Caching, Crashing & Concurrency

Verification under Adverse Conditions

Dissertation
Zur Erlangung des Doktorgrades Dr. rer. nat.
Institut für Software & Systems Engineering
Fakultät für Angewandte Informatik
Universität Augsburg

Stefan Bodenmüller

Universität
Augsburg
University

Reviewers:	Prof. Dr. Wolfgang Reif
	Prof. Dr. Bernhard Bauer
	Prof. Dr. Gidon Ernst
Day of Defense:	July 27, 2022

Abstract

The formal development of large-scale software systems is a complex and time-consuming effort. Generally, its main goal is to prove the functional correctness of the resulting system. This goal becomes significantly harder to reach when the verification must be performed under adverse conditions. When aiming for a realistic system, the implementation must be compatible with the “real world”: it must work with existing system interfaces, cope with uncontrollable events such as power cuts, and offer competitive performance by using mechanisms like caching or concurrency.

The Flashix project is an example of such a development, in which a fully verified file system for flash memory has been developed. The project is a long-term team effort and resulted in a sequential, functionally correct and crash-safe implementation after its first project phase. This thesis continues the work by performing modular extensions to the file system with performance-oriented mechanisms that mainly involve caching and concurrency, always considering crash-safety.

As a first contribution, this thesis presents a modular verification methodology for destructive heap algorithms. The approach simplifies the verification by separating reasoning about specifics of heap implementations, like pointer aliasing, from the reasoning about conceptual correctness arguments.

The second contribution of this thesis is a novel correctness criterion for crash-safe, cached, and concurrent file systems. A natural criterion for crash-safety is defined in terms of system histories, matching the behavior of fine-grained caches using complex synchronization mechanisms that reorder operations.

The third contribution comprises methods for verifying functional correctness and crash-safety of caching mechanisms and concurrency in file systems. A reference implementation for crash-safe caches of high-level data structures is given, and a strategy for proving crash-safety is demonstrated and applied. A compatible concurrent implementation of the top layer of file systems is presented, using a mechanism for the efficient management of fine-grained file locking, and a concurrent version of garbage collection is realized. Both concurrency extensions are proven to be correct by applying atomicity refinement, a methodology for proving linearizability.

Finally, this thesis contributes a new iteration of executable code for the Flashix file system. With the efficiency extensions introduced with this thesis, Flashix covers all performance-oriented concepts of realistic file system implementations and achieves competitiveness with state-of-the-art flash file systems.

Acknowledgement

The completion of this thesis would not have been possible without the continued support of several people.

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Dr. Wolfgang Reif who guided, instructed, and motivated me. I am very thankful for his feedback and critical remarks that helped achieve the results of this work as a concluding part of a long-term team effort.

I am also deeply grateful to Dr. Gerhard Schellhorn for the countless times he was willing to discuss in detail any issues that arose within the project, providing valuable insights and sharing his broad experience in modeling and verification.

Many thanks to Gidon Ernst for sparking my interest in formal verification during my study years, introducing me to the Flashix project. I would like to thank him and my former colleague Jörg Pfähler for laying the foundation of this thesis with their work on modularization and verification methodologies and their application to the Flashix file system. Likewise, I like to thank my former colleague Martin Bitterlich for his work on code generation and for contributing to the models and proofs for top-level concurrency. It was a pleasure to work with them and share the office.

Thanks to all my former and current colleagues at the Institute for Software and Systems Engineering for creating an inspiring working environment. It was and is always very enjoyable to discuss various topics in software engineering, prepare and hold lectures together, or have fun conversations during lunch and coffee breaks.

I am thankful to the many students who have contributed to the Flashix project. In particular, I want to thank Stephan Lipp for his work on verifying C code generated from Flashix models, Christian Lehner for realizing various extensions to the upper Flashix layers, Florian Straßer for his contribution to the introduction of external concurrency, Nikolai Glaab and Felix Pribyl for their involvement in the red-black tree verification, and Martin Bürger for continuously maintaining and evaluating the C and Scala integration of Flashix.

Finally, I would like to express my deep gratitude to my wonderful friends and loving family - especially my mother, father, and sister - for continuously supporting me and believing in me.

Stefan Bodenmüller

Contents

List of Figures

Chapter 1

Introduction

Summary This chapter motivates the application of formal methods to realistic and safety-critical software systems. In order to be manageable, the formal development of large-scale software systems requires a methodology for modularization and incremental verification. When targeting a realistic implementation, considering efficiency aspects like caching or concurrency becomes inevitable. This thesis covers the integration of mechanisms geared towards efficiency into existing formally verified systems and contributes techniques for proving the correctness of these extensions modularly. These techniques are applied to the Flashix case study, a verified file system for flash memory, to realize several caching and concurrency extensions. As a central aspect, all these extensions must retain crash-safety, i.e., must guarantee successful recovery from power failures.

Contents

1.1 Motivation

Formal Development of Software Systems This thesis is located in the research area of developing correct software systems by the application of formal methods. In the software engineering process, various methods are usually employed to ensure a certain quality of developed software. These comprise, for example, the application of extensive requirements engineering and documentation, or exhaustive testing techniques. However, all these methods have a certain residual risk of overlooking errors (bugs), for example, by missing rare edge cases in the design and testing process. In the context of safety-critical systems, programming errors can have catastrophic consequences, e.g., the incident of the Ariane 5 rocket in 1996 [78, 9].

As an alternative approach, *formal methods* apply mathematical techniques to *formally prove* the correctness of software. The term *correctness* has several facets in this context: it ranges from guaranteeing terminating executions of programs over showing the absence of runtime exceptions, like dereferencing a *null*-pointer, to proving a program's input/output behavior matching an abstract specification. Often, formal methods are applied to existing programs or libraries in order to prove their correctness afterwards, which can occasionally bring unexpected bugs to light, like with OpenJDK's TimSort [53]. A more constructive approach is to use formal methods during the development of a software system to guarantee *correctness by construction*. While this approach is generally very costly, it gives much stronger correctness guarantees than traditional software development processes when applied diligently. For that reason, the application of formal methods to software development is becoming increasingly popular in industry in recent years [116, 71], e.g., formal specification and model checking is employed at Amazon Web Services [88].

Flash Memory and its Problems Flash memory, in contrast to magnetic disks, does not have random access for writing. Instead, memory is partitioned into blocks consisting of pages, where pages can be written in ascending order only (no overwriting). Deleting old, obsolete data is possible only by erasing entire blocks, which is slow. Erasing also wears out blocks: they become unreadable after 10^4 to 10^6 erases, depending on the hardware. Therefore erasing has to be distributed evenly over all blocks: the use of an algorithm achieving *wear leveling* becomes necessary. The specific writing characteristics require a specialized file system for use in applications like the Mars rovers, which use *raw flash* for efficiency: problems with flash memory on these [101] were one of the motivations for NASA to propose a challenge to build a verifiable flash file system [72, 45], which is a pilot project of the Grand Verification Challenge [62]. An alternative to a dedicated Flash file system used in today's SSDs is an intermediate file translation layer (FTL) that simulates an ordinary file system. This approach requires additional hard- and firmware and is somewhat less efficient.

Correctness of File Systems Developing a correct file system implementation requires a strictly modular discipline that allows verifying individual components separately due to the scale of the resulting system, e.g., realistic file system implementations often comprise tens of thousands of lines of code. However, there is an important cross-cutting concern: power failures (*crashes*) can interrupt a system run anywhere in the middle of running operations, deleting the state still stored in RAM. A file system, therefore, not only has to be shown to be functionally correct but also to be *crash-safe*: a reboot must lead to a consistent state that contains "no surprises", e.g., no new files that were not created before the crash should appear in this state.

Flashix The Flashix project [13] takes on the challenge of [72] with the goal to develop a verified, realistic file system for flash memory that adheres to the POSIX standard [100].

In the first phase of the Flashix project, a sequential implementation was developed, structured as a *refinement tower* containing ten main layers, where a layer is again partitioned into several modular *components*. Each layer consists of a specification component that is realized by an implementation component, where the implementation code calls operations from specifications of other (sub-)components. To be realistic, we have taken the concepts realized in UBIFS [63] and UBI [51] as a blueprint. UBIFS is the newest existing file system for flash memory integrated into the Linux kernel. The sequential implementation already contains a modular implementation of all important concepts of UBIFS: a journal and an index are used for efficiency, algorithms for *garbage collection* and *wear leveling* are implemented. A write-buffer is used to queue data until a full page can be written to flash memory. The top-level specification is an abstract POSIX specification defining directory trees and file content. An indirection is used to accommodate hard links to files. The refinement tower ends with a specification of MTD, which is the generic Linux interface for flash hardware.

However, a purely sequential implementation of a file system is not efficient since the user has to wait for operations to finish. Waiting is necessary for three reasons. First, internal operations such as wear leveling and garbage collection, which are executed in between operations called by the user, cause waiting. Second, the top-level of POSIX operations can be used sequentially only, so one process has to wait for the other even when disjoint directories and files are addressed. Third, the user also has to wait for reads and writes to files to be performed on Flash memory, which are two orders of magnitude slower than writing to RAM. Therefore, the second phase of the project was mainly concerned with making the file system efficient.

Extending Formally Developed Systems Adaptations or extensions to existing software systems always have the drawback that other systems, modules, functions, etc., could be affected unintentionally by the changes applied to the system, potentially harming its correctness. While common software engineering principles like SOLID [82] can reduce this problem, they can usually not be avoided altogether. This is especially true for large-scale formally verified systems like Flashix. Here, exact specifications of operation and component behavior are necessary to prove desired properties of the system. As soon as one small part of the system changes, high-level properties or their corresponding proofs often become invalid, requiring extensive specification and proof work to fix. This is why maintainability and expandability are a major challenge in the development of verified software.

Caching or concurrency extensions are particularly challenging, especially when crashes are considered. When caches are integrated into a hierarchical system, this should usually be done transparently for the layers above, i.e., higher abstraction layers should ideally not notice the additional caches. However, when *write-back caches* are used, i.e., caches that defer the persistence of updates, data is lost in the event of a crash, affecting crash-safety of the upper layers. By contrast, when operations are executed concurrently on a particular layer of the hierarchy, this concurrency propagates to the layers below. When these layers are not designed to be used concurrently, they have to be adapted to either cope with concurrent calls or to restrict the concurrency accordingly. Therefore, a sophisticated approach is required to realize such extensions while maintaining as much of the original proof work as possible.

1.2 Formal Development of Large Software Systems

The basic methodology for developing verified software systems used in this thesis is about decomposing a system into manageable, interchangeable parts, called *compo-*

nents. Components are state-based and provide well-defined interfaces. The state of a component is encapsulated, i.e., it is hidden and can only be accessed indirectly via its interface operations. State is given in form of algebraic data types and operations are specified by programs of a simple imperative programming language, including loops and recursion.

Component hierarchies are constructed by repeated application of the recursive pattern of Fig. 1.1. A specification component $\mathtt{A}_i$ (depicted as white rectangle) specifies the requirements of a (sub-)system. These requirements are typically given in a very abstract, easy-to-understand, and concise way, using algebraic operations and non-determinism. An implementation component $\mathtt{C}_i$ (depicted as gray rectangle) realizes the requirements stated by $\mathtt{A}_i$. For this, $\mathtt{C}_i$ uses another specification $\mathtt{A}_{i+1}$ as *subcomponent* (depicted by —(○—), i.e., it uses the functionality provided by $\mathtt{A}_{i+1}$ by calling its interface operations. From there on, the pattern is continued: $\mathtt{A}_{i+1}$ is realized by another implementation $\mathtt{C}_{i+1}$, which uses a specification $\mathtt{A}_{i+2}$ and so on. Each implementation $\mathtt{C}_i$ implements a certain concept or functionality only while the subcomponent $\mathtt{A}_{i+1}$ serves as abstraction for the remaining part of the system. The complete system implementation is then built by combining all implementations $\mathtt{C}_0$—(○—$\mathtt{C}_1$—(○— … —(○—$\mathtt{C}_n$.

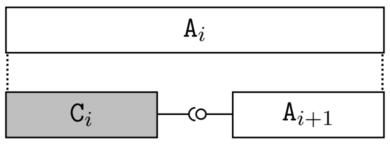


Figure 1.1: Modularization of software systems.

Correctness of the implementation w.r.t. a top-level system specification $\mathtt{A}_0$ is shown incrementally by proving a *refinement* (depicted by dotted lines in Fig. 1.1) for each layer. A implementation $\mathtt{C}_i$ (together with its subcomponent $\mathtt{A}_{i+1}$) *refines* its specification $\mathtt{A}_i$ if clients cannot distinguish between using $\mathtt{C}_i$ or $\mathtt{A}_i$ by looking at the input/output behavior of operations. Refinement is transitive, so proving each refinement individually guarantees correctness of the combined system implementation. Thus, the approach facilitates the formal development of large systems by splitting an otherwise unfeasible verification task into several manageable verification problems.

For the Flashix case study, the correctness requirements are given by a top-level specification of the POSIX standard [100], which defines the expected behavior of file system operations. At the bottom of the hierarchy, a specification of the Linux MTD interface for flash memory specifies assumptions about the flash hardware. We use a code generator to produce executable C and Scala code from implementation components. While the latter is used primarily for testing purposes, the former can be integrated in Linux via the FUSE library (Filesystem in USErspace).

1.3 Mechanizing Proofs: The KIV System

The models and proofs presented in this thesis were created resp. conducted using the KIV system [39, 107]. KIV is a verification tool for the formal development of large-scale, hierarchical software systems. It is actively developed by the Chair of Software and Systems Engineering at the University of Augsburg. Starting as a tool with support for the verification and synthesis of sequential programs using Dynamic Logic in the 80s, it has evolved into a general-purpose theorem prover focusing on developing verified software.

Over the years, KIV has been employed in several major case studies like the WAM compiler for Prolog [105] or the Mondex Challenge [55] or the Flashix project [13]. KIV is also used regularly in courses teaching students the application of formal methods at the University of Augsburg. Furthermore, the research group participated in various international verification competitions, most notably in the VerifyThis Competition series [115], winning multiple awards.

The KIV system is implemented entirely in Scala [92] and is available publicly as a plugin for the Eclipse platform [35] on the KIV website [74]. It provides modern IDE features for editing specifications, such as syntax highlighting, error marking, and the comfortable navigation through projects (via commands like *go to definition*). A similar plugin for the IntelliJ IDEA platform [65] is currently in the making.

At its core, KIV is an interactive theorem prover. The user can conduct proofs manually by interacting with a graphical user interface that allows the context-sensitive application of rules and rewriting. Proofs are saved by storing explicit proof trees, which the user can view visually and manipulate directly. Typically, the proving process is highly automated. To achieve this, KIV uses automated rewriting and powerful heuristics.

The KIV logic is based on a polymorphic higher-order logic, including different formalisms for sequential and concurrent programs. Software systems are developed with a component- and refinement-based approach, using, for example, a weakest-precondition (wp) or Rely-Guarantee (RG) calculus. KIV supports multiple theories developed and applied during the Flashix project by generating specifications and proof obligations, thus reducing the risk of errors.

1.4 Contributions of this Thesis

This thesis is a continuation of the work of Ernst [38] and Pfähler [96]. Consequently, the methodologies for specifying, modularizing, and verifying large-scale and concurrent systems developed by them are taken up and developed further in this thesis, aiming to improve the formal development of efficiency-critical software systems. In particular, the following main contributions are made.

Extensions to Formally Developed Software Systems As one of its main challenges to overcome, this work gives methodologies for performing modular extensions to formally developed systems. Functionalities and concepts can be added to an existing component hierarchy without losing modularity and without having to start verification from scratch. Focusing on the pervasive concepts of *caching* and *concurrency*, it is shown how major parts of existing specification and verification work can be kept by introducing additional proof obligations that can be proved separately and by restricting the impacts of extensions to certain parts of the hierarchy.

Modular Verification Methodology for Destructive Heap Algorithms An approach for simplifying the verification of destructive, heap-based algorithms is given. Reasoning about algorithms manipulating heap data structures is often difficult since complicating concepts like aliasing have to be considered. Methods tailored to heap-based programs, such as *Separation Logic* [102], support the reasoning. However, verification usually still suffers from the complexity of conceptual correctness arguments being intertwined with questions about pointer aliasing and side effects. This thesis gives an approach for *Separating Separation Logic* which allows to reason about functional correctness and the correctness of heap modifications separately.

Correctness and Crash-Safety of Cached and Concurrent Systems The component model of Ernst [38] and Pfähler [96] is adapted for the use in highly cached and concurrent systems. The *operations-based* approach for specifying crash behavior of Pfähler is extended to cover the integration of more fine-grained caches using complex synchronization mechanisms that reorder operations. This thesis gives the novel, history-based correctness criterion *Write-Prefix Crash Consistency* for the use of such caches and provides compatibility of the criterion with the use of other caching mechanisms and concurrency.

Verification of Caching Mechanisms and Concurrency in File Systems This thesis contributes methods for the verification of functional correctness as well as crash-safety of caching mechanisms and concurrency in file systems. A reference implementation for crash-safe caches of high-level data structures is given, adhering to the requirements implied by Write-Prefix Crash Consistency. A respective proof strategy is demonstrated and applied to the case study. A mechanism for the efficient management of fine-grained file locking is presented, which is used for a concurrent implementation of the top layer of file systems, the *Virtual File System Switch*, allowing concurrent calls to the file system interface. Furthermore, a concurrent realization of garbage collection is presented. Both concurrency extensions are proven to be correct by applying *atomicity refinement*, a methodology for proving linearizability [60] (including termination and deadlock-freedom) adopted from Pfähler [96].

Iteration of the Flashix File System Implementation Finally, this thesis contributes a new iteration of working C and Scala code for the Flashix file system. The code is generated from the implementation models developed in this thesis, based on the models developed by Ernst [38] and Pfähler [96]. It contains all the efficiency extensions presented in this thesis and shows significant improvements in terms of performance in comparison to earlier versions, achieving competitiveness with state-of-the-art file systems like UBIFS [63].

Publications The following publications were created in the context of the work presented in this thesis.

1. J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Verification of Order-Preserving Write-Back Caches. In *Proc. of International Conference on Integrated Formal Methods (IFM)*, volume 10510 of *LNCS*, pages 375–390. Springer, 2017

2. G. Schellhorn, G. Ernst, J. Pfähler, S. Bodenmüller, and W. Reif. Symbolic Execution for a Clash-Free Subset of ASMs. *Science of Computer Programming (SCP)*, 158:21–40, 2018

3. G. Schellhorn, S. Bodenmüller, J. Pfähler, and W. Reif. Adding Concurrency to a Sequential Refinement Tower. In *Proc. of International Conference on Rigorous State-Based Methods (ABZ)*, volume 12071 of *LNCS*, pages 6–23. Springer, 2020. Invited Paper

4. S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Integration of Crashsafe Caching into a Verified Virtual File System Switch. In *Proc. of International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *LNCS*, pages 218–236. Springer, 2020

5. S. Bodenmüller, G. Schellhorn, M. Bitterlich, and W. Reif. Flashix: Modular Verification of a Concurrent and Crash-Safe Flash File System. In *Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday*, volume 12750 of *LNCS*, pages 239–265. Springer, 2021

6. S. Bodenmüller, G. Schellhorn, and W. Reif. Verification of Crashsafe Caching in a Virtual File System Switch. *Formal Aspects of Computing (FAC)*, 34(1), 2022

7. G. Schellhorn, S. Bodenmüller, M. Bitterlich, and W. Reif. Software & System Verification with KIV. In *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, volume 13360 of *LNCS*, pages 408–436. Springer, 2022

8. G. Schellhorn, S. Bodenmüller, M. Bitterlich, and W. Reif. Separating Separation Logic – Modular Verification of Red-Black Trees. In *Proc. of International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 13800 of *LNCS*, pages 129–147. Springer, 2022

1.5 Outline

The remainder of this thesis is structured as follows.

Chapter 2 provides some theoretical background required for this thesis. It covers the fundamental logic, the specification of algebraic data types and operations, an imperative programming language together with a calculus for sequential reasoning, and the modeling of and reasoning about heap data structures.

Chapter 3 introduces the methodology for constructing large-scale, hierarchical systems based on components. It provides details on the syntax and semantics of components and a proof method for the refinement of sequential components. Furthermore, a methodology for the modular verification of destructive heap algorithms is presented based on a pointer-based implementation of red-black trees [54].

Chapter 4 gives an overview of the Flashix file system, including a summary of the main layers forming the component hierarchy. The extensions performed in the second project phase and, in particular, in this thesis are highlighted. Furthermore, the models of the top layer are presented as a recap of the work of Ernst [38], as the basis for the following chapters.

Chapter 5 introduces retractions to the semantics of components in order to describe the crash behavior of buffered systems and gives a short recap of its application to a buffer cache integrated into Flashix as part of Pfähler's work [96]. The semantics of [96] are adapted to be able to express crash effects of high-level caches.

Chapter 6 presents the extension of Flashix with high-level caches. It shows how an additional caching layer can be integrated into an existing component hierarchy and how the functional correctness can be verified modularly. An expressive and intuitive crash-safety criterion for cached file systems is given, and a strategy for proving crash-safety of such systems is proposed and applied to the Flashix reference implementation. Finally, it is shown that, when realized correctly, such high-level caches can be combined safely with low-level buffer caches.

