A Concurrent Specification of POSIX File Systems

Gian Ntzik
Imperial College London & Amadeus, UK
gian.ntzik@amadeus.com

Pedro da Rocha Pinto
Imperial College London, UK
pmd09@doc.ic.ac.uk

Julian Sutherland
Imperial College London, UK
jhs110@doc.ic.ac.uk

Philippa Gardner
Imperial College London, UK
pg@doc.ic.ac.uk

Abstract

POSIX is a standard for operating systems, with a substantial part devoted to specifying file-system operations. File-system operations exhibit complex concurrent behaviour, comprising multiple actions affecting different parts of the state: typically, multiple atomic reads followed by an atomic update. However, the standard's description of concurrent behaviour is unsatisfactory: it is fragmented; contains ambiguities; and is generally under-specified. We provide a formal concurrent specification of POSIX file systems and demonstrate scalable reasoning for clients. Our specification is based on a concurrent specification language, which uses a modern concurrent separation logic for reasoning about abstract atomic operations, and an associated refinement calculus. Our reasoning about clients highlights an important difference between reasoning about modules built over a heap, where the interference on the shared state is restricted to the operations of the module, and modules built over a file system, where the interference cannot be restricted as the file system is a public namespace. We introduce specifications conditional on *context invariants* used to restrict the interference, and apply our reasoning to the example of lock files.

2012 ACM Subject Classification Theory of computation → Program verification

Keywords and phrases POSIX, concurrency, file systems, refinement, separation logic, atomicity

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.4

Funding EPSRC Grants EP/H008373/1, EP/K008528/1 and EP/L016796/1

1 Introduction

POSIX [2] is a standard for operating systems, with a substantial part devoted to specifying file-system operations. File-system operations exhibit complex fine-grained concurrent behaviour, in the sense that they comprise multiple actions affecting different parts of the state: typically, multiple atomic[1] reads followed by an atomic update. The standard's description of this complex concurrent behaviour is unsatisfactory: it is fragmented; contains ambiguities; and is generally under-specified. There has been much work on formal, mathematical

[1] Atomic in the sense of *linearisability* [18], where operations appear to take effect at a single discrete point in time.

specifications of POSIX file systems, but no formal description of its concurrent behaviour: the work either restricts to sequential fragments (for example [3, 19, 24, 15, 16, 17, 7, 30]); or takes a coarse-grained view of concurrency that does not capture the POSIX behaviour [33].

Although poorly described, there is a consensus between major file-system implementations on what the concurrent behaviour of POSIX file systems should be. File-system operations (such as unlinking files) typically traverse paths to identify the files or directories on which they will act. Path traversal comprises a sequence of multiple atomic reads, each looking up a component of the path within a directory. Other operations (such as renaming files) exhibit the more complex behaviour of resolving multiple paths. Since POSIX does not specify the order in which multiple paths are resolved, the atomic reads of multiple path traversals can be arbitrarily interleaved. After the path resolution, other atomic actions perform the intended update of the file-system operation. In summary, file-system operations are sequential and parallel combinations of atomic actions.

We provide the first formal concurrent specification of POSIX file systems using a specification language based on concurrent separation logic. Such separation logics provide compositional reasoning about fine-grained concurrency and atomic operations: for example, the TaDA program logic [9, 8] uses a first-order approach to atomicity; the Iris framework [23] encodes the TaDA atomicity, using a higher-order approach initially introduced by Jacobs [20]; and the FCSL logic [26] uses histories. With TaDA, we are able to reason directly about atomic operations by introducing abstract atomic triples. However, such an atomic triple only specifies one atomic action for a given program statement. We cannot specify POSIX file-system operations which perform multiple atomic actions using TaDA. With Iris, it is possible to give a higher-order encoding of the TaDA atomic triples, yielding multiple atomic operations for free. We spent a considerable amount of time trying to use Iris to specify POSIX file-system operations, but found that the Hoare-style specifications were getting too complex. The issue is that the multiple atomic actions in POSIX are not simple linear sequences of atomic steps, but exhibit patterns of control flow which are better associated with program statements than logical assertions. The same issue also arises with FCSL [26].

We introduce TaDA-Refine, a specification language for specifying multiple atomic actions using TaDA assertions in the basic atomic statements, and an associated refinement calculus [4] for verifying clients. Our approach is inspired by the work of Turon and Wand [36], which was the first to combine such a specification language with separation-logic reasoning [32]. They introduced a refinement calculus for reasoning about *atomicity abstraction*, where a specification program appears to perform an operation in one atomic step even though its implementation takes many steps. They can verify that operations on simple data structures, such as incrementing a non-blocking counter, can be abstracted to atomic specification statements. They introduce an ownership discipline, formally captured by the notion of fenced refinement, to verify operations on more complex data structures such as a non-blocking stack. In contrast, we are able to reason about complex data structures using assertions and laws inspired form modern concurrent separation logics.

Our specifications of POSIX file-system operations take the form of simple programs from the TaDA-Refine specification language, built from *atomic specification statements*. An atomic specification statement has the form $\forall \vec{x}. \langle P, Q \rangle$, where P and Q are TaDA assertions for describing shared state. It provides an abstract description of operations that, for arbitrary $\vec{x}$, atomically updates states satisfying precondition P to states satisfying postcondition Q. The associated refinement calculus gives subtle behaviour to these atomic statements. For example, using the stuttering law of refinement, an equivalent specification program is $\forall \vec{x}. \langle P, P \rangle; \forall \vec{x}. \langle P, Q \rangle$. In fact, the combination of stuttering and mumbling laws with the

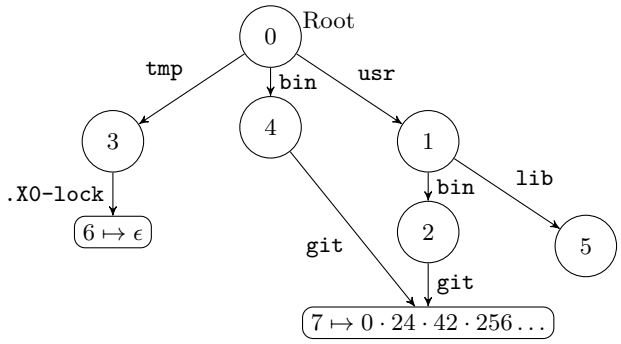

Figure 1 Example snapshot of a file-system graph.

universal quantification means that the atomic statements are robust to the environment changing the values of the $\vec{x}$ over time. In §5 and the technical report [29], we demonstrate that the presence of these laws means that the TaDA-Refine laws, model and soundness proof are significantly simpler than those of TaDA.

We use TaDA-Refine to verify clients of POSIX file systems, often using derived *hybrid specification statements* to reason about both atomic and non-atomic behaviour within one specification. Our client reasoning is different from the usual reasoning about concurrent heap modules using concurrent separation logics. A heap is a private namespace in the sense that a thread can safely access only what its been given through allocation or ownership transfer. Concurrent modules built over a heap restrict the interference on the shared resource they encapsulate, by only allowing access to the resource via the module operations. In contrast, a file system is a public namespace in the sense that a process or thread by default has access to any part of the file system. File access permissions can only enforce restrictions to sets of processes. It is not possible for a thread to keep part of a file system hidden from the rest of the system to restrict interference. Instead, the interference must be explicitly restricted by the reasoning. We do this by introducing *context invariants* to our specifications. We study lock files which provide a simple example to introduce context invariants. A lock file is a regular file under a path. If the lock file exists, the lock it represents is locked. Otherwise, the lock is unlocked. Our context invariant ensures that the path must remain fixed, and the lock file can only be added and removed using the lock operations, not the file-system operations. Other examples of our client reasoning include named pipes which build on lock files, and an email server to demonstrate the importance of reasoning about the full concurrent behaviour of POSIX file systems.

2 POSIX File-system Primer

Most readers will have a basic understanding of POSIX file systems. They will perhaps have less of an understanding of the concurrent behaviour of the file-system operations. We describe the fragment of POSIX used in this paper, and illustrate why the concurrent behaviour is poorly specified in the standard.

2.1 POSIX File-systems

A file system is an abstraction used to organise data, typically stored in some storage medium such as a disk. In POSIX, this abstraction takes the form of a *directed graph*. In figure 1, we give an example of an instance of such a file-system graph. The nodes in the graph

Basic types:

$\iota_0, \iota, j, \ldots \in$ INODES : countable set of inode numbers, ι_0 is the root

$a, b, \ldots \in$ FNAMES : countable set of filenames

$\text{BYTES} \triangleq \{n \in \mathbb{N} \mid 0 \leq n < 2^8\}$ $\quad \text{ERRS} \triangleq \{\texttt{ENOENT}, \texttt{ENOTDIR}, \texttt{ENOTEMPTY}, \ldots\}$

$\text{PATHS'} \ni p ::= a \mid a/p$ $\quad \text{PATHS} \triangleq \text{PATHS'} \cup \{\varnothing_p\}$

File-system structure:

$FS \in \mathcal{FS} \triangleq \text{INODES} \xrightarrow{\text{fin}} \text{LINKS} \uplus \text{FILEDATA}$ $\quad \text{LINKS} \triangleq \text{FNAMES} \xrightarrow{\text{fin}} \text{INODES}$

$\text{FILEDATA} \triangleq \text{BYTES}^*; (\{\emptyset\}^*; \text{BYTES}^?)^*$ where $\emptyset$ denotes a file gap

Notation:

$\mathsf{isfile}(o) \triangleq o \in \text{FILEDATA}$ $\quad \mathsf{isdir}(o) \triangleq o \in \text{LINKS}$ $\quad \mathsf{iserr}(o) \triangleq o \in \text{ERRS}$

$\iota \in FS \triangleq \iota \in \text{dom}(FS)$ $\quad a \in FS(\iota) \triangleq a \in \text{dom}(FS(\iota))$

Figure 2 File-system structure, basic types and some notation.

are *files*. There are different types of files. For this paper, we are primarily interested in *directories* which are denoted as circles in figure 1, and *regular files* which are denoted as curved rectangles. Each file is uniquely identified by an *inode number*, or henceforth simply an inode. In figure 1, the inodes are integers with 0 denoting the root directory.

Directories store *links*[2] to other files. Each link has an associated name which is unique to the directory and, thus, the links give the files their names. In figure 1, the links are given by the labelled edges. Regular files contain *file data* which are sequences of *bytes* which need not necessarily be contiguous. In figure 1, the notation $7 \mapsto 0 \cdot 24 \cdot 42 \cdot 256$ describes a regular file with inode 7 and the sequence of bytes $0 \cdot 24 \cdot 42 \cdot 256$. Regular files can be linked more than once, as is the case with the file with inode 7 in figure 1.

In figure 2, we give the basic mathematical definitions for the file-system structure that we use throughout the paper. We give a simplified view of the file-system structure which is enough to introduce our reasoning. In particular, we omit features such as symbolic links, the special filenames "." and "..", and file-access permissions. These features are orthogonal to reasoning about the concurrent behaviour of file-system operation and are discussed further in Ntzik's thesis [27].

The basic types, given by the sets INODES with a distinguished inode ι_0 for the root directory, FNAMES and BYTES, are self-explanatory. The error types, given by the set ERRS, consists of the errors used by POSIX. We describe them as we use them in examples. The paths describe absolute paths starting from the root directory; a model in [27] also uses relative paths which start from a particular inode. The paths, given by the set PATHS, is either the empty path $\varnothing_p$ or a finite sequence of file names written as $a_1/.../a_n$

A *file-system structure* is a finite map from inodes to their contents, given by the set LINKS⊎FILEDATA. For a directory which stores links to other files, the content is described formally as a finite partial function from filenames to inodes, given by the set LINKS. A directory is empty if its link function has the empty domain. For a regular file, the context is a sequence of bytes of a regular language, given by the set FILEDATA, where $\emptyset$ denotes a gap in the sequence and ϵ denotes the empty sequence. A file-system structure is well formed if there are no dangling links.

[2] In POSIX, the terms link, hard link, directory entry, and entry mean the same thing.

