantees, and thus it suffices to explore one execution from each.

The semantics we present was employed by the model checker PerSeVerE [28], which uses a similar technique. To facilitate effective model checking for persistency, PerSeVerE partitions executions into *persistency equivalence* classes. It allows the algorithm to combat the state space explosion arising from the filesystem semantics. This way, PerSeVerE *effectively* enumerates all possible consistent executions of a given program and all its possible post-crash persisted states, and checks whether the supplied assertions/invariants hold.

4 General persistency framework

In this section, we present the general persistency framework—the basis on which we further build the `ext4` semantics. First, we specify the language which semantics we formalize and introduce the notation. Then we provide the mathematical definition of the persistency framework. We build the model in an axiomatic manner, i.e., executions are modeled by graphs, where nodes represent events (e.g., reading or writing). Further, we provide the consistency predicate filtering out unobservable executions. Finally, we present the general notion of persistency formalizing which subsets of events can possibly be persisted upon a crash.

4.1 Notation and Programming language

Disk Domains

$\mathsf{Floc} \triangleq \mathsf{Inode} \times \mathsf{Offset}$	file locations
$nl \in \mathsf{Dnameloc} \triangleq \mathsf{String}$	file name locations on disk
$ds_f \in \mathsf{Dsizeloc}$	file size locations on disk†
$dl \in \mathsf{Dloc} \triangleq \mathsf{Floc} \uplus \mathsf{Dnameloc} \uplus \mathsf{Dsizeloc}$	disk locations

Memory Domains

$ml \in \mathsf{Mloc}$	memory locations
$d_f \in \mathsf{Fd}$	file descriptors where $f \in \mathsf{Inode}$
$ms_f \in \mathsf{Mloc}$	file size locations in memory†
$ol_{d_f} \in \mathsf{Mloc}$	file offset locations in memory‡

General Domains

$f \in \mathsf{Inode} \triangleq \mathbb{N}$	file inodes
$id \in \mathsf{Id} \triangleq \mathbb{N}$	operation ids
$t \in \mathsf{Tid} \triangleq \mathbb{N}$	thread ids
$o \in \mathsf{Offset} \triangleq \mathbb{N}$	offsets
$\mathsf{Loc} \triangleq \mathsf{Mloc} \cup \mathsf{Dloc}$	locations
$v \in \mathsf{Val}$	byte values
$lck \in \mathsf{Lck} \triangleq \mathsf{Inode} \cup \mathsf{Fd} \cup \{dir\}$	lockables

$\dagger$: defined for all $f \in \mathsf{Inode}$
$\ddagger$: defined for all $d_f \in \mathsf{Fd}$

$$\mathsf{Syscal} \ni \mathsf{c} ::= \mathsf{sync}\,(\,) \mid \mathsf{fsync}\,(d_f) \mid \cdots \qquad \text{(see Fig. 7 for a full list)}$$
$$\mathsf{Comm} \ni \mathsf{C} ::= \mathsf{e} \mid \mathsf{c} \mid \mathsf{C};\mathsf{C} \mid \mathbf{if}(\mathsf{e})\ \mathbf{then}\ \mathsf{C}_1\ \mathbf{else}\ \mathsf{C}_2 \mid \mathbf{while}(\mathsf{e})\ \mathsf{C} \qquad \mathsf{Exp} \ni \mathsf{e} ::= v \mid \cdots$$
$$\mathsf{Prog} \ni \mathsf{P} ::= \mathsf{Tid} \xrightarrow{\text{fin}} \mathsf{Comm}$$

Figure 6: Types and their metavariables (above); the programming language (below)

In operating system, files are represented by several structures in memory and on disk. Moreover, recall that under `ext4` the file metadata (e.g., size) is stored in a different location than its data (§3). We thus use the domains in Fig. 6 (above) to model the file representation. We build the syntax of the formal model from:

(i) A set of on-disk *file locations.* Each file location is a pair $dl{=}(f, o)$, where $f \in \mathsf{Inode}$ is the *file inode* and $o \in \mathsf{Offset}$ is the *offset*, denoting that dl contains the o^{th} byte of the f data.

(ii) A set of on-disk *file name locations* $\mathsf{Dnameloc} \triangleq \mathsf{String}$, where each $nl \in \mathsf{Dnameloc}$ records the inode associated with the file name nl.

(iii) A set of on-disk *file size locations* $\mathsf{Dsizeloc}$, and a mapping from inodes to their on-disk size locations; for brevity, we omit this mapping and write ds_f for the on-disk size location of f.

(iv) A set of *memory locations*, Mloc.

(v) A set of *file descriptors*, Fd, and a mapping from file descriptors to their inodes; we omit this mapping and simply write d_f to denote that the descriptor d is associated with inode f.

(vi) A mapping from inodes to their *in-memory size locations*; we omit this mapping and write ms_f for the in-memory size location of inode f.

(vii) A mapping from file descriptors to their (in-memory) *offset locations.* We write ol_{d_f} for the offset location of file descriptor d_f.

(viii) A set of lockable entities lck comprising Fd (corresponds to the locks in the Open File Table, see Fig. 2), Inode (corresponds to the locks in the in-memory Inode Table), and a separate lock dir used by the directory operations.

The language itself is modeled in a standard way: the program P consists of several threads: $\mathsf{P}{=}\mathsf{C}_1||\cdots||\mathsf{C}_n$, meaning $dom(\mathsf{P}){=}\{t_1, \dots, t_n\}$ and $\mathsf{P}(t_i){=}\mathsf{C}_i$. Each thread is a sequential composition of standard expression, conditionals, loops, and system calls (c). We assume a finite set Val of values, a finite set Tid of thread identifiers and any standard interpreted language for expressions, Exp, containing values. We use v as a metavariable for values, t for thread identifiers, and e for expressions.

For the purpose of generality, we do not define an explicit size for disk sectors and blocks, as these are determined by the underlying filesystem. However, as mentioned in §3, `ext4` provides certain guarantees on the same-sector/same-block accesses. To this end, we assume the existence of two *equivalence relations*, $\mathtt{ssec} \subseteq \mathsf{Dloc} \times \mathsf{Dloc}$ and $\mathtt{sblk} \subseteq \mathsf{Dloc} \times \mathsf{Dloc}$, relating the disk locations on the *same sector* and the *same block*, respectively.

4.2 Events

The semantics defines the correspondence between programs and executions. In the framework we present, each execution is a graph which set of vertices is formed by *events* defined as follows.

Definition 4.1 (Events). An *event* is a tuple $\langle n, t, id, l\rangle$, where $n \in \mathbb{N}$, $t \in \mathsf{Tid}$, $id \in \mathsf{Id}$ and l is an event *label* with one of the following forms: (i) $\mathtt{MR}(ml)$ for a memory read from ml; (ii) $\mathtt{MW}(ml, v)$ for a memory write to ml with value v; (iii) $\mathtt{DR}(dl)$ for a disk read from dl; (iv) $\mathtt{DW}^m(dl, v)$ for a disk write to dl with value v and *mode* $m \in \{\mathtt{norm}, \mathtt{trunc}, \mathtt{zero}, \mathtt{rename}\}$; (v) $\mathtt{FS}(f)$ for a file sync on f; (vi) $\mathtt{S}$ for a sync; (vii) $\mathtt{Open}(d_f, f)$ for opening f and yielding d_f; (viii) $\mathtt{Close}(d_f, f)$ for closing d_f associated with f; (ix) $\mathtt{L}(lck)$ for locking lck; (x) $\mathtt{U}(lck)$ for releasing lck.

The modes are needed for the indication of the event origin (system call); e.g., rename for a disk write corresponding to rename. The set of *memory reads* is $\mathtt{MR} \triangleq \left\{\langle n, id, t, l\rangle \;\middle|\; l{=}\mathtt{MR}(.)\right\}$. The sets of *memory writes* (MW), *disk reads* (DR), *disk writes* (DW), *syncs* (S), *file syncs* (FS), *locks* (L) and *unlocks* (U), as well as the sets $\mathtt{DW}^{\mathtt{trunc}}$, $\mathtt{DW}^{\mathtt{rename}}$, $\mathtt{DW}^{\mathtt{zero}}$ and $\mathtt{DW}^{\mathtt{norm}}$ are defined analogously. The sets of *reads* and *writes* are defined as $\mathtt{R} \triangleq \mathtt{MR}{\cup}\mathtt{DR}$ and $\mathtt{W} \triangleq \mathtt{MW}{\cup}\mathtt{DW}$, respectively; the set of *durable events* is defined as $\mathtt{D} \triangleq \mathtt{DW}{\cup}\mathtt{S}{\cup}\mathtt{FS}$. The set of all events is Events.

Notation. Given an event $e{=}\langle n, t, id, l\rangle$, we write $\mathtt{eid}(e)$, $\mathtt{tid}(e)$, $\mathtt{id}(e)$ and $\mathtt{lab}(e)$ to project its components, respectively. Analogously, given a label l, we write $\mathtt{loc}(l)$, $\mathtt{val}(l)$ and $\mathtt{lck}(l)$, when applicable. For instance, when $l = \mathtt{MW}(ml, v)$ then $\mathtt{loc}(l){=}ml$ and $\mathtt{val}(l){=}v$. We lift $\mathtt{loc}(.)$, $\mathtt{val}(.)$ and $\mathtt{lck}(.)$ to events.

Given a relation r, we write $r^{?}$ and r^{+} for the reflexive and transitive closures of r, respectively. We write r^{-1} for the inverse of r, $r|_A$ for $r{\cap}(A{\times}A)$, $[A]$ for the identity relation $\{(a, a) \mid a{\in}A\}$, and $r_1; r_2$ for the composition of r_1 and r_2: $\{(a, b) \,|\, \exists c.\, (a, c){\in}r_1{\wedge}(c, b){\in}r_2\}$. When A is a set of events, $x \in \mathsf{Loc}$ and $X \subseteq \mathsf{Loc}$, we define $A_x \triangleq \{e{\in}A \mid \mathtt{loc}(e){=}x\}$ and $A_X \triangleq \bigcup_{x \in X} A_x$. We also define $r_x \triangleq r|_{\mathtt{Events}_x}$ and $r_X \triangleq r|_{\mathtt{Events}_X}$.

4.3 Executions

As mentioned, we represent program executions as graphs, which nodes are events, and the edges represent different relations between these events. In our framework, there is one basic relation rf 'reads-from' matching every reading event with the corresponding write which wrote the read value.