In Chapter 7, the method for introducing concurrency into a component hierarchy is described, based on the work of Pfähler [96]. The semantics of components are adapted to allow interleaved executions of operations, and a methodology for proving the correctness of concurrent components with respect to an atomic specification is presented.

Chapter 8 then applies the methodology of Chapter 7 to introduce concurrency on the top levels of Flashix. In order to enable concurrent access to files, a mechanism for managing dynamically allocated locks efficiently is provided. It is employed in the realization of a concurrent Virtual File System Switch. Furthermore, a concurrent implementation of the garbage collection algorithm is presented. Proofs for both concurrency extensions are outlined as well.

To evaluate the efficiency extensions presented in this thesis, Chapter 9 demonstrates how the resulting implementation of the Flashix file system compares to previous versions as well as a real-world flash file system implementation, namely UBIFS.

Finally, Chapter 10 summarizes and gives an outlook on possible future work.

Chapter 2

Theoretical Background

Summary This chapter summarizes the basic logical foundations for this thesis. They are all supported by the interactive verification system KIV, which was used for the specification and verification work of this thesis. In the context of this work, the underlying higher order logic was extended from monomorphic to polymorphic types. The models are built upon structured specifications of algebraic data types and operations. An imperative programming language is used to model software systems, and a sequent-based weakest precondition calculus is used to reason about properties of sequential programs. The programming language and calculus were extended to support exceptions so that the absence of runtime exceptions in programs can be proven. Reasoning about pointer structures is done with Separation Logic using an explicit heap model.

Contents

Publications Most of the logical foundations presented in this chapter are based on [107].

2.1 Polymorphic Higher Order Logic

The logical basis of this thesis is higher order logic (HOL) using polymorphic types [52, 107]. The definition of the set of types Ty is based on a finite set of type constructors $\mathsf{tc}{:}l \in Tc$ with fixed arity l and a countable set of type variables $'a \in Tv$.

Definition 1 (Polymorphic Types)**.** *The syntax of types* $\mathsf{ty} \in \mathit{Ty}$ *is given by the following grammar.*

$$\begin{array}{lll} \mathsf{ty} := & {}'a & \textit{type variable} \\ & \mid \mathsf{tc}{:}l(\mathsf{ty}_1, \ldots, \mathsf{ty}_l) & \textit{type constructor application} \\ & \mid (\mathsf{ty}_1, \ldots, \mathsf{ty}_n) & \textit{n-nary tuple type} \\ & \mid \mathsf{ty}_1 \times \ldots \times \mathsf{ty}_n \to \mathsf{ty}' & \textit{n-nary function type} \end{array}$$

Type sequences must have l elements in the application of a type constructor $\mathsf{tc}{:}l$, at least two elements for tuples, and at least one element for function types. *Type constants* are applications of type constructors $\mathsf{tc}{:}0$ with arity 0. Types that do not contain type variables are called *monomorphic*, types containing type variables are called *polymorphic*. It is assumed that the boolean type constant is predefined, i.e., $\mathsf{bool}{:}0() \in \mathit{Tc}$. Usually, the arity of type constructors as well as the parenthesis of type constants are left implicit when writing types, so for example, the boolean type is written bool.

Expressions $e \in \mathit{Expr}$ are defined over a set of (typed) variables $x{:}\mathsf{ty} \in X$ and a signature $\Sigma = (\mathit{Tc}, \mathit{Op})$ which in addition to type constructors contains (typed) operations $\mathsf{op}{:}\mathsf{ty} \in \mathit{Op}$. Op always includes the usual boolean operations like $\texttt{true} : \mathsf{bool}$, $\neg\, . : \mathsf{bool} \to \mathsf{bool}$ (written prefix), or $. \wedge . : \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool}$ (written infix), equality $. = . :\ {}'a \times {}'a \to \mathsf{bool}$, an if-then-else-operator $\supset$: $\mathsf{bool} \times\ {}'a \times\ {}'a \to {}'a$, as well as tuple constructors (written $(e_1, \ldots, e_n)$) and tuple selectors for every arity (e.g., written $e.\_3$). For *predicates*, i.e., operations with a type $\mathsf{ty}_1 \times \ldots \times \mathsf{ty}_n \to \mathsf{bool}$, the target type bool is typically omitted.

Definition 2 (Higher Order Expressions)**.** *The syntax of higher order expressions* $e \in \mathit{Expr}$ *is given by the following grammar.*

$$\begin{array}{lll} e := & x{:}\mathsf{ty} & \textit{variable} \\ & \mid \mathsf{op}{:}\mathsf{ty} & \textit{operation} \\ & \mid e_0(e_1, \ldots, e_n) & \textit{application} \\ & \mid \lambda\ \underline{x{:}\mathsf{ty}}.\ e & \textit{lambda abstraction} \\ & \mid \forall\ \underline{x{:}\mathsf{ty}}.\ \varphi & \textit{universal quantifier} \\ & \mid \exists\ \underline{x{:}\mathsf{ty}}.\ \varphi & \textit{existential quantifier} \end{array}$$

In the definition, φ denotes a *formula*, i.e., an expression of type bool. Expressions without quantifiers are called *terms* (typically denoted by t), quantifier-free formulas are denoted by ε. $\underline{x{:}\mathsf{ty}}$ denotes a sequence $x_1{:}\mathsf{ty}_1, \ldots, x_n{:}\mathsf{ty}_n$ of typed variables (in this thesis, sequences of any kind will be signified by an underline throughout), the variables of $\underline{x{:}\mathsf{ty}}$ must be pairwise disjoint. The typing rules are standard, e.g., in an application, the type of e_0 must be a function type where the argument types are equal to those of $e_1, \ldots, e_n$. Most types can be inferred by type inference, so in the following, the types of variables and the instance types of operations are left implicit in formulas. Application of the if-then-else-operator is written as $(\varphi \supset e_1; e_2)$. Its result is e_1, when φ is true, and e_2 otherwise.

The semantics of an expression $[\![e]\!]$ is based on algebras $\mathcal{A} = (\mathcal{U}, \{\mathsf{tc}{:}l^{\mathcal{A}}\}, \{\mathsf{op}{:}\mathsf{ty}^{\mathcal{A}}\})$, following the semantics of HOL defined in [52].

The first component of an algebra is the universe $\mathcal{U}$, which is a set of non-empty (potential) carrier sets. The semantics $\mathsf{tc}{:}l^{\mathcal{A}} : \mathcal{U}^l \to \mathcal{U}$ of a type constructor maps the carrier sets of its argument types to the one of the full type. The semantics of booleans, functions and tuples is standard, i.e., $\mathcal{U}$ is assumed to contain the set $\mathbb{B} = \{\mathbf{tt}, \mathbf{ff}\}$ that interprets booleans, and to be closed against forming Cartesian products and functions to interpret function and tuple types. Given a type valuation $w : Tv \to \mathcal{U}$, that maps each type variable to a carrier set, an algebra fixes the semantics $[\![ty]\!](\mathcal{A}, w)$ of a type ty as one of the carrier sets in $\mathcal{U}$.

Definition 3 (Semantics of Types)**.** *Given an algebra $\mathcal{A}$ and a type valuation w, types* $\mathsf{ty} \in Ty$ *are evaluated as follows.*

$$\begin{aligned}
[\![{'a}]\!](\mathcal{A}, w) &= w({'a}) \\
[\![\mathsf{tc}{:}l(\mathsf{ty}_1, \dots, \mathsf{ty}_l)]\!](\mathcal{A}, w) &= \mathsf{tc}{:}l^{\mathcal{A}}([\![\mathsf{ty}_1]\!](\mathcal{A}, w), \dots, [\![\mathsf{ty}_l]\!](\mathcal{A}, w)) \\
[\![(\mathsf{ty}_1, \dots, \mathsf{ty}_n)]\!](\mathcal{A}, w) &= ([\![\mathsf{ty}_1]\!](\mathcal{A}, w), \dots, [\![\mathsf{ty}_n]\!](\mathcal{A}, w)) \\
[\![\mathsf{ty}_1 \times \ldots \times \mathsf{ty}_n \to \mathsf{ty}']\!](\mathcal{A}, w) &= [\![\mathsf{ty}_1]\!](\mathcal{A}, w) \times \ldots \times [\![\mathsf{ty}_n]\!](\mathcal{A}, w) \to [\![\mathsf{ty}']\!](\mathcal{A}, w)
\end{aligned}$$

The interpretation $\mathsf{op}{:}\mathsf{ty}^{\mathcal{A}}$ of an operation over an algebra yields an element of $[\![ty]\!](\mathcal{A}, w)$ for every possible type valuation w. Fixing such an interpretation to a type valuation w is denoted by $\mathsf{op}{:}\mathsf{ty}^{\mathcal{A}}(w)$. Similarly, a valuation v maps each variable $x{:}ty$ to an element of $[\![ty]\!](\mathcal{A}, w)$ for every possible w. Fixing the type valuation of a valuation is written $v(w)$. Finally, the semantics $[\![e]\!](\mathcal{A}, w, v)$ of an expression of type ty yields an element of the carrier set $[\![ty]\!](\mathcal{A}, w)$.

Definition 4 (Semantics of Expressions)**.** *Given an algebra $\mathcal{A}$, a type valuation w, and a valuation v, expressions $e \in Expr$ are evaluated as follows.*

$$\begin{aligned}
[\![x{:}\mathsf{ty}]\!](\mathcal{A}, w, v) &= v(w)(x{:}\mathsf{ty}) \\
[\![\mathsf{op}{:}\mathsf{ty}]\!](\mathcal{A}, w, v) &= \mathsf{op}{:}\mathsf{ty}^{\mathcal{A}}(w) \\
[\![e_0(e_1, \dots, e_n)]\!](\mathcal{A}, w, v) &= [\![e_0]\!](\mathcal{A}, w, v)([\![e_1]\!](\mathcal{A}, w, v), \dots, [\![e_n]\!](\mathcal{A}, w, v)) \\
[\![\lambda\ \underline{x{:}\mathsf{ty}}.\ e]\!](\mathcal{A}, w, v) &= \underline{a} \mapsto [\![e]\!](\mathcal{A}, w, v(\underline{x{:}\mathsf{ty}} \mapsto \underline{a}) \qquad \textit{for } a_i \in [\![\mathsf{ty_i}]\!](\mathcal{A}, w) \\
[\![\forall\ \underline{x{:}\mathsf{ty}}.\ \varphi]\!](\mathcal{A}, w, v) &= \mathbf{tt} \quad \textit{iff } [\![\varphi]\!](\mathcal{A}, w, v(\underline{x{:}\mathsf{ty}} \mapsto \underline{a})) = \mathbf{tt} \\
&\qquad\quad \textit{for all } a_i \in [\![\mathsf{ty_i}]\!](\mathcal{A}, w) \\
[\![\exists\ \underline{x{:}\mathsf{ty}}.\ \varphi]\!](\mathcal{A}, w, v) &= \mathbf{tt} \quad \textit{iff } [\![\varphi]\!](\mathcal{A}, w, v(\underline{x{:}\mathsf{ty}} \mapsto \underline{a})) = \mathbf{tt} \\
&\qquad\quad \textit{for some } a_i \in [\![\mathsf{ty_i}]\!](\mathcal{A}, w)
\end{aligned}$$

A formula φ is valid in $\mathcal{A}$ for given w and v (written $\mathcal{A}, w, v \models \varphi$) when the semantics of φ evaluates to $\mathbf{tt}$. An algebra is a model of an axiom φ ($\mathcal{A} \models \varphi$) iff the formula evaluates to true for all w and v.

In the following we are interested in valuations v that are used as (the changing) states of programs, while $\mathcal{A}$ and w are fixed. In this case, these arguments are often dropped and the semantics of an expression is written $[\![e]\!](v)$. Similarly, $v \models \varphi$ abbreviates in this work that a formula is valid for a given valuation v.

2.2 Structured Specifications of Algebraic Data Types

Structured algebraic specifications are used to build a hierarchy of data type and operation definitions. A specification contains a signature Σ and a set of axioms

Ax: the former contains the type constructors and operation symbols (cf. Sec. 2.1) that are available, the latter defines the properties of the operations. An algebra $\mathcal{A}$ is a model of a specification iff it is a model for all its axioms Ax (written $\mathcal{A} \models Ax$). Specifications can be augmented by additional operations and can be combined using the usual structuring operations like enrichment, union, and renaming. A generic instantiation concept allows to replace an arbitrary subspecification P (the "parameter") of a generic specification G with an actual specification A using a *mapping.* A mapping is a generalized morphism that renames types and operations of P to types and expressions over A. For a correct instantiation, it must be proven that the axioms of P instantiated by the mapping are theorems over A.

Algebraic data types are generated using constructor functions, a data type can be either *free* or *non-free.* For a free data type ty_f, the semantics $[\![\mathsf{ty}_f]\!](\mathcal{A}, w)$ is given directly by the semantics of all terms only containing constructor functions (called *constructor terms*). Thus, the axiomatization of the type is evident and can typically be performed automatically. On the other hand, non-free data types are specified by giving suitable axioms to guarantee that a proper quotient type is defined.

A frequently used (polymorphic) free data type is $\mathsf{list}('a)$ (free data types are introduced with the keyword **data**).

$$\textbf{data}\quad \mathsf{list}('a) = \texttt{[]} \mid \texttt{.} + \texttt{.}\ (\texttt{.}\ \texttt{.head} : 'a\ ;\ \texttt{.}\ \texttt{.tail} : \mathsf{list}('a))$$

A list $\mathsf{list}('a)$ is defined using a constant constructor `[]` (representing the empty list) and a non-constant infix constructor `+`. Non-empty lists consist of an head element of generic type $'a$ and and a remaining tail list. These fields can be accessed via the postfix selector functions `.head` and `.tail`, respectively. Similarly, update functions (written e.g. x`.head:=` *newhead*) yield a list with one field replaced.

Selector (and update) functions are underspecified in the sense that they are not given axioms for all arguments: `[].head` is left unspecified (as is `[].tail`). The semantic function in a model then is still a total function, and `[].head` may be any value, following the standard loose approach to semantics. However, for use in programs, a *domain* is attached to the function, here given as $\lambda\ x.\ x \neq \texttt{[]}$. Calling `.head` outside of its domain in a program (here: with `[]`, where it is "undefined") will raise an exception, explained in detail in Sec. 2.3 and Sec. 2.4.

Other relevant list operations are $\#\ \texttt{.} : \mathsf{list}('a) \to \mathsf{nat}$ (length of a list, nat denotes the type of natural numbers $\mathbb{N}$), $\texttt{.} \in \texttt{.} : 'a \times \mathsf{list}('a)$ (a list contains an element), and $\texttt{.} +\!\!+ \texttt{.} : \mathsf{list}('a) \times \mathsf{list}('a) \to \mathsf{list}('a)$ (concatenation of two lists).

One of the most relevant non-free data types in this thesis is $\mathsf{map}('a, 'b)$, representing a partial function from keys of type $'a$ to values of type $'b$. Maps can be accessed with $\texttt{.}\ \texttt{[}\ \texttt{.}\ \texttt{]} : \mathsf{map}('a, 'b) \times 'a \to 'b$ for retrieving a value stored under a key (written mixfix, e.g., $mp\,\texttt{[}k\texttt{]}$) or with $\texttt{.}\ \texttt{[}\ \texttt{.}\ \texttt{]} : \mathsf{map}('a, 'b) \times 'a \times 'b \to \mathsf{map}('a, 'b)$ for storing resp. updating a key-value pair (e.g., $mp\,\texttt{[}k, v\texttt{]}$). The predicate $\texttt{.} \in \texttt{.} : 'a \times \mathsf{map}('a, 'b)$ is used to determine if there is an entry for a key in the map. Similarly, $\texttt{.} \in_v \texttt{.} : 'b \times \mathsf{map}('a, 'b)$ determines if at least one key-value pair with the requested value exists. The operation $\texttt{.}\ \texttt{--}\ \texttt{.} : \mathsf{map}('a, 'b) \times 'a \to \mathsf{map}('a, 'b)$ deletes the pair stored under a key from the map.

Further commonly used non-free data types are (unordered) sets $\mathsf{set}('a)$ and arrays $\mathsf{array}('a)$. Elements can be added to and removed from a set with the operations $\texttt{.}\ \texttt{++}\ \texttt{.} : \mathsf{set}('a) \times 'a \to \mathsf{set}('a)$ and $\texttt{.}\ \texttt{--}\ \texttt{.} : \mathsf{set}('a) \times 'a \to \mathsf{set}('a)$, respectively. The empty set is denoted by $\emptyset$, the typical set operators $(\cup, \cap, \ldots)$ are used as usual. An array of fixed size is constructed with $\mathtt{mkarray} : \mathsf{nat} \to \mathsf{array}('a)$. Arrays are indexed with natural numbers and zero-based, and elements can be read or updated using a notation similar to maps $(\,.\,\mathtt{[}\,.\,\mathtt{]} : \mathsf{array}('a) \times \mathsf{nat} \to 'a$ and $.\,\mathtt{[}\,.\,\mathtt{]} : \mathsf{array}('a) \times \mathsf{nat} \times 'a \to \mathsf{array}('a))$. The prefix operation $\#\,.$ is used uniformly to denote the cardinality of a set, the length of an array, and the size of a map, i.e., the number of stored key-value pairs.

Like for lists, some array and map operations are undefined for some arguments. For example, the result of accessing an array location outside of its bounds $(ar\,\mathtt{[}n\mathtt{]}$ for $\#ar \le n)$ or looking up a value for an unallocated key $(mp\,\mathtt{[}k\mathtt{]}$ for $\neg\ k \in mp)$ is not specified. Therefore, a domain is also added to these operations to be able to reason about the occurrence of runtime exceptions, as shown in the following sections.

Together with basic types like $\mathsf{nat}, \mathsf{bool}, \mathsf{string}, \ldots$, the data types introduced in this section form a fundamental collection of types for this thesis. This collection is extended with more specific types in the respective chapters.

2.3 Imperative Programs with Exceptions

Software systems in this thesis are modeled using an imperative programming language with recursive procedures, similar to the rules of Turbo ASMs [16].

Definition 5 (Syntax of Programs). *The syntax of programs α, β is given by the following grammar.*

$\alpha :=$	$\underline{x} := \underline{t}$	*parallel assignment*
	$\mid \alpha; \beta$	*sequential composition*
	$\mid$ **if*** ε **then** α **else** β	*conditional*
	$\mid$ **choose*** $\underline{x}$ **with** φ **in** α **ifnone** β	*nondeterministic choice*
	$\mid$ **while*** ε **do** α	*iteration*
	$\mid$ **proc#**$(\underline{t}; \underline{u}; \underline{v})$	*procedure call*
	$\mid$ **throw** $\mathtt{op}$	*exception throwing*
	$\mid$ **atomic** ε $\{\alpha\}$	*atomic block*
	$\mid$ **forall**$\|$ $\underline{x}$ **with** φ **do** α	*weak-fair interleaving*

Assignments $\underline{x} := \underline{t}$ assign each variable x_i the value of term t_i simultaneously. This allows for swaps of the form $x, y := y, x$ in a single statement with no additional variables necessary. Assignments to functions $f := f\,\mathtt{[}a \mapsto b\mathtt{]}$ are usually written $f\,\mathtt{[}a\mathtt{]} := b$. A similar notation is used for assignment to locations of indexed data types like arrays or maps, e.g., $ar\,\mathtt{[}n\mathtt{]} := a$ and $mp\,\mathtt{[}k\mathtt{]} := v$. The program **skip**, that does nothing, is used as an abbreviation for an empty assignment (**skip** $\equiv \langle\rangle := \langle\rangle$).

Two programs α and β can be executed in order by sequential composition (;). Conditionals evaluate the (quantifier-free) formula ε: if the evaluation yields **tt**, α is executed, otherwise β is executed. **choose*** introduces new local variables $\underline{x}$ with a random valuation that satisfies a condition φ, and executes α using these variables. If there is no valuation satisfying φ, β is executed instead. When no restriction on the valuation of $\underline{x}$ is made, i.e., φ is `true`, the condition can be omitted. **while***-loops execute their body α as long as the loop condition ε is satisfied. The non-terminating program **while*** `true` **do skip** is abbreviated by the program **abort**.

A procedure **proc#** can be called with input arguments $\underline{t}$, reference arguments $\underline{u}$, and output arguments $\underline{v}$ by the statement **proc#**$(\underline{t}; \underline{u}; \underline{v})$, where the different argument vectors are separated by semicolons. Input arguments may be arbitrary terms, while only variables are allowed to be passed as reference and output arguments. Typically, a procedure has a declaration of the form **proc#**$(\underline{x}; \underline{y}; \underline{z})\{\alpha\}$ with disjoint formal parameters $\underline{x}$, $\underline{y}$, and $\underline{z}$. When calling a procedure, the types of the actual arguments of the call must match those of the formal parameters. The body α may only use variables of the formal parameters or local variables (introduced by **choose***). Thus, a procedure call cannot implicitly manipulate any global state unless it is passed as an argument and, consequently, syntactically visible to the caller. Furthermore, α must set all output parameters $\underline{z}$, and updates to the input parameters $\underline{x}$ are invisible to the caller of **proc#**.