A file-system structure is shared across all processes and is inherently concurrent. We have given concurrent specifications for operations of a core fragment of POSIX file systems. The fragment comprises the operations `mkdir`, `rmdir`,`link`, `unlink`, `rename`, `stat`, `open`, `close`, `read`, `write`, `lseek`, `opendir`, `closedir`, `readdir`, `pread` and `pwrite`. The fragment is significant, in that it includes most of the primitive structural commands that manipulate the file-system directory structure and the primitive input-output operations that change the contents of regular files. For this paper, we motivate our specifications by focusing in the structural operations `unlink` and `link`, and the input-output operations `read` and `write`. We also use a number of other operations in our client examples. The full specification of the fragment is available in Ntzik's thesis [27].

2.2 Concurrent Behaviour: the unlink operation

The POSIX file-system standard is a mature English standard with a comparatively clear description of the sequential behaviour of file-system operations. The description of the concurrent behaviour of file-system operations is much less clear. It is fragmented, contains ambiguities and is generally under-specified.

A particular difficulty lies with the POSIX atomicity guarantees for file-system operations. To illustrate this point, let us consider the `unlink` operation. Its sequential behaviour is straightforward. According to the POSIX standard (volume XSH, section 3), `unlink`(*path*) removes the link identified by the *path* argument. For example, using the file-system graph of figure 1, `unlink`(`/usr/bin/git`) first resolves the path `/usr/bin`, starting from the root directory, following the links `usr` and `bin` to yield the directory 2. It then removes the link named `git` to the regular file with inode 7 from this directory. If `unlink` is unable to resolve the path because, for example, one of the names in the path does not exist in the appropriate directory, it returns an error. Furthermore, POSIX allows some flexibility with the behaviour, in that it allows implementations to return an error if the path identifies a link to a directory rather than a regular file. In other words, `unlink` is permitted to exhibit non-determinism due to different implementation decisions.

In a different section of the standard (volume XSH, section 2.9.7), `unlink` is specified as *atomic*, which suggests that the whole process of resolving the path and removing the link to the identified file is logically indivisible. However, this is a common misconception. Hidden in the fine print of a section describing the specification rationale for `unlink`, we find the statement:

> "... Any part of the path of a file could be changed in parallel to a call to `unlink`, resulting in unspecified behavior ... ".

Here, *unspecified behaviour* means that we cannot predict whether the operation is going to succeed or error, even if we know the file-system's state when `unlink` is invoked. When the POSIX standard describes `unlink` as atomic (volume XSH, section 2.9.7), it means that the removal of the link that the path identifies is atomic. The path resolution itself comprises a sequence of atomic lookups that traverse the file-system graph by following the path. This fragmentation and ambiguity of the description of the `unlink` operation in the standard applies to all the POSIX operations that resolve paths. Such operations are sequences of atomic read operations followed by an atomic update which removes, adds, moves (renames) and looks up individual links in a directory. This behaviour is demonstrated by virtually all major file-system implementations. It has two interesting implications. First, because the file system can be changed arbitrarily by the concurrent environment between the individual atomic steps comprising an operation, it is impossible to determine whether the operation

is going to succeed or error just by examining the file-system state at the invocation point. The result depends on the concurrent environment and the scheduler interleavings. Thus, POSIX operations exhibit non-determinism due to concurrent interleavings. Second, if an operation succeeds, it does not necessarily mean that the path given as argument exists, or even existed at any single point in time. It merely means that the operation was able to resolve the path.

POSIX exhibits this ambiguity only for the operations that resolve paths. It is important to understand what the intentions of the standard are with respect to their behaviour. We suspect that POSIX does intend for these operations to have an atomic *effect*, but with consideration to implementation performance. A truly atomic implementation, where both the path resolution and the effect at the end of the path takes place in a single observable step, would require synchronisation over the entire file-system graph. For most implementations, the performance impact of this coarse-grained behaviour would be unacceptable. Therefore, the wording of the standard allows path resolution to be implemented non-atomically, as a sequence of atomic steps, where each looks up where the next name in the path leads to. The specification of path resolution (volume XBD, section 4.13), is silent on this matter.

Our interpretation of the standard's intentions is verified in the Austin Group mailing list [1].[3] Path resolution itself consists of a sequence of atomic lookups that traverse the file-system graph by following the path. In the case of `unlink`, the effect of removing the resolved link from the file-system graph is atomic. In fact, this is part of a common tenet followed by virtually all major file-system implementations: removing (`unlink`), adding (`open`, `creat`, `link`), moving (`rename`) and looking up individual links in a directory (path resolution steps) are implemented atomically. In other words, when accounting for concurrency, POSIX operations that resolve paths are sequences of atomic operations.

3 TaDA-Refine Specification Examples

We first introduce our TaDA-Refine specifications of file-system operations by example. In particular, we specify operations on links and I/O operations on regular files. To account for the fact that file-system operations perform sequences of atomic operations, our specifications take the form of "programs" in a simple specification programming language.

3.1 Operations on links

In §2.2, we have informally described the behaviour of the `unlink`(*path*) operation: it performs a sequence of atomic steps, first to resolve the argument *path* and then to remove the link to the file identified by the path. We define the specification of `unlink` using the following TaDA-Refine *specification program*:

$$
\begin{array}{l}
\textbf{let}\ \texttt{unlinkSpec}(\mathit{path}) \triangleq\ \textbf{let}\ p = \texttt{dirname}(\mathit{path});\\
\qquad\qquad\qquad\qquad\ \ \textbf{let}\ a = \texttt{basename}(\mathit{path});\\
\qquad\qquad\qquad\qquad\ \ \textbf{let}\ r = \texttt{resolve}(p, \iota_0);\\
\qquad\qquad\qquad\qquad\ \ \textbf{if}\ \neg\mathsf{iserr}(r)\ \textbf{then}\\
\qquad\qquad\qquad\qquad\qquad \texttt{return link\_\_delete}(r, a) \sqcup \texttt{link\_\_delete\_\_notdir}(r, a)\\
\qquad\qquad\qquad\qquad\ \ \textbf{else}\ \texttt{return}\ r\ \textbf{fi}
\end{array}
$$

The specification program initially splits the *path* argument to the path prefix p and last name a, using `dirname` and `basename` respectively. If *path* is only one name, then `dirname`

[3] Thread: "Atomicity of path resolution", Date: 21 Apr 2015.

returns `null`. The path prefix p is then resolved by calling the function $\texttt{resolve}(p, \iota_0)$. The second argument to `resolve` is the inode number of the directory from which to start the path resolution. In figure 1, this would be the directory with inode 0. To simplify the presentation, we define the specifications in this paper in terms of absolute paths, and therefore we start the resolution from the root directory, which has the known fixed inode ι_0. If the resolution fails with an error code, we return it. If the resolution succeeds, the return value is the inode of the directory containing the link we want to remove. POSIX allows implementations to return an error if the link we want to remove is a link to a directory. This freedom of choice given to implementations introduces *angelic non-determinism*. An implementation is allowed to choose which behaviour to implement. On the other hand, clients must be robust with respect to both behaviours if they wish to be portable. To account for this, we use the non-deterministic *angelic choice* operator ($\sqcup$) to join the atomic operations `link_delete` and `link_delete_notdir`.

`resolve` is defined as a function that recursively follows *path*, starting from the initial directory with inode ι:

$$
\begin{aligned}
\textbf{letrec}\ \texttt{resolve}(\mathit{path}, \iota) \triangleq\ & \textbf{if}\ \mathit{path} = \texttt{null}\ \textbf{then}\ \texttt{return}\ \iota\ \textbf{else} \\
& \quad \textbf{let}\ a = \texttt{head}(\mathit{path}); \\
& \quad \textbf{let}\ p = \texttt{tail}(\mathit{path}); \\
& \quad \textbf{let}\ r = \texttt{link\_lookup}(\iota, a); \\
& \quad \textbf{if}\ \mathsf{iserr}(r)\ \textbf{then}\ \texttt{return}\ r \textbf{else}\ \texttt{return}\ \texttt{resolve}(p, r)\ \textbf{fi} \\
& \textbf{fi}
\end{aligned}
$$

The `head` and `tail` operations return the first name and the path postfix of the path argument. Note that if *path* is a single name, then `tail` returns `null`. In each step, `resolve` calls the atomic operation $\texttt{link\_lookup}(\iota, a)$, to get the inode of the file pointed to by the link named a, if that link exists in the directory with inode ι. If the link a does not exist in the ι directory, or if the ι file is not a directory, `link_lookup` returns an error, the resolution stops and the error is immediately returned. The procedure returns the resolved inode when there is no more path to resolve, i.e. the postfix p of the *path* argument is `null`.

Any implementation of the `unlink` operation must exhibit behaviour given by the specification program `unlinkSpec`. In other words, a correct implementation must be a refinement of our specification program: $\texttt{unlink}(\mathit{path}) \sqsubseteq \texttt{unlinkSpec}(\mathit{path})$.

In §5 and the technical report [29], we formally define our specification language and an associated refinement calculus. The resulting refinement relation, $\sqsubseteq$, is contextual meaning that, in any context, `unlink` can be replaced by `unlinkSpec` to achieve the same behaviour. Therefore, to reason about a client (a particular context), we can replace an implementation with its specification.

To complete our `unlink` specification, we need to define the primitive atomic operations `link_lookup`, `link_delete` and `link_delete_notdir` that do the actual lookup and deletion of a link. Note that these are not POSIX operations, but abstract operations corresponding to the basic atomic actions that POSIX operations perform. We use *atomic specification statements*, $\forall \vec{x}.\, \langle P, Q \rangle$, to denote any program that atomically updates a state satisfying the precondition P to a state satisfying the postcondition Q, inspired by Morgan's specification statements [25]. The universal quantifier binds $\vec{x}$ to both the precondition and postcondition, and declares that the operation is atomic for all values of $\vec{x}$.

We define the atomic operations `link_lookup`, `link_delete` and `link_delete_notdir` as the atomic specification statements given in figure 3. Consider `link_delete` used in the definition of `unlinkSpec` earlier. There are three cases composed with $\sqcap$, which we will

explain shortly. Consider the first case:

$$\forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\, a \in FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0 \rangle$$

In the precondition $\mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))$, the abstract predicate $\mathsf{fs}(FS)$ states that the file-system structure is given by the file-system graph FS, and the pure predicate $\mathsf{isdir}(FS(\iota))$ states that a directory with inode ι must exist in that file-system graph. In the postcondition, we use the notation $f[x \mapsto v]$ to denote the function that maps x to v and all other elements of the domain of f to $f(x)$, and $f \setminus S$ to denote the restriction of f to $\mathrm{dom}(f) \setminus S$. The postcondition states that if, at the point the atomic update takes effect, the link named a exists in the directory with inode ι, then the link is removed and the return variable `ret` is bound to 0. As a convention, we use `ret` within a function to bind its return value.

The other two cases specify erroneous behaviour. The first error case, defined by `enoent`, specifies that if a link named a does not exist in the directory with inode ι then the return variable is bound to the POSIX error code `ENOENT`. The second error case specifies that if the inode ι does not identify a directory then the error code `ENOTDIR` is returned. Note that the error cases do not modify the file system.

The three specification cases are composed with the non-deterministic *demonic choice* operator $\sqcap$. We use demonic choice to account for the non-determinism of a specification due to scheduling behaviour. In the case of `link_delete`, which of the three possible behaviours we observe in a particular execution depends not only on the environment, but also on which of the possible interleavings the scheduler decides to execute. Thus we consider the scheduler to act as a demon and we call such specifications demonic. For example, `link_delete` handles errors by returning the error code to the client. When reasoning about a particular client, if we have information that restricts the environment, for example by requiring some path to always exist, we can elide the cases that are no longer applicable. On the other hand, an implementation of a demonic specification must implement all the cases. For example, an implementation of `link_delete` must implement all three atomic specification statements.

The definition of `link_delete_notdir` is similar, except that it succeeds only when the link being removed does not link a directory, and an extra error case is added for when it does. `link_lookup` has the same error cases as `link_delete`, but does not modify the file system, simply returning the target inode of the link named a, if it exists in the directory with inode ι.

