Towards verified file systems

Thesis submitted for the degree of

Doctor of Philosophy

at the University of Leicester

by

Andrea Giugliano

Department of Informatics

supervised by

Dr. Tom Ridge

December 6, 2018

Abstract

The formal methods community aims to provide a stack of verified software to users. Verified software is proven to be reliable. The rigour of mathematical logic makes it possible to prove that software meets the designer expectations. File system software enables organized data storage, and in most software systems this functionality is critical. This work provides the basis on which to build a formally verified file system. Firstly, a formal and mechanized specification of POSIX (and Linux, Mac OS X, FreeBSD) is defined and used as an oracle to test if modern implementations behave correctly; then it is shown how to extend this specification with timestamps and the challenges this extension entails; finally the definition of an immutable B-tree and the mathematical verification of its operations are mechanically formalized. operations are mechanically formalized.
These achievements bring the development of a verified file system within reach.

Acknowledgements

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted.

My supervisor Dr. Tom Ridge guided me into the world of research, by supporting me in many moments of disorientation and teaching me the concept of elegant simplicity by example.

My colleagues Thomas Türk, David Sheets, Anil Madhavapeddy and Peter Sewell who supported me during the research and that shared the effort needed to publish part of this work.

The people at Microsoft Research Limited who believed in the potential of this work and who decided to fund this Ph.D. position.

My family that enriched my studies and allowed me to study abroad, and so to achieve an MSc, realize that not everyone eats pasta, and open a path to some interesting research.

My beautiful girlfriend who enlightened (and is currently enlightening) the last part of this journey.

My warmest thanks to the Informatics department, formerly the Computer Science department, of the University of Leicester, which gave me the opportunity to meet fantastic people who generously shared their knowledge and time with me.

My dearest thanks to the Leicester community who made my stay in this city so profound: poets such as Bobba Cass taught me that poetry needs to be in your life before it is in your pen; writers at the Phoenix Writers Club taught me how to write for an erudite and enthusiastic public; orchestra directors such as Paul Jenkins who taught me how to sing and perform opera; runners at the Parkour society who helped me discover how obstacles can become exciting opportunities; children who educated me in how to teach African drumming; street artists who taught me that music is from the heart rather than from the brain; homeless people who taught me to not forget.

Thanks also to those who left a sign in history and made me bloom as a person: philosophers, poets, writers and singers such as Alan Watts, Kahlil Gibran, Oscar Wilde and Angelo Lo Forese, respectively.

Contents

List of Tables

List of Figures

1 Introduction

Software engineering is complex. Complexity causes errors, and the experience of software malfunctions is common. Although one can deal with an unresponsive phone by trying to restart it, there are situations in which software errors are not affordable: some unfortunate examples of this important issue happened in the medical field [45], the space field [18, 59, 66], the military field [72, 73], and in security [19]. For instance, in 1999 a NASA mission worth $327.6 million failed because a team of engineers used a different measurement unit to calculate velocity than the one used by the other components of the system: the Mars Climate Orbiter disappeared in space before accomplishing all the mission's targets [67].

In such cases the aim should be to eliminate the risk of malfunction. Although standard techniques provide sufficiently reliable software [32], in mathematics one can certify correctness using proof, and one can apply proof to software also. This process is called software formal verification. The computer science community has built numerous tools to make formal verification more practical. Part of the community considers formal methods essential to obtain software correctness [33, 77]. Yet, formal verification is not suitable in all occasions: indeed, it is a complex process and the automation provided by tools available today makes it only just feasible for relatively small examples [2]. One can divide tools for formal verification in two broad categories [5]:

model checkers check the design with respect to the specified properties encoded in a modeling language. A model checker will attempt to do so automatically with limited human intervention and return one of three results:

1. Properties are satisfied by the design.
2. Properties are not satisfied, for which a counterexample will be given.
3. Indeterminate. The state space is such that the tool cannot compute a result in a reasonable amount of time.

proof assistants combine automated techniques with manual guidance to prove correctness. They are generally more powerful than model checkers; developers can use built-in tactics or develop new ones that aid in proving safety and security propositions.

In general, model checkers (e.g., the boolean satisfiability solver are able to prove many things automatically, by limiting the expressivity of the language and properties that are verified, while proof assistants (e.g., Isabelle/HOL [57]) support rich languages but need human intervention for complex proofs such as those involving induction [11]. The flexibility of proof assistants makes them usable to verify complex software.

Another problem is that in building a software application for a user, one always relies on the correctness of other components (e.g., a software library, the operating system, the drivers, the compiler, the hardware, etc. . .). This means that even after verification, its correctness is strictly bound to the correctness of its dependencies.

Indeed, the formal methods community is trying to build a stack of verified software which can provide a correct basis to build new software applications. For example they already delivered CompCert, a compiler for the C programming language [44]; CompCertTSO, the same compiler with concurrency [70]; CakeML, a verified compiler from core ML to machine code [41]; Vellvm, a verified LLVM optimization pass [80]; RockSalt, a verified software-based fault isolation (SFI) for x86 architectures [54]; seL4 a verified hypervisor [40]; NetSem, a formal specification of network semantics [8, 60]; and verified low-level crypto-protocol implementations [37].

A verified file system implementation is necessary in order to build verified systems that make use of a file system as a trusted component. Such a verified implementation should ideally be competitive with state of the art file systems. Because file systems are complex, their behavior must be formalized and their components need to be mechanically verified, aiming for simplicity and modularity.

Along with networking and core OS functionalities such as process and memory management, the file system is a key part of most systems. A rigorous approach to identify software errors is formalizing a test oracle [76]. To implement this approach, both specification and implementation of a software must exist as source code. The specification is simpler as it behaves as the implementation but does not include unnecessary details (such as optimizations). Having a formal specification and an implementation available, one should expect that the same input values will produce the same results. A mismatching result signals that there is an incongruity between what the designer aimed and the implementation is achieving. In some cases, the mismatch can be due to an erroneous specification. A specification can be wrong if it does not consider all the use cases of the implementation, or if it defines the wrong feature behaviour: in such cases the specification needs to be updated until it reflects the intention of the designer. Having a formal specification removes ambiguities and inconsistencies, which affect most of natural language specifications.

A critical aspect is the correctness of the model: an implementation that satisfies a wrong specification is wrong. Errors in the specification are more likely if the targeted system is complex. However, one can mitigate this issue if existing implementations of the targeted system are available. Indeed, one can compare the behaviors of the specification and the implementations to validate and enhance the specification.

In the case of file systems there is no need to design a new specification, since a well known operating system specification that includes the behaviour of file systems already exists: the POSIX (Portable Operating System Interface) specification developed by IEEE and the Open Group, which is an IEEE, ISO/IEC and Open Group Technical standard [74].

POSIX is not a formal specification and it is inconsistent and ambiguous as the Open Group mailing list testifies. When these issues appear in documents as large as POSIX, their significance worsen as even fixing simple incongruities becomes difficult, since multiple experts need to be consulted to find a suitable solution. A mechanized specification can solve the issues caused by informality, as a computer system is capable of checking the consistency of definitions in an insignificant amount of time.

A mechanized and formal specification for the POSIX file system is SibylFS [62], see Chapter 2. This models in high order logic not only the subset of the POSIX file system semantics, but also the divergences from POSIX that Linux, Mac OS X and FreeBSD have implemented in their own designs. SibylFS is a specification of a non-deterministic system, as its commands may have more than a single allowed output and many concurrent processes use the file system. A test oracle can take an observed trace and it can determine if it matches the behaviour of known systems (e.g., Linux). This feature helps to find errors in the specification and compare file system behaviours between platforms.

Lem is the high order language used to specify SibylFS [55]. Lem syntax is similar to the one of functional languages of the ML family, and it is, indeed, translatable to OCaml for testing, and Coq, HOL4, and Isabelle/HOL for proving properties of the specification. This choice makes SibylFS usable as an OCaml test oracle and at the same time discloses the specification to a wider audience of proof assistant users.

SibylFS was designed to be modular in order to easily add the features that current file systems already offer. For example, modern file systems maintain data describing files and directories, such as who is authorized to change them (permissions) or when these have been accessed or modified (timestamps). This work demonstrates how to formalize POSIX timestamps in SibylFS and what challenges it entails (Chapter 3).

The modular design of SibylFS makes it both usable as an oracle and suitable to be the basis of a verified file system: at the heart of SibylFS there are two maps, one from file identifiers to files and one from directory identifiers to directories; by substituting these two maps with a correct storage model (this must provide a synchronization, writing and reading mechanism), the file system specification can be instantiated to a file system implementation that respects the specification. A good candidate to a storage model is the B-tree data structure. The B-tree data structure implements a persistent on-disk map, which can substitute the maps at the heart of SibylFS specification (Figure 1.1).

The B-tree data structure has various desirable properties such as offering a search algorithm that remains efficient also when applied to big quantities of data. A correct B-tree specification for the *find* and *insert* operations was achieved during this study (Chapter 4). Isabelle/HOL is the high order language used to define

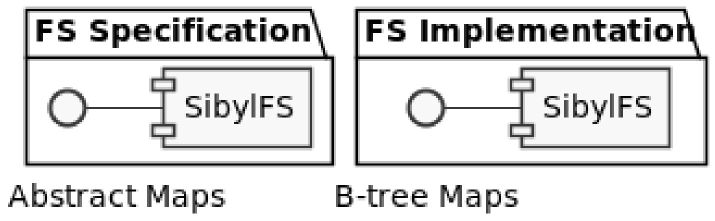


Figure 1.1: SibylFS as a specification or as an implementation

the B-tree and its operations. The proofs of correctness for the B-tree operations are encoded in the procedural style of the Isabelle/HOL proof assistant. The Isabelle/HOL proof assistant is enriched by a library of proved lemmas, which facilitate the proof process. It also allows to translate definitions in various languages (e.g., OCaml, Haskell and Scala), making possible to test definitions and their properties before developing proofs.

The design chosen for the B-tree data structure was extensively reworked to simplify the proof process [85]. The proofs aim to show that the B-tree implements a Map data structure. Knowing that the B-tree interfaces are isomorphic to the interfaces of a Map guarantees that the data structure behaves correctly. Having a correct storage model permits the generation of a verified file system via SibylFS.

This is sufficient to provide a basic functioning file system. However, to produce a file system with performance equivalent to existing modern file systems requires additional components discussed in Chapter 5.

Overview of thesis

The content of this thesis is divided as follows: Chapter 2 presents the SibylFS formal specification and its oracle use case; Chapter 3 presents the timestamp extension to SibylFS; Chapter 4 presents a formal specification of the B-tree data structure and a mechanized proof of correctness for its operations; Chapter 5 summarizes future work to achieve the verification of a usable file system.

2
SibylFS: a formal file system specification

2.1 Overview

Some systems depend critically on the behaviour of file systems, but that behaviour differs in many details, both between implementations and between each implementation and the POSIX (and other) prose specifications.[1] Building robust and portable software requires understanding these details and differences, by systematically describing, investigating, or testing file system behaviour across this complex multi-platform interface. In this chapter we discuss how to characterise the envelope of allowed behaviour of file systems in a form that enables practical and highly discriminating testing. We see a mathematically rigorous model of file system behaviour, SibylFS, that specifies the range of allowed behaviours of a file system for any sequence of the system calls within the formal specification scope. This can be used as a test *oracle* to decide whether an observed trace is allowed by the model, both for validating the model and for testing file systems against it. SibylFS is modular enough to not only describe POSIX, but also specific Linux, Mac OS X and FreeBSD behaviours. An extensive test suite of over 21 000 tests complements the model; this can be run on a target file system and checked in less than five minutes, making it usable in practice. Finally, we see experimental results for around 40 configurations of many file systems, identifying many differences and some serious flaws.

2.2 Introduction

The process of testing a file system and checking the resulting traces with SibylFS is depicted in Figure 2.1.

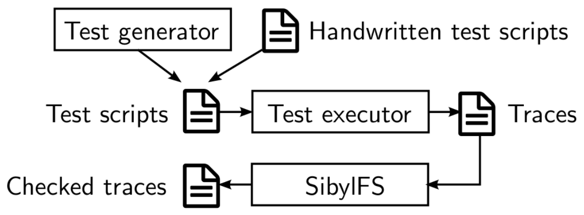


Figure 2.1: File system testing and trace checking

The process starts with a set of test scripts, organized into groups according

[1]Most of this chapter is based on material from a published paper in SOSP2015 [62] which includes the current author as coauthor.

to the libc functions they target. The bulk of these test scripts are generated automatically by the test generator, and are supplemented by hand-written test scripts. Test scripts contain sequences of file system commands that are used by the test harness to drive the real-world file system under test, via the libc interface. An example excerpt from a test script is:

```
@type script
# Test rename___rename_emptydir___nonemptydir
mkdir "emptydir" 0o777
mkdir "nonemptydir" 0o777
open "nonemptydir/f" [O_CREAT;O_WRONLY] 0o666
rename "emptydir" "nonemptydir"
```

after the header, each line is the data for a single libc call (more complex test scripts can involve multiple processes). Each script sets up whichever file system state it needs, starting from an empty file system; they involve up to several hundred libc function calls. The resulting behaviour is recorded in a trace file, as:

```
@type trace
# Test rename___rename_emptydir___nonemptydir
3: mkdir "emptydir" 0o777
RV_none
... [further calls and return values] ...
6: rename "emptydir" "nonemptydir"
EPERM
```

in this example we see the interleaving of commands from the script with the responses received from the real-world system. The *RV_None* return value indicates the call completed successfully.

These trace files are processed by SibylFS to check for conformance with the model. The main part of the SibylFS checker is the model itself, automatically translated from Lem to OCaml and then linked together with a small OCaml wrapper. Checking is done with respect to a particular variant of the model (POSIX, Linux or OS X); in addition, various flags control further checking parameters, such as whether the initial process runs with root privileges or not. The output from this checking phase is a set of checked traces. An excerpt from an example failing trace is:

```
6: rename "emptydir" "nonemptydir"
```

```
   EPERM
# Error:    6: EPERM
#   unexpected results: EPERM
#   allowed are only: EEXIST, ENOTEMPTY
#   continuing with EEXIST, ENOTEMPTY
# trace not accepted
```

For steps in the trace that conform to the model, the checked trace resembles the original trace. For steps that are non-conformant, the checked trace includes an error message and (if possible) diagnostic information to help identify why the behaviour is non-conformant. In the example above the error message indicates that at line 6 in the trace file the real-world file system returned `EPERM`, but the specification allowed only `EEXIST` or `ENOTEMPTY`. Note that the example is an excerpt of a checked rename trace from SSHFS/tmpfs 2.5 on Linux 3.19.1. Individual trace files may contain multiple test calls, and so it is important that the checker tries to continue even when an individual step fails. In the example above SibylFS continues checking the trace under the assumption that `EEXIST` or `ENOTEMPTY` was returned rather than `EPERM`. Analysis of the results also requires automation to assist with the volume of data, as each run produces tens of thousands of checked traces per platform, and the results must be compared between file systems and between model versions (during model development). Checked traces can be rendered to HTML, along with autogenerated indexes and summaries of check results. To analyse the results of multiple runs, typically for different file systems on the same operating system, the system can intelligently combine the results across many different platforms, merging behaviours common to many runs and highlighting the differences. In addition, a model-debugging tool allows model developers to analyze the checking process itself, taking a trace and producing a description of the real-world states that were being tracked by SibylFS at every step of the trace. This has been extremely useful for developing the model, although end users of SibylFS should not need it. The process of constructing the model has been intimately entwined with testing: testing (particularly on new operating systems and file systems) uncovers new real-world behaviours, which are then incorporated into the model; new tests are added and the updated model is then used for another round of testing, with those behaviours now not generating discrepancies. This represents a virtuous circle: at each stage the model becomes more accurate and comprehensive, and the test suite accumulates more and more tests.

2.2.1 Motivation

File systems, in common with several other key systems components, have some well-known but challenging properties:

- they provide behaviourally complex abstractions;
- there are many important file system implementations, each with its own internal complexities;
- different file systems, while broadly similar, behave quite differently in some cases; and
- other system software and applications often must be written to be portable between file systems, and file systems themselves are sometimes ported from one OS to another, or written to support application portability.

Thus file system behaviour, and especially these variations in behaviour, must be understood by those developing file systems, by those aiming to write robust and secure software above them, and by those porting file systems or applications. Normal practice has for decades relied on prose standards and documentation (the POSIX standard [74], Linux Standard Base (LSB) [46], *man* pages) and on test suites [47, 75]. Indeed, this is so well established that many practitioners would not imagine that any alternative exist. But normal practice does not support any of the above: prose documents generally cannot be made complete and unambiguous; they cannot be used as a test oracle to automatically determine whether some observed behaviour is allowed or not; and building test suites without a test oracle requires manual curation of the intended outcome of each test. As we will see from the test results of this work, behavioural differences between file systems have proliferated, some intentional and many clearly bugs.

2.2.2 Scope and limitations

Scope

POSIX describes many aspects of operating systems, but the model considered in this work covers only the part that is relevant to file systems. The specification includes the libc commands: `close`, `closedir`, `link`, `lseek`, `lstat`, `mkdir`, `open`, `opendir`, `pread`, `pwrite`, `read`, `readdir`, `readlink`, `rename`, `rewinddir`, `rmdir`, `stat`, `symlink`, `truncate`, `unlink`, and `write`. This covers the essential commands that are necessary to manipulate and interrogate the directory structure and file contents, and the functions dealing with symlinks (`readlink`, `symlink`). Together this is sufficient to cover a broad range of uses. The model also includes a model of processes and the operating system, again focusing on those aspects that are relevant to file systems. Processes can be created and destroyed. Each process has a working directory which is mainly used when resolving relative paths. For this reason we include `chdir`. Additional per-process structures that we model include the file-descriptor table and the process run state. We also model permissions, including `chmod`, `chown`, and `umask`, and a model of users, groups, and which users belong to which groups. POSIX includes notions of *undefined*, *unspecified* and *implementation-defined* behaviour. Undefined behaviour results from using a libc function with arguments that are invalid according to POSIX. Unspecified behaviour results from a libc function call with arguments that are valid, but for which POSIX leaves the behaviour unspecified. Implementation-defined behaviour is similar to unspecified behaviour, though it is expected that conforming implementations explicitly document their behaviour in such cases. The aforementioned model covers all these cases for the POSIX platform. The variants of the model for real-world platforms describe the actual real-world behaviour, even where POSIX declares the behaviour to be undefined.

Limitations

Currently the model does not include host crash-failure. It includes concurrent file system API calls, and the checking infrastructure supports them, but the test harness does not force libc calls from different processes to overlap in time (e.g., so that both calls execute in-kernel simultaneously). The test harness and test suite do cover interleaved calls from multiple processes, which is important when modeling and testing permissions. It does not include unusual file types (such as FIFO special files) or signals and the associated EINTR error. Also it excludes exotic errors such as EIO (a physical I/O error has occurred) and ENOMEM; from a modeling perspective such errors could potentially occur at any time. It also does not model many resource exhaustion behaviours, such as exceeding the maximum number of entries in a directory or using all available inodes. It does not currently model the *at forms of functions, although it should be straightforward to adapt it to include them. It does not model free space or storage media behaviour in general. One can imagine future work developing, for a particular file system of interest, an executable abstraction function that reads a concrete volume state (perhaps after a host crash and recovery) and calculates the corresponding abstract state of the model. Testing the correspondence between implementation and model at each step, analogously to [60], would likely be extremely discriminating. The model parameterization, while desirable and necessary, also has a cost: running SibylFS is low-cost, but adapting SibylFS to model a file system with significantly different behaviour can involve substantial work (though with a big pay-off: characterizing that behaviour in detail).

2.3 Technical challenges

2.3.1 Non-determinism

In writing a model to be used as a test oracle (i.e., to compute whether observed traces are allowed by the model or not), the treatment of non-determinism is a key issue. If both the model and the implementations are entirely deterministic, at the abstraction level at which they are being observed, then one could just run the model and an implementation on the same input and check whether they have equal output. But for real-world software that is rarely the case: implementation behaviour typically varies, both between implementations and depending on implementation-internal runtime choices, and specifications are often deliberately loose. For example, for file systems:

- Some API calls could give rise to several distinct errors, e.g., `EISDIR`, `EEXIST` or `ENOTEMPTY` when renaming a file to a non-empty directory; which error is actually returned may be determined by the order the checks are made in the file system implementation code.
- The number of bytes returned by a `read` may be less than the number requested, determined by the implementation internal state.
- The order in which `readdir` returns entries from a directory with multiple entries will depend on the implementation and on details of the storage layout of the directory data (neither of which belong in an abstract specification).
- The behaviour of concurrent API calls may be determined by scheduling.

A sound specification must be loose enough to accommodate all such variation (we should not confuse looseness with the question of whether a specification is precise: we want a mathematically precise model, but one that admits a range of allowable behaviours). However, checking a trace against such a specification poses an algorithmic problem, especially when there is internal non-determinism that is not immediately observable. In general one must effectively track the set of all possible implementation states (abstracted to what can affect external observation) at every step of the trace, or, equivalently, calculate the set of constraints on the specification state that arise from a trace of observations. The NetSem project of Bishop et al. [9] produced a specification and trace-checker in that form for TCP/IP and the Sockets API, but it required a sophisticated higher-order logic constraint solver and a backtracking search process, and checking around 1000 traces (of broadly similar character and size to ours) took 2500 CPU-hours, at the limits of practicality. At the same time, there is a tension between writing a model to be as clear as possible and one that supports efficient checking (both quite different from writing a file system implementation, of course). As much as possible we want to avoid polluting the model with algorithmic concerns. Accordingly, for SibylFS we took great care up-front to write the model in a way that would remain clear and be efficiently checkable, without the need for backtracking search or sophisticated constraint solving. SibylFS incorporates different strategies for different sources of non-determinism, as follows.

Simple non-determinism via possible next-state enumeration At the top level the model consists of a type of abstract file system states and a function that, given such a state and an API event (call, return, etc.), returns a finite set of possible next states. We will go into more detail in §2.4. For the simple case of multiple possible API error return values, the model explicitly calculates the set of all expected errors (using novel and concise combinators, as described in §2.3.2) and a subsequent state for each, then when the real-system return value is observed we simply choose the corresponding state. The model uses a similar approach to deal with the number of bytes processed by a read or write, by just enumerating the possible immediate next states. This is attractively simple and suffices for testing. However, it does involve some unnecessary cost for tests with large reads or writes, enumerating many next-states. This blowup is resolved at the next step in the trace, when the actual number of bytes read by the real-world process becomes known. To test with very large reads and writes one could refactor the model slightly to produce continuations abstracted on the API return values, to check them and calculate a single next state. A disadvantage of doing this uniformly is that it makes it hard for the checker to describe, for a failing step, the set of values that would have been allowed (as we saw in the example trace of Chapter 1).

Directory listing non-determinism by hand-crafted specification The `readdir` command is challenging to specify. A process can request a directory handle using `opendir` and then use `readdir` to return the directory entries. These can be returned in any order, therefore this command gives rise to significant non-determinism. However, the real challenge in specifying this command is to deal with modifications of the directory (either by the same process or a different process) while the directory handle[2] is open. If the directory is not modified at any point, then `readdir` returns all the entries in the directory, and each entry is returned exactly once. The POSIX intent is to provide a similar guarantee when the directory is modified, and real-world file systems also provide this guarantee, as far as we can observe: for any entry, if that entry is not modified from the time the directory handle is opened, then that entry will be returned by `readdir` exactly once. If an entry is deleted, and if it has not already been returned by `readdir`, then it may be returned by subsequent calls to `readdir` (if it has already been returned, then it is not returned again if it is deleted). Similarly, if an entry is added, then it may be returned by subsequent calls. So far, the semantics could be modeled by taking a snapshot of the entries when `opendir` is called, and recording which entries have already been returned by `readdir`. On subsequent calls to `readdir`, the entries in the directory at that point could be examined, and the possible entries that could be returned at that point could be determined. The problematic case arises for entries that are initially in the directory, then deleted, then added again (or, vice versa, those that are added then deleted). According to POSIX, these entries may (but need not) be returned. In order to model this behaviour, we are forced to track all changes to a directory from the point that `opendir` is called, as well as the entries that have already been returned by `readdir`. With this information, it is possible to determine the set of entries that must be returned, and those that may be returned, and thus to give a semantics to the whole command. In fact, we need to maintain (rather than compute) sets of must and may entries in a directory. Whenever a directory handle

[2]A directory handle is an identifier that references the contents of a directory.

is read from, it accesses the changes since the last time it was read from, and updates the must and may sets, before non-deterministically splitting to allow any of the entries in must or may be read. This non-determinism is resolved at the next step, when the label reveals the entry actually read. It is worth noting that this is an area where a good specification is conceptually more complicated than any particular implementation: the latter just returns some list of names, while the model has to capture all allowable sequences, ruling out all those that are not possible.

Interleaving concurrency non-determinism via state sets Non-determinism from multiple user processes executing file system API calls concurrently also results in non-deterministic behaviour, e.g., if one process renames a file while another removes it. The SibylFS model and trace checker cope with this by maintaining explicit sets of possible (model) file system states. Currently our test infrastructure cannot generate traces in which individual calls overlap: typically the first call will complete before the second is executed, and the call-and-return pairs will be interleaved. The proposed test suite does not contain tests of this kind: systematically testing non-interleaved concurrent interaction would require extensive additional effort. Typically OS kernels handles parallelism through locking mechanisms [65]. The interleaving approach models the results of parallel calls. When these locking mechanisms are not well implemented, calls may result in race conditions, which may cause system crashes. The current SibylFS model does not aim to model these errors. Another way to avoid internal non-determinism is to expose the implementation behaviour by instrumenting it, so that all internal choices that affect external behaviour are captured as trace events. For a single implementation that might be viable (and indeed desirable, as it would permit checking of internal invariants). But for checking many file systems, the black-box approach that we consider here is more tractable.

2.3.2 Complexity

To give an idea of the challenges in identifying and describing complex real-world behaviours, we can take as an example the process of updating the model for OS X. Variants for POSIX and Linux were already available. In order to understand what parts of OS X behaviour were different from the available variants, we needed to run the tests on OS X with the default HFS+ file system, and check the traces against the POSIX variant of the model. The result would be thousands of failing traces (around 5 000 for open alone).

We would manually analyze the failing OS X traces to identify why they were not allowed according to our understanding of POSIX. This was painstaking work, that took roughly four to six weeks (though still small compared with the effort required to implement a production file system). The next step was to rework the model to incorporate these new OS X behaviours, while remaining concise, structured, and readable. The process is one of inferring, from thousands of observed behaviours, a compact description of those behaviours (as a higher-order logic specification). To make it feasible to write the model and to extend it in this way, it was essential to structure the model in various ways. Different mechanisms have been useful to address different kinds of complexity.

Modules The model is written in a mathematically rigorous language, the typed higher-order logic of the Lem tool [55]. Lem provides a notion of module: a collection of type, pure function, and inductive relation definitions (analogous to the modules of OCaml and other ML-like languages). We used these to structure the model as a set of independent modules, with clearly defined interfaces, as shown in Figure 2.2.

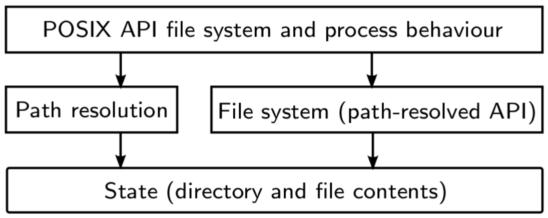

POSIX API file system and process behaviour
Path resolution
File system (path-resolved API)
State (directory and file contents)

Figure 2.2: Modular structure of the model

A file system has to maintain the directory structure and the contents of files, typically using references. This is managed by the *state module*. The *path resolution module* defines how paths reference particular files and directories. The *file system module* represents the bulk of the model. It describes how each command (`link`, `rename`, etc.) behaves, including how they modify the state, and the many possible error cases, but working over fully resolved paths. Finally, the *POSIX API module* glues those together and includes the behaviour of libc and the operating system, introducing the notion of a process, and per-process data structures. This is the top-level module which exposes the interface used for trace checking. The model's module structure does not represent the structure of the existing POSIX specification or the internal structure of any file system implementation; rather, it is the result of an attempt to identify the conceptually key components and their interfaces, while simultaneously minimizing the overall complexity of the model. An important decision was to separate path resolution from the semantics of each command. When processing a command such as `rename p1 p2`, the POSIX API module first resolves the paths p1 and p2 to obtain two resolved paths. These are then used when invoking the file system module equivalent of the rename function. Thus, internally to the model, the file system module API is expressed in terms of resolved paths, not raw strings. This means that the file system model is clean, and unpolluted by the tricky details of path resolution which have been confined in a separate module. Because they have pure value-passing interfaces, modules can be considered in isolation, allowing important invariants to be established. Modularization also allows unit testing of individual modules, which has been useful particularly to get the details of path resolution correct.

Traits Aspects of the model that cut across the modular structure but which are conceptually distinct have been isolated using a trait-like mechanism: there is a *core* model on top of which the user can mix in further traits for particular functionality. The *permissions* trait defines the behaviour of file permissions including functions such as umask. The *timestamps* trait defines how the timestamp information on files is updated, as we will see in Chapter 3.

Monads and combinators Higher-order logic is based on the notion of (pure) functions. Furthermore, various functional programming structuring techniques are introduced in the model, including *monads* and associated *combinators*, to give a uniform structure to definitions. The following specification excerpt illustrates the use of the parallel combinator ||| to specify the checks that the rename function must perform.

```
let fsop_rename_checks ... = ...
 if (fsop_rename_same_rsrc_rdst env rsrc rdst s0) then
   fsm_do_nothing
 else
```

```
( fsop_rename_checks_rsrc_rdst env rsrc rdst
  ||| fsop_rename_checks_root env rsrc
  ||| fsop_rename_checks_subdir env rsrc rdst
  ||| fsop_rename_checks_parentdirs env rsrc rdst
  ||| fsop_rename_checks_perms env rsrc rdst )
```

The conditional first checks whether the source and destination are the same, in which case the rename is a no-op, and the checks do nothing. Otherwise the `rename` function needs to check various conditions: *fsop_rename_rsrc_rdst* checks various combinations of the source and destination that result in errors (for example, `ENOENT` may be raised if the source is missing); *fsop_rename_checks_root* checks attempts to rename the root directory; *fsop_rename_checks_subdir* checks attempts to rename a directory to a subdirectory of itself; *fsop_rename_checks_parentdirs* checks that the parent of the source and destination directory can be found (this check should always succeed; it is included to cover the case that a disconnected file or directory is involved in the rename); *fsop_rename_checks_perms* checks the permissions involved in the rename. Each of these checks may raise many different errors. Moreover, as discussed in §2.3.1, and unlike an implementation, we have to loosely specify the behaviour: any error that arises from any of the checks is valid behaviour. The parallel combinator conceptually allows these checks to be carried out in parallel, and the resulting error may be from any of the individual checks. The shown excerpt concisely and readably expresses *all* the checks involved, and the use of the parallel combinator emphasizes that none of the errors arising from the individual checks has priority over any of the others. The precision and clarity of this model can make it a useful complement to the existing POSIX standard.

2.4 Model

Our model is about 6 000 lines of higher-order logic. In Figure 2.3 we can see the line count for each part of the model.

In this chapter we give only the main types involved, and representative excerpts

Main modules	loc	Others	loc
State	502	Prelude	156
Path resolution	291	Types	888
File system	1388	Monads	130
POSIX API	818	Permissions	208
		Formal properties	1103
		Support files	497
	Total	5981	

Figure 2.3: The model, non-comment lines of specification

from the model to make the discussion manageable.[3] We start by introducing the notion of a labeled transition system: a mathematical way to specify complex real-world systems. We then explore the main modules that make up our model, following Figure 2.2.

2.4.1 Labeled transition systems

Conceptually, SibylFS simply defines a labeled transition system: a non-deterministic infinite-state automaton where the states are abstract (model) file system states and the transitions (mostly) correspond to libc API calls and returns, and are labeled with the call parameter values and return values.

Formally, an LTS can be thought of as a tuple (S, L, S_0, R), where S is a set of states, L is a set of labels, $S_0 \subseteq S$ is a set of start states, and $R \subseteq S \times L \times S$ is a set of triples (known as the transition relation): a triple (s, lbl, s') indicates that, from state s, a transition labeled with lbl to state s' is possible.