Definition 4.2 (Executions). An *execution* $G \in \mathsf{Exec}$ is a tuple of the form $\langle \mathtt{E}, \mathtt{rf}\rangle$, where:

- E is a set of events (Def. 4.1) comprising a set of *initialization events*, $\mathtt{I} \subseteq \mathtt{E}$, by a designated thread t_0 such that: (i) $\mathtt{I} \triangleq \left\{e \in \mathtt{E} \cap (\mathtt{W} \cup \mathtt{U}) \;\middle|\; \mathtt{tid}(e) = t_0\right\}$; (ii) for each $ml \in \mathsf{Mloc}$, the set I contains a single event $w \in \mathtt{MW}$ on ml with value 0, i.e., $\mathtt{loc}(w){=}ml$ and $\mathtt{val}(w){=}0$; (iii) for each $nl \in \mathsf{Dnameloc}$, the set I contains a single event $w \in \mathtt{DW}$ on nl with value $\perp$; and (iv) for each $lck \in \mathsf{Lck}$, the set I contains a single unlock event u on lck (i.e., $\mathtt{lab}(u){=}\mathtt{U}(lck)$).
- $\mathtt{rf} \subseteq (\mathtt{E} \times \mathtt{E}) \cap ((\mathtt{W} \times \mathtt{R}) \cup (\mathtt{U} \times \mathtt{L}))$ is the *reads-from* relation such that (i) for all $(w, r) \in \mathtt{rf} \cap (\mathtt{W} \times \mathtt{R})$: $\mathtt{loc}(w){=}\mathtt{loc}(r)$; (ii) for all $(u, l) \in \mathtt{rf} \cap (\mathtt{U} \times \mathtt{L})$: $\mathtt{lck}(u){=}\mathtt{lck}(l)$; and (iii) $\mathtt{rf}^{-1}$ is functional; (iv) rf is total on its range: for all $a \in \mathtt{E} \cap (\mathtt{R} \cup \mathtt{L})$, there exists a unique $b \in \mathtt{W} \cup \mathtt{U}$ such that $(b, a) \in \mathtt{rf}$; and (v) no two locks read from the same unlock: $\mathtt{rf}^{-1}; [\mathtt{U}]; \mathtt{rf} \subseteq [\mathtt{L}]$.

Apart from the basic relation rf, we define the *derived* relations as follows.

Derived Relations. Given an execution $G = \langle \mathtt{E}, \mathtt{rf} \rangle$, we define the *program-order* relation as:

$$\mathtt{po} \triangleq \mathtt{I} \times (\mathtt{E} \setminus \mathtt{I}) \ \cup \left\{ \begin{array}{l|l} \langle \langle n_1, t_1, id_1, l_1 \rangle, & \langle n_1, t_1, id_1, l_1 \rangle, \langle n_2, t_2, id_2, l_2 \rangle \in \mathtt{E} \setminus \mathtt{I} \\ \quad \langle n_2, t_2, id_2, l_2 \rangle \rangle & \wedge\ t_1 = t_2 \wedge n_1 < n_2 \end{array} \right\}$$

relating initialization events to all others, and the events of each thread in increasing order.

We also define the *same-operation* equivalence relation, $\mathtt{sid}$, on the events of the same operation: $\mathtt{sid} \triangleq \left\{ (a, b) \ \middle|\ \mathtt{id}(a) {=} \mathtt{id}(b) \right\}$. The *same-file*, $\mathtt{sf}$, equivalence relation is defined analogously, relating the events on the same file (inode).

Finally, we lift $\mathtt{ssec}$ to events by defining $(a, b) \in \mathtt{ssec}$ to hold iff $(\mathtt{loc}(a), \mathtt{loc}(b)) \in \mathtt{ssec}$, and define the *block-sequence* relation, $\mathtt{bseq}$, prescribing the order in which a block is written as:

$$\mathtt{bseq} \triangleq \left\{ (w_1, w_2) \ \middle|\ \mathtt{lab}(w_1) = \mathtt{DW}_{(f,o_1)} \wedge \mathtt{lab}(w_1) = \mathtt{DW}_{(f,o_2)} \wedge (o_1, o_2) \in \mathtt{sblk} \wedge o_1 < o_2 \right\}$$

Event generation. In order to keep the framework general, we do not specify the exact correspondence between system calls and executions. This is because this mapping is defined by the filesystem implementation. Thus, we leave this degree of freedom as a *parameter* of the general framework, which will be instantiated for `ext4` in §5.

Parameter 1 (Event generation algorithm). We assume that there is a filesystem-specific algorithm taking a system call as an input and generating a sequence of events as the output.

4.4 Consistency

The filesystem semantics comprises two parts: consistency and persistency. Formalizing persistency also requires consideration of consistency semantics (a.k.a. *memory model*), because persistent effects are largely defined by consistency. For example, whether the file was persisted or not may depend on the fact whether the 'flushing' instruction fsync had been executed.

Underlying memory model. In order to keep our formalism general, we do not define a specific memory model and its associated set of consistent executions. Rather, we *parameterize* the framework in the choice of the underlying memory model. We thus assume a *consistency predicate*, $\mathsf{cons}_{\mathcal{M}}(.)$, which determines whether an execution is $\mathcal{M}$-consistent (i.e., consistent under the $\mathcal{M}$ memory model). This way, our general framework can be *instantiated* for a desired memory model $\mathcal{M}$ by supplying it with the $\mathsf{cons}_{\mathcal{M}}(.)$ predicate.

$\mathcal{M}$-consistency is typically constrained via a *synchronizes-with* relation, $\mathtt{sw}$, prescribing the order induced by synchronization mechanisms, e.g., locks. As we do not restrict our framework to an explicit memory model, we accordingly keep its associated $\mathtt{sw}$ relation as a parameter, with the proviso that $\mathtt{sw}$ includes the $\mathtt{rf}$ edges on disk locations as well as the synchronization induced by lock acquisition. As is standard, we define the *happens-before* relation, $\mathtt{hb}$, as the transitive closure of $\mathtt{po}$ and $\mathtt{sw}$. Intuitively, $\mathtt{hb}$ denotes c-ordering: the order events become visible to other threads.

Parameter 2 ($\mathcal{M}$-consistency). Given an execution $\langle \mathtt{E}, \mathtt{rf} \rangle$, assume a *synchronizes-with* relation, $\mathtt{sw} \subseteq \mathtt{E} \times \mathtt{E}$, denoting a strict partial order such that $[\mathtt{DW}]; \mathtt{rf}; [\mathtt{DR}] \subseteq \mathtt{sw}$ and $[\mathtt{U}]; \mathtt{rf}; [\mathtt{L}] \subseteq \mathtt{sw}$. Assume a consistency predicate, $\mathsf{cons}_{\mathcal{M}}(.) : \mathsf{Exec} \rightarrow \{\mathsf{true}, \mathsf{false}\}$, such that for all executions G, if $\mathsf{cons}_{\mathcal{M}}(G)$ holds, then (i) $\mathtt{hb} \triangleq (\mathtt{po} \cup \mathtt{sw})^+$ is irreflexive; and (ii) $(\mathtt{po} \cup \mathtt{rf})^+$ is irreflexive.

The first requirement ensures that $\mathtt{hb}$ is a strict partial order; the second requirement precludes 'out-of-thin-air' behaviors [20], and is required by GenMC [18], the DPOR framework over which the PerSeVerE model checker was built (see §7).

Persistency order. The last filesystem-specific parameter is the order in which the effects of the durable events persist on the disk. For instance, two sequential writes, depending on the implementation, may or may not be p-ordered. In the latter case, it is possible that after a crash only the second write will be visible.

In our framework, the persistency order manifests in the 'persists-before' relation ($\mathtt{pb}$). We also assume that the events which effects persist *simultaneously* are grouped by the equivalence relation $\mathtt{atom}$.

Parameter 3 (Persistency order). We assume that for any execution there are $\mathtt{pb}$ and $\mathtt{atom}$ relations on its events. We leave the exact way $\mathtt{pb}$ and $\mathtt{atom}$ are constructed as a parameter of the framework.

As will be covered in §5, $\mathtt{pb}$ directly influences the persistency semantics. However, it also defines consistency. Specifically, we require that the executions with $\mathtt{pb}$ cycles be ruled out. Given a memory model $\mathcal{M}$, an execution is consistent iff it is $\mathcal{M}$-consistent (Parameter 2) and its *persists-before* relation is a strict partial order.

Definition 4.3 (Consistency). An execution G is *consistent* according to a memory model $\mathcal{M}$ iff $\mathsf{cons}_{\mathcal{M}}(G)$ holds and $G.\mathtt{pb}$ is irreflexive.

4.5 Persistency: snapshot and frontier

To answer the persistency question "Which values are observable upon a crash?", we define the notion of snapshot and frontier. Informally, a snapshot is a 'prefix' of an execution, comprising the events whose effects have reached the disk before the crash. One execution can have multiple snapshots, as it might crash at different points.

More concretely, a snapshot should (i) contain the initial durable events, as they persist even before the execution starts, (ii) be downward-closed w.r.t. $\mathtt{pb}$ and $\mathtt{atom}$, as so dictates the semantics of $\mathtt{pb}$—the order, in which events reach the disk, and $\mathtt{atom}$—events reaching the disk at the same time.

Definition 4.4 (P-snapshot). A set P is a *p-snapshot* of an execution G iff (i) $(\mathtt{I} \cap \mathtt{D}) \subseteq P \subseteq \mathtt{D}$; (ii) $\mathit{dom}(\mathtt{pb}; [P]) \subseteq P$; and (iii) $\mathit{dom}(\mathtt{atom}; [P]) \subseteq P$.

Although the effects of the snapshot events have reached the persistent storage, not all of them are visible upon a crash. Particularly, if a snapshot contains multiple writes to the same location, only the last of them is observable.

The notion of frontier reflects this intuition. Given a snapshot P of execution G, we define the *frontier* of P, written front_P, to contain *exactly one* write for each disk location dl. Specifically, the one which is $\mathtt{pb}$-latest among the snapshot writes to dl.

Definition 4.5 (P-frontier). Given a p-snapshot P of an execution G, the *frontier* of P is $\mathsf{front}_P \triangleq N \cup S \cup F$ where:

$$N \triangleq \left\{ \max(G.\mathtt{pb}_{\mathit{dl}}|_P) \;\middle|\; \mathit{dl} \in \mathsf{Dnameloc} \right\} \qquad S \triangleq \left\{ \max(G.\mathtt{pb}_{\mathit{ds}_f}|_P) \;\middle|\; \exists w \in N.\ \mathtt{lab}(w) = \mathtt{DW}(\mathit{nl}_f, f) \right\}$$

$$F \triangleq \left\{ \max(G.\mathtt{pb}_{(f,o)}|_P) \;\middle|\; \exists w_1 \in N, w_2 \in S.\ \mathtt{lab}(w_1) = \mathtt{DW}(\mathit{nl}_f, f) \land \exists s.\ \mathtt{lab}(w_2) = \mathtt{DW}(\mathit{ds}_f, s) \land o < s \right\}$$

Note that the frontier consists of three parts: N for name locations, S for file size locations, and F for data locations. The additional conditions in the definition of S and F are to ensure the visibility of the writes in the frontier. Particularly, the data writes has to be to the preallocated region, and thus, the corresponding size update should also be in the frontier.

5 Ext4 formal semantics

In this section, we specialize the framework constructed in §4 for the `ext4` filesystem, whereby defining its semantics. First, using litmus tests, we determine the persistency and consistency guarantees of the `ext4` filesystem. Then we implement these guarantees in the model: first, we specify the correspondence between system calls and events; second, we define the key part of the semantics—the persist-before relation.

5.1 Litmus tests and guarantees

Next, we discuss persistency and consistency features of the `ext4` filesystem via providing an exhaustive set of litmus tests and explaining the `ext4` outcomes. For consistency, it is enough to examine how the filesystem resolves *data race* when two operations, one of which is a write, perform concurrently. The persistency semantics is sensible only for durable events, modifying the disk state.