Now, let us consider the `link`(*source*, *target*) operation. Informally, it creates a new link identified by the path *target* to the file identified by *source*, if it does not already exist. Formally, we give the following refinement specification:

```
link(source, target)
  ⊑ let p_s = dirname(source); let a = basename(source);
    let p_t = dirname(target); let b = basename(target);
    let r_s, r_t = resolve(p_s, ι_0) ∥ resolve(p_t, ι_0);
    if ¬iserr(r_s) ∧ ¬iserr(r_t) then
      return link_insert(r_s, a, r_t, b) ⊔ link_insert_notdir(r_s, a, r_t, b)
    else if iserr(r_s) ∧ ¬iserr(r_t) then return r_s
    else if ¬iserr(r_s) ∧ iserr(r_t) then return r_t
    else if iserr(r_s) ∧ iserr(r_t) then return r_s ⊔ return r_t fi
```

Note that the operation has to resolve two paths before the actual linking is attempted. POSIX does not specify the order in which multiple paths are resolved. Therefore, we compose the two `resolve` invocations in parallel, with $\|$. This allows implementations to

$$\textbf{let}\ \texttt{link\_lookup}(\iota, a) \triangleq$$
$$\forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\, a \in FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = FS(\iota)(a) \rangle$$
$$\sqcap\, \texttt{return enoent}(\iota, a) \sqcap \texttt{return enotdir}(\iota)$$

$$\textbf{let}\ \texttt{link\_delete}(\iota, a) \triangleq$$
$$\forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\, a \in FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0 \rangle$$
$$\sqcap\, \texttt{return enoent}(\iota, a) \sqcap \texttt{return enotdir}(\iota)$$

$$\textbf{let}\ \texttt{link\_delete\_notdir}(\iota, a) \triangleq$$
$$\forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\, a \in FS(\iota) \Rightarrow \mathsf{fs}(FS[\iota \mapsto FS(\iota) \setminus \{a\}]) * \texttt{ret} = 0 \rangle$$
$$\sqcap\, \texttt{return enoent}(\iota, a) \sqcap \texttt{return enotdir}(\iota) \sqcap \texttt{return err\_nodir\_links}(\iota, a)$$

$$\textbf{let}\ \texttt{link\_insert}(\iota, a, j, b) \triangleq$$
$$\forall FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) \wedge \mathsf{isdir}(FS(j)), \\ a \in FS(\iota) \wedge b \notin FS(j) \Rightarrow \mathsf{fs}(FS[j \mapsto FS(j)[b \mapsto FS(\iota)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$
$$\sqcap\, \texttt{return enoent}(\iota, a) \sqcap \texttt{return eexist}(j, b) \sqcap \texttt{return enotdir}(\iota) \sqcap \texttt{enotdir}(j)$$

$$\textbf{let}\ \texttt{link\_insert\_notdir}(\iota, a, j, b) \triangleq$$
$$\forall FS.\, \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)) \wedge \mathsf{isdir}(FS(j)), \\ \mathsf{isfile}(FS(\iota)(a)) \wedge b \notin FS(j) \Rightarrow \mathsf{fs}(FS[j \mapsto FS(j)[b \mapsto FS(\iota)(a)]]) * \texttt{ret} = 0 \end{array} \right\rangle$$
$$\sqcap\, \texttt{return enoent}(\iota, a) \sqcap \texttt{return eexist}(j, b)$$
$$\sqcap\, \texttt{return enotdir}(\iota) \sqcap \texttt{return enotdir}(j) \sqcap \texttt{return err\_nodir\_links}(\iota, a)$$

$$\textbf{let}\ \texttt{enotdir}(\iota) \triangleq \forall FS.\, \langle \mathsf{fs}(FS) \wedge \neg \mathsf{isdir}(FS(\iota)),\, \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOTDIR} \rangle$$

$$\textbf{let}\ \texttt{enoent}(\iota, a) \triangleq \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\, a \notin FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{ENOENT} \rangle$$

$$\textbf{let}\ \texttt{eexist}(\iota, a) \triangleq \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\, a \in FS(\iota) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EEXIST} \rangle$$

$$\textbf{let}\ \texttt{err\_nodir\_links}(\iota, a) \triangleq \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota)),\, \mathsf{isdir}(FS(\iota)(a)) \Rightarrow \mathsf{fs}(FS) * \texttt{ret} = \texttt{EPERM} \rangle$$

Figure 3 Specifications of atomic operations for links and associated error cases.

not only resolve the paths in any order, but also to interleave the two resolutions. The link insertion is attempted when both resolutions succeed. In that case, analogously to `unlink`, we use angelic choice between `link_insert` and `link_insert_notdir`. The former allows the link to be created for any link, even to a directory, whereas the latter considers this erroneous. The atomic specification statements for both are defined in figure 3. Error handling must be robust against errors from both resolutions. Note that if both resolutions error, either error code is returned. In general, a client is unable to determine which path resolution triggered the error.

3.2 I/O operations on regular files

POSIX defines `read` and `write` as the primitive operations for reading and writing data to regular files. The `read` operation reads a sequence of bytes from a regular file to a buffer in the heap, whereas the `write` operation writes a sequence of bytes stored in the buffer to a regular file. These operations do not identify the file they update with a path, but with a *file descriptor* which acts as a reference to a file. To create a file descriptor for a file, a client must first open the file for I/O using the operator `open`$(path, fl)$, where *path* describes the file to be opened and *fl* controls the behaviour of `open` on subsequent I/O operations such as `read` and `write`.

$$\textbf{let } \texttt{write\_off}(fd, ptr, sz) \triangleq \\ \forall FS, o \in \mathbb{N}. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, o, \mathit{fl}) \wedge \mathsf{iswrfd}(\mathit{fl}) * \mathsf{buf}\big(ptr, \overline{b}\big) \wedge \mathsf{len}\big(\overline{b}\big) = sz, \\ \mathsf{fs}\big(FS[\iota \mapsto FS(\iota)[o \leftarrow \overline{b}]]\big) * \mathsf{fd}(fd, \iota, o + sz, \mathit{fl}) * \mathsf{buf}\big(ptr, \overline{b}\big) * \texttt{ret} = sz \end{array} \right\rangle$$

$$\textbf{let } \texttt{write\_badf}(fd) \triangleq \forall o \in \mathbb{N}. \langle \mathsf{fd}(fd, \iota, o, \mathit{fl}) \wedge \texttt{O\_RDONLY} \in \mathit{fl}, \mathsf{fd}(fd, \iota, o, \mathit{fl}) * \texttt{ret} = \texttt{EBADF} \rangle$$

$$\textbf{let } \texttt{read\_norm}(fd, ptr, sz) \triangleq \\ \forall FS, o \in \mathbb{N}. \left\langle \begin{array}{l} \mathsf{fs}(FS) \wedge \mathsf{isfile}(FS(\iota)) * \mathsf{fd}(fd, \iota, o, \mathit{fl}) * \mathsf{buf}\big(ptr, \overline{b_s}\big) \wedge \mathsf{len}\big(\overline{b_s}\big) = sz, \\ \exists \overline{b_t}. \mathsf{fs}(FS) * \mathsf{fd}(fd, \iota, o + \texttt{ret}, \mathit{fl}) * \mathsf{buf}\big(ptr, \overline{b_s} \upharpoonright \overline{b_t}\big) \wedge \overline{b_t} = FS(\iota)[o, sz] * \texttt{ret} = \mathsf{len}\big(\overline{b_t}\big) \end{array} \right\rangle$$

$$\textbf{let } \texttt{read\_badf}(fd) \triangleq \forall o \in \mathbb{N}. \langle \mathsf{fd}(fd, \iota, o, \mathit{fl}) \wedge \texttt{O\_WRONLY} \in \mathit{fl}, \mathsf{fd}(fd, \iota, o, \mathit{fl}) * \texttt{ret} = \texttt{EBADF} \rangle$$

where we write $\forall \vec{x}, x \in X. \langle P, Q \rangle$ to mean $\forall \vec{x}, x. \langle P \wedge x \in X, Q \wedge x \in X \rangle$.

Figure 4 Specification of atomic read and write abstract operations.

POSIX mandates that implementations of `read` and `write` must behave atomically when used on regular files [2]. We give the following refinement specifications to `read` and `write`, defined using the demonic choice of abstract operations given in figure 4:

$$\texttt{read}(fd, ptr, sz) \sqsubseteq \texttt{return read\_norm}(fd, ptr, sz) \sqcap \texttt{read\_badf}(fd)$$

$$\texttt{write}(fd, ptr, sz) \sqsubseteq \texttt{return write\_off}(fd, ptr, sz) \sqcap \texttt{write\_badf}(fd)$$

where fd identifies the appropriate file descriptor and ptr references the buffer storing a sequence of bytes with size sz.

In figure 4, consider the atomic specification statement of `write_off`. The precondition requires fd to be a file descriptor for the file with inode ι, with current file offset o and flags fl. Note that the current file offset is bound by the universal quantifier, meaning that until `write_off` takes effect, the environment can concurrently modify it, with the proviso it remains a valid offset (a natural number). The predicate $\mathsf{iswrfd}(\mathit{fl}) \triangleq \texttt{O\_WRONLY} \in \mathit{fl} \vee \texttt{O\_RDWR} \in \mathit{fl}$ states that file descriptor must have been opened for writing. Furthermore, the predicate $\mathsf{buf}\big(prt, \overline{b}\big)$ states that ptr points to a heap-based buffer storing the byte sequence $\overline{b}$. The postcondition states that the byte sequence $\overline{b}$ stored in the ptr buffer is written to the file, offset from the start of the file (offset 0) by o. Any existing bytes from offset o onward, up to the length of $\overline{b}$, are overwritten. The current file offset associated with the file descriptor is incremented by the number of bytes written, which the operation also returns. The `write_badf` abstract operation returns the `EBADF` error, if the file descriptor has not been opened for writing, and does not modify the file.

Note that we have specified both operations as happening atomically, as is mandated by POSIX. However, not all implementations follow the POSIX specification. For example, in the ext2 file system, the reads and writes are only atomic up to page-size number of bytes. Reads and writes of larger size are split into multiple atomic steps. It is straightforward to specify this kind of implementation-specific behaviour in our specification language. In addition, reading and writing to the heap buffer is not atomic in some modern implementations. In such implementations, the I/O operations behave atomically on the file contents and the file descriptor, but non-atomically on the heap buffer. To account for such behaviour, we require specification statements that combine atomic and non-atomic effects.

In TaDA-Refine, it is possible to derive the *hybrid specification statement*:

$$\forall \vec{x}.\, \exists \vec{y}.\, \{P'\}\langle P(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}$$

We discuss this statement in detail in §5.3. Intuitively, this statement combines the atomic update from $P(\vec{x})$ to $Q(\vec{x},\vec{y})$ with a non-atomic update from P' to $Q'(\vec{x},\vec{y})$. Its purpose is twofold. First, it allows us to specify complex operations that have both atomic and non-atomic effects on different parts of the state, such as the I/O operations of some file-system implementations discussed earlier. Second, it is useful during atomicity proofs of implementations that also sequentially update privately owned resources.

4 TaDA-Refine Client Reasoning I: Lock Files

Ntzik's thesis [27] provides several examples of client reasoning based on the formal specifications of POSIX file-system operations, such as those discussed in §3. Examples of client reasoning include real-world lock files, an implementation of named pipes using regular file I/O and lock files, and a concurrent interaction between an email client and email server that is highly sensitive to the multi-atomic nature of path resolution. In this paper, we concentrate on lock files.

The lock-file module is a widely-used module for implementing locks over the file system. We describe the lock-file module and provide verified specifications for its operations to demonstrate our reasoning with TaDA-Refine. These specifications are, however, limited. They are only valid under the assumption that the file system is shared via the lock-module interface. This assumption is not valid in general as the file system is a public namespace that can be accessed and modified by concurrently executing applications. In §6, we revisit this example and introduce *context invariants* to address this issue.

The lock-file concept is simple. A lock file is a regular file, under a fixed path. If the lock file exists, the lock it represents is locked. Otherwise, the lock is unlocked. For example, `/tmp/.X0-lock` is a typical lock file in contemporary Linux systems and, in figure 1, the lock it represents is locked.

Consider the following implementation of a lock-file module with two operations, $\texttt{lock}(lf)$ and $\texttt{unlock}(lf)$, where lf is the path identifying the lock file:

$$\texttt{unlock}(lf) \triangleq \texttt{unlink}(lf)$$

$$\begin{array}{l}\textbf{letrec}\ \texttt{lock}(lf) \triangleq\\ \quad \textbf{let}\ fd = \texttt{open}(lf, \texttt{O\_CREAT}|\texttt{O\_EXCL});\\ \quad \textbf{if}\ \mathsf{iserr}(fd)\ \textbf{then}\ \texttt{lock}(lf)\\ \quad \textbf{else}\ \texttt{close}(fd)\ \textbf{fi}\end{array}$$

The `lock` operation attempts to create the lock file at path lf by invoking `open`. This operation is used to open files for input/output (I/O) and to create new files. The second argument to `open` is a composition of the flags `O_CREAT` and `O_EXCL`, which causes `open` to create a file at the given path if one does not already exist; otherwise, an error is returned. Thus, if `open` returns an error we try again, with a recursive call to `lock`. If it succeeds, we invoke `close` to close the file descriptor returned by `open`. Note that lock files, essentially, follow the same implementation pattern as spin locks.