We are interested in the subset of LTS that is finite.

Finite LTS S_0 is finite and $\forall s \in S.\{(s,l,s')|(s,l,s') \leftarrow T\}$ is finite.

In other words, an LTS is finite when from every state it can only transition to a finite number of states.

LTS example States have only a counter.

L is $\{Step(n)\}$.

T is $\{< counter = n > \xrightarrow{Step(n)} < counter = n + 1 >\}$.

[3]The interested reader can find explore the model at http://sibylfs.github.io/

S_0 is $\{< counter = 0 >\}$.

An example of trace for this LTS is $\xrightarrow{Step(0)}\xrightarrow{Step(1)}\ldots\xrightarrow{Step(n)}$.

2.4.2 POSIX API module

The POSIX API module defines a labeled transition system. Labels correspond to relevant events: those for a process *calling* a libc function, a value being *returned* to a process from a call, process *creation* and *destruction*, and internal τ events. These are modeled using the Lem type `os_label`.

```
type os_label =
| OS_CALL of (ty_pid * ty_os_command)
| OS_RETURN of (ty_pid * error_or_value ret_value)
| OS_CREATE of (ty_pid * uid * gid)
| OS_DESTROY of ty_pid
| OS_TAU
```

This defines a new datatype (similar to a tagged union or variant type) where values may be one of the five possible variants, distinguished by constructors such as `OS_CALL`, and each holding an immutable tuple of the associated type. For example, if the value *pid* is of type `ty_pid` (representing a process id) and *c* is of type `ty_os_command` (representing a particular instance of a libc function call such as `link`), then `OS_CALL(` *pid* , *c* `)` is a value of type `os_label` (representing the event where process *pid* makes a libc call *c*). The type `ty_os_command` (used in the `OS_CALL` constructor) models the various libc functions and their arguments:

```
type ty_os_command =
| OS_CLOSE of ty_fd
| OS_LINK of (cstring * cstring)
| ...
```

The states of the model must represent real-world system states, including processes, open file descriptors, file descriptions and so on. The key type of model states, `ty_os_state`, is a Lem record type:

```
type ty_os_state ... = <
oss_fid_table : fmap ty_fid (fid state 'dir_ref 'file_ref);
oss_group_table : fmap gid (finset uid);
```

```
oss_pid_table : fmap ty_pid (per_process_state 'dir_ref);
...>
```

The field *oss_fid_table* is a finite map from open file description references (`ty_fid`) to the state of the file description (*fid_state 'dir_ref 'file_ref*); here the pre-primed identifiers are generic type variables, and `fid_state` is actually a type constructor parameterized on arbitrary *'dir_ref* and *'file_ref* types. The field *oss_group_table* is the mapping from group ids to (sets of) user ids. The field *oss_pid_table* holds the per-process information tracked by the operating system. This includes the current working directory, file descriptors and directory handles, process run state, and various permissions-related state, such as the file creation mask, and the real and effective user ids. We have now defined the states and the labels of our LTS. For the transition relation one might expect a relational definition, specifying a set of triples (s, lbl, s'). In this context we prefer a definition that is more computationally convenient, but mathematically equivalent: a function that takes a state and a label, and returns a (finite) set of states. Indeed, we have a top-level function `os_trans` with that type:

```
val os_trans : ty_os_state -> os_label -> finset os_state_or_special
```

There is a subtlety here: the type `finset os_state_or_special` represents a finite set of elements, which are *either* normal states, *or* special states which correspond to POSIX undefined, unspecified and implementation-defined behaviours, as described in §3.3.3. If we ignore special states, the result type indeed represents a set of file system states.

The remainder of the model defines the transition relation: given a state, and a label corresponding to a libc function call, the definition of *os_trans* uses the path resolution module to resolve paths, and then calls the file system module to process the function itself. In addition to this, *os_trans* must deal with processes and concurrency, open file descriptors, file descriptions and so on.

A trace such as that shown in Chapter 1 is a *sequence of labels*. SibylFS checks a trace step by step. At each step i of the trace, SibylFS maintains a finite set S_i of values of type `ty_os_state`, which represents all the states that the real-world file system might be in. For each label lbl_i, SibylFS applies *os_trans* to each element of S_i, and takes a union of the resulting sets to form the set of values S_{i+1} at the next step. The initial set S_0 consists of a single state s_0 representing an empty file system. In effect, given S_0 and the sequence of labels, SibylFS computes a sequence $S_0 \xrightarrow{lbl_1} S_1 \xrightarrow{lbl_2} S_2 \ldots$. If the end of the trace is reached at lbl_n and the set S_n is non-empty, then the trace is *accepted* by the model. If the set S_i of possible file system states at step i is ever the empty set, then this indicates that the trace is *not accepted* by the model.

Path resolution module Path resolution is complicated for several reasons. The resolution of even simple paths (no symlinks, no permissions) can be counter-intuitive on real-world systems, particularly when the path ends in a trailing slash e.g., the path /tmp/f.txt/ is sometimes resolved successfully under Linux, even when f.txt is a non-directory file. Symlinks introduce much additional complexity. For example, symlinks that occur as the last component of a path are sometimes followed and sometimes not, depending on the libc function involved and flags such as those for open; this "follow last symlink" behaviour is further complicated by trailing slashes on the path (a trailing slash makes it more likely the symlink is followed). Permissions further complicate matters. For example, there is the question of how permissions interact with path resolution, and what permissions should be assigned to symlinks.

Our model clearly describes the behaviour of path resolution in terms of the inputs to path resolution, and the output resolved path. The result of path resolution is captured by the *resolved name* type res_name:

```
type res_name 'dir_ref 'file_ref =
| RN_dir of ('dir_ref * ...)
| RN_file of ('dir_ref * name * 'file_ref * ...)
| RN_none of ('dir_ref * name * ...)
| RN_error of (error * ...)
```

Intuitively path resolution can give four possible results: the path can resolve to a directory (constructor RN_dir), a non-directory file (RN_file), or an error can occur during resolution (RN_error), or the path might resolve to "none" (RN_none), representing a non-existent entry in a directory. This last possibility occurs, for example, for functions such as mkdir, where the given path is intended to reference a non-existing entry that will be created by the function.

File system module The file system module defines the behaviour of individual functions such as link and rename. Internal to the model, its API is expressed using resolved names. In §2.3.2 we saw an excerpt from the file system module: the checks that the `rename` command must make.

State module The state module provides a simple model of directory and file contents. The main type is a record type which includes a field *dhs_dirs* (a finite map from directory references to directories) and a field *dhs_files* (a finite map from file references to files):

```
type dir_heap_state_fs = <
dhs_dirs : fmap dh_dir ref_dh_dir;
dhs_files : fmap dh_file ref_dh_file;
... >
```

The interface to the state model is expressed in terms of *references* to files and directories (types `dh_dir_ref` and `dh_file_ref`). The state-model API permits arbitrary linking and unlinking, in particular, our model can handle directory links, and disconnected files and directories can also be modeled (a disconnected file is one that does not appear in the directory tree, but is still accessible).

Contrasting this to the block-structured storage state one might find in a typical file system implementation is instructive: the model can abstract from all the implementation detail while still correctly describing the envelope of allowed behaviour visible at the API we consider, and that abstraction is essential to make the model simple.

2.4.3 Model validity

SibylFS formalizes a subset of the POSIX specification. It also formalizes the behavior of other implementations when these do not respect POSIX definitions.

We can automatically validate that SibylFS models an implementation behavior by using it as a test oracle: if the implementation produces results expected by SibylFS, then SibylFS models the implementation correctly §2.3.2.

However, validate that SibylFS models POSIX is more challenging: although many operating systems claim to be POSIX compliant, the complexity and strictness of POSIX make it possible that they are not §2.6.

In this work two approaches minimize the possibility that SibylFS deviates from POSIX:

- each SibylFS definition matches a POSIX definition:

  most of SibylFS definitions are annotated to record the decision process behind the definition. For example, the annotation

  ```
  (* posix/mkdir.md ENOENT:4; probable POSIX spec error;
    confirmed on austin group mailing list 2014-06-16
    by Geoff Clare *)
  ```

  refers to the definition that POSIX gives for the `mkdir` in case of `ENOENT` error. This example also shows how the development of SibylFS improved the original POSIX model as a side effect. In general definitions are named to match POSIX terms, so that the interested reader can look up the POSIX specification. This approach to validity is similar to the one used in [10], where the authors annotate formal definitions to show the relation between formal and informal specifications.

- SibylFS is used as a test oracle against multiple implementations which are inspired by the POSIX standard:

  assuming that file systems deviate only slightly from POSIX, most of their behavior should be POSIX compliant. The intersection of these behaviors can help validating that SibylFS models POSIX. If SibylFS accepts the intersection of behaviors, it is POSIX compliant (for at least that set of behaviors). Since SibylFS can be used as a test oracle, at each test run deviations were carefully evaluated, and or the model was updated or a deviation was identified for the file system.

Although these cannot prove that SibylFS is equivalent to the subset of POSIX definitions relevant to the file system, in conjunction they simplify the detection of deviations.

2.5 Test suite and harness

In §2.1 we gave an overview of the system, and described the virtuous circle formed by testing and revising the model. We now examine the tests and test execution in more detail.

2.5.1 The tests

Autogenerated scripts test commands such as `link` and `rename` where combinatorial testing is straightforward, feasible, and expected to cover all static real-world behaviour. The combinatorial nature of the tests means that functions such as `link` and `rename` which take two arguments have many more tests than functions such as `rmdir` which take only one. The `open` function has an especially large number of tests because one argument is a bitfield of open flags.

To reduce the test cases to a finite number, we use equivalence partitioning, which requires identifying classes of inputs where a function is assumed to behave "the same", and testing only one member of each class. For example, in a given file system state where neither `f1` nor `f2` exist, the behaviour of `rename f1 f1` should be the same as `rename f2 f2`, so it suffices to test only one of these two possibilities: the assumption is that the exact name of a file is irrelevant. A potential weakness is that these assumptions might not actually hold for real-world file systems. For example, even if neither `f1` nor `.snapshot` exist, it could be that any reference to `.snapshot` triggers unusual file system behaviour so that `rename .snapshot .snapshot` behaves differently to `rename f1 f1`. The proposed tests would typically fail to establish this difference. However, this is an inherent weakness in equivalence partitioning, not specific to the presented use.

The equivalence classes are based on properties (of file system state, and the file system API calls) which we believe affect file system behaviour. For example, properties of paths used in API calls include: whether the path ends in a slash; whether it starts with 0, 1, 2, or ≥ 3 slashes; whether it is the empty string; whether it is a single slash; the type of the resolved path (file, directory, symlink, nonexistent, error); if the resolved path is a directory, then the number of entries in the directory; and whether the path has a symlink component or not. These properties are used to construct equivalence classes. We then make sure that we have *at least one test case* for each logically-possible combination of properties. For API calls involving two paths (such as rename) we consider all combinations of properties of each path individually, together with equivalence classes based on properties of two paths: whether they are equal or not; whether they are different paths to the same file (hard links); and whether one path is a proper prefix of the other. Again, we need to ensure we have at least one test case for each logically-possible combination of properties.

The construction of equivalence classes is carried out manually to achieve the smallest possible set of useful tests: extensive human involvement is necessary to determine which combinations of properties are logically possible. For example, it makes no sense to require that a path corresponds to an empty directory, and is at the same time a proper prefix of a path that corresponds to a file (or directory or symlink). Potentially the model itself could be used to determine that certain combinations are not possible. Maybe this could be done automatically, but would require a significant proof effort for each combination, and there are many logically-impossible combinations. We chose to leave the automatic generation of equivalence classes as future work. Even if we could automatically determine whether a combination was logically possible, constructing missing test cases requires human involvement (see below). Instead of using the model, we should manually inspect each combination for which no test case is available, to certify that the combination is indeed impossible. This takes significant effort, and there is the danger that the human mistakenly labels some combination as impossible, and thereby omits an interesting test case. If a combination is possible, but no test case exists, we manually examine the combination, identify (at least one) missing test case, and extend our automatic test generation to include this case. As an example of the missing test cases our approach uncovered, for commands involving a single path, our test suite initially lacked test cases which resulted in a path resolution error, where the error was not due to a trailing slash on the end of a file. The fix was to include commands that attempt to resolve a nonexistent file *in a nonexistent directory*; a nonexistent file (in an existing directory) does not suffice since it resolves to `RN_none` rather than `RN_error`. OCaml was used to model properties and equivalence classes, and mechanically verify that all logically-possible combinations were matched by at least one test case. The OCaml equivalence classes are defined as sum types which may look like:

```ocaml
type single_path_props =
  ...
  | Is_empty_string
  | Is_slash
  ...
```

These types are then interpreted to predicates:

```
let rec sem1 prop p0 = (
  match p0 with
  | None -> false (* check for null path *)
  | Some p -> (
      match prop with
      ...
      | Is_empty_string -> (p = [""]) (* path was "" *)
      | Is_slash -> (p = ["";""])
...
```

These predicates are used to construct test inputs from predefined partial inputs which may look like:

```
...
let other_paths = [
 (* must have nonexist paths, and some must end in / *)
 ["nonexist_1"]; ["";"nonexist_1";"nonexist_11"]; ["";"nonexist_2"]; ["";"nonexist_2
 (* we also need nonexistent path in an empty dir *)
 ["empty_dir1";"nonexist_3"];
...
```

For commands such as `read` and `write` we need to test sequences of calls, which is inherently hard to test combinatorially. Extensive manual tests were written, attempting to cover all possible behaviours. Preliminary investigation of automated generation of tests was done for these calls but this is future work. An alternative is to use randomized testing.

The standard Open Group POSIX test suite includes hand-written code to check the results of calling libc functions. The use of combinatorial testing, made possible by the SibylFS oracle, allows us to test many more cases: 2 500 autogenerated scripts for `rename` alone, supplemented by further hand-written scripts, whereas the Open Group test suite for `rename` includes around 50 tests. On the other hand, they test a wide range of POSIX functionality, whereas here we test file system functions only.

Testing, interleaving, concurrency and races

The SibylFS model allows interleaving and concurrent behaviours and many test scripts involve multiple processes making interleaved libc calls. However, in-kernel racy behaviours are inherently difficult to elicit from real-world systems, and such racy behaviour, although modeled, is not currently tested. In this section we examine the nature of the interleaving and concurrent behaviours allowed by the SibylFS oracle, and the difficulty in testing racy behaviours.

A file system API function call and return is not modeled as an atomic event. Instead, there is an initial event corresponding to the call, a second internal τ event corresponding to the libc/OS/file system processing the call, and a final event corresponding to the return from the libc call. Additionally, the model satisfies a receptivity property: at any time, any running process can make a libc call, at which point the process blocks until the call returns. This model allows multiple processes to execute calls concurrently. Test scripts can involve multiple processes, each making calls to libc. For example, many of the hand-written test scripts involve multiple processes making interleaved calls to libc, in order to test file system features such as ownership and permissions. The test infrastructure will execute a script line-by-line and no attempt is made to execute calls from different processes at the same time: typically a libc call from one process will complete before the test infrastructure executes a call from another process.

It should be possible to extend the test infrastructure to initiate libc calls from different processes simultaneously, perhaps by assigning different processes to different cores. This would at least make it *possible* for concurrent calls to race in the kernel, but the probability of a race actually occurring would likely be very low (there is no way to force calls to race in the kernel). The next step would be to run such potentially-racy tests many times, to try to increase the chance of racy behaviour being observed. However, the time cost of doing this for a large test suite such as the presented one is prohibitive, and such racy testing should probably be restricted to particular test scenarios where the racy behaviour is expected to be "interesting".

2.5.2 Script execution

Test scripts may involve multiple processes making libc file system calls. Each test script execution forks an interpreter process from the controller process to provide signal and fault isolation. The interpreter process then reads, parses, and dispatches script commands over a high-fd UNIX socket to worker processes running in a `chroot` jail. Each worker runs with real user and group IDs and supplementary group IDs generated to match the permissions relations for the corresponding process in the script. Using `chroot` jails means that we can effectively test as if the file system namespace is empty. This design trades off complete accuracy regarding the behaviour of the root directory (e.g., in a `chroot` jail the root directory link count is typically off-by-one compared to a non-chroot setup), for fast, reliable execution.

2.6 Evaluation and test results

Testing focused on the Linux and OS X operating systems for which we have models. On Linux, we tested tmpfs, Btrfs, ext2, ext3, ext4, F2FS, XFS, HFS+, MINIX, NILFS2, NFSv3/tmpfs, NFSv4/tmpfs, fusexmp/tmpfs (the example FUSE pass-through backed by tmpfs), SSHFS/tmpfs, bind/tmpfs, posixovl/vfat, posixovl/NTFS-3G, aufs/tmpfs/ext4, overlay/tmpfs/ext4, GlusterFS/XFS, and OpenZFS. On OS X, we tested HFS+, NFSv3/HFS+, fusexmp/HFS+, SSHFS/HFS+, fuse-ext2, Paragon ExtFS, and OpenZFS. In addition, on Linux we compared the standard libc (glibc) and the lightweight libc musl, and kernels 3.13, 3.14, and 3.19.

An individual test run currently executes 21 070 tests and produces 46MB of trace data. Because manually analysing system traces of this volume is difficult, we index, filter, and highlight specification deviations in HTML. These tools can also produce merged test runs comparing local specification deviations across multiple platforms with platform differences identified and highlighted. With appropriate experimental design, OS, file system, and libc defects are easy to find.

2.6.1 Performance

To use our specification during file system development or behavioural exploration, individual script execution and trace checking must run quickly. As described in §2.3.1, non-determinism can, without careful management, lead to very long run times. In the checking system, the specification is engineered to control non-determinism and take advantage of trace independence for parallel speedup.

Trace checking the entire test suite with four processes on a machine running Linux 3.14-2 with an Intel Core i7-3520M 2.90GHz CPU with performance governor, Samsung 840 PRO SSD, and 12GB RAM takes about 79s, which is a mean rate of

266 test traces per second. With test suite execution on Linux tmpfs clocking in at 152s, it takes less time to check a trace set than it does to execute the test suite. Our naive single-threaded HTML generator takes about 48s to process a single, unmerged test run. Thus, it seems that the performance of SibylFS is suitable for use during development and continuous integration. The slowest phase of testing is due to user and group creation: we need to employ application-level locking to avoid race conditions on Linux, OS X, and FreeBSD; these race conditions have been reported upstream. The test harness and the checker architecture are not aggressively optimized: for example, a new process is spawned for each trace being checked.

2.6.2 Test results

Trace acceptance

For the "standard" Linux platforms (Linux 3.19, with glibc and either ext2, ext3, or ext4), all but nine of 21 070 traces are accepted by SibylFS. The nine failures are mostly due to the use of a `chroot` jail for testing, i.e., they do not represent real deviations of the underlying file system from the SibylFS model. For other Linux platform variations, failing traces differ mostly in aspects that POSIX indicates are implementation-defined or unspecified. These include default permissions for symlinks, writing zero bytes to bad file descriptors, and specific errors due to removal or renaming of the root directory. On OS X 10.9.5 with the default HFS+ file system, the script which tests `pwrite` with a negative offset fails to execute to completion due to an integer underflow bug in OS X. In total, 34 traces fail to check, due to a handful of issues similar to the Linux failures, and the resolution of symlinks with trailing slashes. The FreeBSD results are similar.

Test coverage

To understand the completeness of the test suite, we can use as a measure the proportion of lines in the model that are covered by a test run. The ideal target of 100% coverage is not possible for two reasons: some lines of the model correspond to situations that are impossible to reach, and clauses for particular platform behaviour will not be exercised when checking traces for another platform. Taking these factors into account, our tests currently cover 98% of the model. The remaining 2% consist of lines that probably could be tested (for example, test process destruction during a test is not currently exercised), and lines for unused internal definitions generated by Lem (which should be excluded from the analysis). The high level of coverage is partially attributable to the decision to use automatically generated test cases in an attempt to exhaustively explore all behaviours. Related work [28] has used randomized testing of the POSIX file system interface applied to a novel file system implementation, achieving 89.06% coverage of the implementation code. If one considers the model as a (non-deterministic) reference implementation, there is a sense in which these figures are comparable.

The coverage figures come with a caveat: these figures show only that the test scripts produce traces whose checking exercises almost all of the model. As noted earlier, it may still be the case that the assumptions underlying equivalence partitioning are invalid or that some real-world behaviour, unrepresented in the model, is not being tested.

The tests aim for complete implementation code coverage, but consider coverage of the model, rather than coverage of the implementations, for two main reasons. First, there is the belief that the model is detailed and accurate (although admittedly this belief partly depends on the testing itself), so that tests which cover the model should also exercise all interesting behaviours of the implementations. Second, attempting to measure implementation coverage is difficult. There are at least three distinct pieces of code which form an implementation (the libc library, the OS, and the file system implementation code), and only parts of each piece are in the domain of the SibylFS model. In order to measure implementation coverage, we would first need to determine, for each of the three pieces, which lines of code are relevant, and which are not, so that we can restrict our coverage checking to the relevant lines. This requires expert knowledge of libc, the OS and the file system code, but should be possible for a single test platform, and there is belief that this would provide further evidence that the tests provide high coverage. However, providing such implementation coverage for each of the many combinations of libc, OS and file system that we consider, would surely be infeasible.

2.6.3 Survey results

During the testing of over 40 different system configurations, numerous deviations from the specification ranging from mundane to critical were discovered. We see a classification of file system defects found during this survey by increasing severity.

Issues in the POSIX specification

POSIX specifies the behaviour of each libc function separately, with clauses for common errors duplicated between functions. Almost inevitably it is difficult to keep these clauses in sync when updating the POSIX text, and minor mistakes have crept in. This formal model is in part a formal counterpart to the informal POSIX specification, and clauses in the formal model that were not uniform suggested underlying issues with the POSIX specification. For `link`, `mkdir` and `open` the POSIX specification was queried of the allowable errors on the Austin Group mailing list; new issues were recorded and subsequently resolved on the Austin Group bug tracker [34].

POSIX specification violation

Core behaviour

If we restrict to *successful* invocations of libc functions, for file system states which do not contain symlinks, and paths that do not end in a trailing slash, and if we ignore permissions and work with a single process, then the behaviour across most system configurations is very similar. On some file systems, specific features such as directory link counts are not supported. Btrfs, SSHFS/tmpfs, and Linux HFS+ all exhibit this violation with SSHFS/tmpfs also not supporting link counts for regular files due to limitations in the SFTP protocol.

Error codes

POSIX often allows different errors in a given circumstance, and this looseness is present in implementations: Linux is substantially different from OS X and, even on the same operating system, different file systems can return different errors in the same situation. There are also cases where error codes not allowed by POSIX are returned; for example, Linux follows the LSB for `unlink` of directories and returns `EISDIR`, where OS X follows POSIX and returns `EPERM`. On OS X, when attempting to `rename` the root directory, `EISDIR` is returned instead of `EBUSY` or `EINVAL`.

Path resolution, trailing slashes, and symlinks

Trailing slashes on paths, even without symlinks, are treated in what appears to be an ad hoc manner. For example, if `f.txt` is a path to a file, then `f.txt/` intuitively should result in an error, but often such a path is resolved successfully. For example, on Linux `link /dir/ /f.txt/` can return `EEXIST` to indicate that the file `f.txt` exists (this is not allowed by POSIX), whereas one might expect `ENOTDIR` to indicate that the path `/f.txt/` cannot be resolved because `f.txt` is not a directory. Symlinks introduce further complications. For example, a path to a symlink followed by a trailing slash is often used to mean "resolve to the target of the symlink (even if a file)", but this is not universally followed on either Linux or OS X. The behaviour when symlinks to symlinks are involved can be confusing. For example, if `s1` is a symlink to a directory, and `s2` is a symlink to `s1`, then on OS X, `readlink s2/` will return the contents of the symlink `s1`, whereas one might expect that the trailing slash would force the path to be resolved to the directory, resulting in an `EINVAL` error returned by `readlink`. Creation of hard links to symlinks using `link` is permitted by Linux and support is specified as implementation-defined. Notably, HFS+ on Linux returns `EPERM` when this is attempted rather than either linking the symlink or following the symlink as OS X does. This behaviour is likely a portability compromise for removable volumes.

Invariants

POSIX specifies that calling open with flags `O_CREAT`, `O_DIRECTORY` and `O_EXCL` on a symlink to an existing directory should fail with `EEXIST`. FreeBSD instead returns `ENOTDIR`. POSIX also mandates a strong invariant: a libc call which returns with an error should leave the underlying file system state unchanged. On Linux and OS X this invariant holds for all our tests. However, in the above scenario, as well as returning the `ENOTDIR` error, FreeBSD deletes the symlink and replaces it with a newly created file. This breaks the POSIX invariant. If the symlink points to a non-existent target rather than a directory, and the flag `O_EXCL` is omitted, then the new file is created as the target of the symlink and `ENOTDIR` is returned, again violating the invariant.

Platform conventions

Some platforms, such as Linux, have well-known and longstanding defects in their POSIX compliance. For example, on Linux, calling `pwrite` on a file descriptor opened with `O_APPEND` will ignore the offset and instead append data to the file. It is crucial that any file system or application software ported to or from Linux follows this convention on Linux and provides or expects POSIX compliance on operating systems that attempt POSIX compliance. This specification and development process ensures that we explicitly express and check behaviour of this kind.

Defects likely to cause application failure

A comparison of SSHFS/tmpfs mount options

An organization's system administrator might consider deploying a shared SSHFS/tmpfs mount to their users and wonder what mount options to use in the configuration scripts. With SibylFS, the administrator can easily compare, in under an hour, the behaviour of various mount configurations *in their specific deployment* of SSHFS/tmpfs and conclude that, using *only* `allow_other` is dangerous because it allows users to violate permissions, using `allow_other` *and* `default_permissions` is safer but still is not adequate for a shared mount deployment due to SSHFS/tmpfs's unconfigurable default creation ownership set to the mount owner (root). Additionally, without a mount option umask, a user process's umask is bitwise `ORed` with 0022 (regardless of the parent process's umask) but when setting a mount option umask of 0000, a user process's umask is ignored entirely. Using this empirical evidence, the system administrator is now informed enough to reject SSHFS/tmpfs for this deployment scenario.

OS X VFS pwrite integer underflow and signal

POSIX specifies a negative offset to `pwrite` should return an `EINVAL` error. Premature, potentially unclean process termination has been observed for this simple error condition. An hypothesis is that the OS X VFS layer incorrectly uses an unsigned integer type for the offset argument to `pwrite` which causes negative values to be interpreted as extremely large positive values, and the operating system then sends a `SIGXFSZ` signal to the process which almost certainly does not handle it.

Various issues in deployed but older versions of Linux

In Ubuntu "Trusty" Linux 3.13.0-34, HFS+ did not support `chmod` and would return `EOPNOTSUPP` for every `chmod` call. This was not the case in Debian "sid" Linux 3.14-2. In OpenZFS 0.6.3-2~trusty, also on Ubuntu "Trusty" Linux 3.13.0-34, files opened with `O_APPEND` would not seek to the end of the file before either `write` or `pwrite` potentially resulting in application malfunction and data loss or corruption.

Defects causing a system halt, data loss, or resource exhaustion

**posixovl/VFAT 1.2 storage leak*

posixovl is an overlay file system which provides POSIX functionality on file systems such as VFAT. The test suite revealed that posixovl/VFAT fails to decrement the hard link count correctly in certain rename scenarios. A simple C program to repeatedly create 64MB files with hard links and delete them using rename was written. On Linux 3.14, this resulted in the process receiving a `SEGFAULT`. On Linux 3.19, we found that the `open` with `O_CREAT` libc call would fail with `ENOENT`. In both cases, the file system would have no remaining space despite being empty - even through an unmount cycle.

OpenZFS on OS X unkillably spins processes in a disconnected directory case

OpenZFS 1.3.0 on OS X 10.9.5 has a defect which, after executing the following sequence of function calls,

```
mkdir("deserted",0700);
chdir("deserted");
rmdir("../deserted");
open("party",O_CREAT | O_RDONLY,0600);
```

causes the calling process to consume 100% CPU and ignore all signals. The file system is still usable by other processes at this time but the OpenZFS volume cannot be unmounted and the machine cannot be shutdown. Force unmounting the OpenZFS volume may succeed and release the storage device or may cause the storage device to become unusable until the next restart.

2.7 Related work

2.7.1 Model checking

Yang et al. [78] have used model checking to find serious file system errors. Their FiSC tool included a simple model of file system state (name, size and link count for files and directories), sufficient for finding errors, but not intended as a realistic model of file systems in the way that SibylFS is. FiSC requires intrusive access to file system internal state: ReiserFS took between one and two weeks of effort to run in FiSC as it violated one of the larger assumptions we made . In contrast, SibylFS tests file systems solely via the libc interface, making it trivial to test new file systems. FiSC is also focused on errors typically arising from host crashes.

SibylFS does not currently model such scenarios at all.

2.7.2 Ad hoc models

FiSC includes a simplified, ad hoc file system model. Such models are reasonably common. For example, the COMMUTER tool [14] includes a model expressed in a symbolic variant of Python. The model is simplified, e.g., filenames have no structure and can only be compared for equality, and there is no support for symlinks. Even these simplified models can take significant time to develop, and are typically not reused across projects. There is reason to believe that the SibylFS model is more detailed and better validated, and hopefully it will be reused in place of such ad hoc models in future file system research projects. The COMMUTER project is similar to this work in other respects. They use equivalence partitioning to ensure only a finite number of tests are generated, which nevertheless "cover all possible paths and data structure access patterns in the model". As with this work, they focus on coverage of the model, rather than implementation code. Moreover, their tests are somewhat simplified because, in addition to the model simplifications described above, their test cases do not deal with directories (other than the root directory).

2.7.3 Differential testing

Differential testing compares the behaviour of multiple implementations to identify possible errors without a reference model [52]. In some cases it can be very effective, e.g., for C compilers [79]: by restricting the domain to C programs that (according the C standard) should be deterministic, any behavioural difference in compiled programs identifies a compiler bug. File systems are more complicated to test because of non-determinism, with a large envelope of allowable behaviours within which file systems are expected to behave differently, so one cannot simply compare runtime behaviours without a reference model that identifies when they are sufficiently similar. SibylFS instead allows differential testing of multiple file systems taking this allowable variability into account. In this sense it improves on differential testing, but the downside is the effort needed to construct the model. Differential testing has also been applied to a novel file system implementation [28] to ensure it behaved the same as a reference implementation (tmpfs on Linux).

The paper also applied randomized testing to file systems, a low-cost alternative (that SibylFS also supports) to the model checking approach described earlier. SibylFS can also be used as a reference implementation by determining the model (selecting one of the many possible states at each step) and previous versions of SibylFS have worked as prototype FUSE file systems under Linux. The good performance of the SibylFS test oracle should also make it feasible to integrate with dynamic verification engines such as EnvyFS [6] or Recon [25].

2.7.4 Formal methods

Previous models of file systems [24, 53] do not aim to capture the full complexity of POSIX or real world file systems. As a result they are usually much simpler than the proposed model: symlinks, permissions and timestamps are ignored, and there is no model of concurrent processes and per-process data structures. Recently Schierl et al. gave an abstract specification of a single file system: UBIFS [69]. However, this is not a general model of POSIX. Work on verified implementations is complementary to this work: it should be possible to prove that a verified implementation behaves according to our model. Implementations of file systems have previously been formally verified [16, 17, 22, 24, 31, 38], but these are highly idealized and do not represent realistic file system implementations. The seL4 team previously produced a verified operating system [40], and some of the researchers are now working on a formally verified file system implementation [39]. Another approach [12] uses a modified Hoare logic inside the Coq theorem prover to attempt to prove correctness of a novel file system implementation. The specification is based on POSIX, but does not attempt to deal with the full variability allowed by POSIX and real-world implementations, since the focus is on a single verified implementation. The authors note, "we found that significant care is needed when writing specifications [. . .] it is easy to write an incomplete specification that does not eliminate the possibility of some bugs". As with other ad hoc models, SibylFS could be used as an alternative, high-quality specification. Recently Ernst et al. [20, 58, 68] achieved a significant milestone by producing a verified implementation based on UBIFS that is actually usable as a flash file system.