5.1.1 Consistency tests

Parallel Writes. Linux provides strong c-atomicity guarantees for writes. Specifically, writing to a file involves acquiring the file's *inode lock* (see Fig. 2), ensuring that all writes to the same file are c-ordered with respect to one another. As each file is associated with a unique inode lock, mutual exclusion is guaranteed regardless of the file descriptor used to carry out the write. Moreover, a call to write (as opposed to pwrite) also acquires the *offset lock* in the file description, making the combination of the offset adjustment and the data write one big c-atomic step. For example, the pread operation in the program below reads either "`bar`" or "`qux`", since both pwrite operations acquire the inode lock:

$$\mathsf{pwrite}\,(d_f, \text{“}\texttt{bar}\text{”}, 0); \;\Big\|\; \mathsf{pwrite}\,(d_f, \text{“}\texttt{qux}\text{”}, 0);$$
$$r = \mathsf{pread}\,(d_f, 3, 0);$$

Reads vs. Overwrites. The c-atomicity guarantees of reads are more subtle: while read calls acquire the offset lock in the file description, pread calls acquire no locks. Therefore, a read call is c-ordered with respect to concurrent read/write calls on the *same file description* as they compete to acquire the offset lock. By contrast, concurrent read calls on different file descriptions and pread calls do not acquire a common lock and thus offer only *byte-level c-atomicity.* As such, if the file of d_f initially contains "`foo`", the pread call below may read "`foo`", "`bar`", "`far`", "`fao`", and so forth.

$$\mathsf{pwrite}\,(d_f, \text{“}\texttt{bar}\text{”}, 0); \;\Big\|\; r = \mathsf{pread}\,(d_f, 3, 0);$$

Reads vs. Appends. When racing with a write *appending* to a file, reads have stronger c-atomicity guarantees: reads consult the file size, which is modified by appends, resulting in stronger synchronization. In general, appends increase the file size not at once but incrementally: they write cache pages one at a time, increasing the file size after each page. Reads first read the file and then the data, and may thus observe the incremental size increases, at the granularity of the page size.

For example, assuming that each page is 3 bytes and that "`foo.txt`" containing "`foo`" is opened with `O_APPEND`, the pread in the program below can read "`foo`", "`foobar`" or "`foobarqux`".

$$\mathsf{write}\,(d_f, \text{“}\texttt{barqux}\text{”}); \;\Big\|\; r = \mathsf{pread}\,(d_f, 42, 0);$$

Note that when a read requests more data than available in the file, it reads as much data as it can.

This behavior is also observable for lseek and appends: if one thread appends multiple pages to a file and another concurrently seeks to `EOF`, lseek may set the offset to an intermediate file size.

Directory operations. Directory operations provide strong consistency guarantees: they are c-atomic against operations that manipulate the same inode (as they acquire the inode lock), as well as against other directory operations on the same directory (as they acquire the directory's inode lock). This gives strong consistency guarantees to file creation, linking, unlinking, etc.

One interesting exception is the rename (nl_{old}, nl_{new}) call when a file with name nl_{new} already exists. Recall that rename is analogous to link followed by unlink. While the link part is c-atomic in that nl_{new} always points to one of the two inodes, rename as a whole is not c-atomic because there is a window in which both nl_{old} and nl_{new} refer to the same inode [21].

Nevertheless, rename provides a mechanism for ensuring update c-atomicity as shown below:

```
d_b = creat ("foo.tmp");                      ||
write (d_b, "bar"); close (d_b);              ||  d_f = open ("foo.txt", O_RDONLY);
rename ("foo.tmp", "foo.txt");                ||  r = read (d_f, 3);    // reads "foo" or "bar"
```

As before, suppose "`foo.txt`" initially contains "`foo`". Regardless of whether open sees the new or the old version of "`foo.txt`", it can seamlessly read the data in the next step. Even if rename happens in between the open and read calls, the right thread still reads "`foo`" as the file description of d_f still points to the same inode even after rename. This inode, albeit no longer accessible via the "`foo.txt`" name after the rename, will not be deleted until all references to it are deleted.

5.1.2 Persistency tests

Overwrites. `ext4` provides very weak persistency guarantees for overwrites. First, it does not guarantee p-atomicity beyond what is provided by the underlying storage, which is typically sector-level p-atomicity for hard drives, but may only be byte-level p-atomicity for other persistent storage media (e.g., non-volatile memory). For a filesystem to guarantee p-atomicity at the block level, it must use techniques such as full *data journaling* or copy-on-write [6], which are not employed by `ext4` by default.

Second, even the order in which overwrites to different sectors are persisted is loosely constrained: writes to sectors within a block persist in order, whereas writes to different blocks may persist in an arbitrary order. That is, although a write issues I/O requests for writing blocks in order, these writes may be freely reordered both by the block I/O layer of the kernel and the disk itself (e.g., to minimize rotation), and by default `ext4` does not prevent this reordering.

For example, consider the following overwrite on d_f associated with file "`foo.txt`" with initial contents "`foo`":

$$\mathsf{pwrite}\,(d_f, \text{“}\texttt{bar}\text{”}, 0); \qquad (\textsc{ow-na})$$

Let us assume that the sector size is one byte and each block comprises 3 sectors. If a crash occurs during OW-NA, then only a *prefix* of "`bar`" may persist to disk before the crash, guaranteeing *sector-level* p-atomicity. That is, in the post-crash disk state "`foo.txt`" may contain "`foo`", "`boo`", "`bao`" or "`bar`", but not "`fao`" or "`far`". This is because blocks are composed of contiguous sectors, and sectors within a block are written in a linear order. If, however, each block contains only one sector and a crash occurs, then outcomes such as "`fao`" and "`far`" are also possible.

Recall from §4.3, in our formal model, we keep the sector and block sizes as parameters; we treat sector writes as p-atomic and assume that sectors constituting a block are written in a linear order.

Appends. `ext4` offers stronger persistency guarantees for appends: it guarantees *block-level* p-atomicity of append *prefixes* and that appends on the same file are p-ordered, as described below.

The block-level p-atomicity guarantees of appends are best seen with an example. Suppose that a file occupies n blocks on disk and a crash occurs while appending k more blocks to it. In the post-crash disk state the file may then contain $n + i$ blocks, where $0 \leq i \leq k$; i.e., a (potentially full) prefix of the appended blocks may persist to disk before the crash. For instance, had we opened "`foo.txt`" in OW-NA with the `O_APPEND` flag, assuming that the block size is `3B`, in the post-crash disk state "`foo.txt`" would contain either "`foo`" or "`foobar`"–recall from § 3.1.1 that opening "`foo.txt`" with `O_APPEND` ignores the absolute offset of pwrite and simply appends to it.

At first glance, this may seem incompatible with the weak p-ordering guarantees of overwrites. If block writes are not p-ordered, how does `ext4` ensure that the size update persists after the appended data? This is enabled by the *journal*: when `ext4` journals the metadata updates of an append (e.g., its size update), it *binds* the commit of the transaction containing the metadata, with the persist of the associated data. That is, the size update transaction commits only once the appended data persists. As such, if a crash occurs before the transaction commits, the file size on disk will not have been updated, and thus the persisted append data (if any) will not be accessible.

5.2 Mapping system calls to events

Here, we present the mapping from the system calls to the sequences of events, whereby specializing Parameter 1 of the general framework. For every system call, we provide an eponymous pseudocode function generating the corresponding sequence of events. In pseudocode, we write $\rightsquigarrow l$ to denote generating an event $e{=}\langle n, t, id, l\rangle$ with label l.

In our algorithmic description, we use several helper functions (e.g., `freshFD`) that do not generate any events; as such, we omit their algorithmic description and describe their behavior intuitively, as necessary. Moreover, we assume that the file descriptors supplied as arguments are valid and forgo their validity checks.

```
procedure BufferRead($f$, $buf$, $count$, $o$)
  $\rightsquigarrow \texttt{MR}(ms_f)$ in $size$
  $m \leftarrow \mathsf{min}(count, size - o)$
  for $i = 0$ to $m - 1$ do
    $\rightsquigarrow \texttt{DR}((f, o + i))$ in $buf[i]$

procedure BufferWrite($f$, $buf$, $count$, $o$)
  $\rightsquigarrow \texttt{L}(f)$
  $\rightsquigarrow \texttt{MR}(ms_f)$ in $size$
  if $\texttt{isPreallocBlock}(o, count, size)$ then
    $end \leftarrow \mathsf{min}(count + o, \texttt{getLastBlockEnd}(f))$
    for $i = size$ to $end - 1$ do $\rightsquigarrow \texttt{DW}^{\texttt{zero}}((f, i), 0)$
    $\rightsquigarrow \texttt{DW}^{\texttt{zero}}(ds_f, end)$
    $size \leftarrow end$
  if $o > size$ then
    for $i = size$ to $o - 1$ do $\rightsquigarrow \texttt{DW}^{\texttt{norm}}((f, i), 0)$
  for $i = 0$ to $count - 1$ do
    $\rightsquigarrow \texttt{DW}^{\texttt{norm}}((f, o + i), buf[i])$
    if $(\texttt{isFstB}(f\ o{+}i{+}1) \lor i{=}count{-}1)) \land o{+}i > size$ then
      $\rightsquigarrow \texttt{DW}^{\texttt{norm}}(ds_f, o + i)\,;\texttt{MW}(ms_f, o + i)$
  $\rightsquigarrow \texttt{U}(f)$

procedure pread($d_f$, $buf$, $count$, $o$)
  BufferRead($f$, $buf$, $count$, $o$)

procedure read($d_f$, $buf$, $count$)
  $\rightsquigarrow \texttt{L}(d_f)$
  $\rightsquigarrow \texttt{MR}(ol_{d_f})$ in $o$
  BufferRead($f$, $buf$, $count$, $o$)
  $\rightsquigarrow \texttt{MW}(ol_{d_f}, o + count)$
  $\rightsquigarrow \texttt{U}(d_f)$

procedure pwrite($d_f$, $buf$, $count$, $o$)
  BufferWrite($f$, $buf$, $count$, $o$)

procedure write($d_f$, $buf$, $count$)
  $\rightsquigarrow \texttt{L}(d_f)$
  $\rightsquigarrow \texttt{MR}(ol_{d_f})$ in $o$
  BufferWrite($f$, $buf$, $count$, $o$)
  $\rightsquigarrow \texttt{MW}(ol_{d_f}, o + count)$
  $\rightsquigarrow \texttt{U}(d_f)$
```