Program statements may raise an exception if a partial function (see Sec. 2.2) is applied to arguments outside of its domain. Each exception is thus coupled with an operation `op`. For example, the subtraction operation $-$ on natural numbers throws its exception in the program $m := n_0 - n_1$ when $n_0 < n_1$. Additionally, the exception of operation `op` can be thrown explicitly by the **throw** program.

The **atomic** and **forall**$\|$ constructs are only relevant when considering programs in a concurrent context (cf. Chapter 7). An **atomic** statement (passively) waits for the environment to make the *guard* ε true. While ε is false, the program is *blocked*. A program where the environment never enables the guard is deadlocked. When the test becomes true, the program α is executed in a single, indivisible step, i.e., there is no interleaving of steps of other threads with α. The typical use of the construct is to model locking, where a thread must wait until the lock is free before it atomically acquires the lock. The program **forall**$\|$ interleaves instances of α for all values that satisfy φ bound to local variables $\underline{x}$. The steps of all instances are interleaved non-deterministically, where the interleaving is only blocked if all instances are blocked. It has a weak-fairness constraint: if an instance of α is enabled continuously (i.e., it is never blocked), it will eventually execute a step, even if other instances are always enabled. A typical use would be **forall**$\|$ n **with** $n < m$ **do** α where the body α uses variable n as the thread identifier.

Besides this base set of program constructs given in Def. 5, some abbreviations are used. For both conditionals and choices, the β clauses are dropped if they are irrelevant.

$$\begin{aligned} \textbf{if* } \varepsilon \textbf{ then } \alpha &\equiv \textbf{if* } \varepsilon \textbf{ then } \alpha \textbf{ else skip} \\ \textbf{choose* } \underline{x} \textbf{ with } \varphi \textbf{ in } \alpha &\equiv \textbf{choose* } \underline{x} \textbf{ with } \varphi \textbf{ in } \alpha \textbf{ ifnone abort} \end{aligned}$$

We use the **let*** program to introduce new variables $\underline{x}$ initialized with values $\underline{t}$ and the **or*** program for an indeterministic choice between two programs α and β. Both constructs can be expressed using **choose***.

$$\begin{aligned} \textbf{let* } \underline{x} = \underline{t} \textbf{ in } \alpha &\equiv \textbf{choose* } \underline{y} \textbf{ with } \underline{y} = \underline{t} \textbf{ in } \alpha_{\underline{x}}^{\underline{y}} \textbf{ ifnone skip} \\ \alpha \textbf{ or* } \beta &\equiv \textbf{choose } bv \textbf{ in } \{\textbf{if* } bv \textbf{ then } \alpha \textbf{ else } \beta\} \end{aligned}$$

Evaluating the conditions of the constructs marked with an asterisk (*) does not take an extra step, instead the evaluation takes place in one atomic step with α or β. While this behavior helps specifying (concurrent) systems or modeling instructions like CAS (compare-and-swap), it generally does not match the expected behavior of common executable programming languages. Therefore, additional constructs (**if**, **choose**, **while**, **let**, **or**) are introduced as abbreviations for the right hand sides of the following equations.

$$\begin{aligned}
\textbf{if}\ \varepsilon\ \textbf{then}\ \alpha\ \textbf{else}\ \beta &\equiv \textbf{if*}\ \varepsilon\ \textbf{then}\ \{\textbf{skip}; \alpha\}\ \textbf{else}\ \{\textbf{skip}; \beta\}\\
\textbf{choose}\ \underline{x}\ \textbf{with}\ \varphi\ \textbf{in}\ \alpha\ \textbf{ifnone}\ \beta &\equiv \textbf{choose*}\ \underline{x}\ \textbf{with}\ \varphi\ \textbf{in}\ \{\textbf{skip}; \alpha\}\\
&\qquad\qquad\qquad \textbf{ifnone}\ \{\textbf{skip}; \beta\}\\
\textbf{while}\ \varepsilon\ \textbf{do}\ \alpha &\equiv \{\textbf{while*}\ \varepsilon\ \textbf{do}\ \{\ \textbf{skip}; \alpha\}\}; \textbf{skip}\\
\textbf{let}\ \underline{x} = \underline{t}\ \textbf{in}\ \alpha &\equiv \textbf{let*}\ \underline{x} = \underline{t}\ \textbf{in}\ \{\textbf{skip}; \alpha\}\\
\alpha\ \textbf{or}\ \beta &\equiv \{\textbf{skip}; \alpha\}\ \textbf{or*}\ \{\textbf{skip}; \beta\}
\end{aligned}$$

The semantics of programs $[\![\alpha]\!]$ is defined as a set of finite or infinite state sequences called *intervals*, following the terminology of interval temporal logic (ITL) [84]. Formally, an interval is of the form

$$I = (I(0), I(0)_b, I'(0), I(1), I(1)_b, I'(1), I(2), \dots, \zeta)$$

where every $I(k)$ and $I'(k)$ are valuations (*states*) which map variables to values. The transitions from $I(k)$ to $I'(k)$ are system transitions and the transitions from $I'(k)$ to $I(k+1)$ from a primed to the subsequent unprimed state are environment transitions. Hence, intervals alternate between system and environment transitions, similar to the reactive sequence semantics in [31].

The length of an interval $\#I$ is defined as the number of system transitions (which is equal to the number of environment transitions) if the interval is finite, otherwise $\#I = \infty$. Thus, an interval with length $\#I$ has $2 \cdot \#I + 1$ states, and notably, the smallest interval with $\#I = 0$ consists of only a single state. The suffix of an interval after n system and environment steps is denoted $I|_{[n..]}$. $I.\mathit{first}$ selects the first state $I(0)$ of an interval, the last state $I'(n)$ of a finite interval $I = (I(0), \dots, I'(n), \zeta)$ is written as $I.\mathit{last}$.

To model passive waiting, the boolean flag $I(k)_b$ denotes whether the program transition from $I(k)$ to $I'(k)$ is *blocked*, i.e., the program waits to continue its execution. In this case, when $I(k)_b = \textbf{tt}$, then $I'(k) = I(k)$.

To model exceptions when running a program, the final state of a finite run carries information ζ whether an exception has occurred. This may be either $\top$ to indicate regular termination without exception or the information that an operation $\mathsf{op} \in \mathit{Op}$ has thrown its exception. To have uniform notation, it is assumed that $\zeta = \infty$ for infinite (non-terminating) runs, so $\zeta \in \mathit{Op} \cup \{\top, \infty\}$. The extraction of this information from an interval I is written $\zeta(I)$.

Two intervals I_0 and I_1 can be concatenated, written $I_0 \fatsemi I_1$, if they agree on their last and first states, respectively, i.e., if $I_0.\mathit{last} = I_1.\mathit{first}$. If I_0 is infinite or ends with an exception ($\zeta(I_0) \in \mathit{Op} \cup \{\infty\}$), $I_0 \fatsemi I_1$ is equal to I_0. If $\zeta(I_0) = \top$, the concatenated interval $I_0 \fatsemi I_1$ contains the duplicate state $I_0.\mathit{last}/I_1.\mathit{first}$ only once.

The semantics of programs is *compositional*, i.e., the semantics of complex programs can be constructed by combining intervals that are members of the semantics of its parts. This has been explained in detail for programs without exceptions in [110], Pfähler added the **atomic** construct to the semantics in his thesis [96]. Def. 6 extends the semantics to include exceptions and the **throw** program.

Definition 6 (Semantics of Programs)**.** *The semantics of a program $I \models \alpha$ is given by the greatest fixed point of the following derivation system.*

$$\frac{I \models \textbf{throw}\ \mathtt{op}}{I \models \underline{x} := \underline{t}} \quad \textit{iff} \quad I(0) \models \omega^{\mathtt{op}}(\underline{t}) \tag{2.1}$$

$$\frac{}{I \models \underline{x} := \underline{t}} \quad \textit{iff} \quad \begin{array}{l} \#I = 1,\ I(0)_b = \mathbf{ff},\ I(0) \models \delta(\underline{t}),\ \zeta(I) = \top, \\ \textit{and } I'(0) = I(0)\{\underline{x} \mapsto [\![\underline{t}]\!](I(0))\} \end{array} \tag{2.2}$$

$$\frac{I_0 \models \alpha \quad I_1 \models \beta}{I \models \alpha; \beta} \quad \textit{iff} \quad I = I_0 ⨟ I_1 \tag{2.3}$$

$$\frac{I \models \textbf{throw}\ \mathtt{op}}{I \models \textbf{if*}\ \varepsilon\ \textbf{then}\ \alpha\ \textbf{else}\ \beta} \quad \textit{iff} \quad I(0) \models \omega^{\mathtt{op}}(\varepsilon) \tag{2.4}$$

$$\frac{I \models \alpha}{I \models \textbf{if*}\ \varepsilon\ \textbf{then}\ \alpha\ \textbf{else}\ \beta} \quad \textit{iff} \quad I(0) \models \delta(\varepsilon)\ \textit{and}\ I(0) \models \varepsilon \tag{2.5}$$

$$\frac{I \models \beta}{I \models \textbf{if*}\ \varepsilon\ \textbf{then}\ \alpha\ \textbf{else}\ \beta} \quad \textit{iff} \quad I(0) \models \delta(\varepsilon)\ \textit{and}\ I(0) \not\models \varepsilon \tag{2.6}$$

$$\frac{I \models \textbf{throw}\ \mathtt{op}}{I \models \textbf{choose*}\ \underline{x}\ \textbf{with}\ \varphi\ \textbf{in}\ \alpha\ \textbf{ifnone}\ \beta} \quad \textit{iff} \quad I(0) \models \exists\ \underline{x}.\ \omega^{\mathtt{op}}(\varphi) \tag{2.7}$$

$$\frac{I\{\underline{x} \mapsto \underline{\sigma}\} \models \alpha}{I \models \textbf{choose*}\ \underline{x}\ \textbf{with}\ \varphi\ \textbf{in}\ \alpha\ \textbf{ifnone}\ \beta} \quad \textit{iff} \quad \begin{array}{l} I(0) \models \forall\ \underline{x}.\ \delta(\varphi) \\ \textit{and } I\{\underline{x} \mapsto \underline{\sigma}\}(0) \models \varphi \end{array} \tag{2.8}$$

$$\frac{I \models \beta}{I \models \textbf{choose*}\ \underline{x}\ \textbf{with}\ \varphi\ \textbf{in}\ \alpha\ \textbf{ifnone}\ \beta} \quad \textit{iff} \quad \begin{array}{l} I(0) \models \forall\ \underline{x}.\ \delta(\varphi) \\ \textit{and } I(0) \models \forall\ \underline{x}.\ \neg\ \varphi \end{array} \tag{2.9}$$

$$\frac{I \models \textbf{throw}\ \mathtt{op}}{I \models \textbf{while*}\ \varepsilon\ \textbf{do}\ \alpha} \quad \textit{iff} \quad I(0) \models \omega^{\mathtt{op}}(\varepsilon) \tag{2.10}$$

$$\frac{}{I \models \textbf{while*}\ \varepsilon\ \textbf{do}\ \alpha} \quad \textit{iff} \quad \begin{array}{l} \#I = 0,\ I(0) \models \delta(\varepsilon),\ \zeta(I) = \top, \\ \textit{and } I(0) \not\models \varepsilon \end{array} \tag{2.11}$$

$$\frac{I \models \alpha; \textbf{while*}\ \varepsilon\ \textbf{do}\ \alpha}{I \models \textbf{while*}\ \varepsilon\ \textbf{do}\ \alpha} \quad \textit{iff} \quad I(0) \models \delta(\varepsilon)\ \textit{and}\ I(0) \models \varepsilon \tag{2.12}$$

$$\frac{I \models \{\mathbf{let*}\ \underline{x} = \underline{t}\ \mathbf{in}\ \alpha\{\underline{y} \mapsto \underline{u}\}\}; \underline{v} := \underline{z}}{I \models \mathbf{proc}\#(\underline{t}; \underline{u}; \underline{v})} \quad \textit{for} \quad \begin{array}{l}\textit{procedure declaration}\\ \mathbf{proc}\#(\underline{x}; \underline{y}; \underline{z})\{\alpha\}\end{array} \tag{2.13}$$

$$\frac{}{I \models \mathbf{throw}\ \mathsf{op}} \quad \textit{iff} \quad \begin{array}{l}\#I = 1,\ I(0)_b = \mathbf{ff},\ I'(0) = I(0),\\ \textit{and}\ \zeta(I) = \mathsf{op}\end{array} \tag{2.14}$$

$$\frac{I|_{[1..]} \models \mathbf{atomic}\ \varepsilon\ \{\alpha\}}{I \models \mathbf{atomic}\ \varepsilon\ \{\alpha\}} \quad \textit{iff} \quad I(0) \not\models \varepsilon\ \textit{and}\ I(0)_b = \mathbf{tt} \tag{2.15}$$

$$\frac{I_0 \models \alpha}{I \models \mathbf{atomic}\ \varepsilon\ \{\alpha\}} \quad \textit{iff} \quad \begin{array}{l}I(0) \models \varepsilon,\ I(0)_b = \mathbf{ff},\ \#I = 1,\ I(0) = I_0(0),\\ I'(0) = I_0'(\#I_0 - 1),\ \zeta(I) = \zeta(I_0) \neq \infty,\\ \textit{and}\ I_0\ \textit{with empty environment}\end{array} \tag{2.16}$$

$$\frac{I_0 \models \alpha \quad I \models \alpha}{I \models \mathbf{atomic}\ \varepsilon\ \{\alpha\}} \quad \textit{iff} \quad \begin{array}{l}I(0) \models \varepsilon,\ I(0) = I_0(0),\ \zeta(I_0) = \infty,\\ \textit{and}\ I_0\ \textit{with empty environment}\end{array} \tag{2.17}$$

$$\frac{I \models \mathbf{throw}\ \mathsf{op}}{I \models \mathbf{forall}\| \ \underline{x}\ \mathbf{with}\ \varphi\ \mathbf{do}\ \alpha} \quad \textit{iff} \quad I(0) \models \exists\ \underline{x}.\ \omega^{\mathsf{op}}(\varphi) \tag{2.18}$$

$$\frac{\bigwedge_{\underline{a} \in \Theta} I_{\underline{a}} \models \mathbf{let*}\ \underline{x} = \underline{a}\ \mathbf{in}\ \alpha}{I \models \mathbf{forall}\| \ \underline{x}\ \mathbf{with}\ \varphi\ \mathbf{do}\ \alpha} \quad \textit{iff} \quad \begin{array}{l}I(0) \models \forall\ \underline{x}.\ \delta(\varphi),\\ \Theta = \{\underline{a} \mid I(0)\{\underline{x} \mapsto \underline{a}\} \models \varphi\},\\ \textit{and}\ I \in \coprod_{\underline{a} \in \Theta} I_{\underline{a}}\end{array} \tag{2.19}$$

The semantics of programs are non-atomic, i.e., each update or condition test is executed in a separate step. $I \models \alpha$ means that the interval I is a possible execution of the program α. $I \models \alpha$ is derived by a recursive derivation system using the *greatest fixed point* in order to derive infinite intervals. These are necessary to capture non-terminating runs of while-loops or recursive calls.

Intervals complying to assignments must be of length 1, so all updates happen within one step. A parallel assignment evaluates all terms $\underline{t}$ in the first state $I(0)$ of the interval and overwrites the values of $\underline{x}$ in the state $I'(0)$ with the resulting values (rule 2.2). However, this rule only applies if the evaluation of $\underline{t}$ is defined $I(0)$, i.e., all partial operations are applied to arguments within their respective domains. Therefore, $I(0) \models \delta(\underline{t})$ must hold for the applicability of rule 2.2, where the *definedness condition* $\delta(e)$ describes that an expression e is defined (the definedness condition for a vector of expressions $\underline{e} = (e_1, \ldots, e_n)$ is given by the conjunction of all conditions $\delta(e_i)$). For example, the term $ar\texttt{[}n, a\texttt{]}\,\texttt{[}m\texttt{]}$, that updates a location n and then reads a location m of an array ar (see Sec. 2.2), produces the definedness condition

$$\delta(ar\texttt{[}n, a\texttt{]}\,\texttt{[}m\texttt{]}) \ \equiv\ n < \#ar \ \wedge\ m < \#ar\texttt{[}n, a\texttt{]}$$

Conversely, rule 2.1 applies if the evaluation of $\underline{t}$ yields an exception. Then the *exception condition* $\omega^{\mathsf{op}}(\underline{t})$ of some operation op must hold in $I(0)$. The exception conditions for all partial operations are calculated following a bottom-up approach: the exception is thrown if and only if an application of `op` violates the domain of `op`, and the evaluation of its arguments does not throw an exception. For the term above, the exception conditions of the two array operations are

$$\omega^{\texttt{[ ]}:\mathsf{array}('a)\times\mathsf{nat}\times'a\to\mathsf{array}('a)}(ar\texttt{[}n, a\texttt{]}\,\texttt{[}m\texttt{]}) \;\equiv\; \#ar \leq n$$
$$\omega^{\texttt{[ ]}:\mathsf{array}('a)\times\mathsf{nat}\to'a}(ar\texttt{[}n, a\texttt{]}\,\texttt{[}m\texttt{]}) \;\equiv\; n < \#ar \;\wedge\; \#ar\texttt{[}n, a\texttt{]} \leq \#m$$

For all other operations, the exception condition is `false`. The exception conditions of vectors $\underline{e}$ are calculated using the standard left-to-right strategy for evaluating arguments and a shortcut semantics for boolean connectives (like Java and Scala). So an e_j may only throw an exception if $\delta(e_i)$ holds for all $0 < i < j$. This guarantees the definedness of expressions like $x \neq \texttt{[]} \wedge x\texttt{.head} = \ldots$, which are often used as conditions in programs, while switching the conjunction would throw the `.head` exception when $x = \texttt{[]}$. Another typical example would be the test $y \neq 0 \wedge x/y = 1$, where the order in the conjunction avoids throwing the division exception for $y = 0$.

Rule 2.1 reduces the interval of an assignment with raised exception to an interval of the **throw** program (rule 2.14). Throwing an `op` takes one step but does not alter the state, but the corresponding interval must signal the occurrence of the exception ($\zeta(I) = \texttt{op}$).

Conditionals, choices, and iterations evaluate a condition at the beginning of its first step. These evaluations can also raise exceptions similar to evaluating terms in assignments. Hence, there are also specific rules for this case (rules 2.4, 2.7, and 2.10). Assuming that the nondeterministic choice construct selects a valuation for $\underline{x}$ satisfying φ by trying all possible values, rule 2.7 is applicable if there is some valuation for which an $\omega^{\texttt{op}}(\varphi)$ is satisfied (as **choose** could try this valuation before finding a suitable valuation). Accordingly, rules 2.8 and 2.9 are only applicable if φ is defined for all possible valuations of $\underline{x}$.

Note that the condition evaluation of conditionals, choices, and iterations does not take an extra step if they are defined: the conditions are evaluated in $I(0)$, and the complete intervals I must correlate to α or β, respectively. Rule 2.8 requires a sequence of values $\underline{\sigma} = (\underline{a}_0, \underline{a}_1, \underline{a}_2, \ldots)$ with the length $2 \cdot \#I + 1$, where the length and types of each $\underline{a}_i$ must correspond to $\underline{x}$, and φ must be valid for $I(0)\{\underline{x} \mapsto \underline{a}_0\}$. $\underline{\sigma}$ fixes the values of $\underline{x}$ for each step of the execution of α: $I\{\underline{x} \mapsto \underline{\sigma}\}$ denotes the substitution of $\underline{x}$ in each state of I with the corresponding values of $\underline{\sigma}$.

If the condition ε of a while-loop holds in the first state of the interval, the loop is unfolded once: α is executed first, and then (if α terminates) the loop is executed again (rule 2.12). As soon as ε does not hold anymore, the loop is exited without a step (rule 2.11). So effectively, $I = I_0 \fatsemi I_1 \fatsemi \ldots$ is a concatenation of intervals $I_i \models \alpha$ with $I_i.\mathit{first} \models \varepsilon$, and either $\#I = \infty$ or $I.\mathit{last} \models \neg\, \varepsilon$.

A procedure call unfolds the procedure declaration, and thus the semantics of a call reduces to the semantics of the procedure body α. The input arguments $\underline{t}$ are bound locally to the formal input parameters $\underline{x}$ so that updates to $\underline{x}$ in the body do not have any side effects on $\underline{t}$. The reference parameters $\underline{y}$ are substituted with the arguments $\underline{u}$ in α, so updates to them are directly visible to the environment (and vice versa, updates by the environment to $\underline{u}$ are also visible to the system). An assignment sets the values of the output arguments $\underline{v}$ after the body has been executed entirely. Thus, if α does not terminate, $\underline{v}$ is never set. Note also that the assignment of the outputs takes an extra step while the binding of the inputs does not.