Preliminary work on specifying the semantics of storage stacks in Isabelle/Isar has been carried out by Alagappan et al. [1]. The researchers argue for expressive logics to capture specifications of each layer, and theorem prover support for proving that file system stacks satisfy the desired guarantees. The researchers list "obtaining specifications" as one of the main challenges. At least for the uppermost layer that is exposed to the application, SibylFS can provide such a specification. For reasoning about POSIX file system behaviour, Gardner et al. [26] proposed a variation of separation logic. The SibylFS model could be used as a basis to prove soundness for this logic.

2.7.5 File system innovation

Recent studies have shown that the workloads imposed on POSIX file systems now vary widely [30], and there are also many new FUSE-based file systems such as Ori [50], OptFS [13], and kernel-based ones such as Betrfs [36] and ReconFS [48] that optimize particular use cases. File system evolution in this style often results in subtle semantic and data corruption bugs [43], and SibylFS is the first rigorous specification that can be used, in a developer-friendly way, to test directly that these implementations remain POSIX compliant.

3 SibylFS extended with timestamps

3.1 Overview

SibylFS, as described in the previous chapter, does not model timestamps. File systems store three timestamps associated to each file and directory. SibylFS does not model these. This chapter starts with a preliminary section that describes abstractly the approach chosen in modeling time, and the challenges it entails §3.2. Then we introduce why timestamps are useful and we state the definitions of POSIX relevant to timestamps §3.3. We then address the technical challenges in modeling timestamps §3.4, and proposes a model of POSIX timestamps that addresses these challenges §3.5. Finally we present how to use the extended specification for testing purposes §3.6, and its limitations §3.7.

3.2 Prelude

In this section we introduce the reader to the complexity of checking timestamps consistency in an operational semantics setting. We also show how one can mitigate this complexity by abstracting the notion of time, and how this can be leveraged to test the timestamps of system calls traces. Subsequent sections will use and expand the concepts introduced here.

3.2.1 Naive trace checking algorithm

Given a trace we can check if a given labeled state transition system (LTS §2.4.1) can produce it. Here we provide a naive algorithm to do so.

Algorithm Maintain a set R of reachable states.

> R is initially S_0.
>
> For an observed trace $\xrightarrow{l_0}\xrightarrow{l_1}\xrightarrow{l_2}$ ... $\xrightarrow{l_n}$.
>
> Compute $R_0 = S_0$; $R_1 = T(R_0, l_0)$ where $T(X, l)$ is the set of states S' reachable by the l transition from $s \in X$; $R_2 = T(R_1, l_1)$ etc...

Definition A trace is **valid** iff R_{n+1} is non-empty.

Example $\xrightarrow{Step(0)}\xrightarrow{Step(1)}$ ... $\xrightarrow{Step(n)}$ is valid:

$R_0 = \{< counter = 0 >\}$; $R_1 = \{< counter = 1 >\}$; $R_2 = \{< counter = 2 >\}$; $R_3 = \{< counter = 3 >\}$; etc…

3.2.2 State space explosion

This algorithm may become expensive in a computational setting. Indeed, R_i may become large.

Example $T = \{< counter = n > \xrightarrow{Step(n)} < counter = m >\}$, where $n \leq m \leq 2^n$. The algorithm can still check in this scenario, but the state space becomes large.

3.2.3 Mitigation through symbolic state

A way to solve the state space explosion problem of the algorithm presented earlier is to use a symbolic representation.

In our previous examples where the state is a counter, symbolic representation means to work with a single state $< count = n >$, which represents all $< counter = m >$ where $m \leq 2^n$.

Example We show how the representation of transitions changes with the symbolic representation.

Previous Transitions were $< counter = n > \xrightarrow{Step(n)} < counter = m >$ where $n \leq m \leq 2^n$

Now $< count = m > \xrightarrow{Step(j)} < count = m + 1 >$ where $0 \leq j \leq 2^m$

With this representation we can use the naive algorithm on this transition system without state space explosion.

In summary, a trace is valid for a transition system iff the same trace is valid for the corresponding symbolic transition system.

3.2.4 Types of transition systems

The following diagram shows different types of transition systems that variate on how time is constrained.

$$\textit{discrete}:$$

$$\{clock = n\} \xrightarrow{tick} \{clock = n + 1\}$$

$$\textit{non deterministic}:$$

$$n < m.\{clock = n\} \xrightarrow{time\ passes...} \{clock = m\}$$

$$\textit{bounded non deterministic}:$$

$$n < m < n + \delta.\{clock = n\} \xrightarrow{time\ passes...} \{clock = m\}$$

$$\textit{real valued}:$$

$$r_1 < r_2.\{clock = r_1\} \xrightarrow{time\ passes...} \{clock = r_2\}$$

$$\textit{bounded real valued}:$$

$$r_1 < r_2 < r_1 + \delta.\{clock = r_1\} \xrightarrow{time\ passes...} \{clock = r_2\}$$

$$\textit{with internal actions}:$$

$$r_1 < r_2 < r_3 < r_4.$$

$$\{clock = r_1\} \xrightarrow{time\ passes...} \{clock = r_2\} \xrightarrow{\tau} \{clock = r_3\} \xrightarrow{time\ passes...} \{clock = r_4\}$$

In a discrete transition system the time of a transition is a discrete value. In a non-deterministic system, instead, the time of a transition belongs to a infinite range of values whose only lower bound is defined. In a bounded non-deterministic system we constrain this range to not exceed a certain threshold: suppose we have a file system trace of an execution taking a second; we can safely assume that the time of a transition cannot be more than a second. However this kind of assumption only reduces non-determinism. Both real valued and bounded real valued transition systems differ from their discrete counterparts only for the values they handle: real values introduce a far larger non-determinism than discrete values (e.g., natural numbers). Finally, we introduce a transition system which exposes the characteristics of file system traces. In a transition system with internal actions there are transitions, labeled as τ, which are not observable.

Note Observations of the state obtained after a transition may happen or not, according if a command that observes the time details is run. In the case of τ transitions no observation is possible. The lack of observations for transitions prevents the reduction of non-determinism in the system.

In the following table we compare these types of transition system on the number of states obtainable from one of their transitions.

Table 3.1: Number of states obtainable from single transitions

Type of transition system	Possible number of states after transition
Discrete	*1*
Non-deterministic	∞
Bounded non-deterministic	$\delta - 1$
Real valued	∞
Bounded real valued	∞
With internal actions	∞

3.2.5 How a system call executes

A process executes syscalls. The operating system (OS) executes internal actions concurrently with processes.

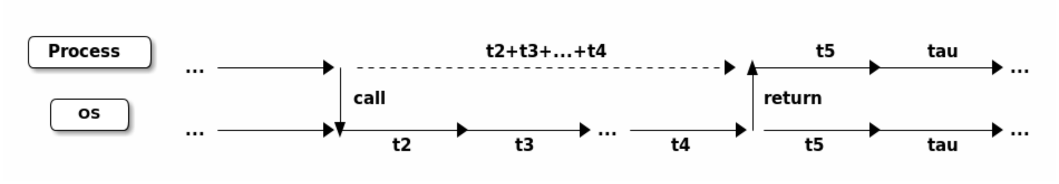


Figure 3.1: Example of syscall execution

Figure 3.1 shows the execution of a syscall from the time perspective of processes and OS: initially the process and the OS are working independently; then the process calls the OS to execute a syscall; the OS performs operations during the time intervals `t1`, `t2` and eventually the syscall returns its output to the caller at the end of `t4`; finally the process and the OS may execute `tau` actions.

An indicative example of real-world recorded trace is:

$$\xrightarrow{call}\xrightarrow{return}$$

In general a trace records only the sequence of calls followed by returns that a syscall causes. In particular we are not seeing any time-passing transition and any internal transition of the process or the OS. This characteristic of recorded traces causes two problems:

1. No time transition observed
2. No internal OS transition observed

For problem (1) we would like to record the time transitions. However for OS internal transitions this would mean to instrument the OS which requires deep understanding of its internals and which is an expensive operation; for process transitions this is difficult as even measuring time involves a syscall, which would change the observed trace.

For problem (2) we can mitigate the problem by requiring that a single internal transition happens between any observed call and return transition. This transition would aggregate any number of internal transitions that happened between call and return in an opaque internal transition. However this model may be not accurate if the OS takes multiple steps and multiple things happen at different times.

3.2.6 Syscalls and file system

The OS has an internal clock. File system's objects get updated with the value of the OS clock. The clock value is not observable until (long) after syscalls return, if at all.

Let's consider as an example the update of the file access time (atime). We write $f[t]$ for a file with atime t.

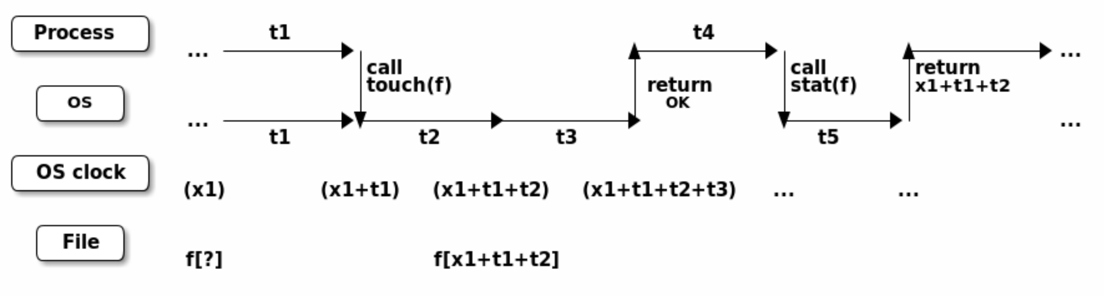


Figure 3.2: Example of syscalls updating and observing file time

In Figure 3.2 we show how the state of the OS clock and of the file's atime change according to the process and OS transitions. These transitions firstly update the file atime with the `touch` syscall and then observe the file atime through the `stat` syscall. As we start we do not know the value of atime, but we determine it after the `stat` syscall.

Note that each non-observed time transition introduces a large (perhaps finite) amount of non-determinism.

3.2.7 Logical time as a set of states

As noted in an earlier example we can solve the huge non-determinism by working with symbolic representations of many states. We call this symbolic representation logical time.

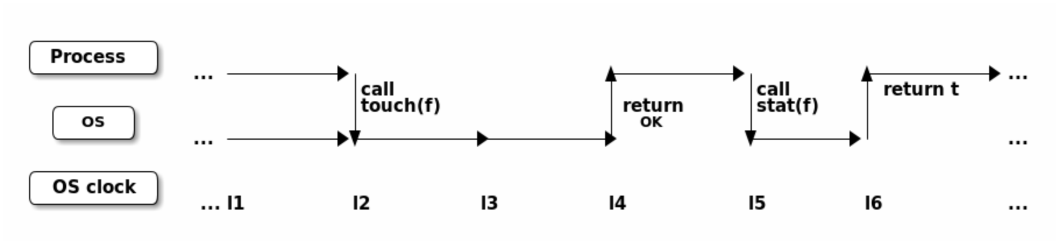


Figure 3.3: Example of syscalls modeled with logical time

Figure 3.3 shows how we can express the example in Figure 3.2 in logical time terms. The main difference is that the OS clock now uses logical times. When the real time t is returned by the `stat` call, we know the value of the logical time `l3`. We also have a constraint on the logical times such that

$$l_1 < l_2 < ... < l_n < ...$$

This constraint is useful as later in the trace we can discover further constraints between logical times and reduce the possible states according to the resolution of this set of constraints.

3.2.8 Logical time and observed times

Sometimes (e.g., `stat` call) the actual value of a logical time can be observed. More generally, we become aware of some constraints on the logical time. In general we have a set of constraints C on the logical time variables. In the trace we took as an example in Figure 3.3 the constraints are of form:

- $l_i = r$: we know the actual time of l_i
- $l_i < l_{i+1}$: wellformed time

In order to check traces we need to determine whether at each point the set C is satisfiable or not. If the set of constraints is not satisfiable then we know that the trace does not conform to the model of the state transition system.

For the forms of constraints we are taking in consideration it suffices to work with a partial map from logical times to real world times. We call this partial map an LP-map.

For example a constraint $l_i = r$ is recorded in the map as $l_i \to \mathit{Some}(r)$.

So an LP-map is a concrete representation of a set of constraints arising from trace checking with symbolic logical times and restricted with the given form of constraints.

3.3 Introduction

This section describes POSIX timestamps and different strategies for updating them §3.3.2, the scope of this study §3.3.3, a naive model for timed transitions §3.3.4 §3.3.5, and why one cannot use such a naive model to model the POSIX standard §3.3.6. Finally it describes how to check whether a real world trace of observed file system interaction involving timestamps conforms to the specification §3.3.7.

3.3.1 Motivation

File timestamps are widely used by applications. For instance the *Make* utility [35] uses these to decide if it needs to update a file or not. Even now timestamps are changing: for example with high resolution timers the developers are improving the accuracy of timestamps interfaces [51]. Higher time resolution is necessary to improve systems based on time, such as user interfaces which become more responsive if they move from a millisecond to a microsecond resolution. Also some file systems update timestamps in a delayed fashion: for example a newly created file can obtain earlier timestamps of its parent directory. An observation of this behavior for the *ext4* file system obtained through SibylFS test harness is in Appendix D. Such an observation required many attempts as the system needs to be heavily loaded to capture this delay, but it motivates the formalization of this behavior. The POSIX specification models this update strategy, which we refer to us periodic strategy. In this chapter we aim to extend SibylFS with a formal model of timestamps. Most of the complexity of this extension derives from modeling the periodic strategy.

3.3.2 Terminology

A *timestamp* represents the unique moment in time when an event occurs. Issues arising from timestamps granularity make timestamps stand for intervals of time.

Timestamp granularity Timestamp granularity indicates how precise the system is at recording times. In general granularity is a monotonic map from real world time to some notion of discrete time, such as the integers or natural numbers. We will call *grain* the unit of discrete time. Supposing such a monotonic map from real numbers to integers, the system would report two events occurring at the same time when their real numbers map to the same integer. So two events occurring separately in time may be recorded as having the same timestamps by the operating system if the distance between them is less than the timestamp granularity. For instance, given an event a happened at time `0.001` and an event b happened at time `0.002` and a grain being one second, one would not know the order of these events, as both a and b would be considered happening at time `0`. Note that when we say that *granularity is one second*, this is an informal statement, as the system clock is not perfectly accurate. The *granularity is one second* means that the size of the set of real numbers which map to the same integer is a second. Naturally, even the concept of second is not accurate as it abstracts over the mechanical clock of the system, which may be imprecise.

The main impact of granularity on the specification is that we may require the timestamps corresponding to two nearby time points to be only less than or equal, while in the real world one may have occurred before the other.

A *file system object* is a file or a directory. The POSIX standard [74] includes three timestamps associated as metadata to each file system object:

```
struct timespec st_atim // Last data access timestamp.
struct timespec st_mtim // Last data modification timestamp.
struct timespec st_ctim // Last file status change timestamp.
```

> [Source: Opengroup Base Defns. sys/stat.h Description section][1]

Each of these records a different type of event:

- the **access time** records the last time the object was read;
- the **modification time** records the last time the object content was written;
- the **change time** records the last time the object's metadata were updated.

A command is a POSIX API call. Calls to the POSIX interface are essentially syscalls. Examples of commands are:

- `mkdir` creates a new directory and alters the timestamps of the parent directory;
- `chmod` changes the object permissions and alters the object timestamps;
- `close` closes a file descriptor for the current process; if the target file is not open by any other process this causes timestamps to be updated;
- `stat` returns the object metadata; all timestamps must have a time value before this command returns.

An update is the association of a time value to a marked timestamp.

Mark for update When a command alters an object timestamp, it does not necessarily set the time immediately; instead the timestamp is marked for update. A mark for update represents a delayed update. A marked timestamp holds a marker for a future value.

There are two update strategies a POSIX-compliant operating system can use:

> An implementation may update timestamps that are marked for update immediately, or it may update such timestamps periodically.[2]

Note that this definition does not specify if a timestamp update should be persisted on disk. In general POSIX focuses on operations working in main memory. Presumably the definition considers writing to disk, as it seems unnecessary to

[1]http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/sys_stat.h.html

[2]Opengroup Base Defns. General Concepts.http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html

distinguish between immediate and periodic updates for inexpensive in-memory updates.

An immediate update strategy indicates that as soon as timestamps are marked by a command, they are also updated. A periodic update strategy means that the marked timestamps of an object may be updated at any time (as required by the operating system), or if one of the following events happens:

1. one of the commands in the `stat` family (i.e., `stat`, `lstat`, `fstat` and `fstatat`) is called on the object;
2. the `close` command is called on a file descriptor referring to an object that is not opened by any other process.

The rationale behind the periodic update strategy is that delaying updates means delaying writing metadata to disk: for instance, a common operation such as listing the content of a directory may force a file system to write the directory metadata to disk, as the access timestamp for that directory has changed. This rationale would be unnecessary if applied to the main memory, as in this context updates are inexpensive. It is not clear if there is any other motivation behind the periodic strategy. The reader may be interested in whether or not periodic updates are observable in the real world. An example of a real world trace that identifies periodic update behavior can be found in Appendix D.

3.3.3 Scope and Limitations

The specification extension discussed in this chapter models exclusively POSIX time for file system operations. The formalized specification can work as a test oracle to check the operating system timestamp implementation agrees with POSIX definitions. It achieves this only by assessing the expected order for a sequence of time values reported in a file system trace of execution: this implies that it is necessary that during the execution, `stat` or `lstat` command are called, as these are the only commands that return timestamps associated to file system objects (without the `*stat` family of commands there is virtually no observable difference between timestamps and specification without timestamps). One could use kernel debugging to observe when timestamps are updated, but this requires a detailed knowledge of file system internals and so it is impractical to analyze the behavior of many file systems.[3]

The extended specification can only be used as a test oracle for small traces. As discussed in §3.5, the main reason is that POSIX allows operating system implementations to delay timestamp updates in order to improve performance. This introduces a large amount of non-determinism into our model of timestamps (as discussed in §3.2). A more detailed examination of this problem is in §3.4 and §3.6.

Non-determinism Non-determinism is the behaviour of a system when its state can transit to multiple and distinct states and it is not decidable in advance which transition will happen. Non-determinism is important because how it is modeled in the specification has a significant impact on testing and the feasibility of testing. In extending SibylFS with timestamps there are multiple sources of non-determinism to consider:

- **Physical time**: the new time value obtained by a timestamp depends strictly on details out of the specification scope (e.g., operating system internals, hardware, physical world effects on the hardware, etc...); the proposed specification deals with this issue both by adding the concept of logical time that maintains the time ordering and by maintaining a map between logical times and physical times.
- **Periodic update**: POSIX provides a timestamp update strategy which allows the operating system to optimize when to assign a value to marked timestamps, but this optimization mechanism of the operating system is out of this specification scope; our timestamps specification abstracts over all possible particular update strategies. So different implementations will have different strategies which should conform to POSIX, and our model should abstract over all of these. A possible reason why the periodic update exists in POSIX may be the problem introduced by the access time: every read of the file's data apparently causes a write. Modern operating systems, such as Linux, allow mount options where the access time is not updated in this fashion; also caching may further alleviate the possible performance implications of this read-caused update.

Since the specification can be used as a test oracle only for small traces (less than

[3]not considering the risk of altering the file system behaviour.

eight commands), it is not possible to verify with great certainty if Linux and Mac OS X implementations of file systems agree with POSIX. This is considered the main flow in the approach we use here. Approaches to improve the specification are discussed in §3.6.2.

3.3.4 Modeling time

Consider the file system executing the following sequence of commands:

```
mkdir p                 // e1
stat p                  // e2
mkdir p/dir             // e3
stat p/dir              // e4
```

Assume that each command associates the current time with the relevant file system object.[4]

If commands are executed sequentially (suppose for example that *time*(*mkdir*(p)) indicates the time at which the *mkdir*(p) command is run), one may require that:

$$\mathit{time}(e_1) < \mathit{time}(e_2) < \mathit{time}(e_3) < \mathit{time}(e_4)$$

and that, therefore, the time data associated to each of the directories respects this order:[5]

$$\mathit{timestamp}(p) < \mathit{timestamp}(p/\mathit{dir})$$

One could validate the model by observing the execution trace of the previous sequence of commands:

```
# The application calls mkdir p
mkdir p
# The operating system completes that call and returns None
# (i.e., no errors encountered)
```

[4]POSIX commands have a start and an end time, rather than considered happening at a single time. We bind commands to a single time to simplify the discussion.

[5]One should consider that two events can be considered happening in the same time grain even if they really happened one after the other if the granularity issue is taken in consideration §3.3.2. In this case the ordering operator will be $\leq$ rather than $<$.

```
 RV_None
# The application calls stat p
stat p
# The operating system returns a stat structure
# with the timestamp associated to p:
# for directory p the timestamps value is 1
 RV_stats {timestamp = 1}
mkdir p/dir
 RV_None
stat p/dir
 RV_stats {timestamp = 2}
```

In this case the model predicts that the sub-directory's timestamp would be updated later than the parent's timestamp. Such a model would reject as invalid the following trace:

```
mkdir p
 RV_None
stat p
 RV_stats {timestamp = 1}
mkdir p/dir
 RV_None
stat p/dir
 RV_stats {timestamp = 0}
```

The time associated with `p/dir` is 0, which is earlier than the time associated with `p`, whereas the sequence of events shows that the `p/dir` was created after `p`.

It might appear that the following trace should be rejected:

```
mkdir p
 RV_None
stat p
 RV_stats {timestamp = 1}
mkdir p/dir
 RV_None
stat p/dir
 RV_stats {timestamp = 1}
```

With the timestamp granularity of one second we may want to accept this trace as the creation of `p/dir` has happened in the same grain of time in which `p` was created.

Note that testing the file system time model against a file system implementation needs to rely on the objects timestamps, since they maintain time information regarding the execution of commands.

3.3.5 Naive model

A naive approach to define a formal time model is to use a transition system. Its transitions are labeled with a command call and the duration of command execution. A transition is defined as $s \xrightarrow[dur]{command} s'$ and s' is the state achieved after running *command* with a transition of duration *dur* from the state s (transition results are ignored to keep the syntax minimal).[6]

Here is an example trace that may be produced by such a model:

$$s_0 \xrightarrow[dur_0]{mkdir(p)} s_1 \xrightarrow[dur_1]{\tau} s_2 \xrightarrow[dur_2]{stat(p)} s_3 \xrightarrow[dur_3]{\tau} s_4 \xrightarrow[dur_4]{mkdir(p/dir)} s_5 \xrightarrow[dur_5]{\tau} s_6 \xrightarrow[dur_6]{stat(p/dir)} s_7$$

This states that a $mkdir(p)$ takes dur_0 to complete before state s_1 is reached. Note that this transition system needs to consider operating system internal actions (and their duration) as special transitions labeled τ. However, in the remainder of this chapter the notation will keep these τ transitions implicit for simplicity. The τ introduces an additional problem: the checking process becomes even more complicated as we have to consider that between any two observable events there may be a finite sequence of τ transitions that we do not observe, and we ignore the time at which they occur.

In order to check time order in such a model, one would need an initial time t_0, which corresponds to the time of s_0, and also know the duration of the commands and τ transitions. Indeed, one could calculate the timestamp associated with an object by adding the previous commands duration to the time associated with object, e.g.,

[6]As before, we ignore start and end times of commands to simplify the discussion.

$$time(p/dir) = t_0 + \sum_{n=0}^{4} dur_n$$

The specification aims to abstracts over all possible implementations. At the level of abstraction at which the specification operates there is no fixed duration for each call. Without expected values for the timestamps, the specification cannot check that they respect any order. Indeed, if an arbitrary time to execute each command is allowed, given the same initial time t_0 in the following trace:

```
mkdir p
RV_None
stat p
RV_stats {timestamp = 1}
mkdir p/dir
RV_None
stat p/dir
RV_stats {timestamp = x}
```

the `x` can stand for any number greater than 1, potentially introducing a huge amount of non-determinism. Any state containing a timestamp reflecting the expected time order is acceptable. The trace checker should then keep track of the fact that any trace which has a timestamp equal to a particular value would be valid.

It is possible to avoid this non-determinism by abstracting the concept of time from a real world clock value to a so called logical time §3.3.6 and by introducing a consistency test on these timestamps §3.3.7.

3.3.6 Logical time

When one wants to check that the timestamps of an execution trace conform to the constraints of POSIX, one has to deal with non-determinism. For instance, in the trace:

```
mkdir p
RV_None
stat p
RV_stats {timestamp = 1}
```

before the observation through `stat`, the timestamp could have any value. An observation reduces infinite amounts of possible states in a single one (Figure 3.4).

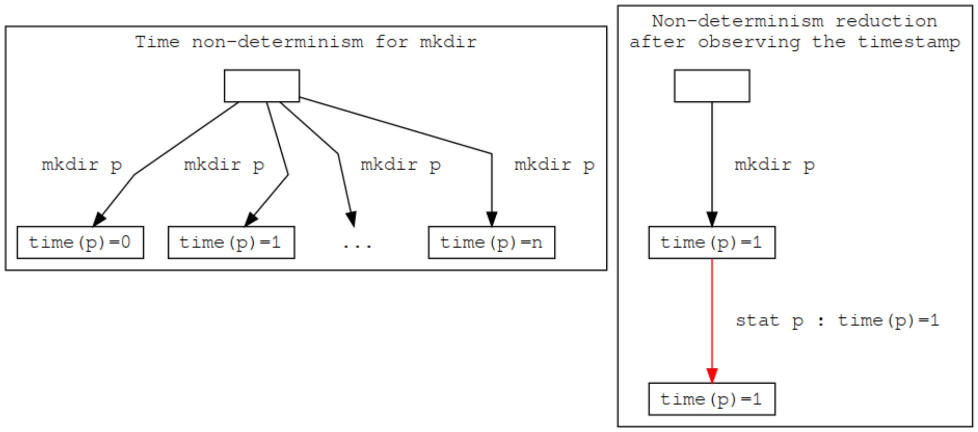


Figure 3.4: Non-deterministic state space explosion

This reduction makes easier to check that timestamps conform to POSIX constraints through our model.

Recall the transition sequence example presented earlier:

$$s_0 \xrightarrow[dur_0]{mkdir(p)} s_1 \xrightarrow[dur_1]{stat(p)} s_2 \xrightarrow[dur_2]{mkdir(p/dir)} s_3 \xrightarrow[dur_3]{stat(p/dir)} s_4$$

In order to check this trace we require that the timestamp of `p` is less than `p/dir` (less than or equal to if we are taking in account the granularity issue).

One way to achieve this is to work with logical time rather than physical time. Using logical times the transition sequence seen above becomes:

$$s_0 \xrightarrow[l_1]{mkdir(p)} s_1 \xrightarrow[l_2]{stat(p)} s_2 \xrightarrow[l_3]{mkdir(p/dir)} s_3 \xrightarrow[l_4]{stat(p/dir)} s_4$$

The logical times are l_1, l_2, l_3 and l_4. Here l_1 represents the time at which s_1 was created. We have the constraint that $i < j$ implies $l_i < l_j$, and each l_i corresponds to a real world time p_i.

Now the timestamp model can work with only logical time and is still able to check that the timestamps observed through the `stat` command respect the transition sequence order: in the example, $l_2 < l_3$ implies $time(p) < time(p/dir)$.

Formulating a toy model of timestamp specification may be useful to explore these concepts in greater detail. This shows the reader two aspects of the final specification in a simplified fashion:

1. how to associate time constraints to file system objects through the use of logical times;
2. how to evaluate if the system state is consistent each time a new physical timestamp is observed.

A starting point for this toy model is to discriminate logical and physical time.

```
type logical_time = int
type physical_time = float
```

A system as whole has a global clock which measures logical time, and every step increases the logical time by one.

```
type clock = logical_time
```

To each file we associate a logical timestamp. Since we are considering a simplified model, we define a logical file as follows:

```
type logical_file = L_file of logical_time
```

In this toy model the state contains a single file with a single timestamp, the clock and the last observed time. The last observed time is a relation between a logical time and a physical time observed through a `stat` transition. There are two types of transitions available: a generic transition that updates the logical timestamp of the file, and a `stat` transition that updates the last observed timestamp of the file.

```
type transition_label = Stat | Modifying_timestamps_command
type last_observed_time = (logical_time * physical_time)
type observed_result = Modifying_command_result
                     | Stat_result of physical_time
type specification_state =
  State of (clock * last_observed_time * logical_file)

(* val tick_clock : specification_state -> specification_state *)
let tick_clock (State(c,lo,lf)) =
(State(c+1,lo,lf))
```

A transition is a function from a state, a label and an observed result to an optional new state: if the transition produces a result that disagrees with the observed result the new state is invalid, and the transition system returns nothing.

```
(* val transition : specification_state -> transition_label ->
observed_result -> specification_state option *)
let transition s l o  = (
```

The clock is the current logical time of the system. Each system transition increments the clock value (`tick_clock` below).

```
let s = tick_clock s in
let (State (c,ot,f)) = s in
(match l with
```

If the transition alters the timestamps, it can only be a transition labeled with `Modifying_timestamps_command`.

```
| Modifying_timestamps_command -> (
  match o with
    | Stat_result t -> None (*unexpected result *)
```

If the received object is the result of a modifying command, the transition runs the command on the specification state in order to obtain the next system state. The result of running the command in the specification (e.g., the status code) should match the observed result, although in this discussion this is not shown to let the reader focus on the time aspect.

```
        | Modifying_command_result ->
(*assume that command result agrees with the observed result *)
        Some (State(c,ot,(L_file c)))
)
```

If the transition observes timestamps, it can only be a `Stat`.

```
| Stat -> (
  match o with
   | Modifying_command_result -> None (*unexpected result*)
```

When a `Stat` command introduces a newly observed timestamp, the specification needs to assess if its state remains consistent in terms of time. In this toy model the logical times maintain the consistency constraints: when we observe a physical time for the file it is bound to its logical time. The last observed time holds a binding between an old logical time and its related physical time. So the relation between the physical times needs to reflect the file logical time and the old logical relation. Note that we use $\leq$ for the physical times in respect to the granularity issue described before.

```
        | Stat_result t ->
          let (l_t, p_t) = ot in
          let L_file(file_l_t) = f in
          let times_are_consistent =
            (l_t = file_l_t and p_t = t)
            or
            (l_t < file_l_t and p_t <= t)
          in
          if times_are_consistent
          then (Some(State(c,(file_l_t, t),f)))
          else None
)))
```

In the following there are some examples states of this toy model:

```
let s0 = (State(0,(0, 0.0),(L_file 0)))

let s1 = (State(1,(1,1.0),(L_file 1)))

let s2 = (State(2,(2,0.5),(L_file 2)))
```

An example of valid transition is:

$$s0 \xrightarrow[Modifying\_command\_result]{Modifying\_timestamps\_command} s1$$

indeed, the constraint $l_0 < l_1$ implies $p_0 \leq p_1$ as in this case we have that $0 < 1$ implies $0.0 \leq 1.0$. An example of invalid transition is:

$$s1 \xrightarrow[Modifying\_command\_result]{Modifying\_timestamps\_command} s2$$

indeed, the observed timestamp does not satisfy the constraint as the physical times are $1.0 > 0.5$.

3.3.7 LP-map wellformedness

The previous section introduced the notion of logical time in order to reduce the complexity of trace checking. Each logical time is associated with a physical time. This approach be generalized to work with multiple and distinct timestamps for each object. This also means that the time observations obtained through the `stat` command are more complex than what we have encountered so far. For simplicity, we introduce some additional non-POSIX commands to use in this simplified model:

1. the `time(object)` command, similar to a `stat`, returns the timestamps;
2. the `update_t1(object)` command updates the first timestamp;
3. the `update_t2(object)` command updates the second timestamp;
4. the `update_all(object)` command updates all timestamps.

A way to generalize the solution from the previous section is to build the set of constraints discussed in §3.2.8 as a map between logical times and physical times that checks the coherence of new logical-physical bindings at their insertion. We call this LP-map.

Consider the following transitions sequence:

$$s_0 \xrightarrow[l_1]{update\_all(o)} s_1 \xrightarrow[l_2]{time(o)} s_2 \xrightarrow[l_3]{update\_t2(o)} s_3 \xrightarrow[l_4]{time(o)} s_4$$

Note, the logical times represent events: for instance, logical clock for s_1 is l_1.