Figure 7: Algorithmic description of mapping system calls to events where $\rightsquigarrow l$ generates an event with label l

```
1: procedure LOOKUP($nl, flags$)
2:   if O_CREAT $\notin flags$ then
3:     $\rightsquigarrow$ DR($nl$) in $f$
4:   else
5:     $\rightsquigarrow$ L($dir$)
6:     $\rightsquigarrow$ DR($nl$) in $f$
7:     if $f = \bot$ then
8:       $f \leftarrow$ freshINode()
9:       $flags \leftarrow flags \setminus$ O_TRUNC
10:      $\rightsquigarrow$ DW$^{\texttt{norm}}$($ds_f, 0$) ; MW($ms_f, 0$)
11:      $\rightsquigarrow$ DW$^{\texttt{norm}}$($nl, f$)
12:    $\rightsquigarrow$ U($dir$)
13:  return ($f, flags$)
```

```
1: procedure open($nl, flags$)
2:   ($f, flags$) $\leftarrow$ LOOKUP($nl, flags$)
3:   if $f = \bot$ then return
4:   $d_f \leftarrow$ freshFD($f$)
5:   $\rightsquigarrow$ MW($ol_{d_f}, 0$) ; Open($d_f, f$)
6:   if O_TRUNC $\in flags$ then
7:     $\rightsquigarrow$ L($f$) ; DW$^{\texttt{trunc}}$($ds_f, 0$)
8:     $\rightsquigarrow$ MW($ms_f, 0$) ; U($f$)
```

```
1: procedure close($d_f$) $\rightsquigarrow$ Close($d_f$)
```

```
1: procedure lseek($d_f, o$)
2:   $\rightsquigarrow$ L($d_f$)
3:   $\rightsquigarrow$ MR($ol_{d_f}$) in $o'$
4:   $\rightsquigarrow$ MW($ol_{d_f}, o$)
5:   $\rightsquigarrow$ U($d_f$)
```

```
1: procedure link($nl_{old}, nl_{new}$)
2:   $\rightsquigarrow$ L($dir$)
3:   $\rightsquigarrow$ DR($nl_{new}$) in $f'$  assert($f' = \bot$)
4:   $\rightsquigarrow$ DR($nl_{old}$) in $f$
5:   $\rightsquigarrow$ DW$^{\texttt{norm}}$($nl_{new}, f$)
6:   $\rightsquigarrow$ U($dir$)
```

```
1: procedure unlink($nl$)
2:   $\rightsquigarrow$ L($dir$) ; DW$^{\texttt{norm}}$($nl, \bot$) ; U($dir$)
```

```
1: procedure rename($nl_{old}, nl_{new}$)
2:   $\rightsquigarrow$ L($dir$)
3:   $\rightsquigarrow$ DR($nl_{old}$) in $f$
4:   $\rightsquigarrow$ DW$^{\texttt{rename}}$($nl_{new}, f$) ; DW$^{\texttt{rename}}$($nl_{old}, \bot$)
5:   $\rightsquigarrow$ U($dir$)
```

```
1: procedure sync () $\rightsquigarrow$ S
```

```
1: procedure fsync($d_f$) $\rightsquigarrow$ FS($f$)
```

Figure 7 (Cont.): Algorithmic description of mapping system calls to events where $\rightsquigarrow l$ generates an event with label l

Open and Close. The open procedure consists of three parts: First, it generates the events relating to the file lookup in the directory (Line 2). Second, it generates the open event itself (Lines 3–5). And finally, the truncation sub-procedure is performed (Lines 6–8).

If the file creation is not needed (`O_CREAT` is not specified), The LOOKUP($nl, flags$) routine, generates a single read event from the associated file name location nl (Line 3) and returns f and the unchanged flags (Line 13). If the creation is required, and the file does not exist, then LOOKUP creates a fresh inode f (via `freshINode`), removes `O_TRUNC` from the flags as the file is just created, initializes the f size both in memory and on disk, associates nl with f and returns f and the updated flags. Second, If f (returned by lookup) is valid, the open part creates a fresh file descriptor d_f (via `freshFD`), initializes its offset and opens f with d_f.

Sync, FSync and LSeek. A sync (resp. fsync (d_f)) call simply corresponds to a single event with label `S` (resp. `FS`(d_f)).

As for the lseek system call, recall that lseek (d_f, o) updates the offset of the file descriptor

d_f to o. Moreover, as with all system calls modifying the file descriptor offset, it does so by first acquiring its lock. As such, the associated algorithm generates a sequence of events to lock d_f, read the current offset o' of d_f, update it to o, and finally release the lock on d_f.

PRead and Read. A call to pread is handled by the BUFFERREAD routine which reads the in-memory file size (Line 2) and generates a sequence of DR events for reading m bytes of file data at offset o, one byte at a time (Line 5), where m is the minimum of count and $\mathit{size} - o$. Recall that the key difference between read and pread is that read updates the descriptor offset. As such, read generates analogous events to those of pread, and further include events for updating the descriptor offset (Lines 3 and 5) and thus acquiring/releasing its lock (Lines 2 and 6).

PWrite and Write. A call to pwrite (i) acquires the inode lock, (ii) carries out the file data write one byte at a time, (iii) increases the file size if necessary (in the case of appends) and (iv) releases the inode lock.

Analogously, the BUFFERWRITE of pwrite generates the lock and unlock events in steps (1) and (4) on Lines 2 and 15, respectively, the events of step (2) on Lines 11–12, and those of (3) on Lines 13–14. Note that the size is updated both in memory and on disk (Line 14) after writing a full block (or the last sub-block of the write), provided that the file size changes (increases).

A pwrite must additionally account for cases where a file with preallocated blocks is appended, or the offset supplied is greater than the file size. The former case (see delayed allocation in § 3.3.2) is handled on Lines 4–6, where the last file block is zeroed if it is preallocated (determined via `isPreallocBlock`)—we assume that a file has at most one preallocated block at the end that is not filled. The on-disk (but not in-memory) file size is then updated accordingly (Line 7); this ensures that the zeroed-out block is observable after recovery from a crash, but not by reads during the execution. The latter case is handled on Lines 9–10, where the bytes between size and o are zeroed.

Lastly, the events generated by write are analogous to those of pwrite and additionally generate the events for updating the file descriptor offset as in read.

Link, Unlink and Rename. The link call acquires the lock on the (only) directory dir (Line 2) and inspects the new name $\mathit{nl}_{\mathit{new}}$ specified (Line 3), ensuring that it is not already taken, i.e., holds $\perp$ (Line 3). It then determines the inode f associated with the old name (Line 4), updates the new name location to point to f (Line 5) and finally releases the directory lock (Line 6). The unlink (nl) call simply unlinks the name location nl from its associated inode by updating it to the designated $\perp$ value. Finally, the events generated by rename are those of link and unlink combined, except that unlike link, rename does not inspect the new name to check that it is available.

Observe that all *disk writes* (in $\mathtt{DW}$) generated by the Fig. 7 algorithms are issued while holding a lock. As such, thanks to lock-induced synchronization (Parameter 2), all writes on the same disk location are related by $\mathtt{hb}$; i.e., $\mathtt{whb}_{dl}$ is total for all $dl \in \mathsf{Dloc}$, where $\mathtt{whb} \triangleq [\mathtt{DW}]; \mathtt{hb}; [\mathtt{DW}]$.

5.3 The persists-before relation

Next, we instantiate Parameter 3 of the framework. To this end, we define the *persists-before* relation $\mathtt{pb} \subseteq \mathtt{D} \times \mathtt{D}$ for the $\mathtt{ext4}$ filesystem, as the least transitive relation such that:

$$(\mathtt{I} \cap \mathtt{D}) \times (\mathtt{D} \setminus \mathtt{I}) \subseteq \mathtt{pb} \qquad \text{(PB-INIT)}$$

$$[\mathtt{DW}]; (\mathtt{hb} \cap \mathtt{ssec}); [\mathtt{DW}] \subseteq \mathtt{pb} \qquad \text{(PB-SECTOR)}$$

$$[\mathtt{DW}]; (\mathtt{hb} \cap \mathtt{bseq}); [\mathtt{DW}] \subseteq \mathtt{pb} \qquad \text{(PB-BLOCK)}$$

$$[\mathtt{DW}_{\mathsf{Floc}}]; (\mathtt{hb} \cap \mathtt{sf}); [\mathtt{DW}_{\mathsf{Dsizeloc}}] \subseteq \mathtt{pb} \qquad \text{(PB-META)}$$

$$[\mathtt{S} \cup \mathtt{FS}]; \mathtt{hb}; [\mathtt{D}] \;\cup\; [\mathtt{D}]; \mathtt{hb}; [\mathtt{S}] \;\cup\; [\mathtt{DW}]; (\mathtt{hb} \cap \mathtt{sf}); [\mathtt{FS}] \subseteq \mathtt{pb} \qquad \text{(PB-SYNC)}$$

$$[\mathtt{DW}_{\mathsf{Dnameloc}} \cup \mathtt{DW}^{\mathtt{trunc}}]; \mathtt{hb}; [\mathtt{D} \setminus \mathtt{DW}_{\mathsf{Floc}}] \subseteq \mathtt{pb} \qquad \text{(PB-DIROPS)}$$

$$(\mathtt{atom}; \mathtt{pb}) \cup (\mathtt{pb}; \mathtt{atom}) \subseteq \mathtt{pb} \qquad \text{(PB-ATOM)}$$

and $\mathtt{atom} \triangleq ([\mathtt{DW} \setminus \mathtt{DW}^{\mathtt{zero}}]; (\mathtt{ssec} \cap \mathtt{sid}); [\mathtt{DW} \setminus \mathtt{DW}^{\mathtt{zero}}]) \cup ([\mathtt{DW}^{\mathtt{rename}}]; \mathtt{sid}; [\mathtt{DW}^{\mathtt{rename}}])$.

Recall that *'persists before' relation* ($\mathtt{pb}$) denotes a $\mathtt{ext4}$-specific partial order in which disk operations are persisted.

The PB-INIT axiom says that the initialization writes are p-ordered before all other durable events.

The PB-SECTOR axiom captures the disk: same-sector writes persist atomically and are never reordered.

The PB-BLOCK axiom models the assumption that sectors within a block are persisted in sequence. As a result, c-ordered same-block writes are also p-ordered, as long as their offsets match the order in which the block is written.

The PB-META axiom ensures that file data updates are p-ordered before their subsequent size updates, as required by the `data=ordered` journaling mode. Intuitively, this relates the event(s) on Line 6 of BufferWrite to that on Line 7, and those on Line 12 to that on Line 14.

The PB-SYNC axiom describes the p-ordering of sync/fsync: (i) all events that are $\mathtt{hb}$-after a sync/fsync are p-ordered after it; (ii) all events that are $\mathtt{hb}$-before a sync are p-ordered before it; and (iii) all disk writes to a file that are $\mathtt{hb}$-before an fsync on the same file are p-ordered before it.

The PB-DIROPS captures the fact that directory operations and file size updates due to truncation are p-ordered before all subsequent operations except overwrites.

Lastly, PB-ATOM ensures that renames and same-sector writes are p-atomic by requiring that `pb` be closed under composition with `atom`. The *atomic* relation `atom` relates the $\mathtt{DW}^{\mathtt{rename}}$ events of the same (rename) operation, as well as the same-sector $\mathtt{DW} \setminus \mathtt{DW}^{\mathtt{zero}}$ events of the same (write/pwrite) operation. Note that we exclude $\mathtt{DW}^{\mathtt{zero}}$ to model the (p-atomicity-violating) anomaly of delayed allocation, thus allowing zero writes to persist without the corresponding data writes.

6 Adaptation for effective formal verification

Although previously in §4 we presented an exhaustive and intuitive notion of persistency as a frontier—a set of writes that are observable upon recovery, it does not provide an *efficient* way of checking persistency. All possible frontiers cannot be enumerated naively, as their number is exponential in the program size. In this section, we give an alternative, more viable definition of persistency. Specifically, we present an *instrumented* execution—the one with a specific recovery routine. Further, we define persistency in an axiomatic manner, and in § 6.2 we prove that for `ext4`, this definition corresponds exactly to the old notion of persistency.