There are two main rules for the **atomic** construct: the program is blocked (and thus, $I'(0) = I(0)$) if the guard ε does not hold in $I(0)$ (rule 2.15), or the guard holds and α is executed in one step, i.e., $\#I = 1$ (rule 2.16). For the latter case, there must be an interval $I_0 \models \alpha$ whose first and last states match $I(0)$ and $I'(0)$, respectively. I_0 has an *empty environment*: all environment steps do not modify the state, i.e., $I_0(k+1) = I'_0(k)$. While I_0 and I may end with an exception, **atomic** programs are not designed for non-terminating programs α, and hence, rule 2.16 does not apply to infinite intervals. Instead, the construct is considered misused if α has a non-terminating run in a initial state satisfying ε. In this case, rule 2.17 allows arbitrary runs of α to prevent arbitrary behavior for non-terminating programs. It is worth mentioning that the guards of atomic blocks are not supposed to throw exceptions as the construct is used for specification purposes only. Thus, there is no associated executable program construct, and as a result, ε is never actually evaluated.

The semantics of **forall** $\|$ reduces to weak-fair interleaving of intervals (cf. [110]). For all possible value vectors $\underline{a}$, an interval $I_{\underline{a}}$ is part of the interleaving: $I_{\underline{a}}$ is a possible execution of α with $\underline{x}$ bound to $\underline{a}$ in the initial state. Similar to **choose**, the rule 2.19 is only applicable if φ is defined for all possible values of $\underline{x}$. Otherwise, an exception is thrown (rule 2.18). An interleaved interval ends with an exception if one $I_{\underline{a}}$ does. Unlike in programming languages where exceptions are thread-local, exceptions are therefore *global* and abort the whole interleaved program. If necessary, the global effect can be avoided by using exception handlers in the interleaved programs. However, in this thesis, exception handlers are not relevant since the programs are designed not to throw any exception, and thus, the absence of exceptions is proved for all programs.

2.4 Weakest Precondition Calculus

The proof system underlying this thesis is a sequent-based calculus [49, 50]. A sequent $\Gamma \vdash \Delta$ abbreviates the formula $\forall \underline{x}.\ \bigwedge \Gamma \to \bigvee \Delta$ where Γ (the antecedent) and Δ (the succedent) are lists of formulas and $\underline{x}$ is the list of all free variables in Δ and Γ. Intuitively, $\Gamma \vdash \Delta$ states that assuming all formulas of Γ, one of the formulas of Δ must hold. The rules of the calculus follow the structure of the formulas and are applied backward to reduce the conclusion to simpler premises until they can be closed using axioms. As an example, consider the following rules for disjunctions: when assuming a disjunction of φ and ψ, the proposition has to be proven for the cases that either φ or ψ holds; to prove that a disjunction of φ and ψ holds, one has to prove either φ or ψ.

$$\frac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta} \qquad\qquad \frac{\Gamma \vdash \varphi, \psi, \Delta}{\Gamma \vdash \varphi \vee \psi, \Delta}$$

Reasoning about sequential programs is done with a weakest-precondition calculus [33], borrowing notation from Dynamic Logic (DL) [58, 56] including its two standard modalities: the formula $[\alpha]\varphi$ (*box*) denotes that for every terminating run the final state must satisfy φ, corresponding to the weakest liberal precondition

$wlp(\alpha, \varphi)$; the formula $\langle\alpha\rangle\varphi$ (*diamond*) guarantees that there is a terminating execution of α that establishes φ. Finally, the formula $\langle\!|\alpha|\!\rangle\, \varphi$ (*strong diamond*) states that the program α is guaranteed to terminate and that all final states reached satisfy φ, which corresponds to the weakest precondition $wp(\alpha, \varphi)$. For deterministic programs the two formulas $\langle\alpha\rangle\varphi$ and $\langle\!|\alpha|\!\rangle\, \varphi$ are equivalent. As a sequent, partial and total correctness of α with respect to pre- and postconditions *pre* and *post* are written as $pre \vdash [\alpha]\ post$ and $pre \vdash \langle\!|\alpha|\!\rangle\, post$.

To handle exceptions the modalities are extended by *exception specifications*

$$\underline{\xi} \equiv \texttt{op}_1 :: \varphi_1,\ \dots\ ,\ \texttt{op}_k :: \varphi_k,\ \textbf{default} :: \varphi_{default}$$

which yields program formulas of the form $[\alpha](\varphi\ ;\ \underline{\xi})$, $\langle\alpha\rangle(\varphi\ ;\ \underline{\xi})$, and $\langle\!|\alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi})$, respectively. The exception specifications allow to give additional postconditions (called *exception postconditions*) for executions that terminate with a specific exception, e.g., φ_1 must hold if α terminates with exception $\texttt{op}_1$. Additionally, an obligatory generic exception postcondition $\varphi_{default}$ handles executions of α that terminate with an exception $\texttt{op} \notin \texttt{op}_1, \dots, \texttt{op}_k$. For exceptions specifications $\underline{\xi}$, the exception postcondition for an $\texttt{op}$ is denoted $\underline{\xi}(\texttt{op})$, yielding either φ_ξ of a matching $\texttt{op} :: \varphi_\xi \in \underline{\xi}$ or $\varphi_{default}$. If one wants to show the absence of exceptions, the exception specifications $\underline{\xi} \equiv \textbf{default} :: \texttt{false}$ can be chosen, which is the default and is typically omitted from program formulas.

Given the interval semantics of programs in the previous section, the semantics of sequential program formulas can be stated as follows.

$$\begin{array}{lll}
v \models [\alpha]\ (\varphi\ ;\ \underline{\xi}) & \textit{iff} & \forall\ I_0.\ I_0\ \textit{has empty environment},\ I_0(0) = v,\ \zeta(I_0) \neq \infty, \\
 & & \qquad I_0 \models \alpha\ \textit{implies} \quad I_0.\textit{last} \models \varphi\ \textit{iff}\ \zeta(I_0) = \top \\
 & & \qquad\qquad\qquad \textit{or}\ I_0.\textit{last} \models \underline{\xi}(\texttt{op})\ \textit{iff}\ \zeta(I_0) = \texttt{op} \\
v \models \langle\alpha\rangle\ (\varphi\ ;\ \underline{\xi}) & \textit{iff} & \exists\ I_0.\ I_0\ \textit{has empty environment},\ I_0(0) = v,\ \zeta(I_0) \neq \infty, \\
 & & \qquad I_0 \models \alpha,\ \textit{and} \quad I_0.\textit{last} \models \varphi\ \textit{iff}\ \zeta(I_0) = \top \\
 & & \qquad\qquad\qquad \textit{or}\ I_0.\textit{last} \models \underline{\xi}(\texttt{op})\ \textit{iff}\ \zeta(I_0) = \texttt{op} \\
v \models \langle\!|\alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}) & \textit{iff} & \forall\ I_0.\ I_0\ \textit{has empty environment},\ I_0(0) = v,\ I_0 \models \alpha \\
 & & \qquad \textit{implies} \quad \zeta(I_0) \neq \infty \\
 & & \qquad\qquad \textit{and} \quad I_0.\textit{last} \models \varphi\ \textit{iff}\ \zeta(I_0) = \top \\
 & & \qquad\qquad\qquad \textit{or}\ I_0.\textit{last} \models \underline{\xi}(\texttt{op})\ \textit{iff}\ \zeta(I_0) = \texttt{op}
\end{array}$$

Note that (sequential) program formulas are evaluated over a valuation v like the expressions defined in Sec. 2.1. In Chapter 7, concurrent program formulas will be introduced that are evaluated over intervals instead of single states. However, the semantics given here can simply be lifted to intervals I by using $I(0)$ as the valuation v.

The main proof technique used in this thesis for verifying program correctness is *symbolic execution.* Basically, a symbolic execution proof step executes the first statement of the program and calculates the strongest postconditions from the preconditions. When the symbolic execution of the program is completed, the proof of the postcondition is performed purely in predicate logic. Similar to rules for logical connectives like the ones shown for disjunctions above, the calculus contains rules for all program constructs given in Def. 5.

$$\frac{\Gamma^{\underline{x}'}_{\underline{x}},\ \delta(\underline{t}^{\underline{x}'}_{\underline{x}}),\ \underline{x} = \underline{t}^{\underline{x}'}_{\underline{x}} \vdash \langle\!|\alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta^{\underline{x}'}_{\underline{x}} \quad \textit{where } \underline{x}' \textit{ fresh} \qquad \mathit{Exc}(\Gamma, \Delta, \underline{t}, \varphi, \underline{\xi})}{\Gamma\ \vdash \langle\!|\underline{x} := \underline{t}; \alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Figure 2.1: Calculus rule for assignments.

Fig. 2.1 shows exemplarily the rule for parallel assignments for total correctness (the rule is identical for the other modalities). The rule uses a vector $\underline{x}'$ of fresh variables to store the values of $\underline{x}$ before the assignment. The assignment is removed and instead the formula $\underline{x} = \underline{t}^{\underline{x}'}_{\underline{x}}$ is added to the antecedent ($\underline{t}^{\underline{x}'}_{\underline{x}}$ denotes the renaming of $\underline{x}$ to $\underline{x}'$ in $\underline{t}$). Note that renaming is possible on all program formulas, while substitution of $\underline{x}$ by general terms $\underline{t}$ in $\langle\!|\alpha|\!\rangle\,\varphi$ is not possible and just yields $\langle\!|\underline{x} := \underline{t}; \alpha|\!\rangle\,\varphi$. Only when the assignment is the last statement of the program and α is missing, the standard premise of Hoare calculus, which replaces the program formula in the premise with $\varphi^{\underline{t}}_{\underline{x}}$, can be used.

Since the expressions $\underline{t}$ can contain partial operations, the evaluation of $\underline{t}$ could raise exceptions. Therefore, *exception premises* $\mathit{Exc}(\Gamma, \Delta, \underline{t}, \varphi, \underline{\xi})$ need to be shown for all potential violations of $\delta(\underline{t})$, i.e., additional premises for all partial operations $\mathsf{op}_1, \ldots, \mathsf{op}_m$ in $\underline{t}$ are added.

$$\begin{array}{l} \mathit{Exc}(\Gamma, \Delta, \underline{t}, \varphi, \underline{\xi}) \equiv \\ \qquad (1)\ \ \Gamma \vdash \omega^{\mathsf{op}_1}(\underline{t}) \rightarrow \langle\!|\textbf{throw}\ \mathsf{op}_1|\!\rangle\,(\varphi;\ \underline{\xi}),\ \Delta \\ \qquad\ \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \\ \qquad (\mathrm{m})\ \Gamma \vdash \omega^{\mathsf{op}_m}(\underline{t}) \rightarrow \langle\!|\textbf{throw}\ \mathsf{op}_m|\!\rangle\,(\varphi;\ \underline{\xi}),\ \Delta \end{array}$$

For example, the assignment $b := ar\,[n, a]\,[m]$ would yield the following two exception premises.

$$\begin{array}{l} \mathit{Exc}(\Gamma, \Delta, ar\,[n, a]\,[m], \varphi, \underline{\xi}) \equiv \\ \qquad (1)\ \Gamma \vdash \#ar \leq n \rightarrow \langle\!|\textbf{throw}\ [\,]|\!\rangle\,(\varphi;\ \underline{\xi}),\ \Delta \\ \qquad (2)\ \Gamma \vdash n < \#ar\ \wedge\ \#ar\,[n, a] \leq \#m \rightarrow \langle\!|\textbf{throw}\ [\,]|\!\rangle\,(\varphi;\ \underline{\xi}),\ \Delta \end{array}$$

Recall that the operation symbol $[\,]$ is overloaded for updating and retrieving a value in an array ($[\,] : \mathsf{array}('a) \times \mathsf{nat} \times 'a \rightarrow \mathsf{array}('a)$ vs. $[\,] : \mathsf{array}('a) \times \mathsf{nat} \rightarrow 'a$). According to this, premise (1) throws the former operation while premise (2) throws the latter.

The rules for **throw** are quite simple. If an operation is thrown for which a specific exception specification $\mathsf{op} :: \varphi_\xi$ is given in $\underline{\xi}$, the program formula is discarded and replaced by the exception postcondition φ_ξ (Fig. 2.2 on the left). If there is no exception specification in $\underline{\xi}$ for the thrown operation op, the default exception postcondition $\varphi_{\mathit{default}}$ must hold (Fig. 2.2 on the right).

The calculus rules for the remaining sequential program constructs function as expected by following the structure of the program, extended with exception premises where necessary. For example, the **if** rules produce premises for the **then**- and **else**-branches, as well as exception premises for evaluating the condition ε. Sequential

$$\frac{\Gamma \ \vdash \varphi_{\xi},\ \Delta}{\Gamma \ \vdash \langle\!|\mathbf{throw}\ \mathsf{op}|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta} \qquad\qquad \frac{\Gamma \ \vdash \varphi_{default},\ \Delta}{\Gamma \ \vdash \langle\!|\mathbf{throw}\ \mathsf{op}|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Figure 2.2: Calculus rules for **throw** with $\mathsf{op} :: \varphi \in \underline{\xi}$ (left) and $\mathsf{op} \notin \underline{\xi}$ (right).

$$\frac{\begin{array}{c}(1)\ \Gamma \vdash inv,\ \Delta \qquad (2)\ Exc(inv, \langle\rangle, \varepsilon, \varphi, \underline{\xi}) \qquad (3)\ inv,\ \neg\,\varepsilon,\ \delta(\varepsilon) \vdash \varphi \\ (4)\ inv,\ \varepsilon,\ \delta(\varepsilon),\ z = t \vdash \langle\!|\alpha|\!\rangle\,(inv \wedge t \ll z;\ \underline{\xi})\end{array}}{\Gamma \ \vdash \langle\!|\mathbf{while}\ \varepsilon\ \mathbf{do}\ \alpha|\!\rangle\,(\varphi\ ;\ \underline{\xi}),\ \Delta}$$

Figure 2.3: Invariant rule for while loops.

composition is resolved by exploiting that the postcondition of a program formula can again be a program formula, e.g., $\langle\!|\alpha|\!\rangle\,(\langle\!|\beta|\!\rangle\,\varphi)$.

Proofs about recursive procedures are typically done using (well-founded) induction. For **while**-loops, typically the invariant rule shown in Fig. 2.3 is used, though the more general induction rule occasionally leads to simpler proofs.

The rule requires an invariant formula *inv* as an input from the user from which multiple premises need to be proven: the invariant must hold at the beginning of the loop (1), *inv* must be stable over the loop body α (4), and *inv* must be strong enough to prove the postcondition φ after the loop was exited (3). Similar to the assignment rule, exception premises are generated for the loop condition ε (2). These cannot include Γ or Δ as ε is evaluated again after each iteration. Instead, the invariant *inv* can be assumed as well as $\delta(\varepsilon)$ for premises (3) and (4). The figure shows the rule for total correctness which requires to give a variant t that decreases with every iteration of the loop ($t \ll z$ in premise (4)). The rule functions analogously for diamond formulas, for box formulas (partial correctness) no decreasing variant is necessary.

When reasoning about sequential programs using the wp-calculus presented in this section, the asterisk (*) program constructs are interchangeable with their non-asterisk counterparts. For example, the rule given in Fig. 2.3 is identical for **while*** programs. However, the rules for the two versions of programs will differ when reasoning about programs in a concurrent context (see Chapter 7).

2.5 Heaps & Separation Logic

Most programs and components presented in this thesis work on algebraic representations of standard data structures. Correctness properties are proven over these algebraic data types, assuming that a straightforward translation to correct implementations is used when generating code from the models. For example, the algebraic lists presented in Sec. 2.2 can be realized as linked lists in C.

On the other hand, there are also more complex data structures whose correct implementation is non-trivial since they are primarily performance-oriented. These include, for example, red-black trees, for which a verified implementation is presented in Sec. 3.2. The algorithms necessary for accessing the data structures are typically destructive and involve explicit manipulation of heap locations and pointers. Thus, verifying programs is usually significantly more difficult when pointer structures are used instead of algebraic data types, as aspects such as memory management or aliasing have to be considered.

Separation Logic (SL) [102] targets the verification of such pointer-based programs. It is best used when there is little aliasing as its basic idea is to separate the heap into disjoint parts. When a data structure is updated, SL facilitates local reasoning about the modified part of the heap while the remaining (unaltered) parts of the heap can be neglected. The central operator used in SL is the *separating conjunction*: the formula $\varphi * \psi$ denotes that the heap can be split into two disjoint parts, i.e., two parts with disjoint domains of references, where one part satisfies the property φ, and the other part satisfies ψ. As a core rule, the *frame rule* enables local reasoning by focusing on the parts of the heap accessed by a program α (here given in the original notation following the notation of Hoare triples).

$$\frac{\{\varphi\}\ \alpha\ \{\psi\}}{\{\varphi * \chi\}\ \alpha\ \{\psi * \chi\}}$$

Most verification tools supporting Separation Logic, like VeriFast [68] or VerCors [12], integrate the heap as part of the semantics of SL formulas. However, in this thesis, heaps are specified as an algebraic data type, based on which SL formulas are axiomatized. This explicit specification is necessary since the used logic requires all parameters of procedures to be explicit. Hence, when reasoning about pointer-based programs, the heap must be an explicit parameter of the program as well.

Heaps are defined as polymorphic non-free data type $\mathsf{heap}('a)$. Similar to maps (cf. Sec. 2.2), a heap can be considered as a partial function mapping references of type ref to objects of a generic type $'a$. But unlike for maps, the allocation of references is explicit, and the reference type contains a distinguished element $\mathtt{null}$ that is never allocated (representing the null pointer). The $\mathsf{heap}('a)$ data type is generated by the constant $\emptyset$ representing the empty heap, by allocating an new reference with $. \mathbin{{+}{+}} . : \mathsf{heap}('a) \times \mathsf{ref} \to \mathsf{heap}('a)$ (written $h \mathbin{{+}{+}} r$), or by updating an allocated location to store an object with $. \mathtt{[}\ . \mathtt{]} : \mathsf{heap}('a) \times \mathsf{ref} \times {'a} \to \mathsf{heap}('a)$ (written $h\mathtt{[}r, \mathit{obj}\mathtt{]}$). A predicate $. \in . : \mathsf{ref} \times \mathsf{heap}('a)$ is defined for checking whether a reference is allocated in a heap, and a function $. \mathtt{[}\ . \mathtt{]} : \mathsf{heap}('a) \times \mathsf{ref} \to {'a}$ is used for looking up objects in the heap (this corresponds to dereferencing a pointer). References can also be deallocated by the function $. \mathtt{--}\ . : \mathsf{heap}('a) \times \mathsf{ref} \to \mathsf{heap}('a)$.

The constructor functions as well as lookup and deallocation are declared as partial functions in order to specify valid accesses to the heap. Hence, the functions are augmented with respective domains: accesses to the heap with the $\mathtt{null}$ reference are always undefined ($r \neq \mathtt{null}$); allocation is only allowed for new references ($\neg\ r \in h$); lookups, updates, and deallocations require the references to be allocated ($r \in h$).

SL formulas are encoded using *heap predicates* $hP : \mathsf{heap}('a) \to \mathsf{bool}$. A heap predicate describes the structure of a heap h. At its simplest, h is the empty heap emp:

$$\vdash \mathsf{emp}(h) \leftrightarrow h = \emptyset$$

The *maplet* $r \mapsto \mathit{obj}$ describes a singleton heap, containing only one reference r mapping to an object obj. It is defined as a higher-order function of type $(\mathsf{ref} \times {'a}) \to$

$\mathsf{heap}('a) \to \mathsf{bool}$:

$$\vdash (r \mapsto obj)(h) \leftrightarrow h = (\emptyset \; \texttt{++} \; r)[r, obj] \land r \neq \texttt{null}$$

More complex heaps can be described using the *separating conjunction* $hP_0 \; \texttt{*} \; hP_1$ asserting that the heap consists of two disjoint parts, one satisfying hP_0 and one satisfying hP_1, respectively. Since it connects two heap predicates, it is defined as a function with type $(\mathsf{heap}('a) \to \mathsf{bool}) \times (\mathsf{heap}('a) \to \mathsf{bool}) \to (\mathsf{heap}('a) \to \mathsf{bool})$:

$$\vdash (hP_0 \; \texttt{*} \; hP_1)(h) \leftrightarrow \exists \; h_0, h_1.\; h_0 \perp h_1 \land h = h_0 \cup h_1 \land hP_0(h_0) \land hP_1(h_1)$$

Analogously, the separating implication (or *magic wand*) is defined:

$$\vdash (hP_0 \; \texttt{-*} \; hP_1)(h) \leftrightarrow \forall \; h_0.\; h_0 \perp h \land hP_0(h_0) \to hP_1(h_0 \cup h)$$

Common non-sharing pointer data structures like singly-/doubly-linked lists or binary trees can be specified using these basic SL definitions. With straightforward abstractions to their algebraic counterparts, the functional correctness (incl. memory safety) of algorithms using such pointer structures can be verified with significantly reduced effort.