A real world trace corresponding to these transitions is:

```
mkdir p
RV_None
stat p
RV_stats {t1 = 0.1; t2 = 0.1; t3 = 0.1}
mkdir p/dir
RV_None
stat p/dir
# the observed result is invalid:
RV_stats {t1 = 0; t2 = 0; t3 = 0}
```

When we observe that $t1 = 0.1, t2 = 0.1$, and $t3 = 0.1$ then we can complete the LP-map like this: $[\mathit{1} \to \mathit{0.1}]$.

However, attempting to insert the second observed time results in the LP-map $[\mathit{1} \to \mathit{0.1}, \mathit{2} \to \mathit{0.0}]$, which is invalid as the logical and physical times do not follow the same temporal order.

Also the LP-map $[\mathit{1} \to \mathit{0.0}, \mathit{1} \to \mathit{0.1}]$ is invalid, as a logical time corresponds to a single real world value.

In summary, there are two restrictions in updating the map:

1. once a logical time is assigned a physical time one cannot change that binding,
2. the order of logical times should match the order of physical times (i.e., $l_i < l_j \implies p_i \leq p_j$).

We formalize these restrictions as a predicate to check before inserting a new binding in the map.

Note that the monotonicity property applies also when different files delay the timestamp update by different amounts: the τ transitions model the delays (they can take any time to complete), the logical times are issued during these transitions, and the logical times bind physical times. An example may help in clarifying this aspect:

consider the trace

```
mkdir p1
RV_None
mkdir p2
RV_None
# the OS assigns the physical time 0.0 to p2's timestamps
# the OS assigns the physical time 1.0 to p1's timestamps
stat p1
RV_stats {t1 = 1.0; t2 = 1.0; t3 = 1.0}
stat p2
RV_stats {t1 = 0.0; t2 = 0.0; t3 = 0.0}
```

here the OS assigns physical time values to the timestamps with different delays. The specification models these commands and returns as follows:

$$\{s_{\mathit{0}}\} \xrightarrow{mkdir(p1)} \{s_{\mathit{1}}m\} \xrightarrow{\tau} \{s_{\mathit{1}}m, s_{\mathit{1}}u\}...$$

A τ transition happens in between a command transition and its return. A command transition only marks timestamps for update ($s_1 m$). During a τ transition marked timestamps of a new state are updated ($s_1 u$). The example trace is acceptable for the state in which the τ transition after `mkdir p2` updates the timestamps of `p2`: indeed, the `stat p1` forces the marked timestamps of p1 to be set to the current time (i.e., 1.0). In this state indeed the logical times are $l_{p2} < l_{p1}$ as the physical times $0.0 <= 1.0$.

Again, a toy model may be useful to show the time validity check based on a logical-physical map.

Firstly, we needed to discriminate between logical and physical time.

```
type logical_time  = int
type physical_time = float
```

The map can be implemented as a list of bindings.

```
(* the map is a list of bindings *)
type lp_map = (logical_time * physical_time) list
```

It is required to search a timestamp in the map.

```
(* val find_pt : logical_time -> lp_map -> physical_time option*)
let find_pt lt m =
  if (mem_assoc lt m)
  then Some (assoc lt m)
  else None
```

A map must be valid after an insertion, so we define the two properties that must hold: logical times must be distinct and physical times reflect the time order of the associated logical times.

```
let distinct_map m =
let logical_times_in_map = (map fst m) in
(* check that each logical time
is present at most once in the map *)
(fold_left
(fun acc e ->
  acc &&
  (1 = (length (find_all (fun e'-> e = e') logical_times_in_map))))
true
logical_times_in_map)
in
```

We can define the distinct property as `distinct_map`:

```
(*val distinct_map : lp_map -> bool *)
let distinct_map m =
```

$$\forall l_1 l_2 . l_1 < l_2 \implies p_1 < p_2$$

```
in
```

We can define the ordering property as `time_ordered_map`:

```
let time_ordered_map =
```

We associate each logical time in the map with all the other logical times.

```
let all_logical_time_pairs =
...
in
```

Take only logical times that were associated in order (e.g., the pair `(2,1)` is rejected, while the pair `(1,2)` is accepted).

```
let only_ordered_lt_pairs =
filter
(fun (lt,lt') -> lt < lt')
all_logical_time_pairs
in
```

We check that physical times reflect the logical time ordering: $\forall l_i l_j . l_i < l_j \implies p_i \leq p_j$.

```
let check_physical_times_order (lt,lt') =
let pt  = find_pt lt  m in
let pt' = find_pt lt' m in
(* lt < lt' then physical times must reflect this order*)
(* equivalence is allowed
 for the time granularity issue *)
(pt <= pt')
```

```
in
for_all check_physical_times_order only_ordered_lt_pairs
```

And we check that the properties hold:

```
(* val is_wellformed : lp_map -> bool*)
let is_wellformed m = (
  (*map property (1) *)
  (distinct_map m)
  &&
  (*map property (2) *)
  (time_ordered_map m))
```

Finally, the map supports insertion. Insertion returns a new map only if this map is wellformed.

Adding a binding in the map appends a pair of logical-physical time to a LP-map, and can be defined as a function:

```
(* val add_binding :
  logical_time  ->
  physical_time ->
  lp_map ->
  lp_map option *)
let add_binding lt pt m = ...
```

If `lt` is already in the domain of `m`, the bound physical time must be equal to `pt`.

Otherwise, we add the new binding to `m`, and check that the new map is wellformed.

Some examples of valid and invalid insertions are:

```
(* create a simple function to extract valid maps
   from the optional constructor *)
let dest_some (Some x) = x
(* create a valid map by adding the logical-physical time pairs
   (0,0.0) and (1,1.0) *)
let m = dest_some (add_binding 1 1.0 (dest_some
                                       (add_binding 0 0.0 [])))

(* trying to insert a new physical time for a logical time
   already in [m] produces an invalid map *)
let _ = assert (None = (add_binding 1 2.0 m))

(* trying to add a physical time that is smaller than
   those already present in [m] breaks the order
   and so produces an invalid map *)
let _ = assert (None = (add_binding 1 0.0 m))
```

The predicate `is_wellformed` checks that the map respects (1) and (2). Asserting the validity of this property after each insertion guarantees that the map is always wellformed.

3.4 Technical challenges

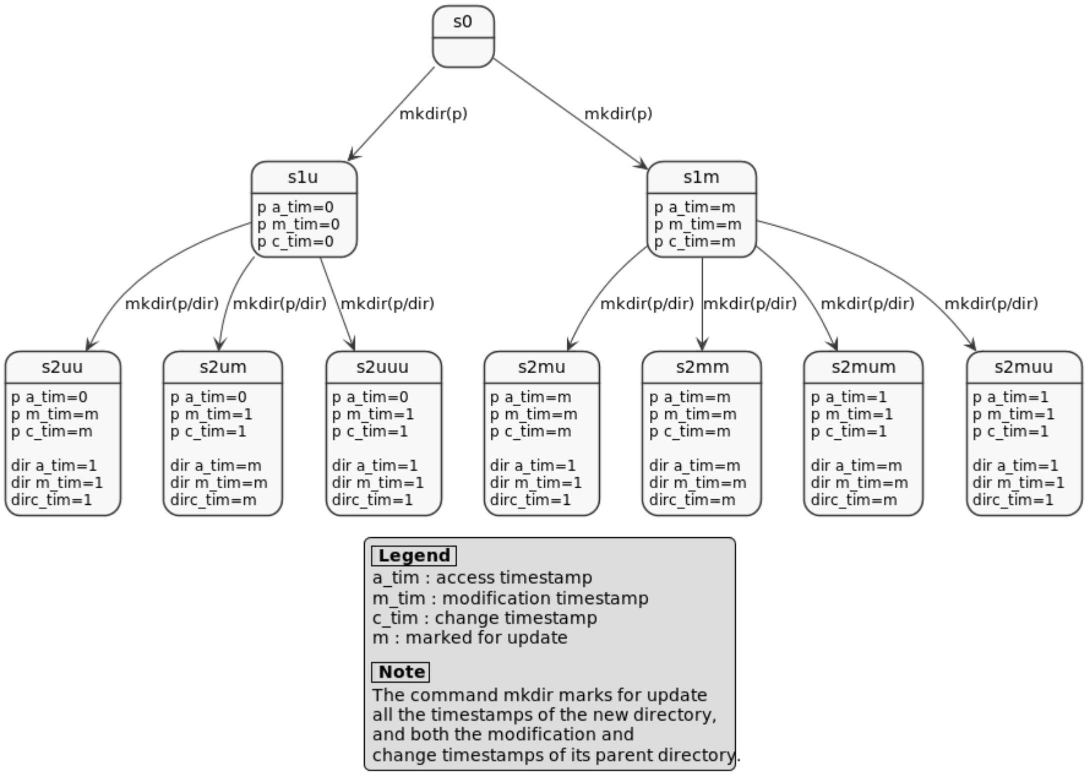
The main challenge in extending SibylFS with a timestamp model is to preserve its test oracle capabilities: indeed, it is hard to discriminate a file system implementation trace of execution that is inconsistent with POSIX time definitions. The fact that only the `*stat` commands allow to observe timestamps implies that only those traces using these commands have a chance to be evaluated.

The same challenge is exacerbated by the definition of periodic update that POSIX gives §3.5.3: as this introduces a serious source of non-determinism. Indeed, the operating system can update timestamps of any object in a delayed fashion. Figure 3.5 shows how the non-deterministic growth of states behaves for the transitions:

$$s_0 \xrightarrow{mkdir(p)} s_1 \xrightarrow{mkdir(p/dir)} s_2$$

from an initial state s_0 that does not contain any file system object, making a directory p results in two possible states where all timestamps have a value or are marked; using the new states as initial states and making a subdirectory p/dir results in multiple states, as the creation of a subdirectory modifies also the timestamps of the parent directory.

The specification cannot anticipate the operating system choices for delayed updates, since it does not aim to model every aspect of an operating system. With the extension presented in this work using the specification as a test oracle for periodic traces is rather limited: when used as a test oracle the specification



Legend
a_tim : access timestamp
m_tim : modification timestamp
c_tim : change timestamp
m : marked for update

Note
The command mkdir marks for update
all the timestamps of the new directory,
and both the modification and
change timestamps of its parent directory.

Figure 3.5: Non-deterministic state space explosion for the transitions $s_0 \xrightarrow{mkdir(p)} s_1 \xrightarrow{mkdir(p/dir)} s_2$; note that any file system object can be updated by the operating system

needs to compute all the valid states achievable for a transition, and with multiple transitions the discussed non-determinism introduces an exponential growth of states as shown in §3.7. An approach that appears promising, and that we could not investigate due to time constraints, is to use logical constraints (Chapter 5) to represent the timestamp order.

3.5 Model

This section discusses how to integrate the time model presented in the previous sections with the original data structures of the SibylFS §3.5.1; how to alter the transition system mechanism to model a clock §3.5.2 and immediate and periodic update strategies §3.5.3.

3.5.1 Time data structures

A starting point to formalize the POSIX time model is defining the data structure used by the POSIX `stat` command. POSIX calls the time data structure *timespec*, and states that it must contain at least the following members:

```
time_t  tv_sec;
long    tv_nsec;
```

where `time_t` is the type that represents seconds, and `long` is the integer type which can represents numbers in the range between -2147483647 and +2147483647.

We define these as:

```
type t_time = nat
type long = int64

(* [ty_os_time] is our corresponding type
   for the POSIX timespec. *)
type ty_os_time =
<| tv_sec:t_time ; tv_nsec:long |>
```

Using natural numbers (in the language used to define SibylFS this is called the `nat` type) has also the advantage of excluding negative values for seconds.

The data structure `ty_os_time` represents a timestamp value observed in the real world.

As discussed in §3.3.4, the definition of logical time is:

```
type ty_logical_time = nat
```

A timestamp may be updated or marked for update, so the datatype needs to model this behaviour as well:

```
type ty_logical_timestamp =
Logical_timestamp of ty_logical_time
| Marked_for_update
```

Having a model of time permits the addition of a logical time field in the file system state representing a clock. This maintains the current time, which the system uses when it needs to update timestamps.

As there is a distinction between internal (logical) and external (physical) representations of time attributes, there are two types of `stat` data:

```
type ty_real_stats =
<| r_st_dev : int;            (** Device number *)
...
r_st_atime : ty_os_timestamp; (** Last access time *)
r_st_mtime : ty_os_timestamp; (** Last modification time *)
r_st_ctime : ty_os_timestamp; (** Last status change time*)
|>

type ty_logical_stats =
<| l_st_dev : int;
...
l_st_atime : ty_logical_timestamp;
l_st_mtime : ty_logical_timestamp;
l_st_ctime : ty_logical_timestamp;
|>
```

These types distinguish explicitly between specification results and real world results (see §3.6).

3.5.2 Logical clock update

The logical clock increases at each operating system transition. In the model a transition is a call to the function `os_trans` §2.4.2, so the clock increment must happen during its execution:

```
val os_trans :
  ty_os_state->
  os_label   ->
  finset (os_state_or_special)
let os_trans s0 lbl = (
 let env = s0.oss_env in
#ifdef aspect_time
 (* clock ticks *)
 let s0 =
   dest_OS_normal(increment_time env (OS_normal s0))
 in
#endif
...
```

The function `increment_time` returns a state with the logical clock increased by one. The `#ifdef aspect_time ... #endif` syntax defines pre-processor declarations, which allow to add or ignore time definitions according to the pre-processor settings. A side effect of this technique is that it makes evident where definitions relevant to time are located.

It is worth mentioning again that one should not expect two distinct logical times to represent two different physical times: for example, given two logical times l_1 and l_2 that are ordered (i.e., $l_1 < l_2$), the corresponding physical times p_1 and p_2 reflect the same order if their granularity can represent $|p_1 - p_2|$ (e.g., a granularity in seconds cannot represent $|0.1s - 0.2s|$).

3.5.3 Immediate and periodic transitions

In SibylFS a command involves three transitions: one to call the command (*Call* transition), one to allow some operating system internal behaviour (τ transition), and one to return the result of the command (*Return* transition). So, calling a command marks timestamps for update, and issuing a τ transition updates their values.

A *Call* transition invokes the command. Introducing the time behaviour involves an extension of the command semantic. Subsequently we will use the `mkdir` command to illustrate how to introduce time definitions. The `mkdir` command creates a new directory, and POSIX expects that it marks all the timestamps of the new directory and the *mtime* and *ctime* of the parent directory (*mtime* because the parent content is changed and *ctime* because the file status, for example the attribute *nlink*, is changed as well). The extended definition is the following:

```
let fsop_mkdir_core env rpath mode =
fsm_get_state >>= (fun s0 ->
(match rpath with
 | RN_none(d0_ref,n,_) -> (
    (* create the new directory and get a reference to it back *)
    let (s1, d1_ref) = env.env_ops.fops_mkdir s0 d0_ref n in
#ifdef aspect_time
    let s1 =
      (* see MKDIR_TS:1 in fs_spec/posix/mkdir.md *)
      mark_timestamps (env.env_ops) s1
        [TS_Access; TS_Modification; TS_Change]
        (Dir_ref_entry d1_ref)
    in
    let s1 =
      (* see MKDIR_TS:2 in fs_spec/posix/mkdir.md *)
      mark_timestamps env.env_ops s1
         [TS_Modification;TS_Change]
         (Dir_ref_entry d0_ref)
    in
#endif
...
    fsm_put_state s1)
 | _ ->
   fsm_special Impossible "error raised before"
end))
```

The original definition uses a monad to model the behaviour of the commands.

This monad composes functions which transform file system states. We extend the original definitions with a sequence of functions that implement the time trait. Our additions can be easily distinguished as they are wrapped in the pre-processor directives as mentioned earlier. So in adding the time definitions we alter the file system's monadic state `s1`, which is finally returned using the function `fsm_put_state`. Note that comments like `see MKDIR_TS:1 in fs_spec/posix/mkdir.md` refer to quotations extracted from the POSIX specification [74]: these references provide future readers a one to one mapping of the formal specification and POSIX definitions.

A τ transition assigns the clock value to marked timestamps according to the update strategy in use. In the immediate update strategy τ assigns the current clock to all marked timestamps (`update_all_marked_timestamps`). In the periodic update strategy τ emulates the behaviour of the operating system: it may update the timestamps of any file system object with marked timestamps (`do_periodic_update`). This makes the transition non-deterministic as it may return multiple new valid states: it creates a new state for each file system object with marked timestamps. A visual representation of the update algorithm is given in Figure 3.6.

The following extracts show that the specification is parameterized by a flag which controls whether or not updates are periodic or immediate:

```
...

#ifdef aspect_time
(* OS_DEFINED_TS in
 fs_spec/posix/base_definitions/ch_4_general_concepts *)
(* in immediate mode there are no Mup timestamps;
  these only arise in Tau transitions *)
let ss = (
 match lbl with
 | OS_TAU -> (
   if ((architecture_of_ty_arch env.env_arch).arch_is_periodic)
   then finset_bigunion_image (do_periodic_update env) ss
   else finset_image (update_all_marked_timestamps env) ss)
 | _ -> ss end)
in
```

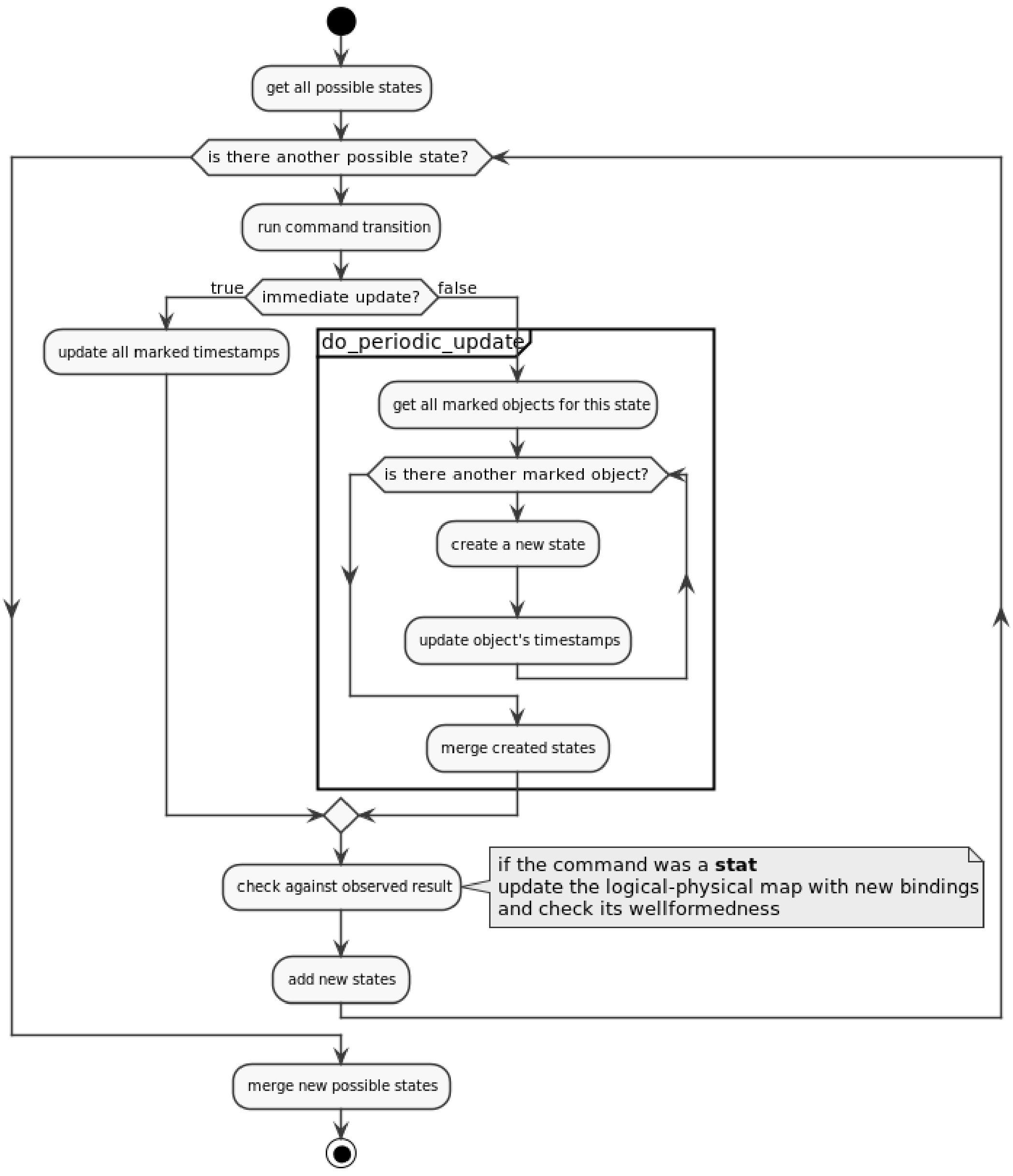

get all possible states
is there another possible state?
run command transition
true
immediate update?
false
update all marked timestamps
do_periodic_update
get all marked objects for this state
is there another marked object?
create a new state
update object's timestamps
merge created states
check against observed result
if the command was a **stat**
update the logical-physical map with new bindings
and check its wellformedness
add new states
merge new possible states

Figure 3.6: Algorithm to use SibylFS with timestamp as an oracle

```
#endif
ss)
```

The characteristics of the operating system running the file system are in the data record `env.env_arch`. The boolean field `arch_is_periodic` discriminates the update strategy used. Given an immediate architecture and a set of initial states, for each state the function `update_all_marked_timestamps` updates marked timestamps. Given a periodic architecture and a set of initial states, the `do_periodic_update` function is applied to each of the initial states. The `do_periodic_update` takes a state and produces a set of states, which represent all the possible results after a periodic update. In each of these result states only one of the marked file system objects obtains concrete time values, while the others remain marked. The sets of result states are then merged in a single set of states (this is done with `finset_bigunion_image` which first applies the function and then flattens the result).

3.6 Oracle

This section describes how to extend the oracle capabilities of SibylFS to consider timestamps §3.6.1, and an attempt to mitigate the state explosion generated by the periodic update strategy §3.6.2.

3.6.1 Extending the SibylFS oracle to handle the LP-map

According to what we discussed in §3.3.7, SibylFS needs to maintain a LP-map in order to discriminate states that are inconsistent in terms of time.

The first step is to make possible to collect physical timestamps. The testing framework needs to parse the time data in the return of the `*stat` commands, insert these into the `timespec` format and finally add them into the `ty_real_stats` data structure. The transformation is summarized in Figure 3.7

The second step is to add the LP-map in the operating system state:

```
type ty_os_state 'dir_ref 'file_ref 'jimpl = <|
  ...
```

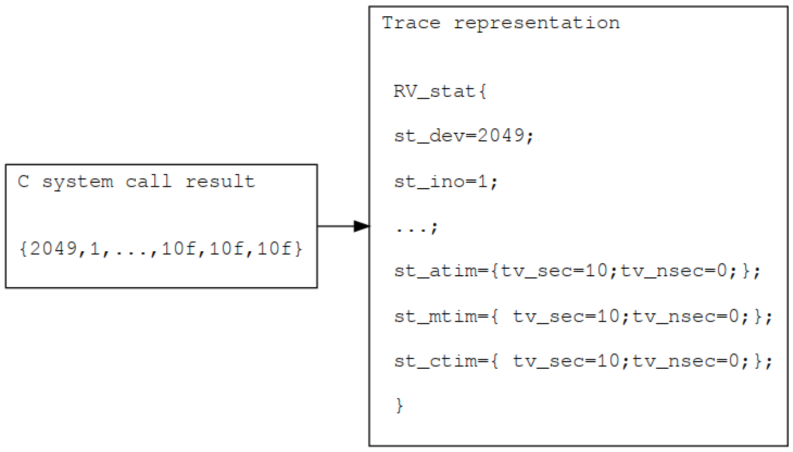


Figure 3.7: Creating observed timestamps from system call output

```
  oss_realtime_table: fmap ty_logical_time ty_os_time
|>
```

A specification state is consistent in terms of time if it contains a wellformed LP-map. So, every insertion in the LP-map must leave it wellformed. Running `*stat` command implies a new insertion. When the LP-map becomes malformed, there are no reachable states for the current `*stat` transition: this means that the file system implementation under test does not assign timestamps correctly.

The following excerpt shows how the specification inserts observed timestamps in the LP-map for a `stat` return:

```
...
| (Value(RV_logical_stats stats1),
Value(RV_real_stats stats2)) -> (
match (differ_only_on_time_formats env stats1 stats2) with
 | false -> (finset_empty ())
 | true -> (
    let r =
      insert_new_entries_in_realtime_map s0
       [(dest_logical_timestamp stats1.l_st_atime,
          dest_os_timestamp stats2.r_st_atime);
        (dest_logical_timestamp stats1.l_st_mtime,
          dest_os_timestamp stats2.r_st_mtime);
        (dest_logical_timestamp stats1.l_st_ctime,
          dest_os_timestamp stats2.r_st_ctime)]
    in
    (match r with
      | Nothing -> (finset_empty ())
      | Just s1 -> (
      finset_singleton
        (OS_normal (update_run_state s1 pid RUNNING)))
end)) end)
...
```

The function `differ_only_on_time_formats` returns true if all the attributes (aside from the timestamps) are equal in the two `stat` results. The timestamp check is left to the insertion function. The function `insert_new_entries_in_realtime_map` has the following signature:

```
val insert_new_entry_in_realtime_map :
  ty_os_state ->
  (ty_logical_time * ty_os_time) ->
  (maybe ty_os_state)
```

This function takes an operating system state and a pair of logical-physical times and may return a new state. It is isomorphic to the `add_binding` function shown in the LP-map example in §3.3.7: it tries to insert each logical-physical time pair into the LP-map, and if after the insertion the LP-map is malformed, it returns nothing.

3.6.2 Mitigating the state explosion

It is not possible to observe marked timestamps in a file system implementation (without instrumenting the OS, which is difficult and not scalable). Since POSIX allows the operating system to update the marked timestamps of file system objects at its own discretion, the specification needs to maintain all the possible states that a file system can reach. The test oracle feature of the specification (as shown in §2.5) is significantly limited due to the state explosion caused by the non-determinism of the periodic updates. As a test oracle SibylFS declares a file system trace acceptable if the specification can validate the trace results. Since the trace represents a sequence of commands that act on a file system state, the test oracle needs to run the corresponding sequence of specification transitions. Each transition may produce many states or none. Each transition is applied on the states obtained by the previous transition. If the transition is deterministic, the set of states will always be a singleton; if, otherwise, the transition is non-deterministic, the set will contain multiple states. As the number of states to check grows, the testing process becomes computationally more expensive (hence slow). If possible, one should avoid this kind of state growth in the first place.

As we have seen, the periodic update strategy introduces a source of non-determinism in the specification. A naive approach we investigated is to embed its non-deterministic information into a single state. In the periodic update strategy we need to create a new state for each marked file system object to reflect the possible operating system updates. It seems wasteful to duplicate whole states that differed only on a single object's timestamps, so we tried to store this information in the marked timestamps. So a marked timestamp would accumulate the times at which the operating system may have issued an update:

```
type ty_logical_timestamp =
  Logical_timestamp of ty_logical_time
  | Marked_for_update of ty_logical_time list
```

With this change non-deterministic transitions as those shown in Figure 3.5 would produce a single state shown in Figure 3.8.

Although, this approach may seem powerful in reducing the cost of non-determinism, it has two problems:

1. unexpected states become valid:

   for example in Figure 3.8 the directory `p` is allowed to have timestamps `[a_tim=0;m_tim=1;c_tim=0]`, although this configuration is not achievable by any of the final states in Figure 3.5;

2. non-determinism is only postponed:

   every time we run a `*stat` transition on an object, the suppressed non-determinism unfolds as the insertion in the LP-map has to assess if the new observed timestamps breaks the time order.

Although (1) is solvable by adding some sort of dependency information to the accumulated times, (2) has not a straightforward solution.

Another approach available to mitigate non-determinism is to store timestamp configurations in the specification state. For example, we may maintain a map that

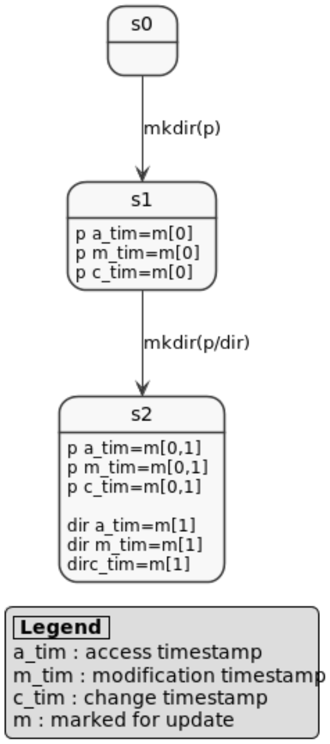


Figure 3.8: Mitigating the time non-determinism of Figure 3.5

has file system objects as keys and sets of timestamps triples as values. In this way the information about timestamps remains precise and the non-determinism does not affect the oracle feature of the model for traces that do not involve timestamps (i.e., those traces that do not invoke `*stat` commands). Only a rough prototype of this approach was made and was not refined further due to time constraints. Also this approach suffers from (2), although promises to be more resilient in multiple ways:

- keeping track of non-determinism simply means updating the state map with new configurations instead of the duplication of state of the naive model;
- allows to unfold only part of the non-determinism efficiently (i.e., `stat f1` would lookup for all the timestamps configurations of the `f1` object present in the state rather than performing the command on each available state as happens in the naive model);
- choosing a more suitable data structure to support retrieval of valid configuration would significantly extend the usage of the extended SibylFS to bigger traces (the naive model chosen can only check traces consisting of about eight commands).

This last approach clearly moves towards enhancing the extended specification with a constraint solver: instead of storing timestamps configurations for file system objects, we should encode these into time constraints and logically exclude the predicates that cannot hold during validation. This is mentioned in Chapter 5 as future work.

We decided that trying to make these models fit would have complicated matters, and we preferred to propose the simplest model available to the reader. For this reason, the models were not implemented in the final deliverable of this work.

3.7 Results

This section discusses how we validated the timestamp extension, and how the immediate and periodic model behaviours relate to a file system implementation.

3.7.1 Validating traces

After implementing the time specification, one can check that it behaves as expected through testing §2.5. In this section we present a minimal collection of tests to validate the most interesting behaviour. These tests fall in three categories (see Appendix C for examples of these traces):

1. positive tests: commands alter timestamps according to POSIX definitions. These tests are subdivided in tests for immediate mode and tests for periodic mode.
2. negative tests: commands alter timestamps differently from what POSIX states and so the specification rejects the trace.
3. concurrent tests: commands run concurrently.

Examples for each of these subsets are in Figures 3.9, 3.10, and 3.11: they use the command `chmod`, which changes the *change* timestamp other than the object permissions.

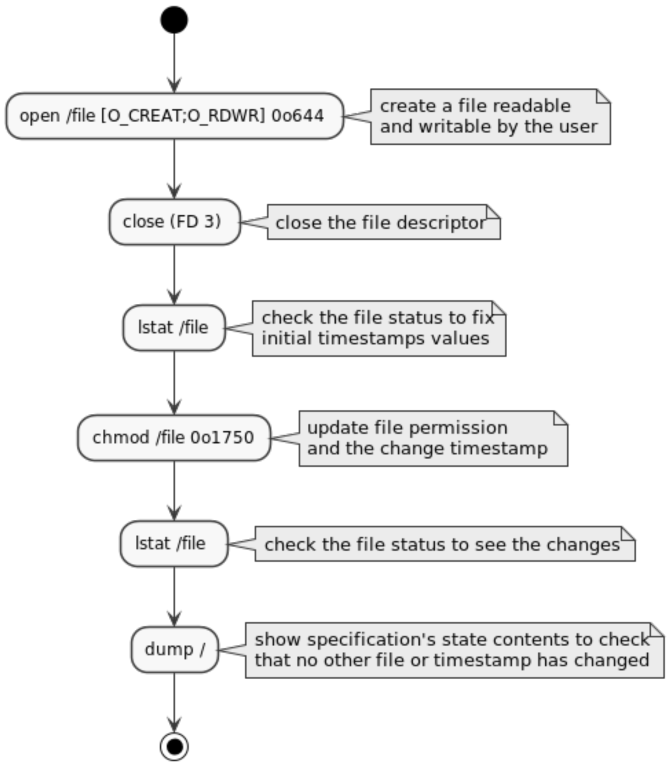

open /file [O_CREAT;O_RDWR] 0o644
create a file readable and writable by the user
close (FD 3)
close the file descriptor
lstat /file
check the file status to fix initial timestamps values
chmod /file 0o1750
update file permission and the change timestamp
lstat /file
check the file status to see the changes
dump /
show specification's state contents to check that no other file or timestamp has changed

Figure 3.9: Checking a trace involving chmod and timestamps

3.7.2 Immediate tests

The granularity issue, discussed in §3.3.2, can hide the periodic behaviour of a file system implementation as the applied delay is typically below the granularity threshold. In most cases an implementation that uses the periodic update strategy would satisfy the immediate tests. This makes the validation unreliable.