The `open` operation exhibits different behaviour depending on the flags used as the second argument and we give its full specification in the technical report [29]. For presentation simplicity we define the specification only in terms of the flags used in `lock`:

$$
\begin{array}{l}
\texttt{open}(\mathit{path}, \texttt{O\_CREAT}|\texttt{O\_EXCL}) \\
\quad \sqsubseteq \textbf{let } p = \texttt{dirname}(\mathit{path}); \\
\qquad \textbf{let } a = \texttt{basename}(\mathit{path}); \\
\qquad \textbf{let } r = \texttt{resolve}(p, \iota_0); \\
\qquad \textbf{if } \neg\mathsf{iserr}(r) \textbf{ then} \\
\qquad\quad \texttt{return link\_new\_file}(r, a) \\
\qquad\qquad\quad \sqcap\ \texttt{eexist}(r, a) \sqcap \texttt{enotdir}(r) \\
\qquad \textbf{else return } r \textbf{ fi}
\end{array}
$$

where $\texttt{link\_new\_file}(\iota, a)$ is defined as follows:

$$\forall FS.\, \left\langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(\iota))\,,\, a \notin FS(\iota) \Rightarrow \exists \iota'.\, \mathsf{fs}\big(FS[\iota \mapsto FS(\iota)[a \mapsto \iota']][\iota' \mapsto \epsilon]\big) * \mathsf{fd}\big(\texttt{ret}, \iota', 0\big) \right\rangle$$

This specifies the creation of a new empty, regular file at inode ι', and the addition of a link named a to the new file within the directory with inode ι, if the link does not already exist. The operation allocates and returns a new file descriptor. The predicate $\mathsf{fd}(\texttt{ret}, \iota', 0)$ asserts that the return value is a file descriptor for the file with inode ι', and the offset from which reads and writes to the file occur, via this file descriptor, is set to 0.

By contextual refinement, we can replace the `open` and `unlink` with their specifications and thus derive a specification for `lock` and `unlock` respectively. However, this would not be useful for reasoning about locks since it fails to capture the abstract lock behaviour. Instead, we want aim to establish a general abstract specification, such as the following:

$$\texttt{lock}(\mathit{lf}) \sqsubseteq \forall v \in \{0, 1\}\,.\, \langle \mathsf{Lock}(s, \mathit{lf}, v)\,, \mathsf{Lock}(s, \mathit{lf}, 1) * v = 0 \rangle$$

$$\texttt{unlock}(\mathit{lf}) \sqsubseteq \langle \mathsf{Lock}(s, \mathit{lf}, 1)\,, \mathsf{Lock}(s, \mathit{lf}, 0) \rangle$$

The abstract predicate $\mathsf{Lock}(s, \mathit{lf}, v)$ states the existence of a lock represented by a lock file at path lf, with state v, the value of which is either 0, if the lock is unlocked, or 1 if the lock is locked. The parameter, $s \in \mathbb{T}_1$, is a variable ranging over an abstract type. It serves to capture invariant information, specific to the implementation of the Lock predicate and is opaque to the client. The specification states that we can abstract each lock-file operation to a single atomic step that updates the state of the lock. In particular, the `lock` specification states that the environment can arbitrarily lock and unlock the lock, but the lock is atomically locked only when it is previously unlocked; the operation blocks while the lock is locked. The `unlock` specification states that the lock can only be atomically unlocked when the lock is locked.

The environmental interference allowed by the specification of the lock operation is due to the stuttering refinement law:

ASTUTTER
$\forall \vec{x}.\, \langle P, P \rangle; \forall \vec{x}.\, \langle P, Q \rangle \sqsubseteq \forall \vec{x}.\, \langle P, Q \rangle$

The environment can interleave between the two sequentially composed atomic specifications, allowing it to change the state of the lock over time, as long as the state remains in the set $\{0, 1\}$, otherwise, the precondition of the lock specification would be violated when it executes, leading to undefined behaviour.

In order to justify the two refinements of the module's specification, we must refine the abstract Lock predicates to the shared lock-file path lf in the file system according to the state of the lock. Additionally, we must enforce that the updates to the abstract state

$$
\begin{array}{l}
\texttt{unlock}(lf) \equiv \texttt{unlink}(lf) \sqsubseteq \\
\quad \textsc{HStutter}\left(\begin{array}{l}
\quad \textbf{let } p = \texttt{dirname}(path); \textbf{let } a = \texttt{basename}(path); \textbf{let } r = \texttt{resolve}(p, \iota_0) \sqsubseteq \\
\quad \forall FS \in \mathcal{LF}(path).\left\langle \mathsf{fs}(FS)\,, \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \right\rangle \\
\quad \textsc{HStrengthen} \\
\quad \sqsubseteq \forall FS \in \mathcal{LF}(path).\{\mathsf{true}\}\langle \mathsf{fs}(FS)\,, \mathsf{fs}(FS)\rangle \left\{ p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \right\}; \\
\quad \textbf{if } \neg\mathsf{iserr}(r) \textbf{ then} \\
\quad\quad \texttt{return link\_delete}(r, a) \\
\quad\quad\quad\quad \sqcup \texttt{link\_delete\_notdir}(r, a) \\
\quad\quad \equiv \texttt{return link\_delete}(r, a) \\
\quad\quad\quad\quad \sqcup (\texttt{link\_delete}(r, a) \sqcap \texttt{err\_nodir\_links}(\iota, a)) \\
\quad\quad \textsc{Absorb} \\
\quad\quad \equiv \texttt{return link\_delete}(r, a) \\
\quad\quad \textsc{DChoiceIntro} \\
\quad\quad \sqsubseteq \forall FS.\langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r))\,, a \in FS(r) \Rightarrow \mathsf{fs}(FS[r \mapsto FS(r) \setminus \{a\}]) * \texttt{ret} = 0\rangle \\
\quad\quad \textsc{ACons, AFrame} \\
\quad\quad \sqsubseteq \forall FS \in \mathcal{LK}(path).\left\langle \begin{array}{c} \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \wedge \neg\mathsf{iserr}(r)\,, \\ \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf) \end{array} \right\rangle \\
\quad\quad \textbf{else} \\
\quad\quad\quad \texttt{return } r \\
\quad\quad\quad \sqsubseteq \langle \mathsf{true},\ \ \texttt{ret} = r\rangle \\
\quad\quad\quad \textsc{ACons} \\
\quad\quad\quad \sqsubseteq \langle \mathsf{true},\ \ \mathsf{true}\rangle \\
\quad\quad\quad \textsc{AFrame} \\
\quad\quad\quad \sqsubseteq \forall FS \in \mathcal{LK}(path).\left\langle \begin{array}{c} \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \wedge \mathsf{iserr}(r)\,, \\ \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \wedge \mathsf{iserr}(r) \end{array} \right\rangle \\
\quad\quad\quad \textsc{ACons} \\
\quad\quad\quad \sqsubseteq \forall FS \in \mathcal{LK}(path).\left\langle \begin{array}{c} \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \wedge \mathsf{iserr}(r)\,, \\ \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf) \end{array} \right\rangle \\
\quad\quad \textbf{fi} \\
\quad\quad \sqsubseteq \forall FS \in \mathcal{LK}(path).\left\langle \begin{array}{c} \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \wedge \mathsf{iserr}(r)\,, \\ \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf) \end{array} \right\rangle \\
\quad\quad \textsc{HStrengthen} \\
\quad\quad \sqsubseteq \forall FS \in \mathcal{LK}(path).\langle \mathsf{fs}(FS)\,, \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf)\rangle \begin{array}{c} \left\{ p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \right\} \\ \{\mathsf{true}\} \end{array}
\end{array}\right) \\
\quad \sqsubseteq \forall FS \in \mathcal{LK}(lf).\{\mathsf{true}\}\langle \mathsf{fs}(FS)\,, \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf)\rangle\{\mathsf{true}\} \\
\quad \equiv \forall FS \in \mathcal{LK}(lf).\langle \mathsf{fs}(FS)\,, \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf)\rangle \\
\quad \textsc{ACons, AUseAtomic} \\
\quad \sqsubseteq \langle \mathbf{Lock}_\alpha(lf, 1) * [\mathsf{G}]_\alpha\,, \mathbf{Lock}_\alpha(lf, 0) * [\mathsf{G}]_\alpha\rangle \\
\quad \sqsubseteq \langle \mathsf{Lock}(s, lf, 1)\,, \mathsf{Lock}(s, lf, 0)\rangle
\end{array}
$$

Figure 5 `unlock()` specification proof sketch.

of the lock by multiple threads follow a protocol: a thread can lock the lock only if it is unlocked and, similarly, can unlock the lock only if it is locked. We use *shared regions* to describe shared resources that are updated according to a particular protocol, using a technique first developed with RGSep [37] and now used by many of the concurrent separation logics [13, 11, 21, 34, 9, 23]. A shared region is an abstract object that encapsulates some underlying (concrete) state that is shared between multiple threads, with the proviso that it is only accessed atomically. We use $\mathbf{t}_\alpha(\vec{y}, x)$, to denote a shared region with identifier α from the set RID, of type $\mathbf{t}$, with parameters $\vec{y}$ and abstract state x. For our current example, we introduce a region type **Lock** where regions of this type are parameterised by the lock-file path and the abstract state of the region corresponds to the state of the lock.

The shared region enforces a protocol on updates to the abstract state via a *labelled transition system*. The transitions between abstract region states are labelled by *guards*. Guards are abstract resources that can be taken from any user-defined separation algebra [6]. For our current example we only need a simple separation algebra with a single, indivisible guard G and the empty guard $\mathbf{0}$. The partial, associative and commutative composition operator between these guards is defined by the axioms: $x \bullet \mathbf{0} = x = \mathbf{0} \bullet x$ for all $x \in \{\mathbf{0}, \mathsf{G}\}$. We define the labelled transition system for the **Lock** as follows:

$$\mathsf{G} : 0 \rightsquigarrow 1 \qquad\qquad \mathsf{G} : 1 \rightsquigarrow 0$$

A thread can only perform a transition between abstract region states if it owns the guard resource associated with the transition; in our current example that is the guard G.

Having defined the **Lock**, its transition system and guards we can now define the interpretation of the abstract Lock predicate and abstract type $\mathbb{T}_1$ in terms of the region as follows:

$$\mathbb{T}_1 \triangleq \text{RID} \qquad \mathsf{Lock}(\alpha, lf, 0) \triangleq \mathbf{Lock}_\alpha(lf, 0) * [\mathsf{G}]_\alpha \qquad \mathsf{Lock}(\alpha, lf, 1) \triangleq \mathbf{Lock}_\alpha(lf, 1) * [\mathsf{G}]_\alpha$$

With the above interpretation we can refine the atomic specification statement of `lock` as follows:

$$\begin{aligned}&\forall v \in \{0,1\} \,.\, \langle \mathbf{Lock}_\alpha(lf, v) * [\mathsf{G}]_\alpha \,, \mathbf{Lock}_\alpha(lf, 1) * [\mathsf{G}]_\alpha * v = 0 \rangle \\ &\quad \sqsubseteq \forall v \in \{0,1\} \,.\, \langle \mathsf{Lock}(s, lf, v) \,, \mathsf{Lock}(s, lf, 1) * v = 0 \rangle\end{aligned}$$

and similarly for the atomic specification statement of `unlock`:

$$\langle \mathbf{Lock}_\alpha(lf, 1) * [\mathsf{G}]_\alpha \,, \mathbf{Lock}_\alpha(lf, 0) * [\mathsf{G}]_\alpha \rangle \sqsubseteq \langle \mathsf{Lock}(s, lf, 1) \,, \mathsf{Lock}(s, lf, 0) \rangle$$

To refine the two updates further, we can apply the AUSEATOMIC refinement law which allows us to refine an update to the abstract region state to an update on the (concrete) state the region encapsulates. A slightly simplified version of this refinement law is as follows:

$$\frac{\forall x. \, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(\mathsf{G})^*}{\forall x. \left\langle I(\mathbf{t}_\alpha(y, x)) * P(x) * [\mathsf{G}]_\alpha \,, I(\mathbf{t}_\alpha(y, f(x))) * Q(x) \right\rangle \sqsubseteq \forall x. \left\langle \mathbf{t}_\alpha(y, x) * P(x) * [\mathsf{G}]_\alpha \,, \mathbf{t}_\alpha(y, f(x)) * Q(x) \right\rangle}$$

The premise of the law requires that an update from the abstract region state x to $f(x)$ must be in the reflexive, transitive closure of transitions guarded by G denoted by $\mathcal{T}_{\mathbf{t}}(\mathsf{G})^*$. In the conclusion, an atomic update, satisfying the premise, on the interpretation of a region refines the same atomic update on the region itself.