Chapter 3

Modular Construction of Sequential Systems

Summary In order to facilitate the formal development of large-scale software systems, a concept for modularization is required. This chapter introduces such a concept in the form of state-based, hierarchical components. A hierarchy of components can be created by connecting them via subcomponent and refinement relationships. The semantics of components are defined in terms of intervals and histories, and a compositional variant of data refinement is given for proving the correctness of components by changing their internal data representation. The methodology is extended by a state-based approach for reasoning about crash-safety.
This chapter also presents a non-trivial application of the approach: an efficient, destructive implementation of red-black trees (a form of self-balancing binary search trees). Correctness of the implementation is shown by modularizing the implementation into multiple components and proving two data refinements, starting from an abstract specification of sets, over an algebraic representation of trees, to a heap-based pointer implementation. Due to this modularization, high-level red-black tree properties could be proven solely on algebraic trees, while complex reasoning about pointer structures using Separation Logic could be limited to fine-grained operations on the tree.

Contents

Publications Different aspects of the formalisms presented in this chapter were published in [97, 13, 15, 107]. These formalisms are largely based on the work of Ernst [38] and Pfähler [96]. The verified version of red-black trees described in this chapter is based on the submitted paper [106].

3.1 Hierarchical Components & Data Refinement

For the formal development of complex software systems, a concept for modularization is necessary. This section presents such a concept: state-based components similar to abstract data types [57, 32] are extended with specifications of the effect of power failures (*crashes*) and connected by data refinements and subcomponent relationships.

Hierarchical, Crash-Aware Components

Def. 7 gives the full-featured version of components, though not all features are relevant for every system, module, or library. The components presented in this thesis will incrementally use more and more parts of the definition: Sec. 3.2 describes a library for red-black trees, consisting of sequential, in-memory components which do not require specifying crash behavior; Chapter 4 shows some components of the upper layers of Flashix in their initial form, i.e., without any caches and purely sequential, but with actual crash behavior since the components are built upon persistent storage; in Chapter 6, caches are added to the Flashix component hierarchy, and thus, their effects on the crash behavior become relevant; finally, Chapter 7 introduces concurrent executions within the component hierarchy, resulting from executions of internal operations or concurrent calls to the file system interface.

Definition 7 (Components)**.** *A component* $\mathtt{C} = (\mathtt{St}, \mathtt{Init}, (\mathtt{Op}_j)_{j\in J}, (\mathtt{IOp}_k)_{k\in K}, \mathtt{Sync}$ $\mathtt{Crash}, \mathtt{Rec}, (\mathtt{C}_l)_{l\in L})$ *consists of a set of states* $\mathtt{St}$*, a set of initial states* $\mathtt{Init} \subseteq \mathtt{St}$*, interface operations* $\mathtt{Op}_j$ *with index set* J*, internal operations* $\mathtt{IOp}_k$ *with index set* K*, a set of synchronized states* $\mathtt{Sync}$ *with* $\mathtt{Init} \subseteq \mathtt{Sync} \subseteq \mathtt{St}$*, a crash relation* $\mathtt{Crash} \subseteq \mathtt{St} \times \mathtt{St}$ *with* $\mathtt{Sync} \subseteq dom(\mathtt{Crash})$*, a recover relation* $\mathtt{Rec} \subseteq \mathtt{St} \times \mathtt{Sync}$*, and a set of subcomponents* $\mathtt{C}_l$ *with index set* L*.*

The state $\mathtt{St}$ of a component is given in the form of a duplicate-free sequence $\underline{st}$ of typed *state variables.*

$$\textbf{state} \quad st_1 : \mathsf{ty}_1 \;\ldots\; st_n : \mathsf{ty}_n$$

The state is encapsulated, so clients of a component cannot access its state variables directly. Instead, state variables can only be read or updated via the operations of a component. Operations are specified with contracts using the operational approach of ASMs [16].

Definition 8 (Operations of Components)**.** *The operations of a component are given as extended procedure declarations. The declarations of interface operations* $\mathtt{Op}_j$ *and internal operations* $\mathtt{IOp}_k$ *over the state variables* $\underline{st}$ *have the form*

```
$\mathbf{op}_j\#(\underline{x}; \underline{y}, \underline{st}; \underline{z})$
  interface
  precondition $pre_j(\underline{x}, \underline{y}, \underline{st})$
$\{\ \alpha\ \}$

$\mathbf{iop}_k\#(; \underline{st})$
  internal
  guard $guard_k(\underline{st})$
$\{\ \alpha\ \}$
```

The state variables are treated as reference parameters in both types of declarations, interface operations may have additional input parameters $\underline{x}$*, reference parameters* $\underline{y}$*, and output parameters* $\underline{z}$*. Interface operations have a precondition given as a formula* pre_j *signalizing when it is allowed to call the operation. Similarly, internal operations have a guard formula* $guard_k$ *determining when the execution of the operation is triggered.*

Recall that the bodies α of the operation declarations in Def. 8 are only allowed to use the variables of their parameters and local variables. Thus, internal operations work only on the component's state and do not accept any inputs or produce outputs. Furthermore, they cannot be called explicitly; instead, an internal operation is assumed to be triggered randomly whenever the state satisfies its guards. Internal operations are typically used to perform administrative tasks, which need to be performed regularly to keep the system usable but should not need to be explicitly triggered by the user. For example, a flash file system requires performing *wear leveling* to extend the lifetime of the flash memory and *garbage collection* to reclaim storage space filled with outdated data. Both precondition and guard clauses are omitted if there are no restrictions, i.e., $pre_j = \mathtt{true}$ or $guard_k = \mathtt{true}$, respectively.

To enable code reuse, parts of the body of interface or internal operations may be outsourced to *auxiliary operations* (they will be tagged with the **auxiliary** keyword in this work). These operations are not part of the component's interface but can only be called from other operations of the component. Hence, they are not listed in Def. 8, but they are often used when modeling components in practice.

Instead of defining the set of initial states `Init` directly, a *initialization procedure* is used. The procedure must be called before any other operation of the component is called and must not interleave with any other operation execution.

Definition 9 (Initialization of Components)**.** *The initial states* `Init` *of a component are produced by an initialization operation, given as procedure declaration.*

```
$\mathbf{init\#}(\underline{x};\,;\underline{st},\underline{z})$
  $\mathbf{initialization\ if}\ \varphi(\underline{st},\underline{z})$
$\{\ \alpha\ \}$
```

An initialization procedure may have input parameters $\underline{x}$ determining the initial state and output parameters $\underline{z}$ that indicate whether initialization was successful or has failed. A restriction condition φ specifies when an execution of the initialization operation has been successful. `Init` *consists of all possible final values of $\underline{st}$ of successful* **init#** *executions.*

Note that, compared to the declarations of Def. 8, the state variables $\underline{st}$ are not added as reference parameters but as output parameters to **init#**. Consequently, α cannot access any old values of $\underline{st}$ unless it has initialized them before.

Although initialization operations often do not require additional arguments, it is useful to parameterize the initialization for some systems. For example, when initializing a file system, i.e., when *formatting* a partition, one might want to define the size of pages used in the Virtual File System Switch. This information is then used to create the core administration data structures and kept for the whole run of the system. In order to take into account that the initialization may also fail, output parameters can be used to signalize such failures to the user. This can be when invalid initialization parameters were given, e.g., a page size of 0, or when hardware errors occur, e.g., the underlying storage refuses to write a page. Formally, successful executions are specified by the *restriction condition* φ based on the output or the produced component state. From a practical point of view, the system user has to retry initializing the system before using it if φ is false after the first attempt. In the context of verification, invariant and refinement properties need only to be shown if φ is true, as we will see in the corresponding proof obligations later in this section.

During or in between the execution of operations, power cuts can occur at any moment. Such crashes have an impact on components: running operations are aborted instantly and state stored in volatile memory is lost. `Sync` and `Crash` specify the effects on the state of a component; syntactically, they are given in the form of formulas over the state variables $\underline{st}$ and the back-primed state variables $\underline{st}^{\backprime}$.

synchronized predicate $\mathit{synced}(\underline{st})$
crash predicate $\mathit{crash}(\underline{st}^{\backprime}, \underline{st})$

The *synchronized predicate* characterizes *synchronized states* in which crashes have (nearly) no effects on the system's state, in the sense that almost all relevant data can be reconstructed from persistent storage. In an unsynchronized state, a crash has intuitively the effect of jumping back to the last synchronized state by reverting whole operations. This crash behavior is usually present when an underlying buffer cache is used, as we will see in Chapter 6. Without such caches, `true` is typically used as synchronized predicate, stating that crashes cannot retract completed operations. In this case, the explicit declaration of the predicate is omitted.

Definition 10 (Retracting Components)**.** *A component* `C` *is retracting if its synchronized states are a proper subset of its state space, i.e.,* $\texttt{Sync} \subset \texttt{St}$*. Otherwise, when* $\texttt{Sync} = \texttt{St}$*,* `C` *is non-retracting.*

The *crash predicate* specifies residual effects of a crash, i.e., the effects a crash has on synchronized states. The predicate describes a relation between a state before the crash, denoted by $\underline{st}^{\backprime}$, and a potential crashed state $\underline{st}$. Note that the crash predicate can be partial if the synchronized predicate is not `true`. As a result, states in which the effects of a crash are hard to express can be left aside when reasoning about correct crash behavior.

On the other hand, `Crash` also determines whether parts of `St` are kept in volatile memory (these parts are called *RAM state* in the following). Intuitively, the values of volatile state variables are lost during a crash, so `Crash` must not restrict the volatile parts of `St`.

Definition 11 (RAM State and RAM Components)**.** *A state variable* st_i *of a component* `C` *is part of the RAM state of* `C` *iff*

$$\forall\ \underline{st}, \underline{st}'.\ \mathit{crash}(\underline{st}, \underline{st}') \to \forall\ st_i''.\ \mathit{crash}(\underline{st}, st_1', \ldots, st_{i-1}', st_i'', st_{i+1}', \ldots, st_n')$$

`C` *is a RAM component iff all its state variables* $\underline{st}$ *are part of its RAM state.*

It follows directly from the definition that the crash predicate of a *RAM component* is equivalent to `true`. Again, we use this as the default case and hence omit the declaration of the crash predicate for RAM components.

Power failures can result in inconsistent states as operations could be interrupted in the middle of their executions. A component can attempt to restore a consistent state after a crash with a recovery `Rec` to handle these situations. Furthermore, the recovery is responsible for re-initializing the RAM state of the component. Similar to initialization, a *recovery procedure* is used to define the recovery mechanism.

Definition 12 (Recovery of Components)**.** *The recovery relation* $\mathtt{Rec}$ *of a component is defined by the possible executions of a recovery operation, given as procedure declaration.*

```
recover#($\underline{x}$; $\underline{st}$; $\underline{z}$)
  recovery if $\varphi(\underline{st}, \underline{z})$
{ $\alpha$ }
```

A recovery procedure may have input parameters $\underline{x}$ *specifying runtime options and output parameters* $\underline{z}$ *indicating whether recovery was successful or has failed. A restriction condition* φ *specifies when an execution of the recovery operation has been successful.* $\mathtt{Rec}$ *consists of all possible pairs of initial an final values of* $\underline{st}$ *of successful* **recover#** *executions.*

An alternative to the relational approach of $\mathtt{Rec}$ would be to have recovery as a distinguished operation of the component (similarly, this could be done for $\mathtt{Init}$). However, recovery is only executed in a sequential context, so there is no need to consider executions of recovery in terms of intervals as we do for regular operations of components (see Def. 24 and following later this section).

The declarations of recovery and initialization procedures are alike in many ways. As recovery can also fail, e.g., due to hardware errors, output parameters can be used to signal so, and a restriction condition is used to specify successful executions. However, unlike initialization procedures, the state is passed as a reference parameter since recovery typically tries to keep as much data from the crashed state as possible. In the context of file systems, the recovery operation corresponds to *mounting* a initialized file system. This file system could have been unmounted unsafely before, potentially due to a power failure, and so the mounting mechanism tries to reconstruct the version of the file system just before the crash, for example, by replaying a log. The input parameters of recovery operations usually correspond to runtime options of the system, e.g., whether a file system should be operated in synchronous or asynchronous mode.

A component $\mathtt{C}$ can use one or more components $\mathtt{C}_l$ as *subcomponents*, written $\mathtt{C}(\mathtt{C}_l)$. $\mathtt{C}$ is then called a *client* of $\mathtt{C}_l$. Visually, such a relation is depicted borrowing notation from UML component diagrams [104, 114], as shown in Fig. 3.1: the *socket* (—( ) depicts a required interface of component $\mathtt{C}$ and the *lollipop* (○—) depicts the corresponding provided interface of $\mathtt{C}_l$. A subcomponent can then again be a client of other components, which can also have subcomponents, and so on. However, a component cannot be used by multiple clients simultaneously, and no cyclic dependencies are allowed either, so the resulting structure is of a tree-like shape. Note furthermore that a component structure is static, i.e., it is established before the initialization of a system and cannot be altered dynamically during the system's runtime, which is a major distinction to object structures in object-oriented programming.

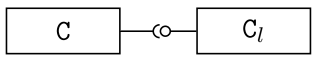


Figure 3.1: Component $\mathtt{C}$ with a subcomponent $\mathtt{C}_l$.

In order to distinguish between elements of multiple (potentially connected) components, we mark parts of a component $\mathtt{C}$ with a superscript $\_^{\mathtt{C}}$, e.g., the state variables of $\mathtt{C}$ are denoted by $\underline{st}^{\mathtt{C}}$.

The combined state space of a hierarchy of components is built using the Cartesian product. The corresponding combined state variable sequence must be duplicate-free to be well-defined, so it is defined using the disjoint union operator.

Definition 13 (Combined State of Components)**.** *The combined state space* $\mathtt{S}^{\mathtt{C}}$ *and the combined set of initial states* $\mathtt{INIT}^{\mathtt{C}}$ *of a component* $\mathtt{C}$ *with subcomponents* $(\mathtt{C}_l)_{l\in L}$ *is determined by the Cartesian product of the respective state set of* $\mathtt{C}$ *with the ones of its subcomponents.*

$$\mathtt{S}^{\mathtt{C}} \equiv \mathtt{St}^{\mathtt{C}} \times \bigtimes_{l\in L} \mathtt{S}^{\mathtt{C}_l} \qquad\qquad \mathtt{INIT}^{\mathtt{C}} \equiv \mathtt{Init}^{\mathtt{C}} \times \bigtimes_{l\in L} \mathtt{INIT}^{\mathtt{C}_l}$$

The combined state variable sequence $\underline{s}^{\mathtt{C}}$ *is given by the disjoint union of the state variables of* $\mathtt{C}$ *with the combined state variables of its subcomponents.*

$$\underline{s}^{\mathtt{C}} \equiv \underline{st}^{\mathtt{C}} \uplus \biguplus_{l\in L} \underline{s}^{\mathtt{C}_l}$$

Using subcomponents allows $\mathtt{C}$ to outsource the realization of partial tasks to one of its subcomponents $\mathtt{C}_l$. The access to subcomponents is restricted, however. $\mathtt{C}$ is not allowed to access (read or update) the state of $\mathtt{C}_l$ directly. Instead, $\mathtt{C}$ can only interact with $\mathtt{C}_l$ by calling its interface operation. This complies with a strict application of the *information hiding* resp. *encapsulation* principle, which is what makes incremental refinement of subcomponents possible in the first place (see Thm. 2 later in this section). When successful, the initialization operation of $\mathtt{C}$ must produce a combined state that is contained in $\mathtt{INIT}^{\mathtt{C}}$. Therefore, the initialization operation must call all its subcomponents' respective initialization operations, and similarly, recovery must call all subcomponent recoveries to establish a combined recovered state. Besides that, there are some more properties a component must satisfy to integrate it correctly in a refinement hierarchy.

Definition 14 (Modularizable Components)**.** *A component* $\mathtt{C}$ *is modularizable if it uses subcomponents* $(\mathtt{C}_l)_{l\in L}$ *such that*

1. $\mathtt{C}$ *does not read or write to the state variables* $\underline{s}^{\mathtt{C}_l}$ *of any of its subcomponents* $\mathtt{C}_l$*,*
2. *the operations of* $\mathtt{C}$ *only call interface operations of its direct subcomponents* $\mathtt{C}_l$*,*
3. *the initialization and recovery operations of* $\mathtt{C}$ *call the respective operations of each subcomponent* $\mathtt{C}_l$*, and they do so before any other operation of* $\mathtt{C}_l$ *is called, and*
4. *each subcomponent* $\mathtt{C}_l$ *is modularizable.*

Note that interface operations are only allowed to be called externally by clients of $\mathtt{C}$. This limitation is relevant for defining the semantics of components later in this section, as calls to interface operations within a component should not be part of the observable behavior of a component. However, this restriction can be easily overcome by moving the respective functionality of an interface operation to an auxiliary operation that multiple interface operations can call.

When crash behavior comes into play, i.e., when there are non-RAM components, components must account for additional constraints. In particular, retracting components cannot be integrated arbitrarily into the hierarchy.

Definition 15 (Crash-Modularizable Components)**.** *A component* C *is crash-modularizable if it is modularizable according to Def. 14 and uses subcomponents* $(\mathtt{C}_l)_{l \in L}$ *such that*

1. *successful executions of the initialization and recovery operations of* C *must produce synchronized states,*

2. *at most, one component of the hierarchy starting at* C *may be retracting according to Def. 10, and if there is one, it must be a leaf of the hierarchy,*

3. *if there is a retracting component, all other components of the hierarchy must be RAM-components according to Def. 11, and*

4. *if there is no retracting component, the crash predicate of all components must be total.*

Components are distinguished between specifications and implementations. The former are used to model the functional requirements of a (sub-)system and are typically kept as simple as possible by heavily utilizing algebraic functions and non-determinism. The approach is as general as specifying with pre- and postconditions, since **choose** st', out' **with** $post(st', out')$ **in** $st, out := st', out'$ can be used to establish any postcondition $post$ over state st and output out.

On the other hand, implementations are typically deterministic and only use constructs that allow generating executable Scala- or C-code from them with our code generator.

Definition 16 (Implementation Programs)**.** *A program is a valid implementation iff it only uses the following subset of program constructs.*

$$x := t \qquad \alpha; \beta \qquad \textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta \qquad \textbf{let } \underline{x} = \underline{t} \textbf{ in } \alpha$$
$$\textbf{while } \varepsilon \textbf{ do } \alpha \qquad \textbf{proc}\#(\underline{t}; \underline{u}; \underline{v})$$

Basically, implementation programs are built from sequential and deterministic program constructs only. Parallel assignments are limited to executing just one assignment at a time since a semantics-preserving sequentialization of parallel assignments might not be possible in some cases, particularly in a concurrent setting. For example, swapping values of state variables would require multiple assignments and the use of additional local variables in real code, which could invalidate correctness properties proven for an atomic swap. So multiple assignments have to be combined via sequential composition. Only the non-asterisk versions of conditionals and iterations are allowed, and the non-deterministic choice construct is restricted to the deterministic introduction of local variables with the **let** abbreviation. [1] Note that, contrary to assignments, multiple local variables can be introduced in **let** because the terms $\underline{t}$ cannot refer to variables of $\underline{x}$. Procedure calls can be used as before.

The **atomic** is not used in implementations but is only a specification instrument, as we will see in Chapter 7. Within the scope of this thesis, implementations will

[1] There is one exception: we use the **choose** construct in implementations of heap-based components to model the allocation of new heap locations by choosing an arbitrary unused reference. It is then up to the code generator to translate these occurrences to according statements, e.g., *malloc* in C.

also not use the **forall** $\|$ construct for spawning new threads dynamically. Instead, each internal operation of a component is executed in a dedicated thread that is active for the complete runtime of the system. Explicit **throw**s are not used as well since the concepts presented here are tailored for systems typically implemented in programming languages without native exceptions, for example, C.

Besides using a suitable subset of program constructs, *implementation components* must use RAM state only.

Definition 17 (Implementation Components)**.** *A component* $\mathtt{C}$ *is an implementation component iff it is a non-retracting RAM component and its operations only use implementation program constructs as defined by Def. 16.*

If a component is not an implementation component, it is considered a *specification component.* Note that as a consequence of Def. 17, only specification components can have persistent state. In order to access persistent storage, like flash memory, an implementation therefore has to use a specification component that models the persistent storage as (part of) its state and has suitable interface operations for reading/manipulating the state. For example, on the bottom layer of the Flashix file system, a simple specification of the Linux MTD subsystem is used to access flash memory (see Sec. 4.1).

Semantics of Components

The behavior of components can be considered as runs of a *labeled transition system* or IO-Automata [81] where the labels correspond to *invocation* and *response* events of operations, together with a distinguish crash event. *Histories* describe the observable behavior of components by tracking the events of component runs, as is common for concurrent systems [60].