However one can expose the periodicity of a file system by heavily exercising it and detecting incongruities of timestamps assignment. A behaviour observed in testing `ext4` (Appendix D) may be: let ts_a be a timestamp and let ts_b be another timestamp, such that $ts_a < ts_b$; let dir_1 be a directory having a modification time equal to ts_b and dir_1/dir_2 a subdirectory having modification time equal to ts_a. The last modification to dir_1 was to add dir_1/dir_2.

But here $ts_a < ts_b$, so it appears that the modification time for dir_1 is updated after dir_1/dir_2's timestamp, whereas in immediate mode, the timestamps should be updated at the same time.

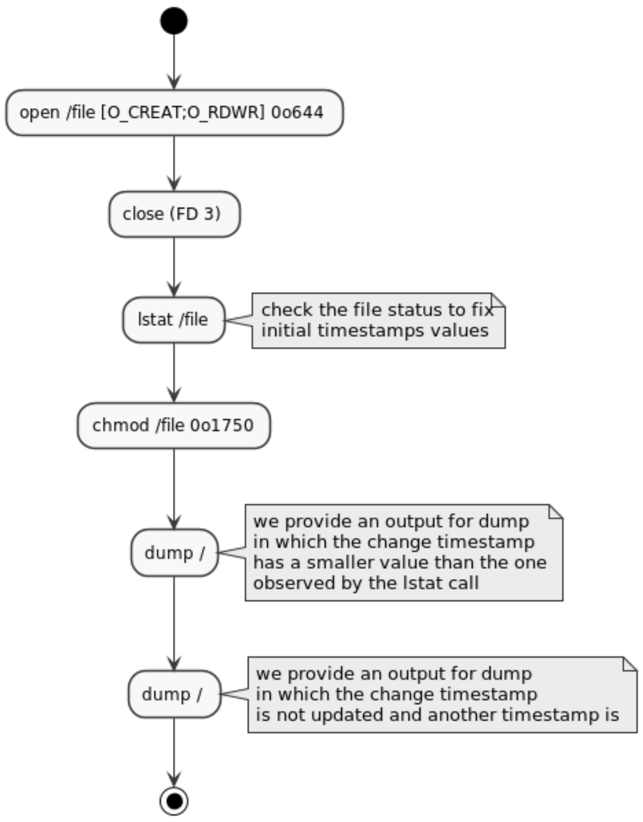

open /file [O_CREAT;O_RDWR] 0o644
close (FD 3)
lstat /file
check the file status to fix initial timestamps values
chmod /file 0o1750
dump /
we provide an output for dump in which the change timestamp has a smaller value than the one observed by the lstat call
dump /
we provide an output for dump in which the change timestamp is not updated and another timestamp is

Figure 3.10: Checking an invalid trace involving chmod and timestamps

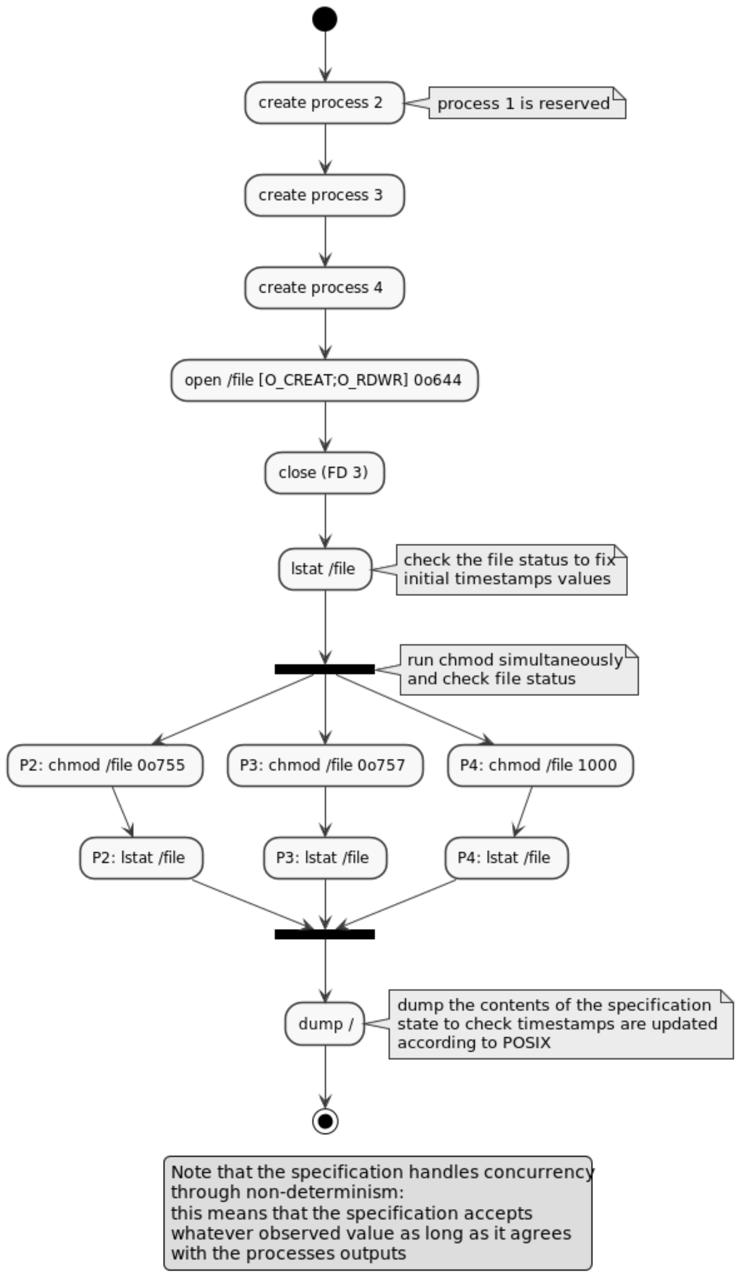

create process 2
process 1 is reserved
create process 3
create process 4
open /file [O_CREAT;O_RDWR] 0o644
close (FD 3)
lstat /file
check the file status to fix initial timestamps values
run chmod simultaneously and check file status
P2: chmod /file 0o755
P3: chmod /file 0o757
P4: chmod /file 1000
P2: lstat /file
P3: lstat /file
P4: lstat /file
dump /
dump the contents of the specification state to check timestamps are updated according to POSIX
Note that the specification handles concurrency through non-determinism:
this means that the specification accepts whatever observed value as long as it agrees with the processes outputs

Figure 3.11: Checking a concurrent trace involving chmod and timestamps

3.7.3 Periodic tests

The periodic flavour of the specification cannot currently be used to test implementations as explained in §3.4. We present here the limits of the periodic specification. Running the specification as a test oracle against a simple trace can provide data to show that the growth of states is exponential. The trace will contain a sequence of `mkdir` calls as this command creates new file system objects:

```
@type trace

# at this point we have already the root directory with timestamps
# marked for update

mkdir /d1 0o000
Tau
RV_none

mkdir /d2 0o000
Tau
RV_none

mkdir /d3 0o000
Tau
RV_none

mkdir /d4 0o000
Tau
RV_none

mkdir /d5 0o000
Tau
RV_none

mkdir /d6 0o000
Tau
RV_none
```

```
mkdir /d7 0o000
Tau
RV_none

mkdir /d8 0o000
Tau
RV_none
```

Figure 3.13 shows the states obtainable from two `mkdir` transitions.

In the following table we can see how the time grows in the previous trace:

Table 3.2: Statistics about checking a periodic trace consisting of a sequence of mkdir calls

# `mkdir` calls	# of dirs	# of states	Elapsed time	User CPU time	System CPU time
1	2	4	0m0.001s	0m0.001s	0m0.000s
2	3	24	0m0.001s	0m0.000s	0m0.001s
3	4	192	0m0.002s	0m0.001s	0m0.000s
4	5	1920	0m0.011s	0m0.009s	0m0.001s
5	6	23040	0m0.137s	0m0.132s	0m0.004s
6	7	322560	0m3.347s	0m3.253s	0m0.093s
7	8	5160960	6m2.876s	6m1.113s	0m1.807s
8	9	?	?	?	?

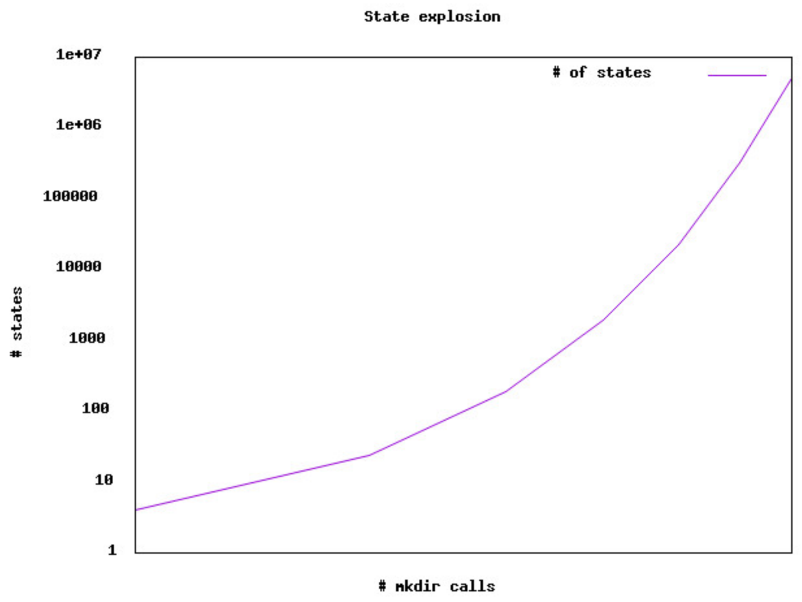


Figure 3.12: Visualization of the state explosion for a trace that just repeats mkdir commands

Table 3.3: Statistics about growth of states

#commands	# of states	f=fact(#commands)	g=$2^{\#commands-1}$	f * g
1	1	1	1	1
2	4	2	2	4
3	24	6	4	24
4	192	24	8	192
5	1920	120	16	1920
6	23040	720	32	23040
7	322560	5040	64	322560
8	5160960	40320	128	5160960

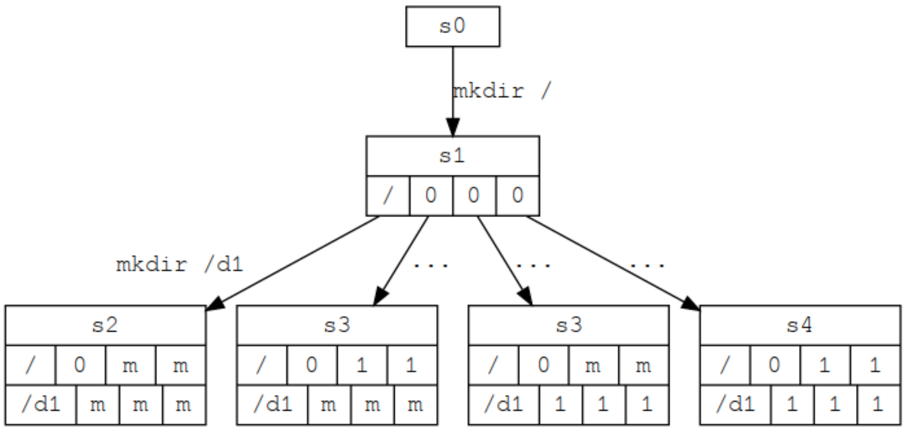


Figure 3.13: Non-determinism explosion for first transition of mkdir trace

Figure 3.12 shows that the growth of states follows roughly an exponential distribution.

3.8 Related work

In 1978, Lamport [42] explores an algorithm to achieve a total order for timed events in a multiprocess distributed system, and he introduces the idea of logical clocks as opposed to a physical clock. He defines the algorithm and associated invariants in terms of logical clocks and then he describes how to relate logical clocks to observed real world clocks. We share the time abstraction with Lamport work, and the total order constraint on logical and physical times that we implement through the LP-map.

In 1990 Nicollin and Sifakis introduce a timed process algebra which adds a special χ action to label transitions that represent the passage of time [56]. This χ action is equivalent to the τ transition of the SibylFS extension, as during this transition a new logical time is assigned to marked timestamps (so introducing passage of time). Another similarity with this work is the differentiation between abstract and physical time to separate the complexity introduced by the execution time of processors: the SibylFS extension however needs to keep track of the physical time to validate traces, while in a timed process algebra context the physical time is just abstracted away.

Alur and Dill define a time theory for automata in [3]. The authors show how to annotate automata with timing constraints, and show that the universality problem (if the timed automata accept all timed traces) is decidable for deterministic systems and undecidable for non-deterministic ones. Although this study annotates a non-deterministic LTS with timing constraints, the problem is decidable as it is very specific: we need to check that only a single trace is acceptable. This is currently unfeasible due to the state explosion generated.

4
B-trees, formally

4.1 Overview

File systems need to move data to and from the disk. Accessing the disk is more expensive than in-memory file system operations. Since accessing the disk is mandatory to persist data, we use data structures that minimize the accesses needed for their operations.

In this chapter we formalize and prove the correctness of one of these optimizing data structures: the B-tree. The definitions and the proofs discussed in this work are mechanized in the Isabelle/HOL theorem prover [85].

This chapter presents an approach to mechanically verify a B-tree. It starts with the basic notions required to formalize the B-tree §4.2, then defines the B-tree we use in this work §4.3. Then the approach to prove the correctness of the B-tree is given §4.4. This approach will require the notion of framestacks §4.5. At this point the reader has enough context to understand how B-tree operations can be defined and proved correct §4.6 §4.7. Finally the chapter discusses a way to refine the model into a B-tree implementation §4.8, and related work in this area §4.9.

4.2 Basics

This section defines the basic concepts required to understand the B-tree formalization.

4.2.1 Maps, keys, values and total orders

A B-tree is an implementation of a map, specialized for block storage. A block storage is the most commonly used software interface to interact with hardware that stores data.

A map is a set of pairs (k, v), where k is the key and v is the value. At most one value is associated to each key. In computer science, maps are finite functions: i.e., the set of pairs are finite. A possible notation for maps can be $(k_1 \rightarrow v_1, k_2 \rightarrow v_2, ...)$.

Maps support three fundamental operations: *find*, *insert*, and *delete*.

- *find* k m: return the value, if any, associated to k in map m,

- *insert k v m*: add (k, v) to m, or destructively update the pair if k is in m, returning the updated map,
- *delete k m*: remove all pairs of the form (k, v) from m and return the resulting map, which can be m itself if there was no pair (k, v) in m.

Typically, keys are totally ordered in map implementations.

Definition A relation $\leq$ on a set S is a total order iff it is a partial order which satisfies the trichotomy law: for all $a \in S, b \in S$ we have $a \leq b$ or $b \leq a$.

The name of the law arises from the version of the axiom that uses the strictly-less operation: $a < b \;\oplus\; a = b \;\oplus\; b < a$, where $\oplus$ here stands for a mutually exclusive $\vee$.

Typical examples of total orders are: natural numbers, integers, reals, strings (ordered lexicographically).

4.2.2 Search trees as implementations of maps

A B-tree is a type of balanced search tree. There are multiple B-trees defined in literature, see §4.9.1 for a brief summary of these. Search trees are well known structures, often covered in basic computer science courses. These are fundamental structures designed to provide efficient search algorithms, which makes them suitable to use as map implementations. An example of a search tree is:

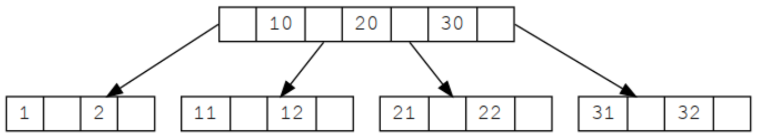


Figure 4.1: Example of search tree with integer keys

In this example, keys are integers. Leaves contain lists of (key,value) pairs, but values have been omitted and only the keys are shown. However, it is worth remembering that for every key (such as 5) in a leaf there is an associated value (e.g., v_5). Leaves cannot be empty, and contain a given key at most once. An exception to this is the empty tree, which is represented by an empty leaf.

The set of pairs (k, v) in the leaves of a search tree constitutes a map.

A node consists of (typically pointers to) children separated by keys. The keys are in increasing order. For example, a tree t has a root $t_0, k_0, t_1, ..., k_{n-1}, t_n$, where t_i

is the subtree i of t, and $k_0 < k_1 < ... < k_{n-1}$. Again, nodes cannot be empty, and a usual restriction is that they must contain at least one key and two children.

A typical node t is a tree:

t_0	k_0	t_1	k_1	...	k_{n-1}	t_n

A search tree t satisfies the following property: for **every** node $t = t_0, k_0, ...$ we have $k_{i-1} \leq keys(t_i) < k_i$, where $keys(t)$ denotes all the keys in the (nodes and leaves of) tree t. If $i = 0$ we may take k_{-1} to be negative infinity, so that $keys(t_0) < k_0$. Similarly for $i = n$ we may take k_n to be positive infinity, so that $k_{n-1} \leq keys(t_n)$.

The fact that this property holds for every node means that every subtree of a search tree is itself a search tree.

An alternative description for the above property can be achieved by considering the keys $k_0, ..., k_{n-1}$ partition the space of keys into $K_0, K_1, ..., K_n$, such that $K_0 < k_0 \leq K_1 < k_1 ... k_{n-1} \leq K_n$, and the union K_i is the set of all keys. Note that in this context the $<$ and $\leq$ operators establish an order relation between an element and all the elements in a partition (e.g., $K_0 < k_0$ is an abbreviation of $\forall x \in K_0 . x < k_0$). This guarantees that for any k there is a unique i such that $k \in K_i$.

For a particular search tree $t = t_0, k_0, ...$ we have $keys(t_i) \subseteq K_i$. This allows us to state things succinctly: for example, rather than saying "find i such that $k_{i-1} \leq k < k_i$", we can say instead "find i such that $k \in K_i$".

Given a key of interest k, a search tree provides an efficient way to navigate to the leaf that possibly contains k and its associated value v.

Informally the algorithm for *find* is as follows: given a search key k, and an initial search tree t, starting at the root node

- in a non-leaf node, find i such that $k \in K_i$ (i.e., $k_{i-1} \leq k < k_i$), with subtree t_i (which possibly contains k) and descend to child t_i.
- stop when you reach a leaf; return the value (if any) associated with k in the leaf.

Given k and t, the *find* algorithm returns the value v associated with k in t (if any). It follows that the algorithm returns v iff the pair (k, v) is in the map corresponding to t. We might call this map $m(t)$ or m_t.

Definition The Isabelle/HOL definition of m_t used in this work is:

```
definition tree_to_map :: "Tree => (key,value_t) map" where
"tree_to_map t = (map_of (List.concat(tree_to_leaves t)))"
```

The `tree_to_map` definition concatenates all the key-value pairs in the leaves and produces a map through `map_of`. The gathering of pairs is achieved through the `tree_to_leaves` definition:

```
function tree_to_leaves :: "Tree => leaf_lbl_t list" where
"tree_to_leaves t0 = (case t0 of
Node(l,cs) => (
(cs |> (List.map tree_to_leaves)) |> List.concat
)
| Leaf(l) => [l]
)
"
```

This definition applies recursively on all the subtrees until it reaches the leaves, from which it can produce the key-value pairs.

Then there are two versions of *find*: $\mathit{find}_m\ k\ m$ which operates on maps, and $\mathit{find}_t\ k\ t$ which operates on search trees. Normally we omit the subscripts, since the particular version of *find* is determined by the type of the last argument (map or tree).

So, the algorithm above implements $\mathit{find}\ k\ t$ for the tree t, and this function "behaves the same as" $\mathit{find}\ k\ m_t$ in the sense that $\mathit{find}\ k\ t = \mathit{find}\ k\ m_t$. In later sections we treat more complicated variants of the "behaves the same as" concept.

Lemma find_t is correct, in the sense that: for all k, t we have $\mathit{find}\ k\ t = \mathit{find}\ k\ m_t$.

Proof We show $\forall\ t\ k.\ \mathit{find}\ k\ t = \mathit{find}\ k\ m_t$ by induction on the size of t. The leaf case is trivial. For the non-leaf case, we have that $t = t_0, k_0, ...$ There is a unique i such that $k \in K_i$. Consider t_i. Apply induction hypothesis to obtain $\mathit{find}\ k\ t_i = \mathit{find}\ k\ m_{t_i}$. Moreover $\ldots\ = \mathit{find}\ k\ m_t$ since m_t is the disjoint union of m_{t_i}. Finally $\mathit{find}\ k\ t = \mathit{find}\ k\ t_i$ by definition of find_t.

4.2.3 State transition systems and invariants

We defined labeled transition systems in §2.4.1. A state transition systems is equivalent to a labeled transition system without labels. We refer to the set of states as S, the set of initial states as S_{init}, and the set of transitions as T.

We use the notation $s\ T\ s'$ to represent $(s, s') \in T$.

A transition sequence (or T-sequence) is a (finite or infinite) sequence $s_0, s_1, \ldots$ where for each pair of states (s_i, s_{i+1}), the pair is a valid transition $s\ T\ s'$.

A trace is a T-sequence where the initial state s_0 is drawn from S_{init}.

A property is, for our purposes, a subset of S. In other words, a property P picks out elements of S that "satisfy the property". $P(s)$ is true iff $s \in P$. Alternatively, we may say that P holds for s. The notation $P(s)$ is often used to emphasize that P is a property of state s.

A T-invariant is a property such that the following holds: if $P(s)$ and $s\ T\ s'$, then $P(s')$. An invariant is relative to the set of transitions T. The word "T-invariant" emphasizes the role of T. This is useful when we work with state transition systems (S', T') that are restrictions of some (S, T). In this settings, when we can distinguish a T-invariant from a T'-invariant, and any T-invariant is T'-invariant. This definition considers transitions sequences and not traces, as the initial state is unconstrained.

A trace-invariant of a state transition system (S, T) is a property $P(t)$ that is a T-invariant and which holds for all initial states.

A state is reachable when there is some trace that includes the state.

An invariant of a state transition system (S, T) is a property $P(s)$ that holds for all reachable states.

Essentially, showing that some property is an invariant of a state transition system usually involves showing that the property is a trace-invariant, which in turn requires showing that the property is a T-invariant. However, it is often useful to separate these concepts.

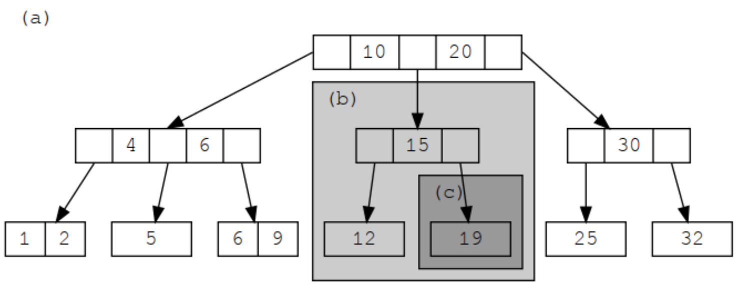


Figure 4.2: Example of descending a search tree by using a framestack

4.2.4 Traversing a tree, context and focus

Tree algorithms typically involve traversing the tree from top to bottom. The algorithm starts at the root of the tree, labeled (a) in Figure 4.2, and descends to a child (b) until finally reaching a leaf (c).

We may view the algorithm as operating on the "root node", and at each step we move to a child node. This is the "graph" view of an algorithm. It emphasizes that nodes are considered in isolation, and pointers are followed to reach other nodes.

Alternatively, we may view the algorithm initially operating on the whole tree, and at each step changing focus to operate on a subtree. This is the "algebraic datatype" view of an algorithm. This viewpoint abstracts over pointers, which simplifies reasoning over the operational semantics and as a consequence this simplifies formal proofs.

In this work, we emphasize the "algebraic" viewpoint. When dealing with real blocks and pointers between blocks we are forced to consider the "graph" view.

At (a), the focus is on the entire tree. The context, which is the part of the tree that we do not focus on, is empty. At (b), we focus on the light-gray subtree, which contains 15, 12, and 19. The context is "the rest of the tree", this refers to all nodes and edges not in the light-gray subtree. The context is comparable to the whole tree, but with a "gap" or "hole" where the subtree (b) should be. We will consider how to model this hole in later sections.

At (c), we focus on the dark-gray subtree containing a single leaf node. The context is "the rest of the tree", that is the whole tree but with a gap or hole where the leaf should be.

Informally, a focus is the subtree we are currently dealing with, and the context is the remaining part of the tree which we use to assemble the final result.

The syntax $t[s]$ is the tree t with subtree s as focus. When we want to refer to the context we write $t[]$. When we want to emphasize the location of the focus s in the original tree t we write $t[s]_p$, where p is the path from the root of t to the focus s.

As we will see in later sections, for *find* the focus is always a subtree and for *insert* the focus is two trees separated by a key as this operation may cause the split of the tree.

4.2.5 Refinement, small step vs big step

"Mathematical" functions map an argument to a result. There is no notion of the "steps" of the function, since the underlying model of a function is as a set of pairs.

In computer science, however, programs typically do take several steps to execute. The distinction is important when the step-based nature of the computation can be observed. Concurrency is one way that the internal workings of an algorithm may be observed by another process. However, even for a single process, we may wish to model the program's steps explicitly. In the case of a B-tree, disk reads and writes can be observed by other processes, but even for a single process we want to argue that, say, the algorithm behaves correctly in the presence of host failure (which may occur at any point during the program's execution). A host failure causes the whole system to enter a "halt" state, which it exits when the user restarts the host. Since the failure can occur at any point, the essential step-based nature of programs that use the disk is observable: a subset of the reads and writes may have happened before the host failed with the possibility of exposing a state different from the one obtained without host failure.

So far, we have considered the *find* operation executing on a map and on a tree.

$$\begin{tikzcd}
\boxed{\texttt{m}_t\texttt{, find k m}_t} \arrow[r] & \boxed{\texttt{m}_t\texttt{, v}} \\
\boxed{\texttt{t, find k t}} \arrow[r] \arrow[u] & \boxed{\texttt{t, v}} \arrow[u]
\end{tikzcd}$$

Figure 4.3: Equivalence of tree and map's find

In Figure 4.3 we show (bottom left) the tree t, and the operation *find* k t evaluating to t (unchanged) and the result v. The top of the diagram shows the equivalent "mathematical function" evaluating on the map m_t. The vertical arrows represent the relation between t and m_t.

In this diagram, both versions of *find* take a single step to evaluate. We now wish to address a more complicated scenario where the implementation of *find* on a search tree takes multiple steps. Each of these steps nominally corresponds to at most one disk operation, so that we properly capture the fact that disk accesses (or, more exactly, requests for disk accesses) occur in sequence over a period of time.

In Figure 4.4, we simplify the previous diagram by suppressing the arguments t and m_t.

$$\begin{tikzcd}
\boxed{\mathtt{m_t,\ find(k)}} \arrow[r] & \boxed{\mathtt{m_t,\ v}} \\
\boxed{\mathtt{t,\ find(k)}} \arrow[r] \arrow[u] & \boxed{\mathtt{t,\ v}} \arrow[u]
\end{tikzcd}$$

Figure 4.4: Equivalence of tree and map's find without arguments

In Figure 4.5, we include multiple steps for the implementation $find_t$. In this diagram we have used the notation $t[s]$ to make clear that the implementation descends from the root of the tree to the subtree s, and then to the subtree s'.

$$\begin{tikzcd}
\boxed{\mathtt{m_t,\ find(k)}} \arrow[rrr] & & & \boxed{\mathtt{m_t,\ v}} \\
\boxed{\mathtt{t,\ find(k)}} \arrow[r] \arrow[u] & \boxed{\mathtt{t[s],\ find(k)}} \arrow[r] & \boxed{\mathtt{t[s'],\ find(k)}} \arrow[r] & \boxed{\mathtt{t,\ v}} \arrow[u]
\end{tikzcd}$$

Figure 4.5: Equivalence of tree and map's find with steps

The sequence of steps of the implementation constitute a "refinement" of the single step of $find_m$.

It is possible to use the formal notion of state transition systems to make the "refinement"[1] notion precise. For the moment it suffices to understand that an implementation is a refinement of a specification when operations, such as *find*, "behave the same" given that we choose to ignore some details of how the implementation behaves (in this case, we choose to ignore the fact that the implementation takes multiple steps whilst the specification consists of only one).

4.3 B-tree definition

In this section we give the formal definition of a B-tree. B-trees are search trees which are:

- balanced (every leaf is at the same distance from the root)
- have minimum and maximum bounds on the sizes of nodes and leaves; individual nodes and leaves can vary in size between these bounds

The bounds on the sizes of nodes and leaves are chosen for performance reasons, and to match the blocksize of the backing block device.

The fact that the tree is balanced, together with a reasonably chosen minimum node size (e.g., half the maximum node size), guarantees $O(log\ n)$ access to any leaf.

The use of minimum and maximum bounds, rather than a fixed size, means that (potentially expensive) tree rebalancing occurs rarely, when compared to, for example, a binary search tree.

A B-tree is a tree where nodes are of the form $t = t_0, k_0, ..., k_{n-1}, t_n$ and leaves are of the form $(k_0, v_0), (k_1, v_1), ...$ Keys in nodes and leaves are ordered in strictly-increasing order.

Definition The Isabelle/HOL definition for a tree is

```
datatype Tree = Node "node_lbl_t * Tree list" | Leaf "leaf_lbl_t"
```

where `node_lbl_t` stands for a list of keys and `leaf_lbl_t` for a list of key-value pairs.

[1]In the Oxford English Dictionary, one definition of refinement is: "the improvement or clarification of something by the making of small changes". Here we clarify HOW the find operation is implemented. The move from a single step to multiple small steps also recalls the essential "breaking down" and "separating out" aspects of refining in various forms.

> This inductive definition only defines a tree, the B-tree constraints are validated through a predicate which is given in Appendix C. This predicate is of key importance for proving that the operations provided always produce wellformed B-trees.

Additionally the following are satisfied:

- the tree is balanced
- all nodes and leaves (except possibly the root) have sizes consistent with the min/max bounds
- for a node $t = t_0, k_0, ...$, have $t_i \subseteq K_i$

A B-tree is, therefore, simply a balanced search tree with size constraints on nodes and leaves.

4.4 Overview of approach to correctness

Our approach to showing the correctness of the B-tree involves two refinements.

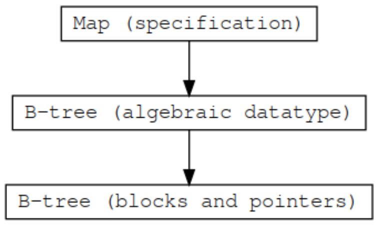


Figure 4.6: Correctness refinements

At the top level, we have the specification, which is simply the "mathematical" description of a map. Each operation "completes" in a "single step". No tree-like data structures are present.

The map level is expressed at a high level of abstraction. The map interface is exposed to users, who are already familiar with the map operations. However, at this level it is not possible to understand the on-disk behaviour of the B-tree, including how it behaves when the host fails.

At the next level, we have the B-tree modeled as an algebraic datatype. At this level the operations take multiple steps (corresponding to disk accesses), but there are no blocks or pointers. It is at this level that the most interesting aspects of B-trees can be captured, including the rebalancing required during *insert* and *delete* operations. However, this level is still not sufficiently detailed to capture the behaviour when the host fails.

At the lowest level, we have the B-tree modeled in full implementation detail, with disk blocks, and pointers between blocks. Caching and the behaviour of the underlying disk are all important at this level. It is at this level that the behaviour under host failure can be expressed.

Each level is a refinement of the level above.