However, in order to refine the `lock` atomic specification statement further according to AUSEATOMIC, we must define the interpretation of the **Lock** region states 0 and 1 to the underlying file-system states.

In the state 0 the lock file does not exist, whereas in the state 1 it does. In both states however, the path to the directory containing the lock file must exist. To assert the aforementioned statements we define the predicate $p \overset{FS}{\mapsto} \iota$ to assert that the path p resolves to the file with inode ι in the file-system graph FS as follows:

$$\varnothing_p \overset{FS}{\mapsto} \iota \triangleq \iota \in \mathrm{dom}(FS) \qquad p \overset{FS}{\mapsto} \iota \triangleq (p, \iota_0) \overset{FS}{\mapsto} \iota \qquad (a, \iota) \overset{FS}{\mapsto} \iota' \triangleq FS(\iota)(a) = \iota'$$

$$(a/p, \iota) \overset{FS}{\mapsto} \iota' \triangleq \exists \iota''.\, FS(\iota)(a) = \iota'' \wedge (p, \iota'') \overset{FS}{\mapsto} \iota'$$

With this predicate we can now define the sets of file systems that correspond to the lock being unlocked and locked respectively as follows:

$$\begin{aligned} \mathcal{ULK}(p/a) &\triangleq \left\{ FS \;\middle|\; \exists \iota.\, p \overset{FS}{\mapsto} \iota \wedge \mathsf{isdir}(FS(\iota)) \wedge a \notin FS(\iota) \right\} \\ \mathcal{LK}(p/a) &\triangleq \left\{ FS \;\middle|\; \exists \iota.\, p \overset{FS}{\mapsto} \iota \wedge \mathsf{isdir}(FS(\iota)) \wedge a \in FS(\iota) \right\} \end{aligned}$$

Additionally, we define the union of the above sets: $\mathcal{LF}(p/a) \triangleq \mathcal{ULK}(p/a) \cup \mathcal{LK}(p/a)$, and the following predicates that describe the updates from FS to FS' that create and remove the lock file in its directory respectively:

$$\begin{aligned} &\mathsf{lk}(FS, FS', p/a) \triangleq \exists \iota, \iota'.\, p \overset{FS}{\mapsto} \iota \wedge FS' = FS[\iota \mapsto FS(\iota)[a \mapsto \iota']][\iota' \mapsto \epsilon] \\ &\mathsf{ulk}(FS, FS', p/a) \triangleq \exists \iota.\, p \overset{FS}{\mapsto} \iota \wedge FS' = FS[\iota \mapsto FS(\iota) \setminus \{a\}] \end{aligned}$$

Now we define the interpretation to the **Lock** region as follows:

$$I(\mathbf{Lock}_\alpha(lf, 0)) \triangleq \exists FS \in \mathcal{ULK}(lf).\, \mathsf{fs}(FS) \qquad I(\mathbf{Lock}_\alpha(lf, 1)) \triangleq \exists FS \in \mathcal{LK}(lf).\, \mathsf{fs}(FS)$$

We can now proceed with the refinement by applying the AUseAtomic law. For simplicity, let us consider the refinement of `unlock`:

$$\begin{aligned} &\langle \exists FS \in \mathcal{LK}(lf).\, \mathsf{fs}(FS) * [\mathsf{G}]_\alpha \,, \exists FS \in \mathcal{ULK}(lf).\, \mathsf{fs}(FS) * [\mathsf{G}]_\alpha \rangle \\ &\qquad \sqsubseteq \langle \mathbf{Lock}_\alpha(lf, 1) * [\mathsf{G}]_\alpha \,, \mathbf{Lock}_\alpha(lf, 0) * [\mathsf{G}]_\alpha \rangle \end{aligned}$$

Now we can work to refine this atomic update on the file-system state to the specification program of `unlink` that we defined in section 3. A proof sketch can be seen in figure 5. First we can frame-off the guard resource as it is no longer required by using the AFrame refinement law which is directly analogous to the frame rule of separation logics:

$$\begin{aligned} &\langle \exists FS \in \mathcal{LK}(lf).\, \mathsf{fs}(FS) \,, \exists FS \in \mathcal{ULK}(lf).\, \mathsf{fs}(FS) \rangle \\ &\quad \sqsubseteq \langle \exists FS \in \mathcal{LK}(lf).\, \mathsf{fs}(FS) * [\mathsf{G}]_\alpha \,, \exists FS \in \mathcal{ULK}(lf).\, \mathsf{fs}(FS) * [\mathsf{G}]_\alpha \rangle \end{aligned}$$

Next, we strengthen the post-condition and eliminate the existential quantification over file-system graphs:

$$\begin{aligned} &\forall FS \in \mathcal{LK}(lf).\, \langle \mathsf{fs}(FS) \,, \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf) \rangle \\ &\quad \sqsubseteq \langle \exists FS \in \mathcal{LK}(lf).\, \mathsf{fs}(FS) \,, \exists FS \in \mathcal{LK}(lf), FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf) \rangle \\ &\qquad \sqsubseteq \langle \exists FS \in \mathcal{LK}(lf).\, \mathsf{fs}(FS) \,, \exists FS \in \mathcal{ULK}(lf).\, \mathsf{fs}(FS) \rangle \end{aligned}$$

Our next challenge is that `unlinkSpec` consists of several steps in sequence. Earlier we introduced the AStutter refinement law, however, `unlinkSpec` requires the manipulation of hidden, intermediary, non-atomic state about the existence of the lock file being manipulated. To deal with this, we introduce a generalisation of the AStutter rule, HStutter,

which chains together the non-atomic preconditions and postconditions as in the sequential composition of Hoare triples:

$$\begin{aligned}&\forall \vec{x}.\, \{P'\}\langle P(\vec{x}), P(\vec{x})\rangle\{P''\}; \forall \vec{x}.\, \exists \vec{y}.\, \{P''\}\langle P(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}\\ &\quad \sqsubseteq \forall \vec{x}.\, \exists \vec{y}.\, \{P'\}\langle P(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}\end{aligned}$$

We will also need the HSTRENGTHEN refinement law which allows us to refine a non-atomic update to a part of the state to an atomic update:

$$\begin{aligned}&\forall \vec{x}.\, \exists \vec{y}.\, \{P'\}\langle P' * P(\vec{x}), Q(\vec{x},\vec{y}) * Q'(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}\\ &\quad \sqsubseteq \forall \vec{x}.\, \exists \vec{y}.\, \{P' * P'\}\langle P(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y}) * Q'(\vec{x},\vec{y})\}\end{aligned}$$

The HSTRENGTHEN refinement law will be used to move stable information about the file system state, i.e. assertions that cannot be invalidated by the environment, into the non-atomic assertions so that they can be used to reason about the behaviour of subsequent steps of the program.

To proceed with our refinement to `unlinkSpec`, we first use the fact that $\forall \vec{x}.\, \langle P, Q\rangle \equiv \forall \vec{x}.\, \{\mathsf{true}\}\langle P, Q\rangle\{\mathsf{true}\}$ and the HSTUTTER refinement law to further refine the current specification:

$$\begin{aligned}&\forall FS \in \mathcal{LK}(lf).\, \{\mathsf{true}\}\langle \mathsf{fs}(FS)\,, \mathsf{fs}(FS)\rangle\left\{p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r)\right\};\\ &\forall FS \in \mathcal{LK}(lf).\, \left\{\begin{array}{c} p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge \\ a \in FS(r)\end{array}\right\}\langle \mathsf{fs}(FS)\,, \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf)\rangle\{\mathsf{true}\}\\ &\quad \sqsubseteq \forall FS \in \mathcal{LK}(lf).\, \langle \mathsf{fs}(FS)\,, \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf)\rangle\end{aligned}$$

Where variables p, a and r correspond to the path prefix, the last name in the path and the inode corresponding to the directory p respectively. The first of these two specifications is then refined using the HSTRENGTHEN refinement law to move all of the content of the non-atomic postcondition into the atomic postcondition. This refinement is valid as the environment is restricted to maintaining the existence of the path lf, the recursive calls to resolve can be thought of as a single atomic step since the result of the individual atomic resolution steps will not be invalidated by the environment.

This specification is further refined to the first three lines of code of `unlinkSpec`. In this specification, we substitute lf for $path$ due to the function call. For the derivation of the recursive `resolve` function we rely on standard fixpoint induction law:

$$\textsc{Ind}\ \frac{\lambda x.\, \phi\,[\psi/A] \sqsubseteq \lambda x.\, \psi}{\mu A.\, \lambda x.\, \phi \sqsubseteq \lambda x.\, \psi}$$

We omit the full derivation for brevity.

$$\begin{aligned}&\textbf{let } p = \texttt{dirname}(path); \textbf{let } a = \texttt{basename}(path);\\ &\textbf{let } r = \texttt{resolve}(p, \iota_0)\\ &\quad \sqsubseteq \forall FS \in \mathcal{LK}(path).\, \left\langle \mathsf{fs}(FS)\,, \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r)\right\rangle\end{aligned}$$

Next, we must verify that:

$$\begin{array}{l}\textbf{if } \neg\mathsf{iserr}(r) \textbf{ then}\\ \quad \texttt{return link\_delete}(r, a)\\ \qquad\quad \sqcup\, \texttt{link\_delete\_notdir}(r, a)\\ \textbf{else return } r \textbf{ fi}\end{array} \quad \sqsubseteq \quad \begin{array}{c}\forall FS \in \mathcal{LK}(path).\\ \left\{p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r)\right\}\\ \langle \mathsf{fs}(FS)\,, \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf)\rangle\\ \{\mathsf{true}\}\end{array}$$

When verifying that an if statement refines an atomic specification, it suffices to verify that both branches of the if statement satisfy the atomic specification given that the precondition is extended with the if statement's condition and its negation for each branch respectively, as is done in figure 5.

First however, before applying this rule, figure 5. applies the HStrengthen refinement law to move the stable information about the file system in the non-atomic precondition back into the atomic precondition.

We can now check that the false branch of the if refines:

$$\forall FS \in \mathcal{LK}(path).\left\langle \begin{array}{c} \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \wedge \mathsf{iserr}(r)\,, \\ \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf) \end{array} \right\rangle$$

As $\mathsf{iserr}(r) \Rightarrow r \in \textsc{Errs}$ and $p \overset{FS}{\mapsto} r \Rightarrow r \in \textsc{Inodes}$, and since $\textsc{Inodes} \cap \textsc{Errs} = \emptyset$, $p \overset{FS}{\mapsto} r \wedge \mathsf{iserr}(r) \Rightarrow \mathsf{false}$ holds. Since $\mathsf{false} \Rightarrow \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf)$, by using the atomic consequence rule, figure 5 reaches the goal for this branch of the if statement.

To finish showing that `unlinkSpec` refines the goal specification, it remains to check the `true` branch of the if statement. First we apply the consequence and frame rule, to frame away $p \overset{FS}{\mapsto} r \wedge a \in FS(r) \wedge FS \in \mathcal{LK}(path)$, on the specification for the true branch of the if statement:

$$\begin{array}{c} \forall FS.\, \langle \mathsf{fs}(FS) \wedge \mathsf{isdir}(FS(r))\,, a \in FS(r) \Rightarrow \mathsf{fs}(FS[r \mapsto FS(r) \setminus \{a\}]) * \texttt{ret} = 0 \rangle \sqsubseteq \\ \forall FS \in \mathcal{LK}(path).\left\langle \begin{array}{c} \mathsf{fs}(FS) \wedge p \overset{FS}{\mapsto} r \wedge \mathsf{isdir}(FS(r)) \wedge a \in FS(r) \wedge \neg\mathsf{iserr}(r)\,, \\ \exists FS'.\, \mathsf{fs}(FS') \wedge \mathsf{ulk}(FS, FS', lf) \end{array} \right\rangle \end{array}$$

Using the specification of `link_delete`() and the DChoiceIntro refinement law, $\phi \sqcap \psi \sqsubseteq \phi$, we can show that the current specification is refined by `link_delete`().