Definition 18 (Events & Histories of Components)**.** *The event set* $\mathcal{L}^{\mathtt{C}}$ *of a component* $\mathtt{C}$ *consists of*

- *invocation events* $inv^{\mathtt{C}}_{tid}(i, \underline{inp})$ *of the interface or internal operation* $i \in J \cup K$ *of* $\mathtt{C}$ *by thread tid with inputs* $\underline{inp}$*,*
- *response events* $res^{\mathtt{C}}_{tid}(i, \underline{out})$ *of the interface or internal operation* $i \in J \cup K$ *of* $\mathtt{C}$ *by thread tid with outputs* $\underline{out}$*, and*
- *a crash event* $\lightning$*.*

A history $h^{\mathtt{C}} = l_0 l_1 l_2 \ldots$ *of a component* $\mathtt{C}$ *is a sequence of labels* $l_i \in \mathcal{L}^{\mathtt{C}}$*. The history* $h^{\mathtt{C}}$ *is crash-free iff it does not contain a crash event, i.e.,* $\forall\ l_i. l_i \in h^{\mathtt{C}} \rightarrow l_i \neq \lightning$*.*

Invocation and response events are marked with the identifier $tid \in Tid$ of the thread calling the operation and record passed input values $\underline{inp}$ and returned output values $\underline{out}$. Reference parameters of operations are treated as both input and output values, so for a reference parameter y, $\underline{inp}$ contains its initial value, and $\underline{out}$ contains its updated value. Note that $\mathcal{L}^{\mathtt{C}}$ and thus histories $h^{\mathtt{C}}$ contain operation events only of $\mathtt{C}$ and no events for operations of any of its subcomponents $\mathtt{C}_l$. This definition is used so that histories match the *observable behavior* of components better, as it will become relevant for the definition of refinement. On the other hand, histories $\widehat{h}^{\mathtt{C}}$

that additionally contain events of subcomponent calls are built over the extended event set $\widehat{\mathcal{L}}^{\mathtt{C}} \equiv \mathcal{L}^{\mathtt{C}} \cup \bigcup_{l \in L} \widehat{\mathcal{L}}^{\mathtt{C}_l}$ [2].

Similar to (durable) linearizability [60, 66], operations in histories are *completed* or *pending*.

Definition 19 (Completed and Pending Operations). *A completed operation in a history $h^{\mathtt{C}}$ is any pair (e_{inv}, e_{res}) of invocation $e_{inv} = inv^{\mathtt{C}}_{tid}(i, in)$ and following matching response $e_{res} = res^{\mathtt{C}}_{tid}(i, out)$. A pending operation is an invocation $e_{inv} = inv^{\mathtt{C}}_{tid}(i, in)$ where e_{inv} has no matching response in $h^{\mathtt{C}}$.*

Pending operations e_{inv} can be *completed* by appending a response e_{res} to $h^{\mathtt{C}}$ such that (e_{inv}, e_{res}) is a completed operation in $h^{\mathtt{C}}\ e_{res}$. The completed part *completed*$(h^{\mathtt{C}})$ of a crash-free history $h^{\mathtt{C}}$ is the maximal subsequence of $h^{\mathtt{C}}$ consisting only of invocations and matching responses.

We distinguish between *sequential* and *concurrent* histories.

Definition 20 (Sequential Histories). *A non-empty history $h^{\mathtt{C}}$ of a component* C *is sequential iff*

- *it is crash-free,*
- *it starts with an invocation event,*
- *every invocation event $inv^{\mathtt{C}}_{tid}(i, \ldots)$ is immediately followed by a matching response event $res^{\mathtt{C}}_{tid}(i, \ldots)$, except if it is last event in $h^{\mathtt{C}}$, and*
- *every response event $res^{\mathtt{C}}_{tid}(i, \ldots)$ is directly preceded by its matching invocation event $inv^{\mathtt{C}}_{tid}(i, \ldots)$.*

The restriction of a history $h^{\mathtt{C}}$ to the events of a thread *tid* (including crash events) is denoted by $h^{\mathtt{C}}|_{tid}$. We only want to consider concurrent histories that originate from interleaving sequential histories of multiple threads. Such histories are called *well-formed*.

Definition 21 (Well-formed Concurrent Histories). *A concurrent history $h^{\mathtt{C}}$ is well-formed iff $h^{\mathtt{C}}|_{tid}$ is sequential for every $tid \in Tid$.*

Definition 22 (Equivalent Concurrent Histories). *Two concurrent histories $h^{\mathtt{C}}_0$ and $h^{\mathtt{C}}_1$ are equivalent iff $h^{\mathtt{C}}_0|_{tid} = h^{\mathtt{C}}_1|_{tid}$ for every $tid \in Tid$.*

In order to define the semantics of components in terms of chained (or interleaved) operation executions, histories are combined with the semantics of programs. We consider intervals I with an empty environment and without blocking steps in the following. Thus, intervals are treated as sequences of states $I = (s_0, s_1, s_2, \ldots)$, omitting the stuttering steps of the environment and the blocked flags.

For a program α, the interval semantics of Def. 6 can be extended with a history $h^{\mathtt{C}}$ that tracks the invocations and responses of calls to operations of C.

[2]Contrary to the definition of the combined state $\underline{s}^{\mathtt{C}}$, the regular union operator is used since $\widehat{\mathcal{L}}^{\mathtt{C}}$ should contain only a single global crash event $\lightning$.

Definition 23 (Observable Histories of Program Executions). *A history $h^{\mathtt{C}}$ is the observable history of a program execution $I \models \alpha$ w.r.t. to component* C*, iff $h^{\mathtt{C}}$ records all invocation and response events of interface and internal operations of* C *in the execution $I \models \alpha$. The judgement $I, h^{\mathtt{C}} \models \alpha$ is called an observable execution of α.*

Having observable executions of programs, observable operation executions can be defined: an interval I and a history $h^{\mathtt{C}}$ describe the execution of an operation of component C if they are an execution of a procedure that defines one of C's interface or internal operations.

Definition 24 (Observable Executions of Operations). *An interval I together with a history $h^{\mathtt{C}}$ form an observable operation execution of a component* C*, written $I, h^{\mathtt{C}} \models \mathtt{OP}^{\mathtt{C}}$, iff there is*

- *an interface operation $\mathtt{Op}_j$ with $j \in J^{\mathtt{C}}$ and a declaration of a corresponding procedure $\mathbf{op}_j\#(\underline{x}; \underline{y}, \underline{s}^{\mathtt{C}}; \underline{z})$ with precondition $\varphi(\underline{x}, \underline{y}, \underline{s}^{\mathtt{C}})$ so that*
  - *$I(0) \models \varphi(\underline{x}, \underline{y}, \underline{s}^{\mathtt{C}})$ and $I, h^{\mathtt{C}} \models \mathbf{op}_j\#(\underline{x}; \underline{y}, \underline{s}^{\mathtt{C}}; \underline{z})$ or*
  - *$I(0) \not\models \varphi(\underline{x}, \underline{y}, \underline{s}^{\mathtt{C}})$, I is arbitrary for the state variables $\underline{st}^{\mathtt{C}}$, and $h^{\mathtt{C}}$ consists of the invoke event for $\mathtt{Op}_j$ and the corresponding response event if I is finite*

  or there is

- *an internal operation $\mathtt{IOp}_k$ with $k \in K^{\mathtt{C}}$ and a declaration of a corresponding procedure $\mathbf{iop}_k\#(; \underline{s}^{\mathtt{C}})$ with guard $\varphi(\underline{s}^{\mathtt{C}})$ so that $I(0) \models \varphi(\underline{s}^{\mathtt{C}})$ and $I, h^{\mathtt{C}} \models \mathbf{iop}_k\#(; \underline{s}^{\mathtt{C}})$*

Operation executions of internal operation are restricted to intervals in which the guard of the corresponding operation holds as it would not be triggered otherwise. Similarly, interface operations should only be called by clients when its precondition is satisfied. If an interface operation is called in a state in which the precondition does not hold, the remaining interval can be arbitrary (in particular, non-termination of the operation is allowed) since the caller violated the contract of the operation. Nevertheless, the history $h^{\mathtt{C}}$ should contain the matching events of the operation call so that observable behavior is as expected.

Definition 25 (Interrupted Executions of Operations). *An interval I together with a history $h^{\mathtt{C}}$ forms an interrupted observable operation execution of a component* C*, written $I, h^{\mathtt{C}} \models \mathtt{OP}^{\mathtt{C}}_{\lightning}$, iff there is an interval I_0 and a history $h^{\mathtt{C}}_0$ such that $I \mathbin{⨾} I_0, h^{\mathtt{C}} \cdot h^{\mathtt{C}}_0 \models \mathtt{OP}^{\mathtt{C}}$. The pair $(I_0, h^{\mathtt{C}}_0)$ is then called a completion of the execution.*

Interrupted operation executions are required to model the occurrence of crashes during the execution of an operation. Naturally, an interrupted operation only produces a prefix of the state transitions and a prefix of the history of a complete operation execution ($h_0 \cdot h_1$ denotes the concatenation of two histories h_0 and h_1). In particular, the response event of an operation is typically missing if it has been interrupted. The definition, however, also comprises operation executions with crashes directly before the invocation or just after the response since I and $h^{\mathtt{C}}$ resp. I_0 and $h^{\mathtt{C}}_0$ can be empty.

The judgements $I, h^{\mathtt{C}} \models \mathtt{OP}^{\mathtt{C}}$ and $I, h^{\mathtt{C}} \models \mathtt{OP}^{\mathtt{C}}_{\lightning}$ denote (interrupted) executions of an arbitrary (interface or internal) operation of $\mathtt{C}$. For a particular operation $\mathtt{Op}^{\mathtt{C}}$ of $\mathtt{C}$, a regular execution is written $I, h^{\mathtt{C}} \models \mathtt{Op}^{\mathtt{C}}$ and an interrupted execution is written $I, h^{\mathtt{C}} \models \mathtt{OP}^{\mathtt{C}}_{\lightning}$.

Given histories and executions of operations, a transition system describing the semantics of components can be defined. In this section, we give a basic semantics for sequential, non-retracting systems, see Def. 27. This basic definition will be extended for retracting components with Def. 34 in Chapter 6 and for concurrent components with Def. 40 in Chapter 7.

Definition 26 (Labeled Transition System)**.** *A labeled transition system $LTS = (St, Initial, \mathcal{L}, \rightarrow)$ consists of a set of states St, a set of initial states $Initial \subseteq St$, a set of labels $\mathcal{L}$, and a state transition relation $\rightarrow \subseteq St \times \mathcal{L} \times St$.*

In the following, a state transition $(s, l, s') \in \rightarrow$ will be written $s \xrightarrow{l} s'$. Multiple state transitions can be combined to (potential infinite) *runs* $r = s \xrightarrow{l_0} s_0 \xrightarrow{l_1} s_1 \ldots s_{n-1} \xrightarrow{l_n} s'$, where *system runs* start in an initial state $s \in Initial$. The labels of a run r form histories: $h(r)$ extracts the full label sequence $l_0 l_1 l_2 \ldots l_n$; $h^{\mathtt{C}}(r)$ extracts the history of r containing only labels $l_i \in \mathcal{L}^{\mathtt{C}}$. Conversely, $s \xrightarrow{h} s'$ abbreviates a run from s to s' that produces the history $h = l_0 l_1 \ldots l_n$, implying there exist states $s_0, s_1, \ldots s_{n-1} \in S$ that form a run $r = s \xrightarrow{l_0} s_0 \xrightarrow{l_1} s_1 \ldots s_{n-1} \xrightarrow{l_n} s'$. The state sequences of runs can be considered as intervals: we write $I(r)$ for the interval $(s, s_0, s_1, \ldots, s')$ of the run r. Combining histories and intervals of runs, the tuple (I, h) is called an *observable run* of LTS, written $(I, h) \in runs(LTS)$, if there is a system run r of LTS with $I = I(r)$ and $h = h(r)$.

Definition 27 (Semantics of Sequential, Non-Retracting Components)**.** *The semantics of a sequential, non-retracting component $\mathtt{C}$ is given by the labeled transition system $C = (\mathtt{S}^{\mathtt{C}}, \mathtt{INIT}^{\mathtt{C}}, \mathcal{L}^{\mathtt{C}} \cup \{\tau\}, \rightarrow^{seq})$. The state transition relation $\rightarrow^{seq}$ is determined by the set of observable system runs $(I, h^{\mathtt{C}}) \in runs(C)$ satisfying the following conditions.*

1. *$I.first \in \mathtt{INIT}^{\mathtt{C}}$.*

2. *$h^{\mathtt{C}} = h^{\mathtt{C}}_0 \cdot \lightning_1 \cdot h^{\mathtt{C}}_1 \cdot \lightning_2 \cdot \ldots \cdot \lightning_n \cdot h^{\mathtt{C}}_n$ consists of crash-free era histories $h^{\mathtt{C}}_0, h^{\mathtt{C}}_1, \ldots, h^{\mathtt{C}}_n$ separated by crash events $\lightning_1, \lightning_2, \ldots, \lightning_n$.*

3. *$I = I_0 \fatsemi (I_0.last, I_1.first) \fatsemi I_1 \fatsemi (I_1.last, I_2.first) \fatsemi \ \ldots \ \fatsemi (I_{n-1}.last, I_n.first) \fatsemi I_n$ is built by the sequential composition of era intervals $I_0, I_1, \ldots, I_n$ connected with interrupt steps $(I_0.last, I_1.first), (I_1.last, I_2.first), \ \ldots \ , (I_{n-1}.last, I_n.first)$.*

4. *Each era $I_i, h^{\mathtt{C}}_i$ with $i < n$ forms an interrupted run of the sequential system, i.e., $I_i, h^{\mathtt{C}}_i \models \{\ \mathtt{OP}^{\mathtt{C}}\ \}^* \ ; \ \mathtt{OP}^{\mathtt{C}}_{\lightning}$.*

5. *The last era $I_n, h^{\mathtt{C}}_n$ forms an uninterrupted system run $I_n, h^{\mathtt{C}}_n \models \{\ \mathtt{OP}^{\mathtt{C}}\ \}^*$.*

6. *Each crash transition $I_i.last \xrightarrow{\lightning_{i+1}} I_{i+1}.first$ with $i < n$ results from applying the crash and recovery relations, i.e., $(I_i.last, I_{i+1}.first) \in \mathtt{CRASH}^{\mathtt{C}} \fatsemi \mathtt{REC}^{\mathtt{C}}$.*

7. *All internal transitions, i.e., transitions that do not correspond to an event of $h^{\texttt{C}}$, are labeled with the internal label τ.*

Condition 1 of Def. 27 ensures that only actual system runs are considered, i.e., runs starting in an initial state. Conditions 2 and 3 introduce *eras*: crashes partition an observable run into crash-free sub-runs.

Eras are only allowed to consist of sequential executions of the components interface and internal operations (conditions 4 and 5). The $\_^*$ operator is used to denote finite or infinite iteration, i.e., α^* abbreviates the statement $\alpha; \alpha; \alpha; \ldots$ executing α finitely or infinitely often. All eras but the last are assumed to consist only of finitely many executions of $\texttt{OP}^{\texttt{C}}$ since the run is interrupted by a crash at some point. This crash may occur during the execution of an operation, so the interrupted eras end with an interrupted execution $\texttt{OP}^{\texttt{C}}_{\lightning}$. Recall, however, that $\texttt{OP}^{\texttt{C}}_{\lightning}$ also covers crashes in between operation executions. The last era is not interrupted by a crash so that it may consist of infinitely many operation executions.

Condition 6 of Def. 27 determines the allowed crash transitions, i.e., how eras are connected. The transition captures the application of the crash effect given by $\texttt{CRASH}^{\texttt{C}}$ and a subsequent application of the recovery mechanism given by $\texttt{REC}^{\texttt{C}}$ ($R_0 \fatsemi R_1$ denotes the composition of two relations R_0 and R_1). The relations $\texttt{CRASH}^{\texttt{C}}$ and $\texttt{REC}^{\texttt{C}}$ in the definition result from combining the respective relation of C with the ones of its subcomponents to a relation over $\texttt{S}^{\texttt{C}} \times \texttt{S}^{\texttt{C}}$. In practice, $\texttt{CRASH}^{\texttt{C}}$ is given by the conjunction of the crash predicates, and $\texttt{REC}^{\texttt{C}}$ is determined by the execution of C's recovery procedure, which must call the recovery procedures of all subcomponents according to Def. 14.

Finally, the remaining transitions with no invocation, response, or crash label assigned are labeled with the label τ representing non-observable internal steps.

Correctness of Components: Invariants & Data Refinement

The functional correctness of a sequential implementation component is typically proven by a data refinement of the corresponding specification component.

Informally, a component C can be considered a refinement of another component A, written $\texttt{C} \leq \texttt{A}$, if both have the same interface and the input/output behavior of C is also a possible behavior of A. If this is the case, clients of A cannot distinguish if they actually use A or if A has been exchanged with the implementation C.

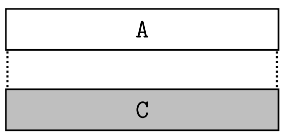


Figure 3.2: Refinement of a specification A by an implementation C.

Fig. 3.2 shows how a typical refinement relationship is depicted graphically. The abstract component A and its refinement C are connected by dotted lines. The two components are also color-coded: throughout this thesis, implementation components are colored gray while specification components are white. Of course, refinement is not restricted to this scenario; in principle, any combination of specification and implementation components is possible. Since refinement is transitive, it can also be applied incrementally: the refinement of a component A with a component C can be divided into a sequence of smaller refinements $\texttt{C} \leq \texttt{A}_n \leq \ldots \leq \texttt{A}_0 \leq \texttt{A}$.

The basic requirement for a refinement relationship between two components is that the are *compatible*.

Definition 28 (Compatibility of Components)**.** *A component* $\mathtt{C}$ *is compatible with a component* $\mathtt{A}$ *iff they have the same index set of interface operations, i.e.,* $J^{\mathtt{C}} = J^{\mathtt{A}}$*, the index set of internal operations of* $\mathtt{C}$ *contains all internal operations of* $\mathtt{A}$*, i.e.,* $K^{\mathtt{A}} \subseteq K^{\mathtt{C}}$*, and all shared operations have the same input and output parameters.*

For the formal definition of refinement, the *observable behavior* of a component is defined in terms of histories produced by runs of its labeled transition system.

Definition 29 (Observable Behavior of Components)**.** *The observable behavior* $\mathit{Obs}(\mathtt{C})$ *of a component* $\mathtt{C}$ *is given by the set of histories* $h^{\mathtt{C}}$ *of observable runs* $(I, h^{\mathtt{C}}) \in \mathit{runs}(C)$ *of the corresponding labeled transition system* C*.*

It is crucial that the observable behavior of component $\mathtt{C}$ only contains events of its interface/internal operations and crash events but no events of $\mathtt{C}$'s subcomponents. Otherwise, refinement would be restricted to components using identical subcomponents in the same way.

Definition 30 (Refinement)**.** *A component* $\mathtt{C}$ *refines a component* $\mathtt{A}$*, written* $\mathtt{C} \leq \mathtt{A}$*, iff* $\mathtt{C}$ *is compatible with* $\mathtt{A}$ *and the observable behavior of* $\mathtt{C}$ *is a subset of the observable behavior of* $\mathtt{A}$*, i.e.,* $\mathit{Obs}(\mathtt{C})|_{\mathcal{L}(\mathtt{A})} \subseteq \mathit{Obs}(\mathtt{A})$*, where* $\mathit{Obs}(\mathtt{C})|_{\mathcal{L}(\mathtt{A})}$ *restricts the operation events of* $\mathit{Obs}(\mathtt{C})$ *to the index sets* $J^{\mathtt{A}}$ *and* $K^{\mathtt{A}}$ *of* $\mathtt{A}$[3]*.*

A common form of refinement is *data refinement*, where a refinement step changes the internal data representation of a component. As shown later in Thm. 1, proofs for such refinements are done with a forward simulation using commuting diagrams. Before that, it must be shown that individual components use their subcomponents correctly, i.e., call their interface operations only within their preconditions. Additionally, invariant formulas can be given for each component, which all its interface and internal operations must maintain. Establishing such invariants simplifies (or even makes it possible in the first place) to prove the forward simulation of a refinement since the invariants of both the abstract and the concrete component can be added as assumptions to the proof obligations.

The invariant of a sequential component $\mathtt{C}$ is given in the form of a formula *inv* over its combined state variables $\underline{s}$.

> **component** $\mathtt{C}$
>
> **invariants** $\mathit{inv}(\underline{s})$

The proof obligations of Lem. 1 are generated for all components and ensure that given invariant holds.

Lemma 1 (Invariants of Sequential, Non-Retracting Components)**.** *For a sequential, non-retracting component* $\mathtt{C}$*, the following proof obligations ensure termination and the absence of exceptions, establish the sequential invariant* $\mathit{inv}(\underline{s})$*, and guarantee that* $\mathtt{C}$ *calls its subcomponents' interface operations only if their preconditions are satisfied.*

[3]The restriction is necessary as $\mathtt{C}$ may introduce additional internal operations which would be visible in the observable behavior. So strictly speaking, $\mathit{Obs}(\mathtt{A})$ would also have to contain corresponding events for these operations to achieve matching observable behavior. We circumvent this technical detail here with the restriction, however, the discrepancy could also be overcome by introducing a dummy operation in $\mathtt{A}$ with the guard `true` and the body **skip**.