We show the correctness of the datatype level by showing it is a refinement from the map specification. And then there is a further refinement from the datatype to the "blocks and pointers" version of the code. The algebraic operations are expressed in small-step semantics in order to guarantee that the proofs of correctness remain valid also under concurrent usage.

4.5 Framestacks: a concrete representation of context

In previous sections we discussed how the *find* algorithm descended a search tree. At each step the algorithm focused on a subtree, and the rest of the tree was dubbed the "context". We now make this notion more precise.

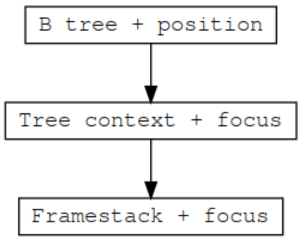


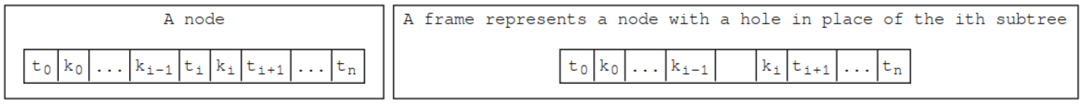
Figure 4.7: Context refinement from a graph view to an abstract and concrete algebraic view

In the diagram of Figure 4.7 we start off with the notion of a position in a tree. This is the "graph" view, where algorithms operate on single nodes in a tree and follow pointers to child nodes. For concreteness, a position or path in a tree might be the sequence of child indexes to reach a particular node. Thus $(0, 1)$ represents the node that can be found by following the 0th child from the root, and then following the 1st child from that node (indexing from 0).

For the "datatype" view, it is more natural to identify a particular position in a tree with the subtree at that position. Then we need to find a way to formalize the context of the "rest of the tree". We could, as above, maintain the original tree and a path to the "current" subtree. However, we choose to develop a different notion of context based on framestacks. The reason is that we need to implement algorithms on the context, and these algorithms can be more easily expressed using framestacks.

The lowest level in the diagram involves reifying the notion of "tree context" as a concrete data structure called a framestack.

Definition A frame is a node with a "missing" child; for a node $t = t_0, k_0, ...,$ a frame is a pair $(t_0, k_0, ..., k_{i-1}), (k_i, t_{i+1}, ...)$. Compared to the original node, the child t_i is missing.


A node
t0 k0 ... ki-1 ti ki ti+1 ... tn
A frame represents a node with a hole in place of the ith subtree
t0 k0 ... ki-1 ki ti+1 ... tn

We may write a frame as t/t_i or $t[]_i$, meaning that in node t the ith child is missing.

Definition A framestack is a list of frames $f_n, ..., f_0$. The frame f_0 corresponds to the root node, and the frame f_n corresponds to the parent node of the current subtree.

The Isabelle/HOL definition of framestack is:

```
datatype 'f framestack = Frame_stack "'f focus_t * context_t"
```

Where a focus is defined as a generic data type to allow its specialization for the specific operation (i.e., *find* or *insert*):

```
datatype 'f focus_t = Focus 'f
```

And a context is defined as a list of tuples that contain the parent leftmost and rightmost keys (bounds) and a pair consisting of the subtree (i.e., `node_t`) associated with its index in the parent node:

```
type_synonym context_t = "(left_bound * (node_t * nat) * right_bound) list"
```

In Figure 4.8 we show the framestack produced by descending a simple tree. We aim to reach the leaf with the key *19*. At the beginning we have an empty framestack, and take the root as our initial frame. We then pick as new frame the child node corresponding to the last key that is smaller than *19*, and add the

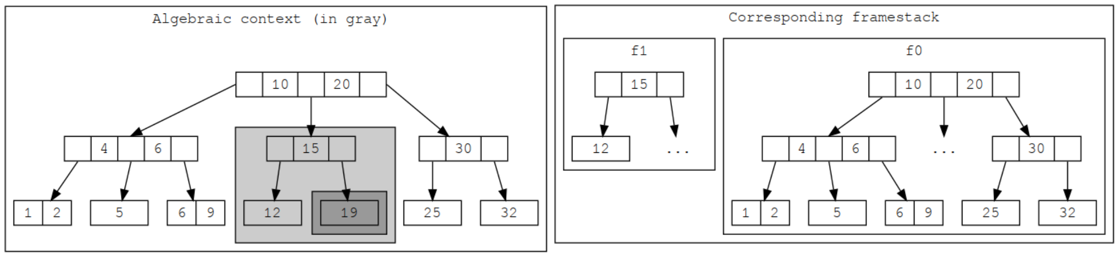


Figure 4.8: Example of algebraic context and framestack equivalence

current frame (labeled as `f0`) to the framestack. We then choose as next frame the leaf with the key we are looking for and the current frame is added to the framestack (labeled as `f1`). At this point we have reached the bottom of the tree.

It is possible to reconstruct a tree given a context and a focus.

Definition given an initial tree, a context is a framestack and a focus is a tree such that the context and focus can be combined to give the initial tree.

The algorithm to combine a framestack and a focus to obtain the initial tree can be expressed formally as:

- if framestack is empty, return the focus;
- otherwise pop the stack, fill the hole with the focus (this may require restructuring, see §4.7), and reiterate by using the obtained tree as the focus.

Given a framestack *ctxt* and a focus t, we use the notation $ctxt(t)$ to express that we are combining the framestack and focus to obtain a complete tree.

4.6 Find

We are now in a position to describe the *find* implementation formally.

4.6.1 Descending a search tree

We define *find* as a transition sequence. Each transition (or step) operates on a state. The state is composed by:

- the context (*ctxt*), as a framestack;
- the focus (t), as a tree;

- the key (k) we are looking for;
- lower (l) and upper (u) bounds on keys appearing in the focus.

We include bounds on keys because they will be significant in our discussion of *insert* and *delete*, although they are not relevant for *find*. However *find* can be used as a subroutine for *insert* and *delete*, where these bounds are relevant.

A *find* state can be then defined as $s = (ctxt, t, k, l, u)$. Descending the tree is equivalent to the sequence $s\ T\ s_1, ..., s_{n-1}\ T\ s_n$, where s_n corresponds to a state where the focus is a leaf.

4.6.2 Basic algorithm

We define the *find* step in pseudo-code as follows (we provide the formal definition in Appendix C):

```
// k - the search key (constant for duration of find steps)
// ctxt - the context (a framestack)
// t - the focus (a tree)
// l,u - lower and upper bound on keys appearing in the focus t
define find_step(ctxt,t,k,l,u) = {
  if (t is a leaf) then return (ctxt,t,l,u)
                                       // NB t is a node...
                                       //   (t0,k0,...,k(n-1),t(n))
  i     <- get_child_index (k0,...) k  // find i st k(i-1)<=k<k(i)
  frame <- t/t_i                       // new frame, with hole at t(i)
  ctxt  <- cons(frame,ctxt)            // add frame to context
  t     <- t(i)                        // set focus to child t(i)
  l     <- if i>0 then k(i-1) else l   // update l
  u     <- if i<n then k(i) else u     // update u
  return (ctxt,t,l,u)                  // return updated values
}
```

Here the state remains unaltered if the focus is a leaf, otherwise we select the child that may contain the target key, we create a frame without this child, we update the bounds on the keys and return the updated state.

In the initial state the context is an empty framestack, the focus is the original tree and our bounds are respectively $+\infty, -\infty$[2] to represent that the root tree has no bounds on keys.

Then we define *find* as the transition sequence:

```
define find(k,t) = {
  ctxt <- empty             // initial values...
  l <- -inf
  u <- +inf
  while(t is not a leaf) {  // repeatedly apply find_step
    ctxt,t,l,u <- find_step(k,ctxt,t,l,u)
  }
  // t is a leaf
  return (value associated with k in leaf t, if any)
}
```

4.6.3 Wellformedness

We say that a tree is wellformed when it satisfies the definition given in §4.3 (a formal definition of wellformedness can be found in Appendix C).

This can be summarized in the following property over a tree: $\mathit{wellformed} = \mathit{wf\_size} \wedge \mathit{wf\_ks\_rs} \wedge \mathit{balanced} \wedge \mathit{keys\_consistent} \wedge \mathit{keys\_ordered}$, where *wf_size* and *balanced* enforce the B-tree specific properties (e.g., boundaries on the node size) and the others check the search tree properties (e.g., the total strictly increasing order of the keys).

We use the wellformedness predicate on the tree to define when the *find* operation is in a wellformed state.

We say that the state of the *find* operation is wellformed when:

- any tree in the framestack is wellformed;
- any tree in the focus is wellformed;
- the lower and upper bounds are the limits of K_i, where i corresponds to the tree missing in the parent frame.

We will refer to this wellformedness property as $W(s)$.

[2]In the Isabelle definitions the bounds are represented as options.

Now we want to say that, given the initial state is wellformed $W(s_0)$, there is a T-invariant $W(s)$ on *find*: this means that *find*'s state remains wellformed for any *find* transition, which implies that all the trees involved stay wellformed as well.

We can prove trivially that this invariant holds using the $W(s_0)$ hypothesis, as this implies that all subtrees are wellformed and *find* transitions do not alter trees and choose bounds according to $W(s)$ definition.

4.6.4 Correctness

The *find* operation is correct when its output is the same as calling $find_m$ on m_t. This is equivalent to say that when we reach a leaf l, the bounds have been correctly chosen, $ctxt(l)$ is the original tree and $(k, v) \in map(ctxt(l)) \iff (k, v) : l$.

The proof of correctness relies on the fact that the keys are in a total strictly-increasing order: on this hypothesis we know that if a pair (k, v) exists, it can only be in the leaf l and that if it does not exist in l, it cannot exist in any other leaf.

We formalize this lemma as:

```
definition invariant_map_equivalence_find :: "bool" where
"invariant_map_equivalence_find == (
! fs.
let map_equivalence_find =
(
 case step_fs fs of
 Some fts' =>
 let k  = fst (dest_f_frame_stack fs) in
 let m  = fs_to_map fs in
 let m' = fs_to_map fs' in
 ((m k) = (m' k))
 | _ => True
)
in
 total_order_key_lte -->
 wellformed_fs fs --> map_equivalence_find
)"
```

where `total_order_key_lte` and `wellformed_fs_fs` represent the total order and $W(s_0)$ hypothesis; `fst (dest_f_frame_stack fs)` returns they key we are looking for; `fs_to_map` is equivalent to $ctxt(l)$; `step_fs` represents the *find* step on the framestack.

We prove the lemma by inducting on $ctxt(l)$; if it is a leaf, then this is trivial as the total order guarantees the uniqueness of the keys; if it is a node we know that there is only one child which contains k for the total order, so we can apply the induction hypothesis over the disjoint union of the children. The mechanized proofs for this command are available at [85].

4.7 Insert

The *insert* operation is more complicated than *find*, because it potentially involves splitting nodes that are too big.

4.7.1 Inserting in a tree

Insert uses *find* to locate the leaf in which to insert a new (k, v) pair.

For concreteness, we define a version of *find* that is used by *insert* (and *delete*):

```
define find' (k,t) = {
  ctxt <- empty              // initial values...
  l    <- -inf
  u    <- +inf
  while (t is not a leaf) {  // repeatedly apply find_step
    ctxt,t,l,u <- find_step(k,ctxt,t,l,u)
  }
  // t is a leaf
  return (ctxt,t,l,u)
}
```

Note that, compared to *find*, *find'* returns a lot more information: the context, the leaf, and the lower and upper bounds are all returned.

Suppose the leaf is of the form $(k_0, v_0), (k_1, v_1), ..., (k_n, v_n)$. If the leaf is not already at the maximum size allowed (i.e., $n < max\_size$), we can just insert the new key-value pair to obtain a new leaf. We can then combine this new leaf with the context to obtain a new tree.

If the leaf already contains k, we can update it with the new key-value pair even if the leaf has achieved its maximum size.

However a more difficult case to handle occurs when the leaf has its maximum size (i.e., $n = max\_size$), and the key is not already present in the leaf. Adding the new pair would result in a leaf that is too big. In this case, the leaf of length $max\_size + 1$ is divided in two leaves, separated by a key. Suppose we divide at position i. Then the first leaf is $leaf_1 = (k_0, v_0), (k_1, v_1), ..., (k_{i-1}, v_{i-1})$, while the second leaf is $leaf_2 = (k_i, v_i), (k_1, v_1), ..., (k_n, v_n)$. The key that separates them is k_l, in the sense that $leaf_1, k_l, leaf_2$ represents a valid partition of the original leaf. Figure 4.9 shows the result of inserting the pair $(4, v)$ (we ignore the value for simplicity) in a tree with a full leaf.

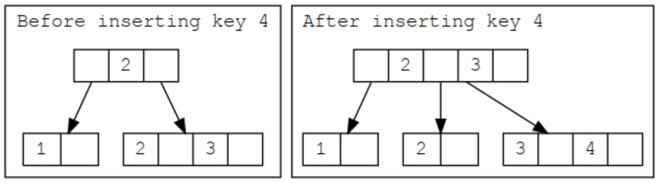


Figure 4.9: Example of insert with splitting

Suppose the context has frame $(..., t_{i-1}, k_{i-1}), (k_i, t_{i+1}, ...)$ at the head (corresponding to the parent of the leaf). At this point, one possible scenario is that we could insert the new leaves to get a new node $(..., k_{i-1}, l_1, k_l, l_2, k_i, ...)$. However, the other possible scenario is that this node in turn is too big, since it has one more child than the original. In this case, we must again split the node, and repeat with the next frame on the framestack.

Eventually we may end up with a root that is too big, at which point we split the root in two, and create a new root with two children. Now, the height of the tree grows by one. Figure 4.10 shows the result of inserting the pair $(6, v)$ in a tree with a full leaf: firstly we insert the key 6 in the leaf that contains already keys 4 and 5 (only keys are considered for simplicity); we obtain an oversized leaf; then we split this leaf and create a focus with a leaf containing 4, the key 5 and a leaf containing 6, and we insert the focus in the context; at this point we try to add key 5 into the parent node and obtain an oversized node; after creating a new focus with the two nodes obtained from splitting and the key 4, finally, we create a new root with the key *4*.

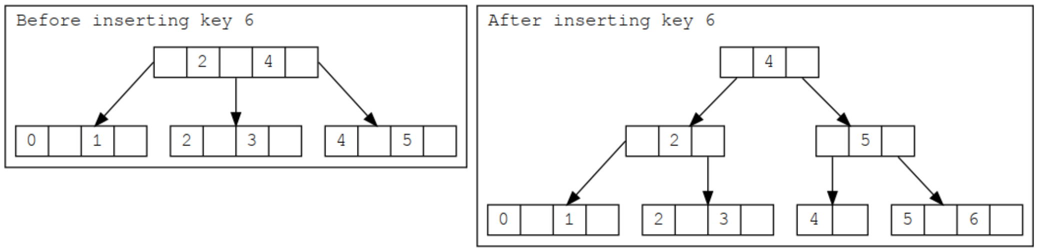


Figure 4.10: Example of insert with splitting and merging of root

4.7.2 Basic algorithm

The operation *insert* has three stages: the first stage involves descending the tree to a particular leaf of interest, the second stage involves altering that leaf to get a new leaf with an extra key or two new leaves separated by a key, and the stage where we assemble the tree to insert the tree (or trees) in the focus, potentially splitting the obtained tree if they become oversized. In the pseudo-code below, we wrap this information in a sum datatype (a formal definition can be found in Appendix C):

```
data insert_stage x = Desc(x) | Bot(x) | Asc(x)
```

Now we can define the *insert* step in pseudo-code as follows:

```
// ist - the stage of the insert step
// fts - the stage of the find step (i.e., (k,ctxt,t,l,u))
// (k,v) - the key and the value to insert
//        (constant for duration of insert steps)
// ctxt - the context (a framestack)
// t - the focus (a tree when descending,
//        may be also a triple (t1,k_m,t2) when ascending)
// l,u - lower and upper bound on keys appearing
//        in the focus t
define insert ist = {
  case ist of
    | Desc(fts,v) ->
      if (t is not a leaf)
      then  // descends one level of the tree
```

```
        ctxt',t',l',u' <- find_step fts
        Desc((k,ctxt',t',l',u'),v)
      else  // sets up for inserting (k,v) in the leaf
        Bot(fts,v)
    | Bot(fts,v)  ->
      k,ctxt,t,l,u <- fts  // NB: t is a leaf a this point
      t' <- insert (k,v) t // obtains a new leaf
                           // from adding (k,v)
                           // (or updating
                           // if key is already in t).
                           // NB: this must keep
                           // the keys ordered.
      if (size of t' >= max size)
      then  // only if it was an addition
        (t',k_l,t'') <- split_leaf t' // it partitions
                                      // the oversized leaf
                                      // in a triple (t1,k_m,t2)
        return Asc(ctxt,(t',k_l,t''),l,u)
      else Asc(ctxt,t',l,u)
    | Asc(ctxt,t,l,u)  ->
      if (ctxt empty)
      then  // if the context is empty, t is the root:
            // no need to ascend further
        return t
      t',ctxt',l',u' <- pop ctxt  // pop the head of ctxt and
                                  // get the lower and upper
                                  // bounds of the upper node
      t' <- fill_hole t' t  // fill the node with a missing child
                            // with t
      if (size of t' >= max size)
      then  // if t was a triple, then t' may need splitting too
        t' <- split_node t'  // it partitions the oversized node
                             // in a triple (t1,k_m,t2)
      Asc(ctxt',t',l',u')
}
```

Here we have different cases according to the stage of the operation. If descending, we just apply the *find* step described earlier. If we reach the leaf, we update the leaf with the pair (k, v). If the obtained leaf is oversized, we divide it into (l_1, k_l, l_2) and use this as the focus for ascending the tree. If ascending, we check that there are parent nodes to ascend: if not the insertion is complete. Otherwise, we obtain the parent node with the hole in it (t') and fill the hole with the focus t. Filling the hole may be as simple as inserting t as the missing child or it may need the addition of a k_m and a second tree t_2, in this case t' was divided into t_1, k_l, t_2. If t' was divided in this manner, inserting more than one child into the hole may imply a new splitting of the focus. This can cascade to the point that a new root is created.

Similarly to what we have done for *find*, we define *insert* as the transition sequence:

```
define insert((k,v),t) = {
  ctxt <- empty                      // initial values...
  l <- -inf
  u <- +inf
  fts <- k,ctxt,l,u
  ist <- Desc(fts,v)
  while(ist is not the new root) {    // repeatedly apply insert_step
    ist <- insert_step(ist)
  }
  // ist is the new root
  return ist
}
```

4.7.3 Wellformedness

We say that an *insert* stage is wellformed when:

- any tree in the framestack is wellformed;
- any tree in the focus is wellformed and the dividing key k_l of the focus (if any) is $\forall k_i \in K_i, k_j \in K_{i+1}. k_i < k_l \leq k_j$;
- the lower and upper bound are the limits of K_i, where i corresponds to the tree missing in the parent frame.

Now we can say that, given the initial state is wellformed $W(s_0)$, there is a T-invariant $W(s)$ on *insert*.

We can prove that this invariant holds using the $W(s_0)$ hypothesis. Note that this proof is more laborious than the one for the *find* operation as we have to demonstrate that the invariant holds for all states of *insert* and for each kind of focus we operate upon (i.e., both the single tree and the partion t_1, k_m, t_2).

4.7.4 Correctness

The *insert* operation is correct when its output is the same as calling $insert_m$ on m_t. This is equivalent to say that:

1. when we reach a leaf l, the bounds have been correctly chosen, $ctxt(l)$ is the original tree and $(k, v) \in map(ctxt(l)) \iff (k, v) : l$;
2. after inserting (k, v) in l we obtain l' such that $ctxt(l') = (ctxt(l) + (k, v))$;
3. given a framestack *ctxt* and a focus t, and their immediate parent $ctxt'$ and focus t', we have $ctxt(t) = ctxt(t')$

We have already shown (1) for the *find* operation. We can prove (2) similarly by showing that the original map is the union of all the maps obtainable from the leaves $ctxt(l') = m_{l_1} + .. + m_{l'} + ... + m_{l_n}$; so we are left to show $m_{l_1} + .. + m_{l'} + ... + m_{l_n} = m_{l_1} + .. + m_l + ... + m_{l_n} + m(k, v)$, which is trivial as k only belong to l and l' for the total order hypothesis. We can prove (3) by showing that the ascending step does not alter the tree contents. The mechanized proofs for this command are available at [85].

4.8 Refinement to block device

So far, we have dealt with B-trees as algebraic datatypes, i.e., as trees. Real storage hardware, such as hard disk drives (HDD) and solid-state drives (SSDs), work with blocks of bytes. Whilst the algebraic viewpoint suffices to discuss the correctness of operations - such as *find*, *insert* and *delete* - we need to explain how to extend our treatment to deal with the additional complexities of block storage.

What are block devices? Most storage operates in terms of a block model. A block is simply a large contiguous chunk of bytes, for example, 1024 bytes or 4096 bytes are common block sizes. Blocks are read and written as a whole, and addressed by block number. Thus, block reads and writes must occur at a byte address that is a multiple of the block size. For example, if the block size is 1024, it is possible to read 1024 bytes from position 2048 (a multiple of the block size) in a single operation, but to read 1024 bytes from position 2049 would require two block reads (one at offset 2048, and one at 3096).

The performance of file systems depends on minimizing the number of block operations. The B-tree data structure allows the branching factor of nodes, and the size of leaves, to be chosen so that a full node fits exactly in one block. This typically reduces the number of block operations needed when executing the B-tree map operations, and therefore makes B-trees a very good fit for block devices. It is worth to note, however, that B-trees impose some "organizational" overhead as well, for instance maintaining the tree structure rather than indexing directly into the data.

The algebraic model treats a tree as a node consisting of children which are subtrees (Figure 4.11).

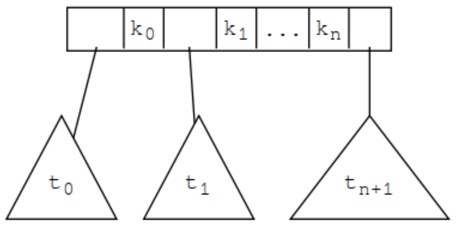


Figure 4.11: Algebraic view of a B-tree

When dealing with block devices, we instead model the children via pointers (Figure 4.12).

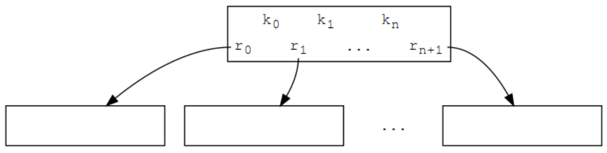


Figure 4.12: Block view of a B-tree

Here r_i is the pointer to the block representing the root of t_i. Given a pointer r to a block representing the root of a tree, it is easy to reconstruct the tree as an algebraic structure. Thus, the block model is a refinement of the tree model: what is represented is the algebraic tree structure, but the block model exposes the pointers that are used to encode this structure on top of a block device.

The real-world nature of block devices requires us to take extra care because all operations may potentially fail in a number of ways. At the most extreme, a USB device can be unplugged at any point, causing the entire block device to disappear. More mundanely, the device may become full and unable to service further requests. These complexities can be dealt with uniformly through the use of a monad (a technique from functional programming) to syntactically hide the numerous error cases which otherwise would cause the B-tree code to become unreadable.

The use of a block device also entails that we must somehow keep track of which blocks are in use. There are several ways to implement this. Perhaps the simplest maintains the "minimum free block" number, and simply increments this when blocks are allocated. When free blocks are exhausted, we can simply transfer all in-use blocks to another device, and continue. This has the downside that one needs two block devices, and most of the time one of the devices is inactive. To be usable, the transfer must occur ahead of time, and asynchronously, so that there is no perceptible pause while the transfer takes place. An alternative is to maintain an explicit free map on the block device itself. Yet another alternative is to reuse an existing logical block manager such as is found in Linux LVM.

Beyond this, we must also address mundane issues such as how to marshal a tree node (with pointers!) to a block-sized byte sequence. Typically nothing depends on exactly how this is done, and we are free to choose whichever marshalling scheme is most suitable.

Suprisingly it is possible to deal with all these issues whilst still keeping the essential B-tree routines short and readable. The main additions to the code are explicit block allocations and frees. To see how this is done, we refer the interested reader to [85], which also includes the development of *delete* routines, and further B-tree operations such as list all keys . Executable OCaml code, extracted from the formal development and packaged so that it is accessible as an OCaml library, can also be found in [84].

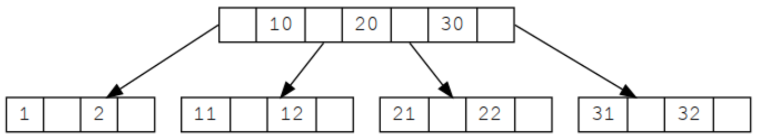


Figure 4.13: A B-tree according to Bayer and McCreight with branching factor 4

4.9 Related work

4.9.1 Proposed versions of the B-tree data structure

Bayer and McCreight introduced the term B-tree for the first time in 1970 [7] (Figure 4.13), and their definition is the following:

> Def. 2.1. Let $h \geq 0$ be an integer, k a natural number. A directed tree T is in the class $\tau\ (k, h)$ of B-trees if T is either empty ($h = 0$) or has the following properties:
>
> i) Each path from the root to any leaf has the same length h, also called the *height* of T, i.e., $h =$ number of nodes in path.
>
> ii) Each node except the root and the leaves has at least $k{+}1$ sons. The root is a leaf or has at least two sons.
>
> iii) Each node has at most $2k + 1$ sons.
>
> iv) Each node holds between k and 2k keys except the root node which may hold between 1 and $2k$ keys.

Then they discuss the properties that a B-tree must have in order to allow retrieval from the store:

> To repeat, the pages on which the index is stored are the nodes of a B-tree [...] and can hold up to $2k$ keys. In addition the data structure for the index has the following properties:
>
> i) Each page holds between k and $2k$ keys (index elements) except the root page which may hold between 1 and $2k$ keys.
>
> ii) Let the number of keys on a page P, which is not a leaf, be l. Then P has $l + 1$ sons.
>
> iii) Within each page P the keys are sequential in increasing order: x_1, x_2, ... x_l; $k \leq l \leq 2k$ except for the root page for which $1 \leq l \leq 2k$.

Furthermore, P contains $l + 1$ pointers $p_0, p_1, \ldots p_l$ to the sons of P. On leaf pages these pointers are undefined.

iv) Let $P(p_i)$ be the page to which p_i points, let $K(p_i)$ be the set of keys on the pages of that maximal subtree of which $P(p_i)$ is the root. Then for the B-trees considered here the following conditions shall always hold:

$$(\forall y \in K(p_0))(y < x_1),$$

$$(\forall y \in K(p_i))(x_i < y < x_{i+1}); i = 1, 2, ..., l - 1,$$

$$(\forall y \in K(p_i))(x_i < y).$$

Less than a decade later B-trees are "de facto, a standard for file organization" [15] at least in databases [27]. During the 1980's this data structure was further developed, and Knuth proposed two versions: the B*-tree and the B+ tree. The B*-tree is a B-tree in which nodes must be at least two-thirds full (Figure 4.14); the main benefits of such a constraint are an optimized utilization of capacity (at least 66%), and a faster lookup phase, since this variant has a smaller height compared to traditional B-trees.

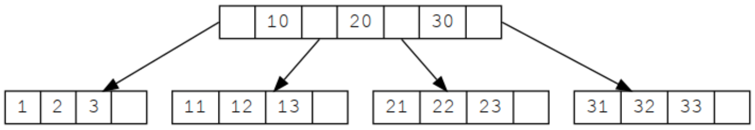


Figure 4.14: A B*-tree according to Knuth with branching factor 4

The B+ tree is a B-tree with keys also in the leaf nodes and with leaf nodes linked together (Figure 4.15); the main benefits of using the B+ tree are evident for sequential operations, because accessing many sequential entries requires a single lookup operation to reach the intial entry and then follow the link between leaves.

In 2008 Rodeh introduces a persistent copy-on-write B+ tree [63]. The persistence of the B+ tree enables the cheap implementation of features such as clones and snapshots. The persistent B+ tree works through a copy-on-write mechanism: every time there is a modification in the tree, the current tree remains intact, and only a copy of the path from the root to the leaf is modified (Figure 4.16). The Btrfs file system implements features like snapshots directly from Rodeh's ideas [64].

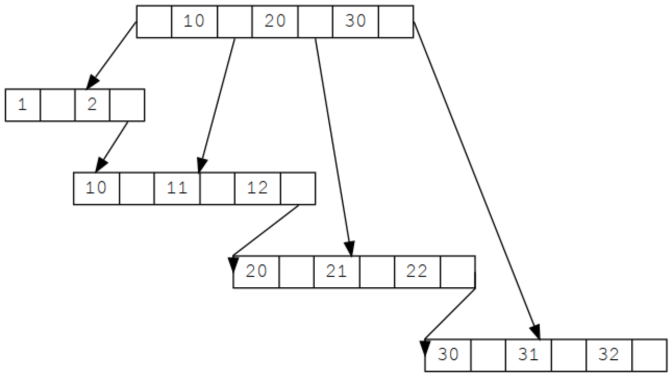


Figure 4.15: A B+ tree according to Knuth with branching factor 4. The empty boxes in the leaf nodes hold the data entries

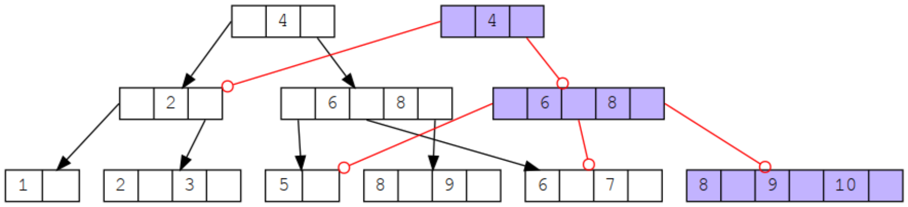


Figure 4.16: The addition of an element in a leaf requires shadowing up to the root

The new tree is achievable by taking the modified path root. With this mechanism one can maintain multiple versions of the same tree by storing the old root and not deleting the nodes belonging to the old path. This B+ tree does not chain its leaves in a linked list, since it makes the copy-on-write mechanism too expensive: the modification of a linked leaf requires the copy of all predecessors and successors in order to have an updated list; but copying all leaves would mean changing every path to the root, i.e., copy the whole tree at any modification. This B+ tree is known in literature as the CoW B-tree.

4.9.2 B-tree verification

In literature there are various attempts to formalize and verify B-trees. These attempts target two different abstract types: ephemeral and persistent ones. Harper distinguish these abstract types as follows:

> The distinction is best explained in terms of the logical future of a value. Whenever a value of an abstract type is created it may be subsequently acted upon by the operations of the type (and, since the type is abstract, by no other operations). Each of these operations may yield (other)

> values of that abstract type, which may themselves be handed off to further operations of the type. Ultimately a value of some other type, say a string or an integer, is obtained as an observable outcome of the succession of operations on the abstract value. The sequence of operations performed on a value of an abstract type constitutes a logical future of that type — a computation that starts with that value and ends with a value of some observable type. We say that a type is ephemeral iff every value of that type has at most one logical future, which is to say that it is handed off from one operation of the type to another until an observable value is obtained from it. This is the normal case in familiar imperative programming languages because in such languages the operations of an abstract type destructively modify the value upon which they operate; its original state is irretrievably lost by the performance of an operation. It is therefore inherent in the imperative programming model that a value have at most one logical future. In contrast, values of an abstract type in functional languages such as ML may have many different logical futures, precisely because the operations do not "destroy" the value upon which they operate, but rather create fresh values of that type to yield as results. Such values are said to be persistent because they persist after application of an operation of the type, and in fact may serve as arguments to further operations of that type. [29]

After providing an implementation of an ephemeral B-tree in Pascal, Fielding proposes a pen-and-paper proof using two refinements with an intermediate level of nested sets [23].

Ernst et al. produced a mechanized verification of an ephemeral B+ tree by integrating shape analysis and interactive theorem proving techniques [21].

Additionally Malecha et al. produced a mechanized verification of an ephemeral B+ tree by using a separation logic framework for the Coq theorem prover [49]; in this work they divide the logic of B-tree operations from the implementation by creating a "p-tree" layer bound to the implementation, and also divide the proof for B-tree validity in geography (the height and arity of the nodes) and keys order, similarly to Sexton and Thielecke. approach [71].