Finally, the Absorb law, $\phi \sqcup (\phi \sqcap \psi) \equiv \phi \equiv \phi \sqcap (\phi \sqcup \psi)$, asserts that a specification made up of an angelic choice between a specification, ϕ, and a second, strictly less permissive one, $\phi \sqcap \psi$, is equivalent to ϕ, as in both cases, it must be satisfied. This law can be used in conjunction with the definition of `link_delete_notdir`(r, a) to show that `return link_delete`$(r, a) \sqcup$ `link_delete_notdir`$(r, a) \sqsubseteq$ `return link_delete`(r, a) [4], which completes the proof.

This proof encapsulates the entirety of the file system within the **Lock** shared region, which effectively prohibits sharing of the file system via means other than the lock-file module's interface. This assumption is not valid in general as the file system is a public namespace that can be accessed and modified by concurrently executing applications. In section 6, we will extend this reasoning to be able to express the necessary restrictions on the context in which the program executes.

5 TADA-Refine Specification Language and Refinement Calculus

We describe TADA-Refine, our concurrent specification language and associated refinement calculus, giving just a sketch here and providing full details in the technical report [29]. We discovered that, due to the stuttering and mumbling laws, we have simpler laws and soundness proof compared with those of TaDA.

[4] `link_delete_notdir`(r, a) can be rewritten as `link_delete`$(r, a) \sqcap$ `return err_nodir_links`(ι, a).

$$
\begin{array}{lll}
\text{Specifications} & \phi, \psi ::= & \phi; \psi \mid \phi \parallel \psi \mid \textbf{let } f = F \textbf{ in } \phi \mid Fe \\
& & \mid \phi \sqcup \psi \mid \phi \sqcap \psi \mid \exists x.\, \phi \mid \forall \vec{x}.\, \langle P, Q \rangle_k \\
\text{Functions} & F ::= & f \mid A \mid \mu A.\, \lambda x.\, \phi \mid \lambda x.\, \phi \\
\text{Expressions} & e ::= & \ldots \\
\text{Assertions} & P, Q, R ::= & \mathsf{false} \mid \mathsf{true} \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \exists x.\, P \mid \forall x.\, P \mid P \Rightarrow Q \\
& & \mid \mathbf{t}^k_\alpha(\vec{y}, x) \mid [\mathsf{G}]_\alpha \mid \ldots
\end{array}
$$

where x denotes a variable, $\vec{x}$ a sequence of variables, A a recursive variable and f a function.

Figure 6 The specification language of TaDA-Refine: specifications and assertions.

5.1 The Specification Language

The syntax of the specification language for TADA-Refine is given in figure 6, using the grammars of the specifications and assertions. Following Turon and Wand [36], we do not distinguish between specifications and concrete programs, taking the view that programs are the most concrete of specifications. The specifications are built from traditional programming constructs: sequential composition $\phi; \psi$, parallel composition $\phi \parallel \psi$, recursion and first-order procedures. We use additional constructs to express specification non-determinism: angelic choice, $\phi \sqcup \psi$, behaves either as ϕ or as ψ; and demonic choice, $\phi \sqcap \psi$, behaves as ϕ and ψ. Angelic and demonic non-determinism are motivated in the `unlink` specification of §3.1. Existential quantification, $\exists x.\, \phi$, behaves as ϕ for some choice of x.

The atomic specification statement, $\forall \vec{x}.\, \langle P, Q \rangle_k$, is motivated in §3.1. It specifies any operation that atomically updates a state satisfying the precondition P to a state satisfying the postcondition Q. The universal quantifier binds $\vec{x}$ to both the precondition and postcondition, declaring that the update is atomic for all possible values of $\vec{x}$. The statement includes a *region level* subscript, k, explained below. The assertions, P and Q, are based on the intuitionistic assertions of TaDA [9]. They are built from the standard connectives from first-order logic and intuitionistic separation logic [31], together with shared region assertions and guard assertions of TaDA and first-order recursive predicates. In addition, we will use a collection of abstract and pure predicates signified by the ... and introduced by example in the other sections. We implicitly require that the pre- and postconditions are *stable*: they must account for interference from other threads. The region assertion, $\mathbf{t}^k_\alpha(\vec{y}, x)$, asserts the existence of a region with superscript k with identifier α of type $\mathbf{t}$ and parameters $\vec{y}$ and abstract state x. The guard assertion, $[\mathsf{G}]_\alpha$, asserts the ownership of guard G for region α. As described in §4, associated with a region type is a guard separation algebra, a labelled transition system and an interpretation function.

A region assertion also has a region-level superscript, k, analogous to the region-level subscript of a specification statement. The region level is an integer, signifying that only regions below level k may be replaced by their interpretation (opened) in the refinement of a specification statement. Their purpose is to ensure that we cannot open the same region twice during a refinement derivation, as this could unsoundly duplicate resources encapsulated by the region.

We keep the specification language minimal. For simplicity and to keep specifications declarative, variables are immutable. Additional programming constructs used in the specifications given throughout this paper can be easily encoded. For example, the specification

$$\textsc{AEElim}\quad \forall \vec{y}, x.\, \langle P, Q\rangle_k \sqsubseteq \forall \vec{y}.\, \langle \exists x.\, P, \exists x.\, Q\rangle_k$$

$$\textsc{ACons}\quad \frac{\forall \vec{x}.\, P \preccurlyeq P' \qquad \forall \vec{x}.\, Q' \preccurlyeq Q}{\forall \vec{x}.\, \left\langle P', Q'\right\rangle_k \sqsubseteq \forall \vec{x}.\, \langle P, Q\rangle_k}$$

$$\textsc{AFrame}\quad \forall \vec{x}.\, \langle P, Q\rangle_k \sqsubseteq \forall \vec{x}.\, \langle P * R, Q * R\rangle_k$$

$$\textsc{AUseAtomic}\quad \frac{\forall x.\, (x, f(x)) \in \mathcal{T}_{\mathbf{t}}(G)^*}{\begin{array}{l}\forall x, \vec{x}.\, \left\langle I(\mathbf{t}^k_\alpha(\vec{e}, x)) * P(\vec{x}) * [\mathsf{G}]_\alpha, I(\mathbf{t}^k_\alpha(\vec{e}, f(x))) * Q(\vec{x})\right\rangle_k \\ \sqsubseteq \forall x, \vec{x}.\, \left\langle \mathbf{t}^k_\alpha(\vec{e}, x) * P(\vec{x}) * [\mathsf{G}]_\alpha, \mathbf{t}^k_\alpha(\vec{e}, f(x)) * Q(\vec{x})\right\rangle_{k+1}\end{array}}$$

$$\textsc{ARLevel}\quad \frac{k_1 \leq k_2}{\forall \vec{x}.\, \langle P, Q\rangle_{k_1} \sqsubseteq \forall \vec{x}.\, \langle P, Q\rangle_{k_2}}$$

$$\textsc{AStutter}\quad \forall \vec{x}.\, \langle P, P\rangle_k; \forall \vec{x}.\, \langle P, Q\rangle_k \sqsubseteq \forall \vec{x}.\, \langle P, Q\rangle_k$$

$$\textsc{AMumble}\quad \forall \vec{x}.\, \langle P, Q\rangle_k \sqsubseteq \forall \vec{x}.\, \left\langle P, P'\right\rangle_k; \forall \vec{x}.\, \left\langle P', Q\right\rangle_k$$

Figure 7 Some refinement laws for the atomic specifications.

let $x = F(e)$ **in** ϕ can be written as $\exists x.\, F(e, x); \phi$ which binds the return variable `ret` to x. Control flow can be encoded with angelic choice. For example, **if** P **then** ϕ **else** ψ **fi** can be written as $(\langle \mathsf{true}, P\rangle; \phi) \sqcup (\langle \mathsf{true}, \neg P\rangle; \psi)$. Encodings for the other syntactic features used in our specifications are given in the technical report [29]. In §5.3, we discuss the encoding of hybrid specification statements.

In the technical report [29], we also define the operational semantics for our specification language as a transition relation, $\phi, h \Downarrow o$, from specifications ϕ and concrete heaps h to outcomes $o ::= h \mid \lightning$, where $\lightning$ denotes a fault.

5.2 The Refinement Calculus

The refinement calculus for TaDA-Refine comprises standard laws of refinement, refinement laws for atomic specification statements adapted from [36], and laws associated with TaDA's program-logic rules. Unlike TaDA, where stuttering and mumbling is hidden in its underlying semantics, the stuttering and mumbling laws are first-class citizens in TaDA-Refine. This enables us to simplify significantly the laws associated with the TaDA rules and the proof of adequacy (Theorem 1). The full calculus is given in the technical report [29].

In figure 7, we provide a selection of refinement laws for atomic specification statements. The AEElim is analogous to the existential elimination rule of Hoare logics. The ACons law is a semantic consequence law, originating from the views framework [10]. It generalises the standard logical consequence relation, using a view-shift relation $\preccurlyeq$ adapted from TaDA instead of the usual logical implication. The AFrame law is a frame law for atomic statements, originating from Turon and Wand's work [36]. The AUseAtomic and ARLevel laws are taken from analogous rules of TaDA. AUseAtomic allows us to refine an atomic update on the state of a shared region into an atomic update on the region's interpretation given by the interpretation function I. Note that by doing so the region level associated with the specification statement is decremented. This prevents the same region to be re-opened again in subsequent refinements. Unlike TaDA, we do not require other laws for shared regions due to the presence of the stuttering and mumbling laws as first-class citizens. Stuttering and mumbling originate from the work on trace semantics by Brookes [5]. AStutter allows us to refine a single atomic update to a sequence of atomic steps, as long as the state before the update is maintained. AMumble allows us to refine a sequence of atomic steps into a

single atomic update, as long as the environment does not invalidate the intermediate states. Note that, by combining the two laws, we can derive a stuttering equivalence.

The laws of the refinement calculus are sound with respect to a denotational refinement relation which is adequate with respect to an operational refinement relation. We define the denotational refinement relation using a denotational semantics of specifications, denoted $[\![\phi]\!]^\rho$ with variable context ρ, which is defined as sets of observed traces. This gives us a denotational refinement relation defined as the subset relation between traces. Our adequacy proof follows the methodology of Turon and Wand [36], significantly adapted to handle TaDA assertions [9]; the details of adequacy and soundness are in the technical report [29].

▶ **Definition 1** (Denotational Refinement). $\phi \sqsubseteq \psi \iff [\![\phi]\!]^\rho \subseteq [\![\psi]\!]^\rho$.

We define the operational refinement relation as a partial-correctness contextual refinement, using our operational semantics given in the technical report [29]:

$$\phi \sqsubseteq_{\text{op}} \psi \iff \forall C, h. \begin{cases} C[\phi], h \Downarrow \lightning \Rightarrow C[\psi], h \Downarrow \lightning \\ C[\phi], h \Downarrow h' \Rightarrow C[\psi], h \Downarrow h' \vee C[\psi], h \Downarrow \lightning \end{cases}$$

where C is a specification context. If the specification ψ faults, then ϕ is allowed to do anything since a fault is treated as *unspecified* behaviour.

▶ **Theorem 2** (Adequacy). *If $\phi \sqsubseteq \psi$, then $\phi \sqsubseteq_{\text{op}} \psi$*

5.3 Derived Hybrid Specification Statement

The derived hybrid atomic statement, $\forall \vec{x}. \exists \vec{y}. \{P'\}\langle P(\vec{x}), Q(\vec{x}, \vec{y})\rangle\{Q'(\vec{x}, \vec{y})\}_k$, extends the atomic specification statement with non-atomic components: the atomic specification statement $P(\vec{x})$ is atomically updated to $Q(\vec{x}, \vec{y})$ for all values of $\vec{x}$; at the same time, the non-atomic precondition P' is updated to $Q'(\vec{x}, \vec{y})$ without any atomicity guarantees. The quantification extends to the end of the statement and is a little subtle. The non-atomic precondition is independent of the universally quantified $\vec{x}$ because the environment may be modifying it before the atomic update takes effect. The two postconditions are linked by the existentially quantified $\vec{y}$, non-deterministically chosen by the implementation at the point the atomic update takes effect.