1. $\vdash \langle\!|\mathbf{init}\#(\underline{x};;\underline{s},\underline{z})|\!\rangle\,(\varphi(\underline{s},\underline{z}) \to inv(\underline{s}))$
2. $pre_j(\underline{x},\underline{y},\underline{s}),\ inv(\underline{s}) \vdash \langle\!|\mathbf{op}_j\#(\underline{x};\underline{y},\underline{s};\underline{z})|\!\rangle\, inv(\underline{s})$ *for all* $j \in J$
3. $guard_k(\underline{s}),\ inv(\underline{s}) \vdash \langle\!|\mathbf{iop}_k\#(;\underline{s})|\!\rangle\, inv(\underline{s})$ *for all* $k \in K$
4. $inv(\underline{s}_0),\ crash(\underline{s}_0,\underline{s}) \vdash \langle\!|\mathbf{recover}\#(\underline{x};\underline{s};\underline{z})|\!\rangle\,(\varphi(\underline{s},\underline{z}) \to inv(\underline{s}))$

□

Recall that initialization and recovery procedures may have restriction conditions φ. Thus, *inv* must only be established in the proof obligations 1 and 4 of Lem. 1 if φ is true after the execution of the procedures. Otherwise initialization respectively recovery have to be rerun before the system can be used.

Component operations with preconditions are treated specially in the weakest-precondition calculus. When a procedure $\mathbf{op}_j\#$ with body α and precondition pre_j is called, the call is replaced with the (instantiated) body α, however, wrapped inside an **if** statement: **if** pre_j **then** α **else abort**. Since the obligations of Lem. 1 all use strong diamond formulas, it has to be shown that the respective precondition holds for each call (a formula $\langle\!|\mathbf{abort}|\!\rangle\,\varphi$ always reduces to `false`). Thus, the obligations implicitly ensure that clients always use their subcomponents safely, which is why the precondition can be assumed in obligation 2.

The absence of exceptions is guaranteed by not giving explicit exception specifications, which comes down to the default exception specifications $\underline{\xi} \equiv$ **default** :: `false` (see Sec. 2.4).

For a refinement proof of $\mathtt{C} \leq \mathtt{A}$, a forward simulation $R \subseteq \mathtt{S}^\mathtt{A} \times \mathtt{S}^\mathtt{C}$ is required. R (called *abstraction relation* in the following) is given in form of a formula *abs* over the state variables $\underline{s}^\mathtt{A}$ and $\underline{s}^\mathtt{C}$.

data refinement $\mathtt{C} \leq \mathtt{A}$
abstraction relation $abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C})$

The proof obligations Thm. 1 ensure a valid refinement of two compatible sequential, non-retracting components.

Theorem 1 (Data Refinement of Sequential, Non-Retracting Components). *The refinement* $\mathtt{C} \leq \mathtt{A}$ *of sequential, non-retracting components* $\mathtt{A}$ *and* $\mathtt{C}$*, where* $\mathtt{C}$ *is compatible with* $\mathtt{A}$*, is implied by the forward simulation* $abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C})$ *satisfying the following conditions.*

1. $\vdash \langle\!|\mathbf{init}^\mathtt{C}\#(\underline{x};;\underline{s}^\mathtt{C},\underline{z}_0)|\!\rangle$
   $\big(\varphi^\mathtt{C}(\underline{s}^\mathtt{C},\underline{z}_0) \to \langle\mathbf{init}^\mathtt{A}\#(\underline{x};;\underline{s}^\mathtt{A},\underline{z}_1)\rangle(abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C}) \wedge \underline{z}_0 = \underline{z}_1)\big)$
2. $abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C}),\ pre^\mathtt{A}_j(\underline{x},\underline{y},\underline{s}^\mathtt{A}),\ inv^\mathtt{A}(\underline{s}^\mathtt{A}),\ inv^\mathtt{C}(\underline{s}^\mathtt{C}),\ \underline{y}_0 = \underline{y}_1$
   $\vdash \langle\!|\mathbf{op}^\mathtt{C}_j\#(\underline{x};\underline{y}_0,\underline{s}^\mathtt{C};\underline{z}_0)|\!\rangle$ *for all* $j \in J$
   $\big(\langle\mathbf{op}^\mathtt{A}_j\#(\underline{x};\underline{y}_1,\underline{s}^\mathtt{A};\underline{z}_1)\rangle(abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C}) \wedge \underline{y}_0 = \underline{y}_1 \wedge \underline{z}_0 = \underline{z}_1)\big)$
3. $abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C}),\ guard^\mathtt{C}_k(\underline{s}^\mathtt{C}),\ inv^\mathtt{A}(\underline{s}^\mathtt{A}),\ inv^\mathtt{C}(\underline{s}^\mathtt{C})$
   $\vdash \langle\!|\mathbf{iop}^\mathtt{C}_k\#(;\underline{s}^\mathtt{C})|\!\rangle\,\big(\langle\mathbf{iop}^\mathtt{A}_k\#(;\underline{s}^\mathtt{A})\rangle abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C})\big)$ *for all* $k \in K^\mathtt{C} \cap K^\mathtt{A}$
4. $abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C}),\ guard^\mathtt{C}_k(\underline{s}^\mathtt{C}),\ inv^\mathtt{A}(\underline{s}^\mathtt{A}),\ inv^\mathtt{C}(\underline{s}^\mathtt{C})$
   $\vdash \langle\!|\mathbf{iop}^\mathtt{C}_k\#(;\underline{s}^\mathtt{C})|\!\rangle\, abs(\underline{s}^\mathtt{A},\underline{s}^\mathtt{C})$ *for all* $k \in K^\mathtt{C} \setminus K^\mathtt{A}$

5. $abs(\underline{s}_0^{\mathtt{A}}, \underline{s}_0^{\mathtt{C}}),\ inv^{\mathtt{A}}(\underline{s}_0^{\mathtt{A}}),\ inv^{\mathtt{C}}(\underline{s}_0^{\mathtt{C}}),\ crash^{\mathtt{C}}(\underline{s}_0^{\mathtt{C}}, \underline{s}^{\mathtt{C}}),\ \underline{s}^{\mathtt{C}} = \underline{s}_1^{\mathtt{C}}$

$$\vdash \langle\!|\mathbf{recovery}^{\mathtt{C}}\#(\underline{x}; \underline{s}^{\mathtt{C}}; \underline{z}_0)|\!\rangle \Big(\big(\varphi^{\mathtt{C}}(\underline{s}^{\mathtt{C}}, \underline{z}_0) \rightarrow \exists\ \underline{s}^{\mathtt{A}}.\quad crash^{\mathtt{A}}(\underline{s}_0^{\mathtt{A}}, \underline{s}^{\mathtt{A}}) \land \langle \mathbf{recovery}^{\mathtt{A}}\#(\underline{x}; \underline{s}^{\mathtt{A}}; \underline{z}_1)\rangle (abs(\underline{s}^{\mathtt{A}}, \underline{s}^{\mathtt{C}}) \land \underline{z}_0 = \underline{z}_1)\big) \land \big(\neg\ \varphi^{\mathtt{C}}(\underline{s}^{\mathtt{C}}, \underline{z}_0) \rightarrow crash^{\mathtt{C}}(\underline{s}^{\mathtt{C}}, \underline{s}_1^{\mathtt{C}})\big)\Big)$$

The proof obligations of Thm. 1 are quite similar to those of standard data refinement [32]. Additional proof obligations arise from dealing with crashes (5) and having internal operations (3 and 4).

Obligations (1) and (5) also cope with potential low-level errors occurring during initialization or recovery: both obligations only require to show that related states have been (re-)established when the operation of C signals a successful execution via the restriction condition $\varphi^{\mathtt{C}}$. When the initialization of C fails, nothing has to be shown about the initialization of A (it may produce an arbitrary state) since initialization has to be rerun. Similarly, the final state of a failed recovery is irrelevant. However, the proof obligation has to ensure that recovery can be retried after a failed attempt. One must show that a crash from the state produced by the failed recovery execution to the initial crashed state is possible (the equation $\underline{s}^{\mathtt{C}} = \underline{s}_1^{\mathtt{C}}$ in the antecedent of obligation (5) fixes the values of $\underline{s}^{\mathtt{C}}$ before the execution of $\mathbf{recovery}^{\mathtt{C}}\#$), which enables applying obligation (5) in the resulting state again. Intuitively, this restricts C's recovery to manipulating only RAM state while leaving persistent state unaffected. This behavior is reasonable as the recovery task is to reconstruct in-memory data structures from persisted data, e.g., one main objective of the Flashix recovery routine is to restore the latest version of the RAM index by loading the last committed version from flash and replaying the uncommitted log entries.

Obligation (2) gives the transitions for interface operations as expected. Informally, one has to prove that, when starting in *abs*-related states, for each execution of an operation $\mathtt{Op}_j^{\mathtt{C}}$ of C there must be a matching execution of $\mathtt{Op}_j^{\mathtt{A}}$ of A that maintains $abs(\underline{s}^{\mathtt{A}}, \underline{s}^{\mathtt{C}})$ with the same inputs and outputs. Note that only the abstract precondition $pre_j^{\mathtt{A}}$ is assumed, so the obligations also require to show that the precondition $pre_j^{\mathtt{A}}$ is strong enough to establish the concrete precondition $pre_j^{\mathtt{C}}$ if *abs* holds.

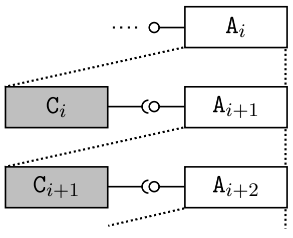


Figure 3.3: Building a modularized refinement hierarchy.

The obligations (3) and (4) give the analog transitions for internal operations. If C *introduces* an internal operation $\mathtt{IOp}_k^{\mathtt{C}}$, i.e., there is no corresponding operation in A, an execution of $\mathtt{IOp}_k^{\mathtt{C}}$ commutes with a no-op of A, given by (4). Otherwise, $\mathtt{IOp}_k^{\mathtt{C}}$ commutes with the corresponding operation $\mathtt{IOp}_k^{\mathtt{A}}$ of A as given by (3). Conversely to preconditions of interface operations, the guard $guard_k^{\mathtt{C}}$ of the concrete operation is assumed, so $guard_k^{\mathtt{C}}$ and *abs* must imply $guard_k^{\mathtt{A}}$.

Proof of Thm. 1. As usual, the proof composes commuting diagrams, starting from two related states resulting from successful executions of (1). For executions of

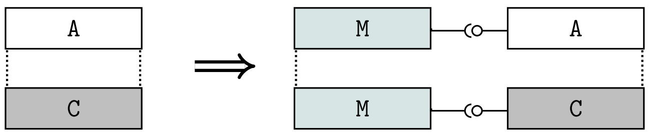


Figure 3.4: Compositionality of Data Refinement.

internal or interface operations, the obligations (2), (3), and (4) give the respective commuting diagrams.Finally, obligation (5) gives the commuting diagram for crash transitions. □

Using subcomponents and refinements, a system is composed of multiple components connected by refinements and subcomponent usages, resulting in a hierarchy like the one shown in Fig. 3.3.

A specification component $\mathtt{A}_i$ is refined by an implementation $\mathtt{C}_i$ that uses a specification $\mathtt{A}_{i+1}$ as a subcomponent. This pattern then repeats in the sense that $\mathtt{A}_{i+1}$ is refined further by an implementation $\mathtt{C}_{i+1}$ that again uses a subcomponent $\mathtt{A}_{i+2}$ and so on. $\mathtt{A}_i$ may also be used as a subcomponent of an implementation $\mathtt{C}_{i-1}$ if it is not the top-level specification. The complete implementation of the system then results from composing all individual implementation components $\mathtt{C}_0(\mathtt{C}_1(\mathtt{C}_2(...)))$.

The central requirement for the correct use of such a modularization approach is *compositionality*: substituting a subcomponent $\mathtt{A}$ of a client $\mathtt{M}$ by a correct implementation $\mathtt{C}$ must not affect the correctness of $\mathtt{M}$. In other words, $\mathtt{C} \leq \mathtt{A}$ must imply $\mathtt{M}(\mathtt{C}) \leq \mathtt{M}(\mathtt{A})$ for a client component $\mathtt{M}$ that uses its subcomponents accordingly (cf. Fig. 3.4). This ensures that the composed implementation of a refinement hierarchy is in fact a correct refinement of its top-level specification $\mathtt{A}_0$, i.e. $\mathtt{C}_0(\mathtt{C}_1(\mathtt{C}_2(...))) \leq \mathtt{A}_0$. This allows us to divide a complex refinement task into multiple, more manageable ones.

Theorem 2 (Compositionality of Non-Retracting Data Refinement)**.** *Given two sequential, non-retracting components* $\mathtt{C}$ *and* $\mathtt{A}$ *with* $\mathtt{C} \leq \mathtt{A}$*, i.e., where* $\mathtt{C}$ *is a valid refinement of* $\mathtt{A}$*, and a crash-modularizable component* $\mathtt{M}(\mathtt{A})$ *that uses* $\mathtt{A}$ *as a subcomponent, then* $\mathtt{A}$ *can be substituted by* $\mathtt{C}$ *in* $\mathtt{M}$ *without affecting* $\mathtt{M}$*'s correctness, i.e.,* $\mathtt{M}(\mathtt{C}) \leq \mathtt{M}(\mathtt{A})$ *holds.*

Proof of Thm. 2. See the first case of the proof of Thm. 1 in [96]. □

Crash-Atomic Operations & Crash-Neutrality

The proof obligations of Lem. 1 and Thm. 1 consider crashes only in between operations (the crash/recovery obligations always assume states in which the sequential invariants and the abstraction relation hold). This simplification is possible due to a criterion called *crash-neutrality*[4].

Definition 31 (Crash-Neutral Operations)**.** *An operation* $\mathtt{Op}^{\mathtt{C}}$ *of a component* $\mathtt{C}$ *is crash-neutral if every interrupted execution* $I, h^{\mathtt{C}} \models \mathtt{Op}^{\mathtt{C}}_{\lightning}$ *has a completion* $(I_0, h^{\mathtt{C}}_0)$

[4]The term *crash-neutrality* is adopted from the work of Ernst [38]. Pfähler [96] uses the term *crash-introducible* for this criterion and repurposes *crash-neutrality* for a slightly different, stronger criterion.

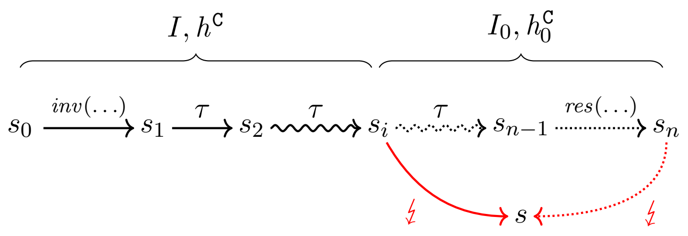


Figure 3.5: Crash-neutrality of operations.

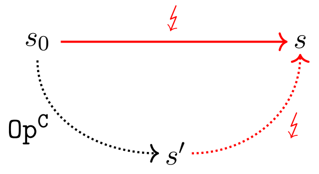


Figure 3.6: Full crash-neutral operation execution.

with $I \fatsemi I_0, h^{\mathtt{C}} \cdot h_0^{\mathtt{C}} \models \mathtt{Op}^{\mathtt{C}}$*, such that* $(I_0.last, s) \in \mathtt{CRASH}^{\mathtt{C}} \fatsemi \mathtt{REC}^{\mathtt{C}}$ *holds for every state* $s \in \mathtt{S}^{\mathtt{C}}$ *with* $(I.last, s) \in \mathtt{CRASH}^{\mathtt{C}} \fatsemi \mathtt{REC}^{\mathtt{C}}$.

As depicted in Fig. 3.5, crashes during the execution of crash-neutral operations can be viewed as crashes that occur directly after the respective operation. An arbitrary interrupted execution $(I, h^{\mathtt{C}})$ of an operation of $\mathtt{C}$ ends in some intermediate state s_i, i.e., $I.last = s_i$, and a crash transition $\lightning$ from this state yields to a *crash-recovered* state s. Crash-Neutrality now guarantees that the interrupted execution can be completed to a state sn which also has a $\lightning$-transition to s. Intuitively, this means that a crash-neutral operation can always "decide" to make only steps that do not alter the persistent state. For a flash file system, this property typically holds trivially for its operations since the flash hardware can always refuse to perform an operation if an error is detected, e.g., when too many bit errors occurred (which may indicate that the degradation of a requested block reaches a critical level). As a consequence, it is possible to "push" a crash that occurs during a crash-neutral operation to the back of it and only consider crashes in between crash-neutral operations. Such operations are considered as *crash-atomic*.

As a special case of crash-neutrality, complete operation executions can be introduced before a crash transition, as shown in Fig. 3.6. Given a crash transition from a state s_0 to a state s, crash-neutrality of an operation $\mathtt{Op}^{\mathtt{C}}$ guarantees that there exists a complete execution of $\mathtt{Op}^{\mathtt{C}}$ starting from s_0 to some state s', from which s can be reached via a crash transition. This follows directly from Def. 25 since the interval and history of an interrupted operation execution can be empty.

Definition 32 (Crash-Neutral Components)**.** *A component* $\mathtt{C}$ *is crash-neutral iff all of its interface operations* $\mathtt{Op}_j^{\mathtt{C}}$ *with* $j \in J$ *are crash-neutral.*

The crash-neutrality property is lifted from single operations to whole components by Def. 32. If all interface operations of a component, i.e., all operations a client can call explicitly, are crash-neutral, the component is as well. Using only crash-neutral subcomponents directly propagates the property to an implementation client component, as stated by Lem. 2.

Lemma 2 (Crash-Neutrality of Implementation Components)**.** *An implementation component (and thus RAM component)* $\mathtt{C}$ *with subcomponents* $(\mathtt{C}_l)_{l \in L}$ *is crash-neutral if each subcomponent* $\mathtt{C}_l$ *is crash-neutral.* □

Fig. 3.7 shows the basic argumentation for Lem. 2 (a detailed proof is given in [38] and [96]). A regular execution of an operation $\mathtt{Op}^{\mathtt{C}}$ of an implementation component $\mathtt{C}(\mathtt{A})$ consists of steps accessing the RAM state of $\mathtt{C}$ (labeled "RAM ops"

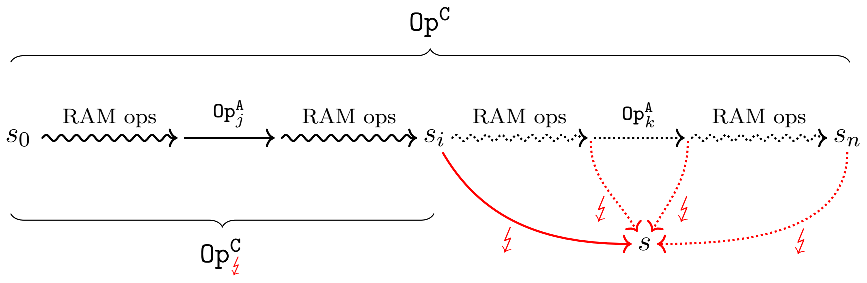

$\mathtt{Op}^{\mathtt{C}}$
s_0
RAM ops
$\mathtt{Op}^{\mathtt{A}}_j$
RAM ops
s_i
RAM ops
$\mathtt{Op}^{\mathtt{A}}_k$
RAM ops
s_n
s
$\mathtt{Op}^{\mathtt{C}}_{\lightning}$

Figure 3.7: Crash-neutrality of an implementation component $\mathtt{C(A)}$ using a crash-neutral subcomponent $\mathtt{A}$.

in the figure) and of calls to interface operations $\mathtt{Op}^{\mathtt{A}}$ of its subcomponent $\mathtt{A}$. Since $\mathtt{C}$ is an implementation component, accesses to persistent state are performed only via the $\mathtt{A}$. Given an interrupted execution $\mathtt{Op}^{\mathtt{C}}_{\lightning}$ ending in some intermediate state s_i and a crash-recovered state s, crash-neutrality of the operation (and thus of $\mathtt{C}$) follows again from a completion of $\mathtt{Op}^{\mathtt{C}}$ where a crash-recovery from the final state to s is possible. Such a completion can be constructed by adding crash-neutral steps of the form of Fig. 3.6 incrementally. Steps accessing RAM state are by definition crash-neutral as the complete RAM state is lost during a crash (see Def. 11). Thus, RAM ops can be added to the interrupted execution, and the crash transition can be postponed until after these steps. For calls to interface operations of the crash-neutral subcomponent, like the call of $\mathtt{Op}^{\mathtt{A}}_k$ in the example, a crash-neutral transition can be added as well, and the crash can be pushed further back. This can be repeated until a complete execution of $\mathtt{Op}^{\mathtt{C}}$ is constructed.

Note that the execution constructed this way may produce a different call sequence than a regular, uninterrupted execution of $\mathtt{Op}^{\mathtt{C}}$. Crash-neutral executions of operations are typically executions in which the requested task could not be completed successfully. So the client component could, for example, employ a retry mechanism or cancel the execution preliminarily, which would result in fewer or additional calls.

Lem. 2 requires all subcomponents of $\mathtt{C}$ to be already crash-neutral in order to be applicable. While this seems quite restrictive at first, this can be achieved with relatively less effort in practice. Therefore, we only consider atomic components, i.e., components whose operations cannot be interrupted by a crash, as subcomponents. Then only a *wp* proof obligation remains to be proved for each component's operations individually.