6.1 Instrumented execution

Recovery observer. The recovery observer is a procedure representing the code that is called after a crash. We assume that no more crashes can occur during its execution. Thus for modeling the recovery observer it is sufficient to determine the observable contents of the filesystem, in other words, to answer the question *"Which values can be read from disk?"*. Moreover, the recovery observer is executed in a single-threaded environment, therefore, no consistency effects are present.

We thus represent the recovery observer as a sequence of pread system calls preceded by open calls. We assume that in the recovery mode open is called *without* `O_TRUNC` and `O_CREAT`.

Algorithm 1 Mapping open calls in the recovery observer to events

```
procedure open($nl, flags$) assert($\texttt{O\_TRUNC} \notin flags \wedge \texttt{O\_TRUNC} \notin flags$)
    $\rightsquigarrow$ DR($nl$)
```

The algorithm generating events for the pread system calls differs from that of non-recover calls in Fig. 7. Specifically, as all in-memory data disappear after a crash, the recovery pread calls read the metadata from the disk rather than the memory (Algorithm 2, Line 2). Note that both open and pread produce only DR events.

Algorithm 2 Mapping pread calls in the recovery observer to events

```
procedure pread($d_f$, $buf$, $count$, $o$)
    $\rightsquigarrow$ DR($ds_f$) in $size$
    $count \leftarrow \mathsf{min}(count, size - o)$
    for $i = 0 \ldots count - 1$ do
        $\rightsquigarrow$ DR(($f$, $o + i$)) in $buf[i]$
```

Instrumented execution. As the recovery observer differs from the regular threads, *executions* produced by programs with the recovery observer are also different. Specifically, we *instrument* regular executions with a distinguished set of events $\mathtt{REC}$, corresponding to the recovery routine.

Definition 6.1 (Instrumented execution). An *instrumented execution* is a tuple $\langle \mathtt{E}, \mathtt{rf} \rangle$ such that:

- $\langle \mathtt{E}, \mathtt{rf} \rangle$ is an execution (Def. 4.2); and
- the event set is partitioned, $\mathtt{E} = \mathtt{NREC} \uplus \mathtt{REC}$, into *non-recovery events*, $\mathtt{NREC}$, and *recovery events*, $\mathtt{REC}$, comprising disk reads by a designated thread t_r: $\mathtt{REC} \triangleq \left\{ e \in \mathtt{E} \cap \mathtt{DR} \;\middle|\; \mathtt{tid}(e) = t_\mathsf{r} \right\}$.

Well-formed executions. The executions we consider are not arbitrary ones but generated from programs. Hence, they satisfy a number of properties automatically. Proposition 6.2 incorporates those of them that are used in our proofs.

Proposition 6.2 (Well-formed instrumented execution). All instrumented executions G generated by PERSEVERE are *well-formed* in that they satisfy the following properties:

- $\forall (f, o) \in \mathsf{Floc}, r \in \mathtt{DR}_{(f,o)} \cap \mathtt{REC}.\ \exists r'.\ \mathtt{lab}(r') = \mathtt{DR}(ds_f, s) \wedge o < s \wedge (r', r) \in \mathtt{po}$

  (READ-SIZE-FIRST)
- $\forall (f, o) \in \mathsf{Floc}, r \in \mathtt{DR}_{(f,o)} \cup \mathtt{DR}_{ds_f}.\ \exists r'.\ \mathtt{lab}(r') = \mathtt{DR}(nl_f, f) \wedge (r', r) \in \mathtt{hb}$

  (READ-NAME-FIRST)

The property READ-SIZE-FIRST guarantees that reading the data from a file in the recovery observer is preceded by reading its size. Moreover, the size read (s) justifies the offset (o) read from: $o < s$ The READ-NAME-FIRST guarantees that reading a file size or data is preceded by reading its name. All `ext4` executions satisfy these properties: each recovery read is generated by Algorithm 2, which reads the file size (Line 2) `po`-before reading any data. Similarly, each algorithm generating file read events, checks for the file descriptor validity first, it thus does not generate a write to a file that has not been opened before.

Persistency. Having introduced instrumented execution, we now are ready to adjust the notion of persistency. With recovery observer, one does not have to enumerate *all* crash scenarios (objectified in snapshots and frontiers) to define persistency, but only consistent with the observer. Specifically, it is required that the values read by the observer be persisted on disk. To put it more formally, **there should exist a snapshot, which frontier contains all the write events read by the recovery procedure**.

It turns out, the latter requirement can be expressed *efficiently* in an axiomatic fashion. In persistent executions, we forbid paths of a specific form (REC).

Definition 6.3 (Persistency). An instrumented execution G is *persistent* iff:

- $\langle \mathtt{NREC}, \mathtt{rf}|_{\mathtt{NREC}} \rangle$ is consistent according to Def. 4.3 (when $G = \langle \mathtt{NREC} \uplus \mathtt{REC}, \mathtt{rf} \rangle$); (CON) and

- $[\mathtt{REC}];\mathtt{rb};\mathtt{atom}^?;\mathtt{pb}^?;\mathtt{rf};[\mathtt{REC}] = \emptyset$, where $\mathtt{rb} \triangleq \cup_{x\in\mathsf{Loc}}\mathtt{rf}_x^{-1};\mathtt{whb}_x$ (REC)

6.2 Equivalence theorem

The following theorem clarifies the equivalence between two definitions of persistency. It shows that consistent instrumented executions (Def. 6.3) are exactly those, whose recovery routine reads from a correct frontier of some execution. In other words, the data observable after restart is formed by some interrupted execution.

Theorem 6.4 (Equivalence). For all persistent instrumented executions G, there exists a p-snapshot P of G such that $dom(\mathtt{rf};[\mathtt{REC}]) \subseteq \mathsf{front}_P$. For all consistent executions $G{=}\langle \mathtt{E}, \mathtt{rf}\rangle$, p-snapshots P of G, relations $\mathtt{rf}'$ and recovery reads $\mathtt{REC} \subseteq \mathtt{DR}$ by t_r ($\forall r \in \mathtt{REC}.\ \mathtt{tid}(r){=}t_\mathsf{r}$), if $rng(\mathtt{rf}'){=}\mathtt{REC}$ and $dom(\mathtt{rf}') \subseteq \mathsf{front}_P$, then the instrumented execution $\langle \mathtt{E} \uplus \mathtt{REC}, \mathtt{rf} \uplus \mathtt{rf}'\rangle$ is consistent.

We present a proof of Theorem 6.4. The proof, as well as the theorem, consists of two parts: Recovery Soundness and Recovery Completeness.

Informally, Recovery Soundness guarantees that recovery events of any consistent instrumented execution (Def. 6.3) read from the frontier of a p-snapshot. Conversely, Recovery Completeness ensures that any p-snapshot-based characterization of the events observable on recovery can also be expressed as a consistent instrumented execution.

6.2.1 Recovery Soundness

Given a well-formed consistent instrumented executions G, we prove that there exists a p-snapshot such that $G.\mathtt{REC}$ events read from its frontier.

Let P be the union of the initial durable events and the $\mathtt{pb}$-prefix of the writes read from by the recovery observer:

$$P \triangleq \mathtt{I}{\cap}\mathtt{D} \cup dom(\mathtt{atom}^?;\mathtt{pb}^?;\mathtt{rf};[\mathtt{REC}])$$

Next, we prove that (i) P is a p-snapshot of G and (ii) $\mathtt{REC}$ events read from front_P.

Lemma 6.5 (Canonical snapshot). P defined as above is a *p-snapshot* of G according to Def. 4.4.

Proof. We show that properties (1)–(3) of Def. 4.4 are satisfied by P. Since P is a union of two sets, we split each proof into two parts: the first for $\mathtt{I}\cap\mathtt{D}$, and the second for $dom(\mathtt{atom}^?;\mathtt{pb}^?;\mathtt{rf};[\mathtt{REC}])$.

(i) $(\mathtt{I}\cap\ \mathtt{D}) \subseteq P \subseteq \mathtt{D}$

The first inclusion holds by construction.

For the second one, we are to show the inclusion of both parts of P:

(a) trivially, $\mathtt{I}{\cap}\mathtt{D} \subseteq \mathtt{D}$,

  (b) subsequently splitting the reflexive closure of $\mathtt{atom}$ and $\mathtt{pb}$ into two cases, $\mathit{dom}(\mathtt{atom}) \subseteq \mathtt{DW} \subseteq \mathtt{D}$, $\mathit{dom}(\mathtt{pb}) \subseteq \mathtt{D}$, and $\mathit{dom}(\mathtt{rf}; [\mathtt{REC}]) \subseteq \mathtt{DW} \subseteq \mathtt{D}$.

(ii) $\mathit{dom}(\mathtt{pb}; [P]) \subseteq P$

Again, we represent P on the left-hand side as a union and consider two cases:

  (a) $\mathit{dom}(\mathtt{pb}; [\mathtt{I}{\cap}\mathtt{D}]) \subseteq \mathtt{I}{\cap}\mathtt{D} \cup \mathit{dom}(\mathtt{pb} \cap ((\mathtt{E} \setminus \mathtt{I}) \times \mathtt{I})) = \mathtt{I}{\cap}\mathtt{D} \cup \mathit{dom}(\mathtt{pb} \cap \mathtt{pb}^{-1}) = \mathtt{I}{\cap}\mathtt{D} \subseteq P$,

  (b) $\mathit{dom}(\mathtt{pb}; [\mathit{dom}(\mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}])]) = \mathit{dom}(\mathtt{pb}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}]) \subseteq \mathit{dom}(\mathtt{pb}; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}]) = \mathit{dom}(\mathtt{pb}; \mathtt{rf}; [\mathtt{REC}]) \subseteq P$.

(iii) $\mathit{dom}(\mathtt{atom}; [P]) \subseteq P$ Analogously, we have two cases:

  (a) $\mathtt{atom}; [\mathtt{I}{\cap}\mathtt{D}] \subseteq [\mathtt{I}{\cap}\mathtt{D}]$,

  (b) $\mathit{dom}(\mathtt{atom}; [\mathit{dom}(\mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}])]) = \mathit{dom}(\mathtt{atom}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}]) \subseteq \mathit{dom}(\mathtt{atom}; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}]) \subseteq P$.

□

Let the sets N, S and F of the frontier front_P be as defined in Def. 4.4.

There are three types of disk writes read by the recovery observer: name writes, size writes, and data writes, i.e., $\mathit{dom}(\mathtt{rf}; [\mathtt{REC}]) \subseteq \bigcup \mathtt{DW}_{\mathsf{Dnameloc}} \cup \mathtt{DW}_{\mathsf{Dsizeloc}} \cup \mathtt{DW}_{\mathsf{Floc}}$. For each type of writes we show its inclusion in the corresponding part of the frontier, namely N, S, and F, respectively.

A set of disk write events D is included in $\left\{\max(\mathtt{pb}_x|_P) \;\middle|\; Q(x)\right\}$ for some arbitrary predicate Q, iff (i) the locations of D satisfy Q—we call this property *inclusion*, (ii) the events in D are $\mathtt{pb}|_P$-maximal for each location; i.e., $\forall \mathit{dl} \in \mathsf{Dloc}.\ \mathtt{DW}_{\mathit{dl}} \cap D \cap \mathit{dom}(\mathtt{pb}_{\mathit{dl}}|_P) = \emptyset$—we call this property *maximality*. This way, for each type of writes we show its *inclusion* and *maximality*.

The following lemma provides the *maximality* in a uniform way for all three types of writes.