The hybrid specification statement is a derived construct, defined as a specification program comprising a sequence of atomic specification statements.

▶ **Definition 3** (Hybrid Specification Statement). The *hybrid specification statement* is defined by:

$$\begin{array}{l} \forall \vec{x}. \exists \vec{y}. \{P'\}\langle P(\vec{x}), Q(\vec{x}, \vec{y})\rangle\{Q'(\vec{x}, \vec{y})\}_k \triangleq \\ \quad \exists p. \forall \vec{x}. \langle P' * P(\vec{x}), P' \wedge p * P(\vec{x})\rangle_k; \\ \quad \left(\begin{array}{l} \mu A. \lambda p.\ \exists p'. \forall \vec{x}. \langle p * P(\vec{x}), p' * P(\vec{x})\rangle_k; Ap' \\ \qquad \sqcup \exists \vec{x}, \vec{y}. \exists p''.\ \langle p * P(\vec{x}), p'' * Q(\vec{x}, \vec{y})\rangle_k; \\ \qquad\qquad \left(\begin{array}{l} \mu B. \lambda p''.\ \exists p'''. \langle p'', p'''\rangle_k; Bp''' \\ \qquad \sqcup \langle p'', Q'(\vec{x}, \vec{y})\rangle_k \end{array} \right) p'' \end{array} \right) p \end{array}$$

The first atomic specification statement solely serves to capture the states satisfied by the non-atomic precondition P' into the variable p, so that it can be passed as an argument to the subsequent recursive function. It is a silent atomic step: since it does not change the state, the first atomic specification statement is not observable by AStutter. The recursive function that follows consists of two branches that are non-deterministically chosen using

HMumble
$$\forall \vec{x}.\, \exists \vec{y}.\, \{P'\}\langle P(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}_k \sqsubseteq$$
$$\forall \vec{x}.\, \exists \vec{y}.\, \{P'\}\langle P(\vec{x}), P'(\vec{x})\rangle\{P''\}_k; \forall \vec{x}.\, \exists \vec{y}.\, \{P''\}\langle P'(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}_k$$

HStutter
$$\forall \vec{x}.\, \{P'\}\langle P(\vec{x}), P(\vec{x})\rangle\{P''\}_k; \forall \vec{x}.\, \exists \vec{y}.\, \{P''\}\langle P(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}_k \sqsubseteq$$
$$\forall \vec{x}.\, \exists \vec{y}.\, \{P'\}\langle P(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}_k$$

HStrengthen
$$\forall \vec{x}.\, \exists \vec{y}.\, \{P'\}\langle P' * P(\vec{x}), Q(\vec{x},\vec{y}) * Q'(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y})\}_k \sqsubseteq$$
$$\forall \vec{x}.\, \exists \vec{y}.\, \{P' * P'\}\langle P(\vec{x}), Q(\vec{x},\vec{y})\rangle\{Q'(\vec{x},\vec{y}) * Q'(\vec{x},\vec{y})\}_k$$

Figure 8 Select hybrid specification statement refinement laws.

angelic choice. Note that these branches operate on both the non-atomically updated state captured by the logical variables $p, p', \ldots$ and the atomically updated state specified by the P and Q assertions. The first branch updates the non-atomic state p, while maintaining the atomic precondition $P(\vec{x})$ for all $\vec{x}$, and then recursively continues with the resulting non-atomic state p'. Effectively, this specifies that while P' is being modified in multiple steps the concurrent environment may change $\vec{x}$ as long as it maintains $P(\vec{x})$. The second branch of the angelic choice terminates the recursion by performing the atomic update from $P(\vec{x})$ to $Q(\vec{x},\vec{y})$, for some $\vec{x}$ and $\vec{y}$ (determiend by the the atomic update takes affect). The same update may also update the non-atomic part of the state. After this point, the non-atomic part of the state can continue to be updated until we reach a state satisfying $Q'(\vec{x},\vec{y})$; the atomic part cannot be further updated by the thread, and the environemnt need not be constrained, as the atomic step has been done by the thread.

We can derive refinement laws for hybrid specification statements which are analogous those given for the atomic specification statements. Most are straightforward extensions accounting for the non-atomic state component. We focus on the most interesting cases in figure 8. The HStrengthen law allows us to refine part of the non-atomic component update into an atomic update, essentially making the whole operation "more" atomic. The HMumble law simply extends AMumble to the hybrid case. However, consider HStutter. For the atomic component it acts in the same as AStutter, whereas for the non-atomic component it acts as the sequencing rule of Hoare logics. In fact, a property of these derived hybrid laws is when the atomic pre- and postconditions are `true` then the hybrid refinement laws correspond to standard Hoare rules, and when the non-atomic pre- and postconditions are `true` then the hybrid refinement laws correspond to the laws of atomic specification statements.

6 TaDA-Refine Client Reasoning II: Context Invariants

In section 4, we introduced the key elements of our client reasoning by examining a simple lock file module and proving a refinement between the abstract specification and implementation for the `unlock` operation. However, it encapsulates the entire file system within the **Lock** shared region and thus also the abstract predicate `Lock`. This prevents us from using the abstract specification to reason about clients that make further use of the file system.

We are unable to abstract the details of how a module's implementation shares the file system and maintain compositionality at the same time. This is due to the nature of POSIX file systems. In POSIX all possible file system paths are usable by everyone at all times,

even if they do not exist. The concept of a path being private to an application simply does not exist [5]. In other words the file system is a *public namespace*. In contrast, the traditional heap memory is a private namespace. Heap addresses are usable only when allocated. When an address is first allocated, it is only known to the allocating thread and thus we can programmatically control how they are shared with other threads. Dereferencing an unallocated heap address is undefined behaviour, typically resulting in a crash.

Effectively, in POSIX file systems all locations are shared, with everyone. Therefore, in §4, by fully encapsulating the file system state in the **Lock** shared region, we restrict all file system access to the lock-file module. This is too restrictive; we cannot reason about the module's use in contexts that also use the file system. The solution is to place restrictions on the context itself.

In the case of the lock-file module, we require that the context keeps the sub-graph formed by the path to the lock-file directory unmodified, and that the only way to create or remove the lock file is via the module's operations. Otherwise, the context could interfere with the resolution of the path, rendering it unresolvable or diverting the resolution to a different location. The lock-file module cannot enforce such restrictions on its own. Instead, these restrictions form a proof obligation for the context. We express such proof obligations with *context invariants*. For our lock-file module, LFCtx denotes the context invariant under which its specification holds.

In order to define LFCtx, we first encapsulate the file system within the global file-system shared region of type **GFS**. There is only a single instance of this region with a known identifier which we keep implicit. All clients accessing the file system do so via this region. The region's state is a file-system graph, $FS \in \mathcal{FS}$, with the straightforward interpretation: $I(\mathbf{GFS}(FS)) \triangleq \mathsf{fs}(FS)$. The guards and labelled transition system of this region, are defined by the context. However, we define LFCtx to place restrictions on the guards and transition system.

To aid our definitions, we introduce some notation: $\mathcal{G}_{\mathbf{t}}$ denotes the set of guards associated with the region type $\mathbf{t}$; $\mathsf{G} \bullet \mathsf{G}'$ denotes the partial, associative and commutative composition of guards; $\mathsf{G}\#\mathsf{G}'$ states that the composition of guards G and G' is defined; and $\mathcal{T}_{\mathbf{t}}(\mathsf{G})^*$ denotes the transitions for guard G of the region type $\mathbf{t}$, where the superscript * denotes the reflexive-transitive closure. We also define the following auxiliary predicates:

$$!\mathsf{G} \in \mathcal{G}_{\mathbf{t}} \triangleq \mathsf{G} \in \mathcal{G}_{\mathbf{t}} \wedge \mathsf{G} \bullet \mathsf{G} \textbf{ undefined}$$
$$(x, y) \dagger_{\mathbf{t}} \mathsf{G} \triangleq (x, y) \in \mathcal{T}_{\mathbf{t}}(\mathsf{G})^* \wedge \forall \mathsf{G}' \in \mathcal{G}_{\mathbf{t}}.\, \mathsf{G}'\#\mathsf{G} \Rightarrow (x, y) \notin \mathcal{T}_{\mathbf{t}}(\mathsf{G}')^*$$

The predicate $!\mathsf{G} \in \mathcal{G}_{\mathbf{t}}$ states that there is only one instance of the guard G of the region type $\mathbf{t}$, and $(x, y) \dagger_{\mathbf{t}} \mathsf{G}$ states that in regions of type $\mathbf{t}$, the transition from state x to y is defined only for G. Additionally, we define the expression $FS\restriction_p$ to identify the sub-graph of FS that is formed by the path p as follows:

$$FS\restriction_p \triangleq FS\restriction_p^{\iota_0} \qquad FS\restriction_a^{\iota} \triangleq \iota \mapsto (a \mapsto FS(\iota)(a))$$
$$FS\restriction_{a/p}^{\iota} \triangleq \iota \mapsto (a \mapsto FS(\iota)(a)) \uplus FS\restriction_p^{FS(\iota)(a)}$$

[5] File-access permissions restrict access to only the processes with the same privileges.

We can now define the context invariant as follows:

$$\begin{array}{c}\mathsf{LFCtx}(p/a) \triangleq \\ \exists FS \in \mathcal{LF}(p/a).\, \mathbf{GFS}(FS) \wedge !\,[\mathsf{LF}(p/a)] \in \mathcal{G}_{\mathbf{GFS}} \\ \wedge \forall FS \in \mathcal{ULK}(p/a).\, \exists FS'.\, \mathsf{lk}(FS, FS', p/a) \wedge (FS, FS') \dagger_{\mathbf{GFS}} \mathsf{LF}\,(p/a) \\ \wedge \forall FS \in \mathcal{LK}(p/a).\, \exists FS'.\, \mathsf{ulk}(FS, FS', p/a) \wedge (FS, FS') \dagger_{\mathbf{GFS}} \mathsf{LF}\,(p/a) \\ \wedge \forall \mathsf{G} \in \mathcal{G}_{\mathbf{GFS}}.\, \forall FS, FS' \in \mathcal{LK}(lf).\, (FS, FS') \in \mathcal{T}_{\mathbf{GFS}}(\mathsf{G})^* \Rightarrow FS\restriction_p = FS'\restriction_p\end{array}$$

The first line of the definition restricts the states of the global file-system region to those in which the path to the lock-file directory exists and the lock file itself may exist or not. Additionally, it requires the indivisible guard $\mathsf{LF}\,(p/a)$ to be defined for the global file-system region. The second and third lines of the definition state that transitions creating or removing the lock file in its directory are only defined for the guard $\mathsf{LF}\,(p/a)$. Therefore, only ownership of this guard grants a thread the capability to transition between locked and unlocked states. Finally, the last line of the definition requires all transitions between lock-file states to maintain the same file-system sub-graph for the path p. This guarantees that the context does not concurrently modify the sub-graph such that the path resolution is diverted to a different location.

Assuming the context satisfies $\mathsf{LFCtx}(lf)$, we can now redefine the interpretation of the **Lock** region as:

$$I(\mathbf{Lock}_\alpha(lf, 0)) \triangleq \exists FS \in \mathcal{ULK}(lf).\, \mathbf{GFS}(FS) * [\mathsf{LF}(lf)]$$

$$I(\mathbf{Lock}_\alpha(lf, 1)) \triangleq \exists FS \in \mathcal{LK}(lf).\, \mathbf{GFS}(FS) * [\mathsf{LF}(lf)]$$

Instead of the **Lock** fully encapsulating the file system itself in section 4, we now encapsulate only the possible ways in which the file system is shared with everyone else by means of the global file system region.