The work of Sexton and Thielecke is probably the closest in the literature to the work we present here. They present proofs of correctness for *find* and *insert*, and operate on a version of B-trees where leaf nodes have sibling pointers. The main similarities and differences with our work are:

- They work with a data structure with explicit links between sibling leaves. Our B-trees are simpler, with no links between leaves. Their presentation is closer to traditional B-trees, whereas ours is similar to recent copy-on-write presentations, where leaf links are omitted [63].
- We have mechanized our definitions and proofs [85]. For non-mechanized proofs, there are (almost inevitably) errors and typos.
- Both works make use of operational semantics.
- They additionally use separation logic. The use of separation logic affects the presentation of lemmas and proofs, but one could argue that this is not a huge difference, since separation logic assertions could be translated into operational equivalents fairly directly.
- They operate on the level of stores, where (for example) page identifiers are explicitly present in the proof. We have proofs at the level of abstract tree datatypes (with framestacks), and the refinement to working with a block device is presented as a further step. This further step additionally includes aspects of separation that they treat using separation logic. Thus, we have separated out the algebraic aspect from the separation/store aspect, whereas they treat both simultaneously.
- Our presentation is in terms of a small-step semantics. If the data structures are not altered by concurrent processes, then the proofs remain essentially the same, and so proofs directly accommodate other concurrent users of the store. Their proofs can perhaps be extended to cope with concurrency (for example, by using a rely/guarantee version of separation logic), but this at best is not immediately clear.
- Their proofs tend to involve inductions on the tree structure, and are (to our taste) rather complicated. Our proofs are direct inductions on the steps taken by the algorithm, to show that various invariants hold. Correctness of the operations follows directly from the invariants. Thus, the proof structure differs between these two works.

5 Conclusion and further work

In the previous chapters we discussed why file systems need verification, we formalized the interesting subset of an existing operating system specification and used it as an oracle against existing file systems. We showed how to extend this formal specification with the timestamps feature, and we formalized a model of copy-on-write B-tree and mechanically verified its commands in order to provide a correct and efficient storage model for future file systems. These achievements are the basis on which one can implement a verified file system that may be attractive for industrial use.

We left unexplored some of the interesting riddles that this study conducted us to:

1. an efficient oracle for SibylFS extended with timestamps:

   in §3.7.3 we demonstrated that validating file system implementations behavior for timestamps is limited to the execution of few commands. We decided to propose the naive timestamp extension to keep the discussion clear. However we considered exploring a timestamp model based on constraints. The core idea is that, in order to avoid the state explosion (due to assigning a value to marked timestamps), we should compute timestamp value directly from the time constraints we extract from a trace. Accumulating time constraints (e.g., $t_1 < t_2$) should not affect the file system state of the model.

The LP-map approach we showed accumulates time constraints, affecting the file system state of the model. Such a system would leave the burden of validating the consistency of observed timestamps in a trace to a constraint solver;

2. specify and verify the B-tree *delete* operation:

   for completeness and real world usability of the B-tree we need to formalize and verify the last basic operation of the B-tree following the approach described in Chapter 4;

3. provide an optimized refinement of the B-tree for the block device:

   we used the algebraic view of the B-tree to model its formal specification, but this needs to be refined to work optimally using the block device. Not only the refined B-tree operations need to store the data fitting the blocks size and blocks representation, but they have also to handle the erroneous cases that a physical device comes with (e.g., a full device). This model has also to manage the blocks used for the B-tree and the references to them. Finally, it needs to be optimized to compete with industrial storage model for file system (e.g., by implementing caching mechanisms for operations);

4. develop a future file system:

   finally we can use all of these achieved artifacts to produce a verified file system. Ideally, we could do this by injecting the refined B-tree as the storage model of SibylFS. This would reuse the file system definitions of the specification as the core of the file system.

We know that some of these open points are already being addressed (e.g., modelling the delete operation and refining the B-tree to the block storage layer [61]) and we have faith that a future file system will soon be born.

In summary, this work does not only show that formal methods are a valid tool to achieve industrial results, but also that the effort they require can produce a plethora of benefits: the identification of numerous specification and implementation inconsistencies, the discovery of simpler models and the development of functionally correct software.

Appendix A: SibylFS main excerpts

This appendix is a short introduction to the internal mechanisms of SibylFS. The building blocks of the specification are the abstract datatypes representing POSIX basic types useful to define the file system behaviour:

```
type ty_bytes = T_list_array.t

type file_contents = ty_bytes

(* a C string, it may be a null pointer *)
type cstring = CS_Null | CS_Some of string

(*file descriptors*)
type ty_fd = FD of nat

(*directory handlers*)
type ty_dh = DH of nat

type inode = Inode of nat

type error =
 E2BIG
| EACCES
| EAGAIN
| ...

type open_flag =
| O_EXEC
| O_RDONLY
| ...
```

Some abstract datatypes are useful to model the file system as a labeled transition systems. The `ty_os_command` datatype for example labels operating system transition which execute commands on the file system. It is a sum type which defines the commands considered by the specification. For instance, the specification considers the `link` command, whose signature expects two strings as arguments (representing the source and the target paths).

```
type ty_os_command =
 | OS_CLOSE of ty_fd
 | OS_LINK of (cstring * cstring)
 | ...
```

So the `ty_os_command` type labels a transition with the command and inputs to execute the command through the operating system layer, during this execution another transition may happen in the file system layer (most of the commands modeled have an effect on the file system). The function which runs the command on the file system is the following:

```
let os_run_os_command env pid cmd s0 = begin
...
(* some setup that requires OS information *)
let pp path =
  process_path env s0.oss_fs_state ppstate.pps_cwd cmd path in

let run_fs_cmd fs_cmd = os_run_fs_command env pid fs_cmd s0 in
...

let run_os_cmd (cmd : ty_os_command) = (
   match cmd with
   (* real os-commands *)
   | OS_CLOSE fd        -> ...
   ...
   | OS_LINK (s,d)      -> (run_fs_cmd (FS_LINK(pp s, pp d)))
```

The `os_run_os_command` arguments are:

- `env` a record data structure containing information about the file system

(e.g., is a POSIX, Linux, Mac OS X, or FreeBSD file system? Are hard links to directories permitted? Etc…)

- `pid` the identifier of the process that should run the command
- `cmd` the command label (e.g., `OS_LINK path1 path2`)
- `s0` the current state of the specification

Note how `run_fs_cmd` requires operating system information to work. For instance, each operating system process has a current working directory which is a necessary information to resolve path correctly (i.e., `process_path` in the code above).

The function that runs the file system command is the following:

```
let os_run_fs_command env pid cmd s0 = begin
 let rs = fs_trans env s0.oss_fs_state cmd in

 let monad_state_to_os_state (ms : monad_state 'impl ret_value) =
 (match ms with
   | Normal_state(fs_st,v) ->
       OS_normal
         (update_pending_return_and_fs_state s0 pid (Value v) fs_st)
   | Error_state(fs_st,e) ->
       OS_normal
         (update_pending_return_and_fs_state s0 pid (Error e) fs_st)
   | Special_state(special,s) -> ( OS_special(special,s))
 end) in
 finset_image monad_state_to_os_state rs

end
```

The function `finset_image` applies a given function to each element of a set. The file system state is a monad in order to make the design uniform and modular §2.3.2.

The function `monad_state_to_os_state` transforms the file system monadic state to its operating system counterpart. Normal states model successful executions, error states erroneous ones, and special states model behaviours that the specification does not aim to consider.

The `fs_trans` function executes the command and returns all the reachable states:

```
...

let fsop_link env spath dpath = (fsm_get_state >>= (fun s0 ->
 let spath = (
   if (is_mac_os_x_arch env) then
     (mac_os_x_map_rpath env s0 spath)
   else spath)
 in
 fsop_link_checks env spath dpath >>=
   (fun _ -> fsop_link_core env spath dpath))

...
let fs_link = fsop_link
...

let ty_fs_command_to_fsmonad env cmd = match cmd with
   | FS_LINK (s,d) -> (fs_link env s d)
   ...
...

let fs_trans env s0 cmd = (
 let m = ty_fs_command_to_fsmonad env cmd in
 let rs = run_fsmonad m s0 in
 rs)
```

So the function `fs_trans` prepares the file system monad through the call to `ty_fs_command_to_fs_monad`, and then it discloses the reachable states from the monad with `run_fsmonad`.

The function `ty_fs_command_to_fs_monad` binds a command label with a command definition. For instance, the `FS_LINK` label is bound to the `fs_link` function, which models the `link` command.

The `link` command source path resolution changes slightly according to the file system architecture considered (see `is_mac_os_x_arch` case in the above excerpt).

Note that the `fsop_link` function uses the monad to compose the error checking `fsop_link_checks` and the command execution `fsop_link_core`.

The error checking returns a set of erroneous states. For instance, erroneous state would result by giving the `link` command arguments such as an existing target path or a not existing source path:

```
let fsop_link_checks env spath dpath = (
 fsm_get_state >>= fun s0 ->
 (* sanity check spath in parallel with dpath *)
 ( match dpath with
     | RN_error(e,<|re_rn=fopt|>) -> (
       fsm_cond_raises [
         (e,true);
         (* tr/23 probable non-POSIX Linux behaviour
            (path should not resolve) *)
         (* coverage:mac_os_x:posix:irrelevant *)
         (EEXIST, is_linux_arch env
                  && isJust fopt
                  && (e = ENOTDIR));
         (* tr/27 apparent non-POSIX behaviour
            - symlink is followed but a different error results *)
         (* coverage:linux:posix:irrelevant *)
         (EEXIST, is_mac_os_x_arch env
                  && isJust fopt
                  && (e = ENOTDIR))
         ])
     | RN_file _ -> (fsm_raise EEXIST) (* posix/link.md EEXIST:1 *)
     | RN_dir _ -> (fsm_raise EEXIST)  (* posix/link.md EEXIST:1 *)
     | RN_none (d0_ref,n,rp) -> (
         let cwd = rp.rp_cwd in
         let nl = rp.rp_nl in
         let path =
           CS_Some(Resolve.ty_name_list_to_string nl)
         in
         let b0 =
           Resolve.res_name_is_symlink env.env_ops
             s0
             (Resolve.process_path_no_follow_no_trailing_slash
               env
```

```
                s0
                cwd
                path)
          in
        (if (is_linux_arch env ||
                (is_mac_os_x_arch env && not b0))
             && rn_ends_with_slash dpath
         then
            (* tr/24, mac hfsplus_loop/link/results/
               check_exec_link___link_nonempty_dir2__
               f2.txt___nonexist_2__-int.trace *)
            fsm_raise ENOENT
         else
            fsm_do_nothing)
        |||
        ((* a symlink to a non-existing entry on mac
            is treated as though the path resolved
            to the symlink itself
            hfsplus_loop/link/results/
            check_exec_link___link_empty_dir1_____
            nonempty_dir1__d2__sl_dotdot_no_such_
            target-int.trace *)
            if is_mac_os_x_arch env &&
               b0 &&
               not (rn_ends_with_slash dpath)
            then fsm_raise EEXIST
            else fsm_do_nothing))
  end
) ||| (
  match spath with
    | RN_error(e,_) -> (fsm_raise e)
    | RN_none _ -> (fsm_raise ENOENT) (* posix/link.md ENOENT:2 *)
    | RN_file _ -> (
      let cond =
        is_linux_arch env &&
        Resolve.res_name_is_symlink env.env_ops s0 spath &&
          rn_ends_with_slash spath
        in
        fsm_cond_raises
          [(ENOTDIR, (is_RN_none dpath &&
                      rn_ends_with_slash dpath &&
                      (not (is_mac_os_x_arch env)))
           ); (* posix/link.md ENOTDIR:4
                hfsplus_loop/link/results/
                check_exec_link___link_nonempty_dir1__
                d2__f3.txt___nonempty_dir1__d2__sl_
                dotdot_no_such_target__-int.trace *)
           (ENOTDIR, cond); (* FIXME tr/?? *)
           (EPERM, cond);   (* FIXME tr/?? *)
           (ENOENT, cond)]) (* FIXME tr/?? *)
     | RN_dir(d0_ref, _) -> (
          (* FIXME we should check link
             /a/exist_dir /b/f1.txt/ *)
#ifdef aspect_perms
         if (env.env_prms.cp_has_dir_link_create_privilege s0 d0_ref
             && arch_allows_dir_links env) then
#else
         if (arch_allows_dir_links env) then
#endif
           fsm_special
             FIXME "link: directory links unsupported in this spec"
         else
           fsm_raise EPERM)  (* posix/link.md EPERM:2 *)
   end)
#ifdef aspect_perms
 ||| fsop_link_checks_perms env spath dpath
#endif
)
```

Notice that the `#ifdef` blocks are the specification extension points referred as traits in §2.3.2. The aim of the error checking phase is to return the set of all reachable error states. Indeed, the state combinator `|||` merges the erroneous states in the monad:

```
let fsm_parallel_composition_drop m1 m2 = Fsmonad (fun s ->
 let (st_s1_n, st_s1) =
   finset_partition is_Normal_state (run_fsmonad m1 s) in
 let (st_s2_n, st_s2) =
   finset_partition is_Normal_state (run_fsmonad m2 s) in

 let res = finset_union st_s1 st_s2 in
 (* remove obvious duplicates *)
 let res' = finset_cleanup monad_state_shallow_eq res in
 if (finset_is_empty st_s1_n || finset_is_empty st_s2_n) then
   res'
 else
   finset_insert (Normal_state (s, dummy_return_value))  res'
)

let (|||) = fsm_parallel_composition_drop
```

If there are no error states, the main core behaviour of `link` will produce a set of normal states:

```
let fsop_link_core env spath dpath = (
 fsm_get_state >>= fun s0 ->
 (match (spath, dpath) with
   | (RN_file(_,_,i0_ref,_), RN_none(d0_ref, n, _)) -> (
         let s0 = env.env_ops.fops_link_file s0 i0_ref d0_ref n in
         fsm_put_state s0)
   | _ -> fsm_special Impossible "error raised before"
  end))
```

The `env.env_ops.fops_link_file` function provides the algorithmic behaviour of the command. The specification provides an in memory model of `link` since its present use case is to be an oracle for existing file systems:

```
let dhops_link s0 ent d0_ref name = (
 let s1 = dh_update_dir_entries s0 d0_ref name (Just ent) in
 (* increase st_nlink for the fs object *)
 match ent with
 | File_ref_entry(f_ref) ->
```

```
      let file = fromJust(dh_lookup_file s1 f_ref) in
      dh_update_files
        s1
        f_ref
        (Just(<|file with dhf_nlink=(file.dhf_nlink+1)|>))
   | Dir_ref_entry(d_ref) ->
      let dir = fromJust(dh_lookup_dir s1 d_ref) in
      (* increase the link count of the parent *)
      let parent = fromJust (dh_lookup_dir s1 d0_ref) in
      let s1 =
        dh_update_dirs
          s0
          d0_ref
          (Just(<|parent with dhd_nlink=(parent.dhd_nlink+1)|>))
      in
      (* increase the link count of the dir *)
      dh_update_dirs
        s1
        d_ref
        (Just(<|dir with dhd_nlink=(dir.dhd_nlink+1)|>)) end
)
```

This `link` implementation uses heaps to store files and directories. The `link` function adds the given file to the contents of the directory that should include the new link.

Appendix B: SibylFS extended with timestamps main excerpts

This appendix provides excerpts of SibylFS extended with the timestamps feature [4], mainly focusing on the timestamp update events and examples of traces used to validate the time features of the specification.

Periodic update events

The stat update

The stat update requires that all the timestamps marked for update obtain a time value:

```
val fsop_stat_core:
fs_ops ->
res_name ->
fsmonad

let fsop_stat_core ops rn = (
 fsm_get_state >>= (fun s0 -> (
  (match rn with
  | RN_file(d0_ref,n,i0_ref,rp) ->
   #ifdef aspect_time
   let s0 = (* see STAT_TS in fs_spec/posix/timestamps *)
    update_timestamps_fs_state ops s0
     (File_ref_entry i0_ref)
   in
   #endif
   (fsm_put_state_return s0
    (RV_logical_stats (ops.fops_stat_file s0 i0_ref)))
  | RN_dir(d0_ref,rp) ->
   #ifdef aspect_time
   let s0 = (* see STAT_TS in fs_spec/posix/timestamps *)
    update_timestamps_fs_state ops s0 (Dir_ref_entry d0_ref)
   in
   #endif
   (fsm_put_state_return s0
    (RV_logical_stats (ops.fops_stat_dir s0 d0_ref)))
  | _ -> fsm_special Impossible "error raised before" end))))
```

where `update_timestamps_fs_state` assigns logical clock value to the marked timestamps.

The `close` update

The `close` update happens when the operating system attempts to close the last file descriptor that is still open for a file system object. The definition of the `close` event concerns both the file system, which may need to assign the current clock value to marked file, and the operating system state, which contains the list of open objects. In SibylFS this scope is available during the translation from operating system label to file system label:

```
...
let run_os_cmd (cmd : ty_os_command) = (
 match cmd with
 (* real os-commands *)
 | OS_CLOSE fd                   ->
#ifdef aspect_time
    (* we need to check that the fd is valid *)
 let fid_of_fd = lookup_fid_of_fd s0 pid fd in
   match fid_of_fd with
   | Nothing ->
      (* if the fd does not exist
       we do not need to update timestamps *)
      (os_close env pid fd s0)
   | Just (_,fid_state) ->
      (* we need the entry corresponding
        to the to-be-closed fd *)
      let entry = fid_state.fids_entry in
      (* see FILE_CLOSED_TS in fs_spec/posix/timestamps*)
      finset_image
        (update_timestamps_for_entry_if_no_open_fds
           env entry)
        (os_close env pid fd s0)
   end
#else
   (os_close env pid fd s0)
#endif
...
```

The function `update_timestamps_for_entry_if_no_open_fds` takes the operating system states generated by the `close` command, and for each of these it updates the timestamps of the given entry, if no other process has an open file descriptor for it.

Examples of manual test traces

This section provides an example trace for each test category. As a reminder, a positive trace tests that the specifications accepts valid timestamps updates; a negative trace tests that the specification rejects invalid updates; a concurrent trace tests that the specification can handle concurrent updates.

All the examples use the command `chmod`: this command changes the object permissions and alters the *change* timestamp.

Positive trace

```
@type trace
#chmod test on file
```

```
4: open "/f1.txt" [O_CREAT;O_RDWR] 0o644
 Tau
 RV_num(3)

6: close (FD 3)
 Tau
 RV_none

 # end of state setup

10: lstat "/f1.txt"
 Tau
 RV_stat {
           st_dev=36;
           st_ino=34396;
           st_kind=S_IFREG;
           st_perm=0o644;
           st_nlink=1;
           st_uid=0;
           st_gid=0;
           st_rdev=0;
           st_size=0;
           st_atim={tv_sec=1428336036;tv_nsec=0;};
           st_mtim={tv_sec=1428336036;tv_nsec=0;};
           st_ctim={tv_sec=1428336036;tv_nsec=0;}; }

12: chmod "/f1.txt" 0o1750
 Tau
 RV_none

# chmod should update the st_ctime of the file
# (see CHMOD_TS in fs_spec/posix/chmod)

16: lstat "/f1.txt"
 Tau
 RV_stat {
           st_dev=36;
           st_ino=34396;
           st_kind=S_IFREG;
           st_perm=0o1750;
           st_nlink=1;
           st_uid=0;
           st_gid=0;
           st_rdev=0;
           st_size=0;
           st_atim={tv_sec=1428336036;tv_nsec=0;};
           st_mtim={tv_sec=1428336036;tv_nsec=0;};
           st_ctim={tv_sec=1428336036;tv_nsec=1;}; }

18: dump-result "/"
    "/"|D|st_dev=36;
          |st_ino=36065;
          |st_kind="S_IFDIR";
          |st_perm=511;
          |st_nlink=2;
          |st_uid=0;
          |st_gid=0;
          |st_rdev=0;
          |st_size=60;
          |st_atim={tv_sec=1428336036;tv_nsec=0;};
          |st_mtim={tv_sec=1428336036;tv_nsec=0;};
          |st_ctim={tv_sec=1428336036;tv_nsec=0;};
    "/f1.txt"|F
          |"da39a3ee5e6b4b0d3255bfef95601890afd80709"
          |st_dev=36;
          |st_ino=34396;
          |st_kind="S_IFREG";
          |st_perm=1000;
          |st_nlink=1;
          |st_uid=0;
          |st_gid=0;
```

```
           |st_rdev=0;
           |st_size=0;
           |st_atim={tv_sec=1428336036;tv_nsec=0;};
           |st_mtim={tv_sec=1428336036;tv_nsec=0;};
           |st_ctim={tv_sec=1428336036;tv_nsec=1;};
 end dump-result
```

This trace firstly calls an `lstat` to see the timestamps that the new file has (i.e., the `stat` implies an addition of an entry logical-physical time in the LP-map), then calls `chmod` and finally calls `dump` (i.e., a command that lists time metadata for all the objects present in the file system) to show that the specification accepts the correct timestamp being updated.

The relative `chmod` periodic trace is:

```
@type trace
#chmod test on file

4: open "/f1.txt" [O_CREAT;O_RDWR] 0o644
   Tau
   RV_num(3)

6: close (FD 3)
   Tau
   RV_none

 # end of state setup

10: lstat "/f1.txt"
    Tau
    RV_stat {
             st_dev=36;
             st_ino=36516;
             st_kind=S_IFREG;
             st_perm=0o644;
             st_nlink=1;
             st_uid=0;
             st_gid=0;
```

```
         st_rdev=0;
         st_size=0;
         st_atim={tv_sec=1428336896;tv_nsec=0;};
         st_mtim={tv_sec=1428336896;tv_nsec=0;};
         st_ctim={tv_sec=1428336896;tv_nsec=0;}; }

12: chmod "/f1.txt" 0o1750
    Tau
    RV_none

    dump-internal

    # chmod should update the st_ctime of the file
    # (see CHMOD_TS in fs_spec/posix/chmod)

16: lstat "/f1.txt"
    Tau
    RV_stat {
         st_dev=36;
         st_ino=36516;
         st_kind=S_IFREG;
         st_perm=0o1750;
         st_nlink=1;
         st_uid=0;
         st_gid=0;
         st_rdev=0;
         st_size=0;
         st_atim={tv_sec=1428336896;tv_nsec=0;};
         st_mtim={tv_sec=1428336896;tv_nsec=0;};
         st_ctim={tv_sec=1428336896;tv_nsec=1;}; }

# the dump sees that the ctime of f1.txt and
# the root dir have been updated at the same time
# (periodic update)

18: dump-result "/"
```

```
    "/"|D|st_dev=36;
          |st_ino=36513;
          |st_kind="S_IFDIR";
          |st_perm=511;
          |st_nlink=2;
          |st_uid=0;
          |st_gid=0;
          |st_rdev=0;
          |st_size=60;
          |st_atim={tv_sec=1428336896;tv_nsec=1;};
          |st_mtim={tv_sec=1428336896;tv_nsec=1;};
          |st_ctim={tv_sec=1428336896;tv_nsec=1;};
    "/f1.txt"|F
          |"da39a3ee5e6b4b0d3255bfef95601890afd80709"
          |st_dev=36;
          |st_ino=36516;
          |st_kind="S_IFREG";
          |st_perm=1000;
          |st_nlink=1;
          |st_uid=0;
          |st_gid=0;
          |st_rdev=0;
          |st_size=0;
          |st_atim={tv_sec=1428336896;tv_nsec=0;};
          |st_mtim={tv_sec=1428336896;tv_nsec=0;};
          |st_ctim={tv_sec=1428336896;tv_nsec=1;};
 end dump-result
```

This trace is similar to the previous, although it tests that update delays are supported by the specification. Notice that the periodicity is given by the timestamps of the root directory, which are updated at the same time of the change timestamps of the file.

Negative trace

```
@type trace
#chmod test on file

4: open "/f1.txt" [O_CREAT;O_RDWR] 0o644
   Tau
   RV_num(3)

6: close (FD 3)
   Tau
   RV_none

   # end of state setup

10: lstat "/f1.txt"
    Tau
    RV_stat {
             st_dev=36;
             st_ino=39492;
             st_kind=S_IFREG;
             st_perm=0o644;
             st_nlink=1;
             st_uid=0;
             st_gid=0;
             st_rdev=0;
             st_size=0;
             st_atim={tv_sec=1428343545;tv_nsec=1;};
             st_mtim={tv_sec=1428343545;tv_nsec=1;};
             st_ctim={tv_sec=1428343545;tv_nsec=1;}; }

12: chmod "/f1.txt" 0o1750
    Tau
    RV_none

    # chmod should update the st_ctime of the file
    #  (see CHMOD_TS in fs_spec/posix/chmod)

# we try to go back in time on the timestamp to update
18: dump-result "/"
       "/"|D|st_dev=36;
             |st_ino=39489;
             |st_kind="S_IFDIR";
             |st_perm=511;
             |st_nlink=2;
             |st_uid=0;
             |st_gid=0;
             |st_rdev=0;
             |st_size=60;
             |st_atim={tv_sec=1428343545;tv_nsec=1;};
             |st_mtim={tv_sec=1428343545;tv_nsec=1;};
             |st_ctim={tv_sec=1428343545;tv_nsec=1;};
       "/f1.txt"|F
             |"da39a3ee5e6b4b0d3255bfef95601890afd80709"
             |st_dev=36;
             |st_ino=39492;
             |st_kind="S_IFREG";
             |st_perm=1000;
             |st_nlink=1;
             |st_uid=0;
             |st_gid=0;
             |st_rdev=0;
             |st_size=0;
             |st_atim={tv_sec=1428343545;tv_nsec=1;};
             |st_mtim={tv_sec=1428343545;tv_nsec=1;};
             |st_ctim={tv_sec=1428343545;tv_nsec=0;};
    end dump-result


# we try to alter the wrong timestamp

18: dump-result "/"
```

```
    "/"|D|st_dev=36;
          |st_ino=39489;
          |st_kind="S_IFDIR";
          |st_perm=511;
          |st_nlink=2;
          |st_uid=0;
          |st_gid=0;
          |st_rdev=0;
          |st_size=60;
          |st_atim={tv_sec=1428343545;tv_nsec=1;};
          |st_mtim={tv_sec=1428343545;tv_nsec=1;};
          |st_ctim={tv_sec=1428343545;tv_nsec=1;};
    "/f1.txt"|F|"da39a3ee5e6b4b0d3255bfef95601890afd80709"
          |st_dev=36;
          |st_ino=39492;
          |st_kind="S_IFREG";
          |st_perm=1000;
          |st_nlink=1;
          |st_uid=0;
          |st_gid=0;
          |st_rdev=0;
          |st_size=0;
          |st_atim={tv_sec=1428343545;tv_nsec=1;};
          |st_mtim={tv_sec=1428343545;tv_nsec=2;};
          |st_ctim={tv_sec=1428343545;tv_nsec=1;};
  end dump-result


18: dump-result "/"
    "/"|D|st_dev=36;
          |st_ino=39489;
          |st_kind="S_IFDIR";
          |st_perm=511;
          |st_nlink=2;
          |st_uid=0;
          |st_gid=0;
```

```
            |st_rdev=0;
            |st_size=60;
            |st_atim={tv_sec=1428343545;tv_nsec=1;};
            |st_mtim={tv_sec=1428343545;tv_nsec=1;};
            |st_ctim={tv_sec=1428343545;tv_nsec=1;};
      "/f1.txt"|F|"da39a3ee5e6b4b0d3255bfef95601890afd80709"
            |st_dev=36;
            |st_ino=39492;
            |st_kind="S_IFREG";
            |st_perm=1000;
            |st_nlink=1;
            |st_uid=0;
            |st_gid=0;
            |st_rdev=0;
            |st_size=0;
            |st_atim={tv_sec=1428343545;tv_nsec=1;};
            |st_mtim={tv_sec=1428343545;tv_nsec=1;};
            |st_ctim={tv_sec=1428343545;tv_nsec=2;};
   end dump-result
```

This trace attempts invalid updates according to the POSIX specification: firstly we try to assign a smaller time value than the current clock to the change timestamp; then we try to update a timestamp different from the one `chmod` would update. The last `dump` command is successful as we gave the `dump-result` of the correct update to test that the timestamp was update correctly.

Concurrent trace

```
@type trace
# concurrent chmod tests for timestamp
# updates (valid for both immediate and periodic)

Pid 2 -> create User_id 2 Group_id 2
Pid 3 -> create User_id 2 Group_id 2
Pid 4 -> create User_id 2 Group_id 2
Pid 1 -> add_user_to_group User_id 2 Group_id 2
Tau
Pid 1 <- -

Pid 1 -> add_user_to_group User_id 3 Group_id 2
Tau
Pid 1 <- -

Pid 1 -> add_user_to_group User_id 4 Group_id 2
Tau
Pid 1 <- -

Pid 2 -> open /f1.txt [O_CREAT;O_RDWR] 0o666
Tau
Pid 2 <- -

Pid 2 -> close (FD 3)
Tau
Pid 2 <- -

# end state setup

Pid 2 -> stat /f1.txt
Tau
Pid 2 <- RV_stat {st_dev=2049;
                  st_ino=2;
                  st_kind=S_IFREG;
                  st_perm=0o644;
                  st_nlink=1;
                  st_uid=2;
                  st_gid=2;
                  st_rdev=0;
                  st_size=0;
                  st_atim={tv_sec=9;tv_nsec=0;};
                  st_mtim={tv_sec=9;tv_nsec=0;};
                  st_ctim={tv_sec=9;tv_nsec=0;};}
```

```
Pid 3 -> chmod /f1.txt 0o755
Pid 4 -> chmod /f1.txt 0o757
Pid 2 -> chmod /f1.txt 1000

Tau
Tau
Tau

Pid 3 <- -
Pid 4 <- -
Pid 2 <- -

Pid 2 -> stat /f1.txt
Tau
Pid 2 <- RV_stat {st_dev=2049;
                  st_ino=2;
                  st_kind=S_IFREG;
                  st_perm=0o1750;
                  st_nlink=1;
                  st_uid=2;
                  st_gid=2;
                  st_rdev=0;
                  st_size=0;
                  st_atim={tv_sec=9;tv_nsec=0;};
                  st_mtim={tv_sec=9;tv_nsec=0;};
                  st_ctim={tv_sec=10;tv_nsec=0;};}

Pid 3 -> stat /f1.txt
Tau
Pid 3 <- RV_stat {st_dev=2049;
                  st_ino=2;
                  st_kind=S_IFREG;
                  st_perm=0o1750;
                  st_nlink=1;
                  st_uid=2;
                  st_gid=2;
                  st_rdev=0;
                  st_size=0;
                  st_atim={tv_sec=9;tv_nsec=0;};
                  st_mtim={tv_sec=9;tv_nsec=0;};
                  st_ctim={tv_sec=10;tv_nsec=0;};}

Pid 4 -> stat /f1.txt
Tau
Pid 4 <- RV_stat {st_dev=2049;
                  st_ino=2;
                  st_kind=S_IFREG;
                  st_perm=0o1750;
                  st_nlink=1;
                  st_uid=2;
                  st_gid=2;
                  st_rdev=0;
                  st_size=0;
                  st_atim={tv_sec=9;tv_nsec=0;};
                  st_mtim={tv_sec=9;tv_nsec=0;};
                  st_ctim={tv_sec=10;tv_nsec=0;};}
```

This trace creates three operating system processes, and alters simultaneously the permissions of the same object. As for the earlier traces, timestamps are observed before and after using `chmod`. Note that the specification handles concurrency with non-determinism: it maintains multiple valid states representing the possible parallel runs.

Appendix C: B-tree main excerpts

A *Tree* is a recursive type. A *Leaf* contains an equal number of keys and values, while a *Node* contains a list of keys of length n and a list of children of length $n + 1$:

```
type_synonym leaf_lbl_t = "(key * value_t) list"
type_synonym node_lbl_t = "key list"
```

```
datatype Tree = Node "node_lbl_t * Tree list" | Leaf "leaf_lbl_t"
```

Note that we may use the term treestack as equivalent to the framestack discussed in §4.5.