Lemma 6.6 (Write Maximality). For any $\mathit{dl} \in \mathsf{Dloc}$, write events in $\mathit{dom}([\mathtt{DW}_{\mathit{dl}}]; \mathtt{rf}; [\mathtt{REC}])$ are $\mathtt{pb}_{\mathit{dl}}|_P$-maximal.

Proof. Assume that $\emptyset \neq \mathit{dom}([\mathtt{DW}_{\mathit{dl}}]; \mathtt{rf}; [\mathtt{REC}]) \cap \mathit{dom}(\mathtt{pb}_{\mathit{dl}}|_P; [P])$.

Then $\emptyset \neq \mathit{dom}([\mathtt{DW}_{\mathit{dl}}]; \mathtt{rf}; [\mathtt{REC}]) \cap \mathit{dom}(\mathtt{pb}_{\mathit{dl}}|_P; [\mathtt{I}{\cap}\mathtt{D} \cup \mathit{dom}(\mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}])]) \subseteq \mathit{dom}([\mathtt{DW}_{\mathit{dl}}]; \mathtt{rf}; [\mathtt{REC}]) \cap \mathit{dom}(\mathtt{pb}_{\mathit{dl}}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}])$.

In terms of graphs, it means that there exists an $\mathtt{rf}$-edge (u, v) from $\mathtt{DW}_{\mathit{dl}}$ to $\mathtt{REC}$, and there exists a path p: $\mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}$ starting in u and ending in $\mathtt{REC}$. Then there exists a path starting in v, then going to u, and then going along p. In other words, $\emptyset \neq [\mathtt{REC}]; \mathtt{rf}^{-1}; [\mathtt{DW}_{\mathit{dl}}]; \mathtt{pb}_{\mathit{dl}}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}]$. Since $\mathtt{pb}_{\mathit{dl}} = \mathtt{whb}_{\mathit{dl}}$ for each disk location dl, $\mathtt{rf}^{-1}; [\mathtt{DW}_{\mathit{dl}}]; \mathtt{pb}_{\mathit{dl}} \subseteq \mathtt{rb}$, and thus, $\emptyset \neq [\mathtt{REC}]; \mathtt{rb}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}]$, which violates REC. □

We next show the *inclusion* of each type of writes. The *inclusion* is straightforward for name writes, because the predicate $Q(\mathit{dl}) =$ '$\mathit{dl} \in \mathsf{Dnameloc}$' for the frontier subset N is trivially satisfied.

Therefore, the wanted property for name writes is obtained:

Corollary 6.7 (Canonical Name Frontier). $\forall nl \in \mathsf{Dnameloc}.\ dom([\mathtt{DW}_{nl}];\mathtt{rf};[\mathtt{REC}]) \subseteq N$

To get an analogous property for size writes and data writes, first, we prove the *inclusion* lemmas for them.

Lemma 6.8 (Size Writes Inclusion). $\forall f \in \mathsf{Inode}, w \in dom([\mathtt{DW}_{ds_f}];\mathtt{rf};[\mathtt{REC}]).\ \exists w' \in N.\ \mathtt{lab}(w') = \mathtt{DW}(nl_f, f)$

Proof. Since $[\mathtt{DW}_{ds_f}];\mathtt{rf};[\mathtt{REC}] \neq \emptyset$, there exists $r \in \mathtt{REC}$ s.t. $\mathtt{loc}(r) = ds_f$. By the READ-NAME-FIRST, there exists $r' \in \mathtt{REC}$ s.t. $\mathtt{lab}(r') = \mathtt{DR}(nl_f, f)$.

Let us take w' s.t. $(w', r') \in \mathtt{rf}$. We are to show that (i) $w' \in N$ (ii) $\mathtt{lab}(w') = \mathtt{DW}(nl_f, f)$.

(i) By the properties of $\mathtt{rf}$, $(\mathtt{loc}(w'), \mathtt{val}(w')) = (\mathtt{loc}(r'), \mathtt{val}(r'))$, thus $\mathtt{lab}(w') = \mathtt{DW}(nl_f, f)$.

(ii) $w' \in dom([\mathtt{DW}_{nl}];\mathtt{rf};[\mathtt{REC}])$, thus by Corollary 6.7, $w' \in N$.

□

This then gives us the wanted property for the size reads:

Corollary 6.9 (Canonical Size Frontier). $\forall ds \in \mathsf{Dsizeloc}.\ dom([\mathtt{DW}_{ds}];\mathtt{rf};[\mathtt{REC}]) \subseteq S$

The following lemma provides the *inclusion* property for the Data Writes.

Lemma 6.10 (Data Writes Inclusion). $\forall (f, o) \in \mathsf{Floc}, w \in dom([\mathtt{DW}_{(f,o)}];\mathtt{rf};[\mathtt{REC}]).\ \exists w_1 \in N, w_2 \in S.\ \mathtt{lab}(w_1) = \mathtt{DW}(nl_f, f) \wedge \exists s.\ \mathtt{lab}(w_2) = \mathtt{DW}(ds_f, s) \wedge o < s.$

Proof. Since $[\mathtt{DW}_{(f,o)}];\mathtt{rf};[\mathtt{REC}] \neq \emptyset$, there exists $r \in \mathtt{REC}$ s.t. $\mathtt{loc}(r) = (f, o)$. By the READ-SIZE-FIRST, there exists $r' \in \mathtt{REC}$ s.t. $\mathtt{lab}(r') = \mathtt{DR}(ds_f, s)$ and $o < s$. Let us take w_2 s.t. $(w_2, r') \in \mathtt{rf}$. The chosen write w_2 satisfies the requirement in the lemma statement ((i) $w_2 \in S$ and (ii) $\mathtt{lab}(w_2) = \mathtt{DW}(ds_f, s)$) :

(i) $w_2 \in dom([\mathtt{DW}_{ds}];\mathtt{rf};[\mathtt{REC}])$, thus by Corollary 6.9, $w_2 \in S$,

(ii) by the properties of $\mathtt{rf}$, $(\mathtt{loc}(w_2), \mathtt{val}(w_2)) = (\mathtt{loc}(r'), \mathtt{val}(r'))$, thus $\mathtt{lab}(w_2) = \mathtt{DW}(ds_f, s)$.

Applying Lemma 6.8 to w_2, we obtain $w_1 \in N$ s.t. $\mathtt{lab}(w_1) = \mathtt{DW}(nl_f, f)$. □

Combining *inclusion* and *maximality* for the data writes, we conclude the following:

Corollary 6.11 (Canonical Data Frontier). $\forall (f, o) \in \mathsf{Floc}.\ dom([\mathtt{DW}_{(f,o)}];\mathtt{rf};[\mathtt{REC}]) \subseteq F$

Combining corollaries 6.7, 6.9 and 6.11, we get:

Corollary 6.12 (Canonical Frontier). $dom(\mathtt{rf};[\mathtt{REC}]) \subseteq \mathsf{front}_P$

And finally, Lemma 6.5 and Corollary 6.12 give us the soundness result:

Theorem 6.13 (Recovery Soundness). For all well-formed consistent instrumented executions G, there exists a p-snapshot P of G such that $dom(\mathtt{rf};[\mathtt{REC}]) \subseteq \mathsf{front}_P$.

6.2.2 Recovery Completeness

Theorem 6.14 (Recovery Completeness). For all consistent executions $G{=}\langle \mathtt{E}, \mathtt{rf}\rangle$, p-snapshots P of G, relations $\mathtt{rf}'$ and recovery reads $\mathtt{REC} \subseteq \mathtt{DR}$ by t_r ($\forall r \in \mathtt{REC}.\ \mathtt{tid}(r){=}t_r$), if $rng(\mathtt{rf}'){=}\mathtt{REC}$, $dom(\mathtt{rf}') \subseteq \mathsf{front}_P$, and the instrumented execution $G'{=}\langle \mathtt{E} \uplus \mathtt{REC}, \mathtt{rf} \uplus \mathtt{rf}'\rangle$ is well-formed, then G' is consistent.

Proof. To show instrumented consistency of G' we prove its consistency according to Def. 4.3 and show that REC holds. For brevity, we drop the G' prefix before relation name and write e.g., $\mathtt{rf}$ instead of $G'.\mathtt{rf}$.

CON Since $G'.\mathtt{NREC} = G$, its consistency follows from the premise of the lemma.

REC We show that $[\mathtt{REC}]; \mathtt{rb}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}] = \emptyset$. Here we exploit the fact that the *p-snapshot* P is downward-closed w.r.t. $\mathtt{pb}$ and $\mathtt{atom}$ (Def. 4.4). Let $\mathtt{WHB} \triangleq \bigcup_{dl \in \mathsf{Dloc}} \mathtt{whb}_{dl}$ and $\mathtt{PB} \triangleq \bigcup_{dl \in \mathsf{Dloc}} \mathtt{pb}_{dl}$. We then have:

$$
\begin{array}{lll}
& [\mathtt{REC}]; \mathtt{rb}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}] & \text{(by definition of } \mathtt{rb}\text{)} \\
= & [\mathtt{REC}]; \mathtt{rf}^{-1}; \mathtt{WHB}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}] & \text{(as } dom(\mathtt{rf}; [\mathtt{REC}]) \subseteq \mathsf{front}_P\text{)} \\
\subseteq & [\mathtt{REC}]; \mathtt{rf}^{-1}; [\mathsf{front}_P]; \mathtt{WHB}; \mathtt{atom}^?; \mathtt{pb}^?; [\mathsf{front}_P]; \mathtt{rf}; [\mathtt{REC}] & \text{(as } \mathsf{front}_P \subseteq P\text{)} \\
\subseteq & [\mathtt{REC}]; \mathtt{rf}^{-1}; [\mathsf{front}_P]; \mathtt{WHB}; \mathtt{atom}^?; \mathtt{pb}^?; [P]; \mathtt{rf}; [\mathtt{REC}] & \text{(as } P \text{ is } \mathtt{pb}\text{-downward-closed)} \\
\subseteq & [\mathtt{REC}]; \mathtt{rf}^{-1}; [\mathsf{front}_P]; \mathtt{WHB}; \mathtt{atom}^?; [P]; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}] & \text{(as } P \text{ is } \mathtt{atom}\text{-downward-closed)} \\
\subseteq & [\mathtt{REC}]; \mathtt{rf}^{-1}; [\mathsf{front}_P]; \mathtt{WHB}; [P]; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}] & \text{(as } [\mathsf{front}_P]; \mathtt{WHB}; [P] \subseteq \\
= & \emptyset & [\mathsf{front}_P]; \mathtt{PB}; [P] = \emptyset\text{)}
\end{array}
$$

□

7 Application in model checking

As described in §6, we adapted the `ext4` model for formal verification methods. In this section, we describe how the semantics was used on practice, for the real software verification. Although the application is not regarded as a contribution of the present work, we describe it in order to present the semantics in context and to emphasize the potential of the proposed technique.

The developed `ext4` semantics was utilized in PerSeVerE stateless model checker. We begin the section with an overview of the PerSeVerE algorithm on examples. Then we describe bugs in popular text editors (`nano` and `emacs`) found by PerSeVerE.

7.1 PerSeVerE in a nutshell

The semantics we present was utilized in an effective model checking algorithm, PerSeVerE, for automatically verifying sequential or concurrent C/C++ programs that perform file I/O using the POSIX system calls. The key challenge PerSeVerE tackles is combating the state space explosion arising from the filesystem semantics.