The global file system shared region is an abstraction breaker. All modules accessing the file system use it. Therefore, all file-system module specifications are effectively parametric to its definition. Thus we amend the original lock-file specification of section 4 accordingly:

$$\mathsf{LFCtx}(lf) \vdash \texttt{lock}(lf) \sqsubseteq \forall v \in \{0, 1\}\,.\, \langle \mathsf{Lock}(s, lf, v)\,, \mathsf{Lock}(s, lf, 1) * v = 0 \rangle$$

$$\mathsf{LFCtx}(lf) \vdash \texttt{unlock}(lf) \sqsubseteq \langle \mathsf{Lock}(s, lf, 1)\,, \mathsf{Lock}(s, lf, 0) \rangle$$

It remains to prove the refinement between the implementation and our specification. In figure 9 we give a sketch proof for the `lock` operation. Throughout the proof we assume that LFCtx holds. On the other hand, LFCtx is a proof obligation for the context.The main difference from the proof of `unlock` in § 4 is that we use AUseAtomic twice; first to refine the atomic update on the Lock predicate into an update on the global file system region **GFS** and then again to refine that update to an atomic update on the underlying file system in the refinement of `close` and `open`. The refinement of `open` proceeds similarly to `unlock` and is given in the technical report [29].

7 Related Work

There has been substantial work on the formal specification of key fragments of POSIX file systems, even leading to a verification challenge by Joshi and Holzmann [22]. Refinements from specifications to implementations have been widely studied [3, 19, 15, 16]. Of particular note are the specifications based on Z notation, and the use of refinement calculus to construct verified implementations [24, 15, 16]. Recently, specifications based on separation

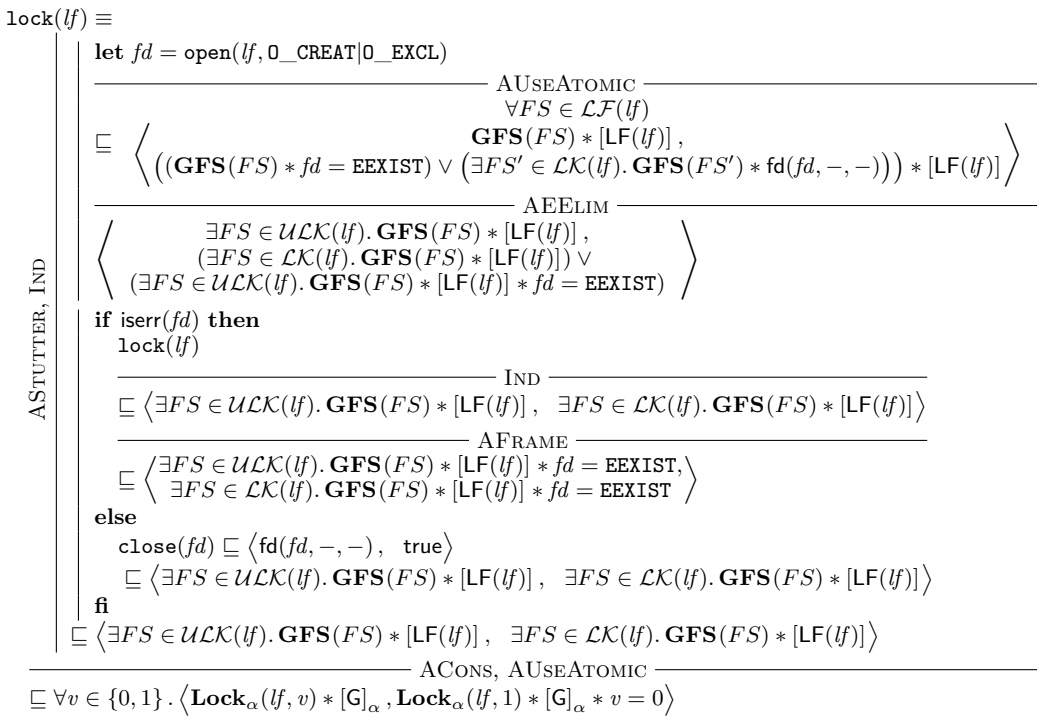

Figure 9 `lock()` specification proof sketch.

logic [31] have been introduced, focusing on scalable client reasoning [17, 30]. This work [17] demonstrates that first-order reasoning scales poorly when reasoning about file-system clients, hence the introduction of a specification based on separation logic. Taking inspiration from this work, we demonstrate scalable reasoning about clients of POSIX file-systems using TaDA-Refine. Separation logics have also been used to build a verified fault-tolerant file-system implementation in the Coq theorem prover [7] and to verify elements of the Linux Virtual Filesystem Switch (VFS) [12]. All the aforementioned works are on sequential fragments of POSIX and do not handle concurrency.

Fisher *et al.* develop Forest [14], a declarative DSL in Haskell for safe manipulation of file systems. Forest clients use the typing discipline to specify the file-system structures they need and file-system access preserves the application invariants identified by static types. This work is an attempt to bridge the gap between the untyped world of sequential file-systems and the strongly-typed world of programming languages.

Ridge *et al.* have developed a coarse-grained concurrent specification of a fragment of the POSIX file system, based an operational semantics [33] with adaptations in the semantics to capture real-world implementations. The specification is used as a test oracle in a substantial test suite which they generate for major real-world implementations. However, since their concurrent specification is coarse-grained, their tests can only expose sequential behaviour. We can derive such coarse-grained specifications from the specifications we give in this paper, by a trivial application of the AMumble refinement law. Such coarse-grained specifications could be used to verify coarse-grained implementations. However, they are not suitable as a general POSIX specification for client reasoning, as the implicit assumption of additional synchronisation is too strong.

We have given a specification of the complex concurrent behaviour associated with POSIX file systems by introducing TaDA-Refine, a concurrent specification language based on TaDA assertions [9, 8] and an associated refinement calculus. Our approach is inspired by the work of Turon and Wand [36]. However, we do not adopt their notion of fenced refinement to reason about fine-grained concurrent data structures as its applicability is more limited than more recent mechanisms of expressing sharing protocols and capabilities found in concurrent program logics. More significantly, fenced refinement proofs carry the built-in assumption that the module's state can only ever be changed by the module's operations which is not appropriate for reasoning about file-system clients. For this purpose we adopt TaDA's assertions and introduce context invariants for client reasoning. Furthermore, we introduce the hybrid specification statement as useful derived construct for reasoning about combinations of atomic and non-atomic effects.

Recently, various concurrent separation logics have been introduced to support reasoning about atomic operations [20, 35, 34, 9, 23, 26]. However, the examples have generally been limited to those using operations comprising single atomic steps. In contrast, our work on specifying POSIX file systems requires operations comprising multiple atomic steps. With higher-order logics such as [35, 34, 23], it is possible to encode multi-atomic specifications as auxiliary code in the style of [20]. With logics based on histories such as [26], it should also be possible to support multi-atomic specifications, although it is unclear if this method is applicable to operations that have concurrent path traversals such as `link`. It is important to note that, for client reasoning, all these formalisms require additional constraints on the context analogous to our context invariants.

8 Conclusions & Future Work

We have developed TaDA-Refine, a concurrent specification language and an associated refinement calculus which is able to specify the complex concurrent behaviour of POSIX file systems. To the best of our knowledge, this is the first specification of file-system concurrency that captures the intended POSIX semantics. Here, we have verified the lock-file client module. In Ntzik's thesis [27], we have also verified an implementation of named pipes which regular file I/O and lock file, and the concurrent interaction between and email client and server that is sensitive to the multi-atomic nature of path resolution. This client verification is not straightforward due to the file system being a public namespace. We introduce specifications conditional on context invariants to restrict interference.

Our research on file-system specification and client verification is far from over. We believe we have formalised the established consensus of the concurrent behaviour of POSIX file systems. Our methodology is, however, flexible enough to explore other choices, if desired. We plan to extend the specification to larger fragments, for example, covering symbolic links and file-access permissions. These are orthogonal to POSIX file-system concurrency and should not affect our reasoning methodology presented here.

For this paper, we justify our specification by appealing to the standard and the community consensus regarding the atomicity of operations. In future, we will justify our specification with respect to implementations. We plan to systematically justify the specification against real-world implementations by generating tests and using the specification as a test oracle, similarly to the approach of Ridge *et al* [33]. Another approach, following our refinement laws, is to refine the specification to a fine-grained concurrent reference implementation. Both approaches will require a mechanised version of our POSIX specification. Additionally, we want to build on the works of Chen *et al.*[7] and Ntzik *et al.* [28] to extend our specifications with fault-tolerance guarantees. We also want to study Network File Systems (NFS), which exhibit concurrent behaviours that are not sequentially consistent.

References

1 The Austin Group Mailing List. https://www.opengroup.org/austin/lists.html. Accessed: September 30, 2016.

2 POSIX.1-2008, IEEE 1003.1-2008, The Open Group Base Specifications Issue 7. URL: http://pubs.opengroup.org/onlinepubs/9699919799/.

3 Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 373–390. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-30482-1_32.

4 Ralph-Johan Back and Joakim Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.

5 Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996. doi:10.1006/inco.1996.0056.

6 C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 366–378, July 2007. doi:10.1109/LICS.2007.30.

7 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 *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM. doi:10.1145/2815400.2815402.

8 Pedro da Rocha Pinto. *Reasoning with Time and Data Abstractions*. PhD thesis, Imperial College London, 2016.

9 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-662-44202-9_9.

10 Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 287–300, New York, NY, USA, 2013. ACM. doi:10.1145/2429069.2429104.

11 Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, MatthewJ. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-14107-2_24.

12 Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif. Verification of a virtual filesystem switch. In Ernie Cohen and Andrey Rybalchenko, editors, *Verified Software: Theories, Tools, Experiments*, volume 8164 of *Lecture Notes in Computer Science*, pages 242–261. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54108-7_13.

13 Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM. doi:10.1145/1480881.1480922.

14 Kathleen Fisher, Nate Foster, David Walker, and Kenny Q. Zhu. Forest: A language and toolkit for programming with filestores. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 292–306, New York, NY, USA, 2011. ACM. doi:10.1145/2034773.2034814.

15 L. Freitas, Zheng Fu, and J. Woocock. Posix file store in z/eves: an experiment in the verified software repository. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 3–14, July 2007. doi:10.1109/ICECCS.2007.36.

16 Leo Freitas, Jim Woodcock, and Andrew Butterfield. Posix and the verification grand challenge: A roadmap. *2014 19th International Conference on Engineering of Complex Computer Systems*, 0:153–162, 2008. doi:10.1109/ICECCS.2008.35.

17 Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the posix file system. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54833-8_10.

18 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

19 WimH. Hesselink and MuhammadIkram Lali. Formalizing a hierarchical file system. *Formal Aspects of Computing*, 24(1):27–44, 2012. doi:10.1007/s00165-010-0171-2.

20 Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 271–282, New York, NY, USA, 2011. ACM. doi:10.1145/1926385.1926417.

21 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-20398-5_4.

22 Rajeev Joshi and GerardJ. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007. doi:10.1007/s00165-006-0022-3.

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

24 Carroll Morgan and Bernard Sufrin. Specification of the unix filing system. *Software Engineering, IEEE Transactions on*, SE-10(2):128–142, March 1984. doi:10.1109/TSE.1984.5010215.

25 Carroll Morgan and Trevor Vickers. *On the refinement calculus*. Springer Science & Business Media, 2012.

26 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and GermánAndrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 290–310. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54833-8_16.

27 Gian Ntzik. *Reasoning About POSIX File Systems*. PhD thesis, Imperial College London, sep 2016.

28 Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner. *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, chapter Fault-Tolerant Resource Reasoning, pages 169–188. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-26529-2_10.

29 Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. A concurrent specification of POSIX file systems (technical report). Technical Report 2018/3, Department of Computing, Imperial College London, 2018. URL: https://www.doc.ic.ac.uk/research/technicalreports/2018/#3.

30 Gian Ntzik and Philippa Gardner. Reasoning about the posix file system: Local update and global pathnames. In *Proceedings of the 2015 ACM SIGPLAN International Conference*

on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, pages 201–220, New York, NY, USA, 2015. ACM. doi:10.1145/2814270.2814306.

31 J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74, 2002. doi:10.1109/LICS.2002.1029817.

32 John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.

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 *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 38–53, New York, NY, USA, 2015. ACM. doi:10.1145/2815400.2815411.

34 Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer Berlin Heidelberg, 2014. doi:10.1007/978-3-642-54833-8_9.

35 Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. Modular reasoning about separation of concurrent data structures. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-37036-6_11.

36 Aaron Joseph Turon and Mitchell Wand. A separation logic for refining concurrent objects. *ACM SIGPLAN Notices*, 46(1):247–258, 2011.

37 Viktor Vafeiadis and Matthew Parkinson. *A Marriage of Rely/Guarantee and Separation Logic*, pages 256–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74407-8_18.