5.1 B-tree wellformedness

A *Tree* is a wellformed B-tree if it satisfies the following property:

```
definition wellformed_tree :: "ms_t => Tree => bool" where
"wellformed_tree ms t0 == (
  let b1 = wf_size ms t0 in
  let b2 = wf_ks_rs t0 in
  let b3 = balanced t0 in
  let b4 = keys_consistent t0 in
  let b5 = keys_ordered t0 in
  let wf = b1&b2&b3&b4&b5 in
wf
)"
```

The definition of wellformedness changes slightly according to the size of the node under examination (see the *wf_size* predicate). We introduce a type *ms_t* to flag when leaves and nodes have sizes smaller than the allowed minimum (to handle the root cases and support the *delete* operation):

```
datatype min_size_t = Small_root_node_or_leaf
  | Small_node
  | Small_leaf


type_synonym ms_t = "min_size_t option"
```

Each predicate in the wellformedness definition is applied recursively to each subtree contained in the given node (i.e., when the *Tree* is not a leaf).

The properties definitions:

Wf_size The size of a leaf is the number of entries it contains, while the size of a node is its number of children. All the nodes in a B-tree have an upper bound on their size, and non-root nodes have also a lower bound.

```
consts min_leaf_size :: nat
consts max_leaf_size :: nat
consts min_node_keys :: nat
consts max_node_keys :: nat
```

Although one can obtain a valid B-tree with whatever value of the constants, we designed the B-tree commands assuming the following properties on the constants:

```
definition wellformed_constants :: "bool" where
"wellformed_constants == (
let wf_node_constants =
(1 <= min_node_keys
&
(max_node_keys = 2 * min_node_keys
| max_node_keys = Suc (2 * min_node_keys))
)
in
let (wf_leaf_constants) =
(1 <= min_leaf_size
&
(max_leaf_size = 2 * min_leaf_size
| max_leaf_size = Suc (2 * min_leaf_size))
)
in
wf_node_constants & wf_leaf_constants
)"
```

So minimal boundary should be greater than 0, and the maximum can be equal to the doubled minimal boundary or the successor number to allow odd keys sizes.

This property changes slightly, if it is dealing with a *Leaf* root:

```
definition get_min_size :: "(min_size_t * Tree) => nat" where
"
get_min_size mt == (
case mt of
(Small_root_node_or_leaf,Node _) => 1
| (Small_root_node_or_leaf,Leaf _) => 0
| (Small_node, Node _) => min_node_keys-1
| (Small_leaf,Leaf _) => min_leaf_size-1
| (_,_) => undefined
)
"
definition wf_size_1 :: "Tree => bool" where
"wf_size_1 t1 == (
case t1 of
Leaf xs => (
let n = length xs in
(n >= min_leaf_size) & ( n <= max_leaf_size))
| Node(l,cs) => (
let n = length l in
(1 <= n) & (n >= min_node_keys) & (n <= max_node_keys)

)
)
```

```
"

definition wf_size :: "ms_t => Tree => bool" where
"wf_size ms t0 == (
case ms of
None => (forall_subtrees wf_size_1 t0)
| Some m => (
let min = get_min_size (m,t0) in
case t0 of
Leaf xs =>
let n = length xs in
(min <= n) & (n <= max_leaf_size)
| Node(l,cs) => (
let n = length l in
(min <= n) & (n <= max_node_keys)
& (List.list_all (forall_subtrees wf_size_1) cs))
))"
```

Indeed, when the type *ms_t* is `Some Small_root_node_or_leaf` and the *Tree* is a *Leaf*, the *Leaf* can be empty. When it is a *Node*, it must contain at least a child, and the children must satisfy the *wf_size_1* predicate. If the *Node* is not the root of the whole *Tree*, it must satisfy the *wf_size_1* predicate as its children. Note that at least a child must exist, as an empty *Node* is considered meaningless.

Wf_ks_rs In a well formed tree each *Node* has *n* keys and *n+1* children:

```
definition wf_ks_rs_1 :: "Tree => bool" where
"wf_ks_rs_1 t0 == (
case t0 of Leaf _ => True
| Node(l,cs) => ((1+ length l) = (length cs))
)"

definition wf_ks_rs :: "Tree => bool" where
"wf_ks_rs t0 == forall_subtrees wf_ks_rs_1 t0"
```

Balanced In a wellformed B-tree every path from the root to any *Leaf* should have the same length. The height of a *Tree* is the number of steps to reach a

Leaf from the given *Tree*:

```
function height :: "Tree => nat" where
"height t0 = (
case t0 of
Leaf _ => (1::nat)
| Node(_,cs) => (1 + Max(set(List.map height cs)))
)"
```

A *Tree* is balanced if all the children have the same height:

```
definition balanced_1 :: "Tree => bool" where
"balanced_1 t0 == (
case t0 of Leaf(l) => True
| Node(l,cs) => (
(cs = []) | (
List.list_all (% c. height c = height (cs!0)) cs))
)"

definition balanced :: "Tree => bool" where
"balanced t == forall_subtrees balanced_1 t"
```

Keys_consistent In a wellformed B-tree the keys of the children are always bound by the keys of the current node. So for example:

```
Consider the following root node:
  Node([key1,key2],[child1,child2,child3]).

Represented as the mixed list:

  | child1 | key1 | child2 | key2 | child3 |

There are three consistency constraints over the node keys:

  1. child1 keys are smaller than key1

  2. child2 keys are bigger or equal than key1
     and smaller than key2
  3. child3 keys are bigger or equal than key2
```

The following definition expresses this property:

```
definition keys_1 :: "Tree => key list" where "keys_1 t0 ==
(case t0 of Leaf xs => (List.map fst xs)
| Node (l,cs) => (l)
)"

definition keys :: "Tree => key list" where
"keys t0 ==
  (t0 |> tree_to_subtrees |>
  (List.map keys_1) |> List.concat)"

definition key_indexes :: "Tree => nat list" where
"key_indexes t == (
  case t of
  Leaf xs => (upt 0 (length xs))
  | Node (l,_) => (upt 0 (length l)))"

definition keys_consistent_1 :: "Tree => bool" where
"keys_consistent_1 t0 == (
case t0 of Leaf(l) => True
| Node(label,children) => (
let b1 = (! i : set(key_indexes t0).
  let k0 = label!i in
  let kls = keys(children!i) in
  check_keys None kls (Some k0))
in
let b2 = (! i : set(key_indexes t0).
  let k0 = label!i in
  let krs = keys(children!(i+1)) in
  check_keys (Some k0) krs None)
in
b1 & b2
))
"
```

```
definition keys_consistent :: "Tree => bool" where
"keys_consistent t == forall_subtrees keys_consistent_1 t"
```

The snippet `! i : set` is the Isabelle/HOL textual representation for: $\forall i \in set$, which means that all elements of the `set` must satisfy the given predicate.

The definition *keys* is recursive and collects all the keys of a *Tree*.

The following definition checks that a list of keys *ks* is in the interval defined by *kl* and *kr*:

```
definition check_keys
 :: "key option => key list => key option => bool"
where
"check_keys kl ks kr == (
let b1 = (
case kl of None => True
| Some kl => (! k : set ks. key_le kl k)
)
in
let b2 = (
case kr of None => True
| Some kr => (! k : set ks. key_lt k kr)
)
in
b1 & b2
)"
```

where *key_lt* stands for "key less than other key" and *key_le* for "key less than or equal other key", they are parametric definition standing for ordering operators over keys. These operators are made generic to make them reusable in other contexts:

```
consts key_lt :: "key => key => bool"

definition key_eq :: "key => key => bool" where
  "key_eq k1 k2 == (~ (key_lt k1 k2)) & (~ (key_lt k2 k1))"

definition key_le :: "key => key => bool" where
```

```
  "key_le k1 k2 == (key_eq k1 k2) | (key_lt k1 k2)"
```

The generality can be achieved by leaving the `key_lt` operator dependent on the type of the key, and building the other operators on it.

Keys_ordered The lists of keys contained in any kind of node must be sorted in ascending order; this enforces that all the key in a node are different. This is defined in the following:

```
definition keys_ordered_1 :: "Tree => bool" where
"keys_ordered_1 t0 == (
let is = set (butlast (key_indexes t0)) in
case t0 of
Leaf xs =>
  let ks = (xs |> List.map fst) in
  ! i : is. key_lt (ks!i) (ks!(i+1))
| Node (ks,_) =>
  ! i : is . key_lt (ks!i) (ks!(i+1))
)
"

definition keys_ordered :: "Tree => bool" where
"keys_ordered t == forall_subtrees keys_ordered_1 t"
```

The ascending order is enforced by *key_lt*, that is the *less than* operator on the *Key* type.

5.2 B-tree find

The *find* command returns the value corresponding to the given key *k*, or nothing, if the tree does not contain such an association. The *find* step function returns the leaf node reached by looking up *k*. The focus for *find* contains the searched key and a subtree:

```
type_synonym f_focus_t = "key * Tree"
```

The *find* step takes a framestack (here called `f_tree_stack`) and may or may not return a new framestack. Indeed, the step searches for the key *k* in the focus' tree: if it is a node the search is not finished and a new framestack is returned; if it is a leaf nothing is returned, as the search is already completed and the leaf is in the focus of the given framestack.

```
definition step_fts
:: "f_tree_stack => f_tree_stack option"
where
"step_fts fts == (
(* we extract the key we are looking for,
   the tree in which to search,
   and the context we got so far *)
let (k,t,ctx) = dest_f_tree_stack fts in
(* we get the lower and upper bounds of the parent node *)
let (lb,rb) =
case ctx of Nil => (None,None)
| (lb,_,rb)#_ => (lb, rb)
in
(case t of
(* if the tree in which to search is a Leaf, we can stop *)
Leaf _ => None
(* if the tree in which to search is a Node,
   we need to descend further *)
| Node(ks,rs) =>
(* we find the index of the first key that is bigger than our key *)
let i = search_key_to_index ks k in
(* we find the lower and upper bound for the current tree *)
let (l,u) = get_lower_upper_keys_for_node_t ks lb i rb in
(* we add the tree to the context *)
let ctx2 = (l,((ks,rs),i),u)#ctx in
(* we create a new stack with the ith child as focus
   and the new context *)
Some(Tree_stack(Focus(k,(rs!i)),ctx2))
))"
```

Note that when a leaf node is reached, *find* does not return a key value pair to make the algorithm reusable for *insert*: once in a leaf, another function looks up for the value.

The complete *find* would loop the step until a leaf is reached, and return the value corresponding to the key, if there is one.

5.2.1 Find correctness

Two properties are needed to prove *find* correctness:

1. the preservation of B-tree wellformedness:

   ```
   definition invariant_wf_fts :: "bool" where
   "invariant_wf_fts == (
   ! fts.
   let wellformed_fts' =
   (
   let fts' = step_fts fts in
   case fts' of None => True
   | Some fts' => wellformed_fts fts'
   )
   in
    total_order_key_lte -->
    wellformed_fts fts --> wellformed_fts'
   )"
   ```

   each step must leave the tree stack well formed, which implies that the final tree is well formed as well.

2. the Map *find* interface equivalence:

   ```
   definition invariant_map_equivalence_find :: "bool" where
   "invariant_map_equivalence_find == (
   ! fts.
   let map_equivalence_find =
   (
    case step_fts fts of
    Some fts' =>
    let k  = fst (dest_f_tree_stack fts) in
    let m  = fts_to_map fts in
    let m' = fts_to_map fts' in
    ((m k) = (m' k))
    | _ => True
   ```

```
    )
    in
     total_order_key_lte -->
     wellformed_fts fts --> map_equivalence_find
    )"
```

the Map obtained by the descended framestack is equivalent to the one obtained from the initial framestack.

5.3 B-tree insert

The *insert* command returns a new tree containing the given *(key,value)* pair. This command uses three types of steps: locating the relevant leaf where the new (k,v) will be inserted; adding the entry; restructure the tree to satisfy the B-tree properties (if needed).

The *find* step is reused to descend the tree. The *insert* step is a bit more complicated when it deals with corner cases: for instance inserting in a full leaf requires tree restructuring. In these cases the *insert* algorithm splits the node. So the *insert* focus can both contain a single tree or two trees separated by a separating key:

```
datatype its_focus_t =
Inserting_one "Tree"
| Inserting_two "Tree * key * Tree"
```

The high level algorithm is defined as:

```
definition its_step_tree_stack
:: "its_state => (its_state) option"
where
"its_step_tree_stack ist == (
case ist of
(* if the tree stack allows find transitions we descend the tree *)
Its_down (fts,v0) =>
let fts1 = step_fts fts in
(case fts1 of
(* if running a step returns nothing we have reached a Leaf *)
None =>
(* when a Leaf is reached we can insert the key-value pair
   and start ascending *)
Option.bind (step_bottom fts v0) (% x . Some (Its_up x))
(* if running a step returns some tree stack we have to keep descending*)
| Some x => Some(Its_down(x,v0)))
(* if the tree stack allows only insert transitions we ascend the tree*)
| Its_up ts => Option.bind (step_up ts) (% x . Some (Its_up x)))
"
```

The *find* steps descend the tree, labeling the state as *Its_down.* At the bottom of the tree, *find* steps cannot be applied anymore, and the insertion of the given pair takes place, and start to ascend the tree to restore the validity properties (labeling the state as *Its_up*).

5.3.1 Insert correctness

Showing the two properties for the *insert* interface is more demanding than showing the ones for *find*:

1. the preservation of B-tree wellformedness:

   find steps do not invalidate this property, but both bottom and ascending steps may do it. So, the property that must be invariant for *step_up* follows:

   ```
   definition invariant_wf_ts :: "bool" where
   "invariant_wf_ts == (
   (*assume ts' is a valid tree stack that results
     from an ascending transition on ts*)
   in
     total_order_key_lte -->
     wellformed_constants -->
     wellformed_ts ts --> wellformed_ts'
   )
   "
   ```

   The property means that given a total order on the keys, valid constants (e.g., minimum size of a node is half than the maximum size), and a wellformed framestack, an ascending transition on the original framestack must produce a wellformed one.

This proof is particularly challenging, as we need to prove both the well-formedness of the focus (which can be a single tree or a pair of trees), and the wellformedness of the head of the framestack updated with the tree(s) in the focus.

Then a similar proof must be developed for *step_bottom* and the other transitions:

```
definition wf_its_state :: "its_state => bool" where
"wf_its_state its == (
case its of
Its_down (fts,_) => wellformed_fts fts
| Its_up its => wellformed_ts its
)"

definition invariant_wf_its_state :: "bool" where
"invariant_wf_its_state == (
(*assume ts' is a valid tree stack that results
  from a descending transition on ts*)
in
  total_order_key_lte -->
  wellformed_constants -->
  wf_its_state its --> wf_its_state_its'
)
"
```

2. the Map *insert* interface equivalence:

```
definition invariant_insert_map_its :: "bool" where
"invariant_insert_map_its = (
!its.
(case its of
Its_down(fts,_) =>
(case step_fts fts of
 None => invariant_step_bottom_map_its
 | _ => invariant_map_equivalence_find)
| _ => invariant_step_up_map_its))
"
```

the correctness of the *insert* step depends on the correctness of each of its phases. The descending invariant was covered in §5.2, while the bottom and ascending are shown below:

```
definition invariant_step_bottom_map_its :: "bool" where
"invariant_step_bottom_map_its = (
!fts k v.
let its = step_bottom fts v in
let m_eq_m' = (
 let m = fts_to_map fts in
 (case its of None => True
 | Some its =>
 let m' = its_to_map its in
 m' = m(k \<mapsto> v)))
in
total_order_key_lte -->
wellformed_fts fts -->
m_eq_m')"
```

m, the Map corresponding to the initial framestack, and m' , the Map corresponding to the framestack after an insertion, must differ only on a single binding *(k,v)*.

```
definition invariant_step_up_map_its :: "bool" where
"invariant_step_up_map_its = (
!its.
let its' = step_up its in
let m_eq_m' = (
 let m = its_to_map its in
 (case its' of None => True
 | Some its' =>
 let m' = its_to_map its' in
 m = m'))
in
total_order_key_lte -->
wellformed_ts its -->
m_eq_m')
"
```

m, the Map corresponding to the initial framestack, and m', the Map corresponding to the framestack after an ascending transition, must be equivalent.

Appendix D: trace that shows ext4 periodicity

Using the *dbench* [81] tool on a machine running Linux 3.14-2 with an Intel Core i7-4600M 2.90GHz CPU with performance governor, ATA3 OPAL2.0 SSD, and 16GB RAM an example of the periodic update behaviour manifested in running a trace on the `ext4` file system.

The following trace does the following:

1. creates a directory `dir_1`;
2. creates two broken symlinks;
3. creates a subdirectory `dir_1/dir_2`;
4. creates another broken symlink.

As the `ext4` file system applies a periodic strategy to timestamps update, it can happen that the timestamps attributed to the created files do not follow the order $1 < 2 < 3 < 4$. Indeed, in the following trace $3 < 1$ holds: the subdirectory has earlier timestamps than the parent directory.

```
@type trace

# test mkdir

5: mkdir "/dir_1" 0o777
Tau
RV_none

7: symlink "justwaiting" "/s"
Tau
```

```
RV_none

9: symlink "justwaiting" "/s1"
Tau
RV_none

11: mkdir "/dir_1/dir_2" 0o777
Tau
RV_none

13: symlink "blabla" "/symlink_1"
Tau
RV_none

15: lstat "/dir_1"
Tau
RV_stat { st_dev=2053;
st_ino=3195944;
st_kind=S_IFDIR;
st_perm=0o755;
st_nlink=3;
st_uid=0;
st_gid=0;
st_rdev=0;
st_size=4096;
st_atim=
{tv_sec=1421231636;
tv_nsec=225414037;
};
st_mtim={tv_sec=1421231636;
tv_nsec=229413986;
};
st_ctim={tv_sec=1421231636;
tv_nsec=229413986;
};
```

```
}

17: lstat "/s"
Tau
RV_stat { st_dev=2053;
st_ino=3195952;
st_kind=S_IFLNK;
st_perm=0o777;
st_nlink=1;
st_uid=0;
st_gid=0;
st_rdev=0;
st_size=11;
st_atim=
{tv_sec=1421231636;
tv_nsec=225414037;
};
st_mtim={tv_sec=1421231636;
tv_nsec=225414037;
};
st_ctim={tv_sec=1421231636;
tv_nsec=225414037;
};

}

19: lstat "/s1"
Tau
RV_stat { st_dev=2053;
st_ino=3195977;
st_kind=S_IFLNK;
st_perm=0o777;
st_nlink=1;
st_uid=0;
st_gid=0;
st_rdev=0;
```

```
st_size=11;
st_atim=
{tv_sec=1421231636;
tv_nsec=225414037;
};
st_mtim={tv_sec=1421231636;
tv_nsec=225414037;
};
st_ctim={tv_sec=1421231636;
tv_nsec=225414037;
};

}

21: lstat "/symlink_1"
Tau
RV_stat { st_dev=2053;
st_ino=3195982;
st_kind=S_IFLNK;
st_perm=0o777;
st_nlink=1;
st_uid=0;
st_gid=0;
st_rdev=0;
st_size=6;
st_atim=
{tv_sec=1421231636;
tv_nsec=229413986;
};
st_mtim={tv_sec=1421231636;
tv_nsec=229413986;
};
st_ctim={tv_sec=1421231636;
tv_nsec=229413986;
};
```

```
}

23: lstat "/dir_1/dir_2"
Tau
RV_stat { st_dev=2053;
st_ino=3195979;
st_kind=S_IFDIR;
st_perm=0o755;
st_nlink=2;
st_uid=0;
st_gid=0;
st_rdev=0;
st_size=4096;
st_atim=
{tv_sec=1421231636;
tv_nsec=225414037;
};
st_mtim={tv_sec=1421231636;
tv_nsec=225414037;
};
st_ctim={tv_sec=1421231636;
tv_nsec=225414037;
};

}
```

This happens because `ext4` is periodic and divides the update into two steps:

1. setting flag bits on the in-memory representation of the inode indicating that a given timestamp must be updated before writing to disk or begin observed (e.g., see `ext4`'s `chown` command *ctime* update [83]);
2. assigning the current clock value to timestamps, if the flag bits require it, before observing or writing then to disk (see `ext4`'s routine to update timestamps [82]).

Bibliography

[1] Alagappan, R., Chidambaram, V., Pillai, T.S., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2015. *Beyond storage APIs: Provable semantics for storage stacks.*

[2] Alglave, J., Donaldson, A.F., Kroening, D. and Tautschnig, M. 2011. *Making software verification tools really work.*

[3] Alur, R. and Dill, D. 1991. *The theory of timed automata.* Springer.

[4] Andrea Giugliano, T.R. 2015. Posix Formal Timestamps Specification. https://github.com/sibylfs/sibylfs_src/tree/sibylfs_with_POSIX_timestamps. Accessed: 2018.05.10.

[5] Armstrong, R.C., Punnoose, R.J., Wong, M.H. and Mayo, J.R. 2014. Survey of existing tools for formal verification. *Sandia Report SAND2014-20533, Sandia National Laboratories, Albuquerque.* (2014).

[6] Bairavasundaram, L.N., Sundararaman, S., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2009. *Tolerating file-system mistakes with EnvyFS.* USENIX Association.

[7] Bayer, R. and McCreight, E. 1970. *Organization and maintenance of large ordered indices.* ACM.

[8] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M. and Wansbrough, K. 2006. *Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations.*

[9] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M. and Wansbrough, K. 2005. *Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets.*

[10] Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A. and Smith, G. 2014. *A trusted mechanised JavaScript specification.* ACM.

[11] Bridge, J.P. 2010. *Machine learning and automated theorem proving.* University of Cambridge, Computer Laboratory.

[12] Chen, H., Ziegler, D., Chlipala, A., Kaashoek, M.F., Kohler, E. and Zeldovich, N. 2015. *Specifying crash safety for storage systems.*

[13] Chidambaram, V., Pillai, T.S., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2013. *Optimistic crash consistency.* ACM.

[14] Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T. and Kohler, E. 2015. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS).* 32, 4 (2015), 10.

[15] Comer, D. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (Jun. 1979), 121–137.

[16] Damchoom, K. and Butler, M. 2009. *Applying event and machine decomposition to a flash-based filestore in Event-B.* Springer.

[17] Damchoom, K., Butler, M. and Abrial, J. 2008. Modelling and proof of a tree-structured file system in Event-B and Rodin. *Formal Methods and Software Engineering.* (2008), 25–44.

[18] Dowson, M. 1997. The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes.* 22, 2 (1997), 84.

[19] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M. and others 2014. *The matter of heartbleed.* ACM.

[20] Ernst, G., Schellhorn, G., Haneberg, D., Pfähler, J. and Reif, W. 2014. *Verification of a virtual filesystem switch.* Springer.

[21] Ernst, G., Schellhorn, G. and Reif, W. 2011. *Verification of B+ Trees: An experiment combining shape analysis and interactive theorem proving.* Springer-Verlag.

[22] Ferreira, M. and Oliveira, J. 2009. An integrated formal methods tool-chain and its application to verifying a file system model. *Formal Methods: Foundations*

and Applications. (2009), 153–169.

[23] Fielding, E. 1980. *The specification of abstract mappings and their implementation as B Trees.* Oxford University Computing Laboratory, Programming Research Group.

[24] Freitas, L., Woodcock, J. and Fu, Z. 2009. POSIX file store in Z/Eves: An experiment in the verified software repository. *Sci. Comput. Program.* 74, 4 (Feb. 2009), 238–257.

[25] Fryer, D., Sun, K., Mahmood, R., Cheng, T., Benjamin, S., Goel, A. and Brown, A.D. 2012. *Recon: Verifying file system consistency at runtime.* USENIX Association.

[26] Gardner, P., Ntzik, G. and Wright, A. 2014. *Local reasoning for the POSIX file system.* Springer.

[27] Graefe, G. and others 2011. Modern B-tree techniques. *Foundations and Trends in Databases.* 3, 4 (2011), 203–402.

[28] Groce, A., Holzmann, G.J. and Joshi, R. 2007. *Randomized differential testing as a prelude to formal verification.*

[29] Harper, R. 2001. Programming in standard ML. (2001).

[30] Harter, T., Dragga, C., Vaughn, M., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H. 2012. A file is not a file: Understanding the I/O behavior of Apple desktop applications. *ACM Trans. Comput. Syst.* 30, 3 (2012), 10.

[31] Hesselink, W. and Lali, M. 2009. Formalizing a hierarchical file system. *Electronic Notes in Theoretical Computer Science.* 259, (2009), 67–85.

[32] Hoare, C.A.R. 1996. *How did software get so reliable without proof?* Springer.

[33] Hoare, C.A.R., Misra, J., Leavens, G.T. and Shankar, N. 2009. The verified software initiative: A manifesto. *ACM Comput. Surv.* 41, 4 (2009).

[34] IEEE, T. and Group, T.O. 2013. Austing group bug tracker. http://austingroupbugs.net. Accessed: 2018.05.10.

[35] IEEE, T. and Group, T.O. 2013. POSIX make definition. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/make.html. Accessed: 2018.05.10.

[36] Jannen, W. et al. 2015. *BetrFS: A right-optimized write-optimized file system.*

USENIX Association.

[37] Jurjens, J. 2005. *Verification of low-level crypto-protocol implementations using automated theorem proving.* IEEE Computer Society.

[38] Kang, E. and Jackson, D. 2008. Formal modeling and analysis of a flash filesystem in Alloy. *Abstract state machines, B and Z.* (2008), 294–308.

[39] Keller, G., Murray, T., Amani, S., O'Connor, L., Chen, Z., Ryzhyk, L., Klein, G. and Heiser, G. 2014. File systems deserve verification too! *ACM SIGOPS Operating Systems Review.* 48, 1 (2014), 58–64.

[40] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S. 2009. *SeL4: Formal verification of an OS kernel.* ACM.

[41] Kumar, R., Myreen, M.O., Norrish, M. and Owens, S. 2014. *CakeML: A verified implementation of ML.* ACM.

[42] Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM.* 21, 7 (1978), 558–565.

[43] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu 2013. *A Study of Linux File System Evolution.*

[44] Leroy, X. 2009. Formal verification of a realistic compiler. *Commun. ACM.* 52, 7 (Jul. 2009), 107–115.

[45] Leveson, N.G. and Turner, C.S. 1993. An investigation of the Therac-25 accidents. *Computer.* 26, 7 (1993), 18–41.

[46] Linux Foundation Linux Standard Base (LSB). http://www.linuxfoundation.org/collaborate/workgroups/lsb. Accessed 2018.05.10.

[47] Linux Test Project Linux Test Project testsuite. http://linux-test-project.github.io/. Accessed 2018.05.10.

[48] Lu, Y., Shu, J. and Wang, W. 2014. *ReconFS: A reconstructable file system on flash storage.* USENIX.

[49] Malecha, J.G., Morrisett, G., Shinnar, A. and Wisnesky, R. 2010. *Toward a verified relational database management system.*

[50] Mashtizadeh, A.J., Bittau, A., Huang, Y.F. and Mazi'eres, D. 2013. *Replica-*

tion, history, and grafting in the Ori file system. ACM.

[51] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A. and Vivier, L. 2007. *The new ext4 filesystem: Current status and future plans.* Citeseer.

[52] McKeeman, W.M. 1998. Differential testing for software. *Digital Technical Journal.* 10, 1 (1998), 100–107.

[53] Morgan, C. and Sufrin, B. 1984. Specification of the Unix filing system. *Software Engineering, IEEE Transactions on.* 10, 2 (1984), 128–142.

[54] Morrisett, G., Tan, G., Tassarotti, J., Tristan, J.-B. and Gan, E. 2012. *RockSalt: Better, faster, stronger SFI for the x86.* ACM.

[55] Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T. and Sewell, P. 2014. *Lem: Reusable engineering of real-world semantics.*

[56] Nicollin, X. and Sifakis, J. 1994. The algebra of timed processes, ATP: theory and application. *Inf. Comput.* 114, 1 (1994), 131–178.

[57] Nipkow, T., Paulson, L.C. and Wenzel, M. 2002. *Isabelle/HOL: A proof assistant for higher-order logic.* Springer Science & Business Media.

[58] Pfähler, J., Ernst, G., Schellhorn, G., Haneberg, D. and Reif, W. 2014. *Crash-safe refinement for a verified flash file system.* University of Augsburg.

[59] Reeves, Glenn E., Neilson, Tracy A. 2005. *The Mars Rover Spirit FLASH anomaly.* Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics; Space Administration, 2005.

[60] Ridge, T., Norrish, M. and Sewell, P. 2008. *A rigorous approach to networking: TCP, from implementation to protocol to service.*

[61] Ridge, T. and others 2017. A B-tree library for OCaml. (2017).

[62] Ridge, T., Sheets, D., Tuerk, T., Giugliano, A., Madhavapeddy, A. and Sewell, P. 2015. *SibylFS: Formal specification and oracle-based testing for POSIX and real-world file systems.*

[63] Rodeh, O. 2008. B-trees, shadowing, and clones. *Trans. Storage.* 3, 4 (Feb. 2008), 2:1–2:27.

[64] Rodeh, O., Bacik, J. and Mason, C. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS).* 9, 3 (2013), 9.

[65] Rubini, A. and Corbet, J. 2001. *Linux device drivers.* " O'Reilly Media, Inc.".

[66] Sagdeev, R. and Zakharov, A. 1989. Brief history of the Phobos mission. *Nature.* 341, 6243 (1989), 581–585.

[67] Sauser, B.J., Reilly, R.R. and Shenhar, A.J. 2009. Why projects fail? How contingency theory can provide new insights–a comparative analysis of nasa's mars climate orbiter loss. *International Journal of Project Management.* 27, 7 (2009), 665–679.

[68] Schellhorn, G., Ernst, G., Pfähler, J., Haneberg, D. and Reif, W. 2014. *Development of a verified flash file system.* Springer Berlin Heidelberg.

[69] Schierl, A., Schellhorn, G., Haneberg, D. and Reif, W. 2009. Abstract specification of the UBIFS file system for flash memory. *FM 2009: Formal Methods.* (2009), 190–206.

[70] Sevcík, J., Vafeiadis, V., Nardelli, F.Z., Jagannathan, S. and Sewell, P. 2013. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM.* 60, 3 (2013), 22.

[71] Sexton, A. and Thielecke, H. 2008. Reasoning about B+ Trees with operational semantics and separation logic. *Electron. Notes Theor. Comput. Sci.* 218, (Oct. 2008), 355–369.

[72] Skeel, R. 1992. Roundoff error and the Patriot missile. *SIAM News.* 25, 4 (1992), 11.

[73] Slabodkin, G. 1998. Software glitches leave navy smart ship dead in the water. *Government Computer News.* 13, (1998), 33727–1.

[74] The IEEE and The Open Group 2008. *The Open Group Base Specifications Issue 7 – IEEE Std 1003.1, 2008 Edition.* IEEE.

[75] The Open Group POSIX Conformance Test Suite. http://www.opengroup.org/testing/downloads.html. Accessed 2018.05.10.

[76] Weyuker, E.J. 1982. On testing non-testable programs. *The Computer Journal.* 25, 4 (1982), 465–470.

[77] Woodcock, J., Larsen, P.G., Bicarregui, J. and Fitzgerald, J.S. 2009. Formal methods: Practice and experience. *ACM Comput. Surv.* 41, 4 (2009).

[78] Yang, J., Twohey, P., Engler, D. and Musuvathi, M. 2006. Using model

checking to find serious file system errors. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 393–423.

[79] Yang, X., Chen, Y., Eide, E. and Regehr, J. 2011. Finding and understanding bugs in C compilers. *SIGPLAN Not.* 46, 6 (Jun. 2011), 283–294.

[80] Zhao, J., Nagarakatte, S., Martin, M.M. and Zdancewic, S. 2012. *Formalizing the LLVM intermediate representation for verified program transformations.* ACM.

[81] Dbench webpage. https://dbench.samba.org/. Accessed 2018.05.10.

[82] Ext4 file system attr.c - routine updating timestamps periodically. https://github.com/tytso/ext4/blob/v3.10-rc7/fs/attr.c#L199. Accessed 2018.05.10.

[83] Ext4 file system open.c - chown flagging ctime time attribute. https://github.com/tytso/ext4/blob/v3.10-rc7/fs/open.c#L534. Accessed 2018.05.10.

[84] Tjrbtree repository. https://github.com/tomjridge/tjr_btree. Accessed 2018.05.10.

[85] https://github.com/ag91/isa_btree. Accessed 2018.05.10.