To see this, consider a sequential program with N independent file operations and no synchronization calls. These operations may persist to disk in any order (i.e., $N!$ ways) and any prefix of such orders may have completed before a crash (i.e., $N \times N!$ possible states). However, this naive enumeration of persistency ordering is far from optimal. A much better way is not to enumerate the orders in which operations persist, and instead to consider whether each of the N operations persisted before the crash (i.e., 2^N states). Moreover, it is typically the case that only a small part of these operations (M much less than N) are relevant for the invariant in question, so it suffices to enumerate 2^M states. When there are synchronization calls, persistency of one operation implies persistency of all prior operations that are separated by a synchronization call, which further reduces the number of states.

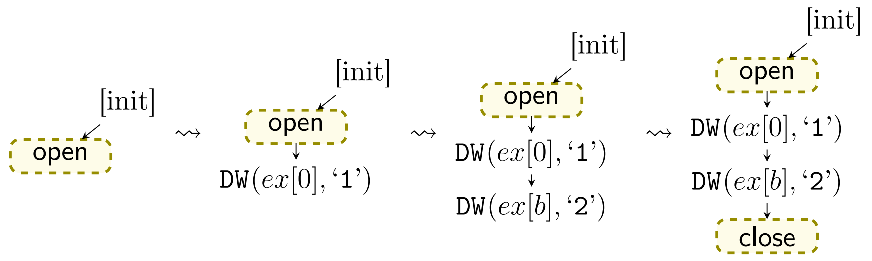
The key idea for exploring this vast state space efficiently is to model the assertions about the persisted state as a *recovery observer* that runs in parallel to the main program P and whose accesses are subject to different consistency axioms from those of P. By ensuring that the axioms do not require a total persistency order, the model checker never enumerates this order explicitly and thus significantly reduces the number of states to explore. Finally, following an axiomatic semantics enables integrating PerSeVerE into existing efficient algorithms for enumerating the (weak) behaviors of concurrent programs, thereby leveraging the state-of-the-art implementations.

To illustrate the PerSeVerE exploration process, let us consider the following program manipulating the "`ex.txt`" file which comprises multiple blocks of the '`0`' character on disk:

```
$d_f$ = open ("ex.txt");          $d_{f'}$ = open ("ex.txt");
pwrite ($d_f$, '1', 0);       ↯   $c_1$ = pread ($d_{f'}$, 1, $b$);
pwrite ($d_f$, '2', $b$);          $c_2$ = pread ($d_{f'}$, 1, 0);        (REC-WW+RR)
close ($d_f$);                    close ($d_{f'}$);
                                  assert($\neg(c_1 = $ '2' $\land c_2 = $ '0'));
```

Let b denote the starting offset of the file's second block; and the code to the right of ↯ denote the recovery observer, inquiring whether it is possible upon recovery to see the second write but not the first; i.e., $c_1 =$ '2' $\land$ $c_2 =$ '0' (this is indeed possible in `ext4`). We next show how PERSEVERE generates all possible outcomes of REC-WW+RR including one where $c_1 =$ '2' $\land$ $c_2 =$ '0'.

First, it explores the main program P (to the left of ↯) line by line, generating the single possible execution (for brevity, we omit the lock/unlock events of pwrite and the read events reading the in-memory file size):



Second, per each explored main program execution (in our case, the single one), PERSEVERE starts the recovery routine exploration. The instruction $c_1 =$ pread $(d_{f'}, 1, b)$ generates the event DR$(ex[b])$, which can read two different values: '2' written by P or the initial data at b. PERSEVERE explores both options, leading to executions $\textcircled{2}$ and $\textcircled{3}$, respectively (see Fig. 8, top).

Let us assume that PERSEVERE continues with $\textcircled{2}$. PERSEVERE next adds the DR event corresponding to $c_2 =$ pread $(d_{f'}, 1, 0)$, which can similarly read from two values: value '1' written by P or the initial value on disk. As before, PERSEVERE explores both options, leading to executions $\textcircled{21}$ and $\textcircled{22}$ depicted in Fig. 8 (middle).

Assuming that PERSEVERE continues with $\textcircled{21}$, it finally adds the close events (not depicted), at which point the resulting execution is complete, thus concluding the first exploration. Next, PERSEVERE backtracks to $\textcircled{22}$ to explore it further, and concludes the second exploration after adding the close event.

Finally, since there are no more alternative reads-from options for DR$(ex[0])$, PERSEVERE backtracks to execution $\textcircled{3}$ as shown in Fig. 8 (bottom), and explores it further in a similar manner to the exploration starting from $\textcircled{2}$.

We conclude with an observation. Because of the absence of `pb` edges in execution $\textcircled{1}$, the reads-from options of the recovery observer of REC-WW+RR are unrestricted, and so four executions were generated. This would be different if the two writes of P were to the

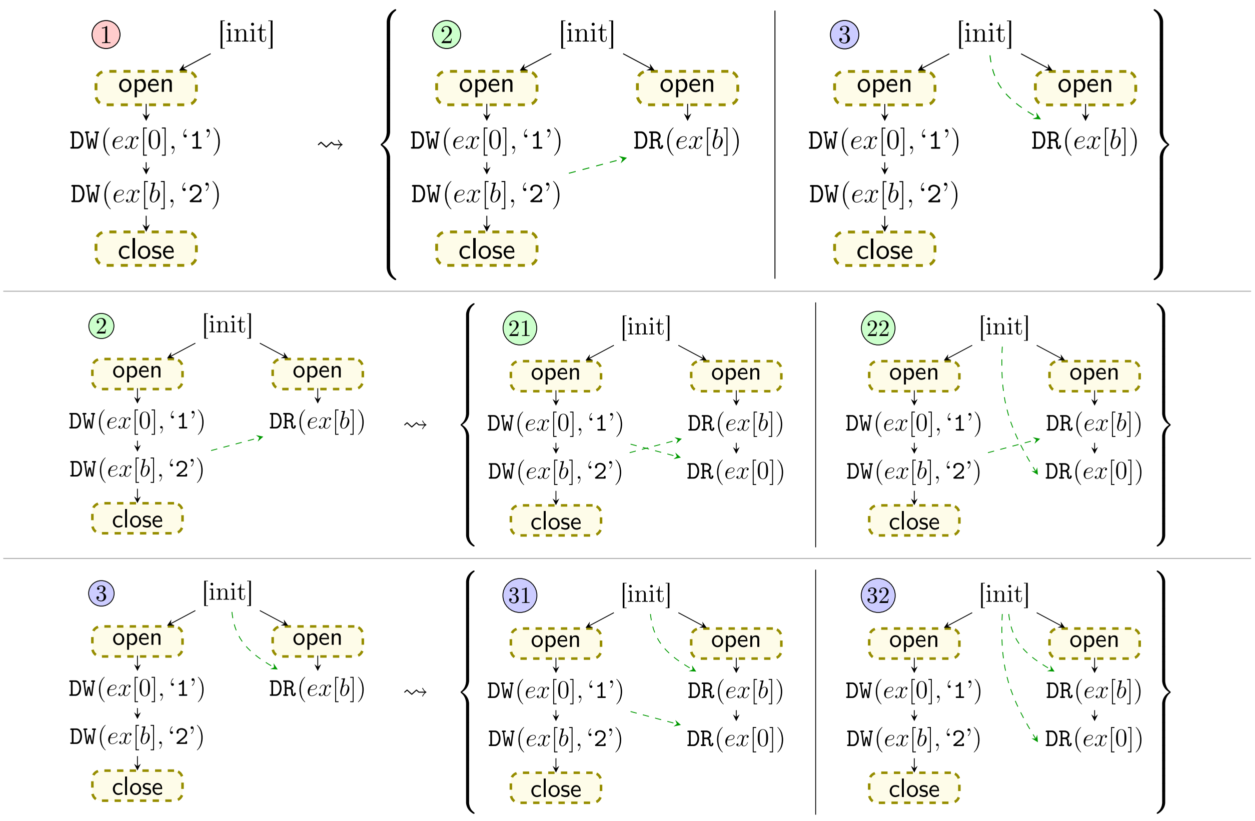


Figure 8: Exploration of the REC-WW+RR recovery observer: step 1 (top), step 2 (middle) and step 3 (bottom)

same block (i.e., if b pointed to a disk location in the first block), as there would be a pb edge between the two writes because of PB-BLOCK. Specifically, execution 22 where DR($ex[0]$) reads '0' would be deemed inconsistent as it would create an edge in [REC]; rb; pb; rf; [REC], thus violating REC.

7.2 Persistency bugs in text editors

Next, we briefly overview several issues in text editors found and reproduced by PerSeVerE. The key problem PerSeVerE discovered is that the text editors do not provide crash safety, although it was intended. Specifically, a crash occurring during the save procedure may lead to the file corruption.

Save in `nano`. The procedure followed by `nano` when saving an open buffer `BUF` as "`f.txt`" is shown in Fig. 9 (left): "`f.txt`" is first truncated (if it exists), and then the contents of `BUF` are written with one write system call. Surprisingly, however, there is no call to fsync after write: the editor may claim that the file is saved and even exit (when the user exits after saving) without the data persisted to disk. This is misleading from the point of view of users, who naturally expect that saving a file implies persistency.

close does *not* wait for the data to be flushed before returning. This then allows a race window where it is possible to see the truncated file without seeing the data that is

```
save("f.txt") :
    $d_f$ = open ("f.txt", O_FLAGS);
    write ($d_f$, BUF);
    close ($d_f$);

backup("f.txt") :
    $d_f$ = open ("f.txt", O_RDONLY);
    $d_b$ = open ("f.txt∼", O_FLAGS);
    $b$ = read ($d_f$);     write ($d_b$, $b$);
    close ($d_b$);     close ($d_f$);
```

Figure 9: Save procedure in text editors (left) and unsafe backup procedure (right). Here `O_FLAGS = O_WRONLY|O_CREAT|O_TRUNC`.

subsequently written to it.

Of course, data loss could occur even if fsync were used, since a crash could occur during fsync. There are, however, two observations to note.

First, users commonly expect a file to be safely persisted to disk once the editor reports it as "saved". If a crash occurs during an fsync, i.e., while saving, data will be lost, but the user would expect this as saving is incomplete. Having an editor exiting gracefully, only to learn later that the file is corrupted is counter-intuitive, to say the least.

Second, to avoid data loss in such cases where crashes occur while saving, editors should make temporary data backups. However, this backup strategy has the very problem suffered by the save procedure: it follows the same writing pattern which does not guarantee that data persists once a file is closed. Once again, without fsync the backup may not be safely persisted to disk before the save procedure starts. It is therefore possible to obtain a corrupted file *and* a corrupted backup. This backup procedure can be made crash-safe by adding an fsync after write. As such, if a crash occurs while saving the original file, the backup copy would be available on disk.

Save in `vim` and `emacs`. The save procedures of `vim` and `emacs` are as in Fig. 9 with an fsync after write, ensuring that file data has safely persisted to disk once the save is complete. Moreover, they use a different backup strategy that, for most cases, does not suffer from the problems discussed above. Specifically, when a file is first modified in a session, `emacs` typically creates a backup as follows:

```
rename ("f.txt", "f.txt∼");
```

`vim` follows a similar strategy and we omit it for brevity. The above procedure with a simple rename call is crash-safe under `ext4`. More concretely, if a crash occurs before the rename commits, then the original file will be intact. As the file truncation from save is c-ordered after rename (i.e., the backup), it will also be p-ordered after it (PB-DIROPS). Therefore, even if a crash occurs during fsync leading to a truncated file, the backup will have persisted, thus avoiding data loss.

In certain cases, however, `emacs` follows a different backup strategy than the one mentioned above. One such case is when the original file has incoming hard links, in which case renaming the original file would undesirably direct the hard links to the backup's name. As such, `emacs` *copies* the original file instead, with a procedure similar to that of `nano` in

Fig. 9 (right). In particular, as the `emacs` backup strategy in such cases does not use `fsync`, it is *not crash-safe* as discussed above.

8 Related work

Formal specification and verification of filesystems, persistent programming, and persistency semantics are highly active research areas, as evidenced by the rapidly growing literature and codebases (e.g., [5, 14, 15, 16, 33, 36, 37]).

To our knowledge, the present work is the first project that encompasses both consistency and persistency guarantees in one (axiomatic) framework.

Several tools have been successful in finding persistency bugs in applications under different filesystems (e.g., [7, 24, 29, 35, 41, 42]). The models used by these tools are products of empirical studies and thorough testing. Unfortunately, however, they do not come with formal semantics. On the other hand, such tools can be usually used under many different filesystems.

A notable exception to the above (and most closely related to our work) is the work of [4], providing a framework for specifying and synthesizing the persistency semantics of different filesystems, as well as the FERRITE tool which exhaustively enumerates the persistency behaviors of litmus tests against the models of different filesystems. In contrast to PERSEVERE, FERRITE leverages SMT techniques to execute litmus tests symbolically against filesystem specifications, and may explore all prefixes of a given `pb` relation to check whether a given safety property holds. Moreover, FERRITE does not model the consistency guarantees of different filesystems, and only focuses on persistency. Among the filesystems they model is `ext4`, but their model is not precise and it is not clear which aspects of `ext4` are covered, e.g., `data=writeback` or `data=ordered`.

Finally, [31, 32] use axiomatic semantics to model persistency guarantees of *Non-volatile memory* (NVM). NVM is a modern technology incorporating the performance of DRAM and the *durability* of HDD. As with hard disk drives, NVM provide persistent storage. Thus, formal models and semantics of filesystems resemble those for NVM to a large extent.

9 Conclusion

In this work, we have provided a formal persistency and consistency semantics of the `ext4` filesystem. Specifically, we have achieved the following results:

(i) We developed the general formal persistency framework based on axiomatic memory model formalization. The formalization covers a substantial part of filesystems, including the fundamental file and directory operations: reading, writing, offset seeking, renaming, linking, and unlinking.

(ii) The developed framework was specialized for the well-known `ext4` filesystem to obtain its formal semantics. The semantics exhaustively covers peculiarities of `ext4`, including *delayed allocation*, and reflects its atomicity/ordering guarantees in `pb` (persists before) relation. The completeness of the semantics was verified on a set of litmus tests.

(iii) We adapted the `ext4` semantics for formal verification and proved the equivalence between the original model and the adaptation. The adapted model was effectively used in PerSeVerE [28] stateless model checker. Further, PerSeVerE was applied to verify popular text editors: vim, Emacs, and nano, which allowed us to discover and report several critical crash-safety bugs.

This work has been published at POPL (principles of programming languages) conference proceedings, January 2021 edition [17].

Further, the presented model can be used to verify the implementations of other applications depending on IO (e.g., databases), and check if they provide sufficient crash-safety guarantees. Apart from that, we believe that the developed framework can be extended to model persistency guarantees of Non-Volatile Memory and other filesystems, such as `xfs` [38] and `btrfs` [34].

References

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. "Optimal dynamic partial order reduction." In: *POPL 2014*. New York, NY, USA: ACM, 2014, pages 373–384.

[2] *Advanced Format*. [Online; accessed 20-May-2020]. 2020.

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig. "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory." In: *ACM Trans. Program. Lang. Syst.* 36.2 (July 2014), 7:1–7:74.

[4] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. "Specifying and Checking File System Crash-Consistency Models." In: *ASPLOS 2016* 44.2 (2016), pages 83–98.

[5] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. "Using Crash Hoare logic for certifying the FSCQ file system." In: *SOSP 2015*. the 25th Symposium. Monterey, California: ACM Press, 2015, pages 18–37. (Visited on June 17, 2020).

[6] *Copy-on-write*. [Online; accessed 20-May-2020]. 2020. URL: `https://en.wikipedia.org/wiki/Copy-on-write`.

[7] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. "Verifying Systems Rules Using Rule-Directed Symbolic Execution." In: *ASPLOS 2013*. Houston, Texas, USA: Association for Computing Machinery, 2013, pages 329–342.

[8] *Ext4 data loss*. [Online; accessed 20-May-2020]. 2009. URL: `https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781`.

[9] ext4 Linux kernel. *ext4 Data Structures and Algorithms*. [Online; accessed 20-May-2020]. 2020. URL: `https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html`.

[10] ext4 corruption. *ext4: Filesystem corruption on panic*. [Online; accessed 20-May-2020]. 2015. URL: `https://bugs.chromium.org/p/chromium/issues/detail?id=502898`.

[11] Cormac Flanagan and Patrice Godefroid. "Dynamic partial-order reduction for model checking software." In: *POPL 2005*. New York, NY, USA: ACM, 2005, pages 110–121.

[12] Patrice Godefroid. "Model Checking for Programming Languages using VeriSoft." In: *POPL 1997*. Paris, France: ACM, 1997, pages 174–186.

[13] Patrice Godefroid. "Software Model Checking: The VeriSoft Approach." In: *Form. Meth. Syst. Des.* 26.2 (March 2005), pages 77–101.

[14] Rajeev Joshi and Gerard Holzmann. “A Mini Challenge: Build a Verifiable Filesystem.” In: *Formal Asp. Comput.* 19 (June 11, 2007), pages 269–272.

[15] Eunsuk Kang and Daniel Jackson. “Formal Modeling and Analysis of a Flash Filesystem in Alloy.” In: *ABZ 2008*. Edited by Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca. Volume 5238. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 294–308. (Visited on June 17, 2020).

[16] Gabriele Keller, Toby Murray, Sidney Amani, Liam O’Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. “File systems deserve verification too!” In: *PLOS 2013*. Farmington, Pennsylvania: ACM Press, 2013, pages 1–7. (Visited on June 17, 2020).

[17] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. “PerSeVerE: Persistency Semantics for Verification under Ext4.” In: *Proc. ACM Program. Lang.* 5.POPL (January 2021).

[18] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. “Model Checking for Weakly Consistent Libraries.” In: *PLDI 2019*. New York, NY, USA: ACM, 2019.

[19] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. “Taming Release-acquire Consistency.” In: *POPL 2016*. St. Petersburg, FL, USA: ACM, 2016, pages 649–662.

[20] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. “Repairing Sequential Consistency in C/C++11.” In: *PLDI 2017*. Barcelona, Spain: ACM, 2017, pages 618–632.

[21] *Linux man pages*. [Online; accessed 20-May-2020]. 2020. URL: `http://www.man7.org/linux/man-pages/index.html`.

[22] Richard Gooch. *Overview of the Linux Virtual File System*. [Online; accessed 20-May-2020]. 1999. URL: `https://www.kernel.org/doc/html/latest/filesystems/vfs.html`.

[23] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, and Laurent Vivier. *The new ext4 filesystem: current status and future plans*. 2007. URL: `https://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf` (visited on June 17, 2020).

[24] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. “Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing.” In: *OSDI 2018*. OSDI’18. Carlsbad, CA, USA: USENIX Association, 2018, pages 33–50.

[25] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. “Finding and Reproducing Heisenbugs in Concurrent Programs.” In: *OSDI 2008*. USENIX Association, 2008, pages 267–280.

[26] Gian Ntzik. “Reasoning About POSIX File Systems.” PhD thesis. Imperial College London, 2016.

[27] Scott Owens, Susmit Sarkar, and Peter Sewell. “A Better x86 Memory Model: x86-TSO.” In: *TPHOLs 2009*. Munich, Germany: Springer, 2009, pages 391–407.

[28] *PerSeVerE project*. [Online; accessed 6-May-2021]. 2021. URL: `http://plv.mpi-sws.org/persevere/`.

[29] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications.” In: *OSDI 2014*. Broomfield, CO: USENIX Association, October 2014, pages 433–448. URL: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai`.

[30] POSIX. *The Open Group Base Specifications Issue 7*. [Online; accessed 20-May-2020]. 2018. URL: `https://pubs.opengroup.org/onlinepubs/9699919799/`.

[31] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. “Persistency semantics of the Intel-x86 architecture.” In: *Proc. ACM Program. Lang.* 4 (POPL December 20, 2019), 11:1–11:31.

[32] Azalea Raad, John Wickerson, and Viktor Vafeiadis. “Weak Persistency Semantics from the Ground Up.” In: *Proc. ACM Program. Lang.* 3 (OOPSLA October 10, 2019), 135:1–135:27.

[33] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. “SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems.” In: *SOSP 2015*. Monterey, California: ACM Press, 2015, pages 38–53. (Visited on June 17, 2020).

[34] Ohad Rodeh, Josef Bacik, and Chris Mason. “BTRFS: The Linux B-Tree Filesystem.” In: *ACM Trans. Storage* 9.3 (August 1, 2013), 9:1–9:32. (Visited on June 17, 2020).

[35] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. “Error propagation analysis for file systems.” In: *SIGPLAN Not.* 44.6 (June 15, 2009), pages 270–280. (Visited on June 17, 2020).

[36] Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler, Dominik Haneberg, and Wolfgang Reif. “Development of a Verified Flash File System.” In: *ABZ 2014*. Volume 8477. Berlin, Heidelberg, 2014, pages 9–24. (Visited on June 17, 2020).

[37] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. “Push-Button Verification of File Systems via Crash Refinement.” In: *OSDI 2016*. OSDI’16. Savannah, GA, USA: USENIX Association, 2016, pages 1–16. URL: `https://www.usenix.org/system/files/conference/osdi16/osdi16-sigurbjarnarson.pdf`.

[38] Adam Sweeney. “Scalability in the XFS file system.” In: *USENIX ATC 1996*. 1996, pages 1–14. URL: `https://www.usenix.org/legacy/publications/library/proceedings/sd96/sweeney.html`.

[39] Stephen C Tweedie. “Journaling the Linux ext2fs Filesystem.” In: (1998), page 8. URL: `http://e2fsprogs.sourceforge.net/journal-design.pdf`.

[40] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. “Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It.” In: *POPL 2015*. Mumbai, India: ACM, 2015, pages 209–220.

[41] Junfeng Yang, Can Sar, and Dawson Engler. “EXPLODE: a lightweight, general system for finding serious storage system errors.” In: *OSDI 2006*. Seattle, Washington: USENIX Association, November 6, 2006, pages 131–146. (Visited on June 17, 2020).

[42] Mai Zheng, Joseph Tucek, Dachuan Huang, Elizabeth S Yang, Bill W Zhao, Feng Qin, Mark Lillibridge, and Shashank Singh. “Torturing Databases for Fun and Profit.” In: (2014), page 17.