Lemma 3 (Crash-Neutrality of Atomic Specification Components)**.** *An atomic component* $\mathtt{C}$ *is crash-neutral if the following proof obligation holds for all its interface operations* $\mathtt{Op}^{\mathtt{C}}_j$ *with* $j \in J$*.*

$$pre^{\mathtt{C}}_j(\underline{x}, \underline{y}, \underline{s}^{\mathtt{C}}),\ \ inv^{\mathtt{C}}(\underline{s}^{\mathtt{C}}),\ \ crash^{\mathtt{C}}(\underline{s}^{\mathtt{C}}, \underline{s}^{\mathtt{C}}_0) \vdash \langle \mathbf{op}^{\mathtt{C}}_j\#(\underline{x}; \underline{y}, \underline{s}^{\mathtt{C}}; \underline{z})\rangle\ crash^{\mathtt{C}}(\underline{s}^{\mathtt{C}}, \underline{s}^{\mathtt{C}}_0)$$

□

The obligation of Lem. 3 corresponds directly to the transitions depicted in Fig. 3.6. However, instead of proving the property for full $\lightning$ transitions (determined by the relation $\mathtt{CRASH}^{\mathtt{C}} ; \mathtt{REC}^{\mathtt{C}}$), the obligation is reduced to $\mathtt{CRASH}^{\mathtt{C}}$ transitions.

This simplifies the obligation significantly as no additional executions of the recovery procedure **recovery**$^{\mathtt{C}}\#$ need to be considered. The restriction is justified because a crash alone is usually sufficient to mask the effect of one possible execution of the operation.

In Flashix, for example, most operations of specification components are specified so that an execution satisfying the obligation exists trivially: the body chooses non-deterministically between a successful execution, potentially making persistent updates, and a *no-op* execution. This behavior emerges from the underlying hardware model of flash memory. An operation on flash memory is either performed successfully returning a success code or fails without altering the state and returns an error code signalizing the reason for the failure, e.g., `ENOSPC` if there is not enough space left on the flash memory for performing the operation.

$\{\ \ldots\ ;\ err \coloneqq \mathtt{ESUCCESS}\ \}$ **or** $\{\ err \coloneqq \mathtt{ENOSPC}\ \}$

Specification components on higher levels of the hierarchy adopt this behavior by potentially failing with a low-level error code if their implementations could access the flash hardware. Of course, not all operations on each level can fail that way, e.g., specifications often also provide operations that are performed purely on RAM state by their implementations and therefore are expected not to fail. Hence, the crash predicates must be chosen appropriately to ensure crash-neutrality of these operations.

The hardware model also justifies the restriction to atomic specification components in Lem. 3. Operations on flash memory are guaranteed to be executed atomically with respect to crashes, e.g., writing a flash page either writes the whole page or no byte at all. Due to this underlying atomicity, Lem. 3 can be applied bottom-up to a refinement hierarchy like the one shown in Fig. 3.3, yielding crash-neutrality on all levels.

3.2 Verification of Destructive Red-Black Trees

Red-black trees [54, 112] are typically used as an efficient data structure for ordered sets (or multisets). They are also used in the Flashix file system in various places. For example, two red-black trees are employed in the Erase Block Manager (`EBM`) to grant efficient access to free and used erase blocks. As a non-trivial in-memory data structure, red-black trees are a well-suited case study for illustrating the core concept introduced in Sec. 3.1, including the modularization approach but omitting advanced topics like crash-safety or concurrency.

In order to abstract from the complex implementation details of red-black trees (traversal, rotations, ...), a simple specification component `RBSet` is used that abstracts the tree data structure to a algebraic set (cf. Sec. 2.2) of ordered elements. Other components then can use `RBSet` as a subcomponent, which simplifies formal reasoning about the client component while the resulting system still uses an efficient heap implementation.

```
component RBSet
state    $rbs$ : set(tord)
```

The element type tord of *rbs* is a generic element for which a total order $. < . : \mathsf{tord} \times \mathsf{tord} \to \mathsf{bool}$ is defined. The order relation is specified by the usual properties of strict total orders.

irreflexivity:	$\vdash \neg\, a < a$
transitivity:	$\vdash a < b \wedge b < c \to a < c$
totality:	$\vdash a < b \vee a = b \vee b < a$

The order properties are a sufficient characterization of the elements stored in the red-black tree to implement and prove the correctness of the algorithms. For the actual use of the implementation, one can instantiate the generic element type with a concrete type by giving a suitable order relation, e.g., with natural numbers nat and the *less* predicate $. < . : \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool}$. In Flashix, a slightly more complex type is used, which stores physical erase block numbers (abbreviated *PNUMs*) and erase counts (abbreviated *ECs*) indicating the wear of an individual block.

$$\textbf{data}\quad \mathsf{blockcounter} = \texttt{block-counter}(.\ .\texttt{pnum} : \mathsf{nat}\ ;\ .\ .\texttt{ec} : \mathsf{nat})$$

This information is stored persistently on flash, but for an efficient implementation of *wear leveling*, it is also kept and updated in main memory using the red-black tree implementation given in this section. Wear leveling needs to find the blocks with the lowest resp. highest erase count, so consequently, the order relation for $\mathsf{blockcounter}$ is defined as the order of the erase count field.

$$\vdash \texttt{block-counter}(pnum_0, ec_0) < \texttt{block-counter}(pnum_1, ec_1) \leftrightarrow ec_0 < ec_1$$

Figure 3.8 shows a complete listing of `RBSet`'s procedure declarations. Note that all procedure names have a prefix "**rbset_**" indicating the component they are defined in. In the remainder of this thesis, all component procedures will have such a prefix for a better distinction between the procedures of different components.

Initially, *rbs* is empty ($\emptyset$) and it can be modified by inserting or removing elements *elem* (by **rbset_insert**#and **rbset_remove**#). Both insertion and removal have an additional output parameter *exists* signaling whether *elem* was in the *rbs* before the update and, thus, whether the operation changed the set. Additional interface procedures check whether the set is empty and whether an element is in the set. The minimal and the maximal[5] element can be selected as required by wear leveling. The red-black tree implementation given here is a pure in-memory data structure, so it does not use an underlying persistent storage. Accordingly, all components presented in this section are RAM components (cf. Def. 11) with the crash predicate `true`. After a crash, clients of the component are responsible for repopulating the data structure, so the recovery procedure of `RBSet` simply initializes *rbs* with $\emptyset$.

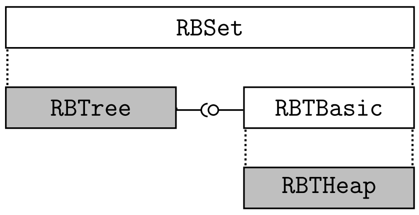


Figure 3.9: The refinement hierarchy for red-black trees.

[5]The actual model selects the maximum below a threshold, which is used to achieve a better distribution in wear leveling. However, this technical detail is abstracted here since it is not relevant for the implementation and verification of red-black trees.

```
rbset_init#()
  initialization
{
  $rbs := \emptyset$
}

rbset_recover#()
  recovery
{
  $rbs := \emptyset$
}

rbset_insert#($elem$; ; $exists$)
  interface
{
  $exists := elem \in rbs$,
  $rbs := rbs$ ++ $elem$
}

rbset_remove#($elem$; ; $exists$)
  interface
{
  $exists := elem \in rbs$,
  $rbs := rbs$ -- $elem$
}
```

```
rbset_isEmpty#(; ; $empty$)
  interface
{
  $empty := rbs = \emptyset$
}

rbset_lookup#($elem$; ; $exists$)
  interface
{
  $empty := elem \in rbs$
}

rbset_getMin#(; ; $elem$)
  interface
  precondition  $rbs \neq \emptyset$
{
  $elem$ := $rbs$.min
}

rbset_getMax#(; ; $elem$)
  interface
  precondition  $rbs \neq \emptyset$
{
  $elem$ := $rbs$.max
}
```

Figure 3.8: Abstract representation of red-black trees: the component `RBSet`.

This component is refined by a pointer-based implementation of red-black trees. However, this refinement is split into two parts to reduce the complexity of the necessary reasoning about the heap done with Separation Logic. The result is the refinement hierarchy shown in Fig. 3.9. The first refinement step `RBTree(RBTBasic)` $\leq$ `RBSet` shows that the set abstraction can be implemented by a red-black tree and that this implementation actually maintains all red-black tree properties. However, this is done using an algebraic datatype instead of a heap data structure.

In the second refinement step `RBTHeap` $\leq$ `RBTBasic`, it is then proven that a heap implementation conforms to this algebraic datatype. The goal of this partition is to keep the operations of `RBTBasic` (and hence those of `RBTHeap`) as simple as possible. The more complex algorithmic parts are handled in `RBTree` while `RBTBasic` only provides an interface for primitive manipulations of the algebraic tree. This includes for example the insertion of an element at a given point within the tree, or a single left- or right-rotation of one particular subtree.

When used directly, these operations are of course inefficient, but note that code can already be generated from `RBTree(RBTBasic)`, which is useful for testing invariants and results of example runs.

Since the interfaces of `RBTBasic` and `RBTHeap` must be identical, one difficulty that arises from this intermediate abstraction is that `RBTree` cannot use a uniform notion to describe a location within the tree. While using references is the natural way for a heap implementation like in `RBTHeap`, this is not possible for the algebraic representation of `RBTBasic`. Instead, a practical method for navigating within an algebraic tree is using paths, i.e., lists of *left* or *right* flags. Since data refinement can refine state only (not inputs or outputs), the states of `RBTBasic` and `RBTHeap` are augmented with auxiliary paths and references, respectively, to circumvent this restriction. As a consequence, the interfaces of `RBTBasic` and `RBTHeap` are extended by operations for navigating through the tree, like "*move up one node*" or "*move to the left child*", and tree manipulations are performed at the locations of the auxiliary paths/references only.

Correctness of Algebraic Red-Black Trees

Red-black trees can be specified algebraically as polymorphic free data type rbtree($'a$), using a constant constructor `SENTINEL` (representing the leaves of the tree) and a non-constant constructor `node`.

$$\begin{aligned}\textbf{data}\quad \mathsf{rbtree}('a) \;&=\; \mathtt{SENTINEL} \mid \mathtt{node}(\mathtt{.elem} : {'a}\ ;\ \mathtt{.color} : \mathsf{rbcolor}\ ;\\ &\qquad\qquad \mathtt{.left} : \mathsf{rbtree}('a)\ ;\ \mathtt{.right} : \mathsf{rbtree}('a))\\ \textbf{data}\quad \mathsf{rbcolor} \;&=\; \mathtt{RED} \mid \mathtt{BLACK}\end{aligned}$$

Nodes have a color (either `RED` or `BLACK`, defined by the enumeration type rbcolor), a left and a right subtree, and an element of generic type $'a$. These fields can be accessed via the postfix selector functions `.elem`, `.color`, `.left`, and `.right`. A type variable $'a$ for the type of elements stored in the tree is used in the definition. So in principle, the data type can be used with any element type.

However, to express the properties of binary search trees, the generic, totally ordered element type tord is used like in `RBSet`. The complete algebraic state is encapsulated in the `RBTBasic` component, which is used by the client component `RBTree`.

component `RBTBasic`
state $\mathit{rbt} : \mathsf{rbtree}(\mathsf{tord}),\quad \mathit{curPath} : \mathsf{list}(\mathsf{lrdesc}),\quad \mathit{auxPath} : \mathsf{list}(\mathsf{lrdesc})$

Besides the algebraic red-black-tree *rbt*, the state contains two paths *curPath* and *auxPath* to store locations within *rbt*. Most of the time *curPath* is used only, but for removal, it is necessary to store a second path *auxPath* that points to the element after the deleted one. The implementation in `RBTHeap` replaces these paths by two references that point to a heap storing individual tree nodes, as we will see later in this section. Paths are modeled as lists (see Sec. 2.2) instantiated with an enumeration type lrdesc.

$$\textbf{data}\quad \mathsf{lrdesc} \;=\; \mathtt{LEFT} \mid \mathtt{RIGHT}$$

Various operations are defined for accessing an rbtree with paths, for example $p \in \mathit{rbt}$ for checking a path p to be in a tree *rbt* (and pointing to an actual node), $\mathit{rbt}\texttt{[}p\texttt{]}$ for selecting the subtree at a path p, or $\mathit{rbt}\texttt{[}p, \mathit{rbt}'\texttt{]}$ for updating the tree at a path p with a new tree rbt'. All these operations are defined recursively over the structure of the tree resp. the path, e.g., the predicate $p \in \mathit{rbt}$ is defined by the following axioms.

$\vdash \neg\ p \in \texttt{SENTINEL}$

$\vdash \texttt{[]} \in \texttt{node}(elem, col, left, right)$

$\vdash \texttt{LEFT + } p \in \texttt{node}(elem, col, left, right) \leftrightarrow p \in left$

$\vdash \texttt{RIGHT + } p \in \texttt{node}(elem, col, left, right) \leftrightarrow p \in right$

Initialization and recovery of `RBTBasic` just sets *rbt* to a `SENTINEL` and *curPath* resp. *auxPath* to the empty path `[]`. The reading operations (on the right of Fig. 3.8) are implemented by checking the tree's root, performing a standard binary search, or traversing to the leftmost or rightmost element, respectively. The more complex operations are inserting or removing an element, as they manipulate the tree and thus require performing fixup mechanisms to ensure the balance of the tree.

Figure 3.10 lists the implementation of **rbtree_insert#** in the `RBTree` component and shows how the state is modified via the interface of `RBTBasic`. For primitive `RBTBasic` operations, the comments in green show the statements of the operation in front of it. All operations start by resetting the paths to point to the root, i.e., to the empty list `[]` (line 1). Then the tree is traversed to an element (or a position) of interest (the loop at lines 5-17); for insertion, a binary search for the element to be inserted (*elem*) is performed. For this purpose, the element at *curPath* is read (line 6) and compared with elem in each iteration. If *elem* is found (line 7), the operation can be aborted with *exists* := `true` since no duplicates are inserted into the tree. Otherwise, *curPath* is extended by either `LEFT` or `RIGHT` depending on the comparison between $elem_0$ and *elem* (line 9-13). When a `SENTINEL` is reached, the search is stopped (line 15). If *elem* was not found in the tree, it is added at the current position inside of a new node without children (line 20). Finally, the routine **rbtree_insertFixup#** is called to fix up the tree as the insertion may have broken the red-black tree properties and the tree may need to be re-balanced (line 21).

During the traversal of the tree, *auxPath* is updated alongside *curPath* to point to the parent of the current position (lines 10 and 13). Although this is unnecessary for the implementation on algebraic trees (the parent path can simply be obtained by dropping the last element of *curPath*), it is required for the pointer-based implementation. Since *curRef* has reached a `null` reference, the **add#** of `RBTHeap` could not link the new node at the right place as no information about the parent of the leaf is available. This is why the binary search stores the reference to the parent of *curRef* in *auxRef* (lines 10 and 13) so that **add#** can update the child references of the parent to point to the new node.

The implementation of **rbtree_remove#** in the `RBTree` component is shown in Figure 3.11. Again, the paths are reset initially (line 1) and the tree is traversed. For removal, the element to be removed (*elem*) is searched and *curPath* will be updated to point to *elem* if it is found using an auxiliary procedure When a leaf is reached (checked in line 3), the search is stopped and the removal is aborted with *exists* := `false` as there is no element to delete (line 31). If *elem* was found, the element must be replaced in order to restore the red-black tree properties. In case the node has a leaf as left or right child, substitution of the node is performed simply by replacing it with the other child (line 8 resp. 14). Otherwise, *curPath* is stored in *auxPath* (line 16) and is then updated by **rbtbasic_right#** and **rbtree_leftmost#** to point to

```
rbtree_insert#($elem$; ; $exists$)
  interface
  postcondition    $\texttt{elems}(rbt) = \texttt{elems}(rbt') \mathbin{\texttt{++}} elem$
                   $\land\ (exists \leftrightarrow elem \in rbt')$;
{
  rbtbasic_reset#(); // $curPath$ := [], $auxPath$ := []
  $exists := \texttt{false}$;
  let $stop = \texttt{false},\ elem_0 = ?$ in {
    rbtbasic_isEmpty#(; ; $stop$); // $stop := (rbt = \texttt{SENTINEL})$
    while $\neg\, stop \land \neg\, exists$ do {
      rbtbasic_getElem#(; ; $elem_0$); // $elem_0 := rbt\texttt{[}curPath\texttt{].elem}$
      if $elem = elem_0$ then $exists := \texttt{true}$
      else {
        if $elem < elem_0$ then
          rbtbasic_leftWithParent#()
          // $auxPath := curPath,\ curPath := curPath \mathbin{\texttt{+}} \texttt{LEFT}$
        else
          rbtbasic_rightWithParent#();
          // $auxPath := curPath,\ curPath := curPath \mathbin{\texttt{+}} \texttt{RIGHT}$
        rbtbasic_isLeaf#(; ; $stop$); // $stop := (rbt\texttt{[}curPath\texttt{]} = \texttt{SENTINEL})$
      }
    };
    if $\neg\, exists$ then {
      rbtbasic_add#($elem$); // add new node with $elem$ at $curPath$
      rbtree_insertFixup#();
      // restore balance and red-black tree properties
    }
  }
}
```

Figure 3.10: `RBTree` procedure for inserting an element *elem*.

the next greater element. This element is the minimal element of the right subtree of *elem* and thus cannot have a left child ($curPath + \texttt{LEFT}$ points to a `SENTINEL`). Therefore, the element can be moved to *auxPath* (line 21) and the tree at *curPath* can be replaced with its right child (line 20). Depending on the specific situation before the replacement, the tree may need to be fixed afterward: the operations **rbtbasic_replLeft#** and **rbtbasic_replRight#** return via *doFix* whether fixing is necessary. The routine **rbtree_removeFixup#** is then called to restore the balance of the tree starting at *curPath*.

The removal algorithm is essentially the same as the one in [28]. However, we implement `SENTINEL` nodes as null pointers instead of using a dummy node that would be necessary to get the parent of a leaf. This results in our operations already working on the parent node, which is why we have to pass certain information explicitly, for example, *isLeftChild* in Figure 3.11 signals whether the left or right child has been deleted. On the other hand, we can omit certain cases by detecting them before an operation is performed. As an example of this, our **rbtree_removeFixup#** is not called if *curPath* points to a leaf or if there is nothing to fix. Otherwise the various cases and rotations are identical to [28].

```
rbtree_remove#($elem$; ; $exists$)
  interface
  postcondition    $\mathtt{elems}(rbt) = \mathtt{elems}(rbt') \mathbin{\texttt{--}} elem$
                 $\wedge\ (exists \leftrightarrow elem \in \mathtt{elems}(rbt'))$;
{
  rbtbasic_reset#(); // $curPath := \texttt{[]},\ auxPath := \texttt{[]}$
  rbtree_search#($elem$); // sets $curPath$ to position of $elem$ or to a leaf
  rbtbasic_isLeaf#(; ; $exists$); // $exists := (rbt\texttt{[}curPath\texttt{]} = \mathtt{SENTINEL})$
  if $\neg$ $exists$ then {
    let $doFix$ = ?, $isLeftChild$ = ?, $cond$ = ? in {
      rbtbasic_hasLeft#(; ; $cond$); // $cond := curPath + \mathtt{LEFT} \in rbt$;
      if $\neg$ $cond$ then {
        rbtbasic_replRight#(; ; $doFix$, $isLeftChild$);
        // replace node at $curPath$ with its right child
        // and move $curPath$ up one node
      } else {
        rbtbasic_hasRight#(; ; $cond$); // analogous to hasLeft
        if $\neg$ $cond$ then {
          rbtbasic_replLeft#(; ; $doFix$, $isLeftChild$);
        } else {
          rbtbasic_initAux#(); // $auxPath := curPath$
          rbtbasic_right#(); // $curPath := curPath + \mathtt{RIGHT}$
          rbtree_leftmost#(); // extend $curPath$ to leftmost inner node
          rbtbasic_getElem#(; ; $elem$); // $elem := rbt\texttt{[}curPath\texttt{]}\mathtt{.elem}$
          rbtbasic_replRight#(; ; $doFix$, $isLeftChild$);
          rbtbasic_setElemAux#($elem$); // $rbt\texttt{[}auxPath\texttt{]}\mathtt{.elem} := elem$
        }
      };
      if $doFix$ then {
        rbtree_removeFixup#($isLeftChild$);
        // restore balance and red-black tree properties
      };
      $exists$ := true;
    }
  } else {
    $exists$ := false // element was not found
  }
}
```

Figure 3.11: RBTree procedure for removing an element.

For verification, the functional properties must be specified as invariant of RBTree. In between interface calls, *rbt* must be a valid red-black tree and must be a valid search tree, i.e., its elements must be ordered.

> **component** RBTree **using** RBTBasic
> **invariants** $\mathtt{isRbtree}(rbt) \wedge \mathtt{isOrdered}(rbt)$

A non-empty red-black tree is characterized by three main properties: the root of the tree is colored BLACK, both children of a RED node have to be BLACK, and each path of any node to a leaf must contain the same number of BLACK nodes.

$$\vdash \mathtt{isRbtree}(rbt) \leftrightarrow \\ \quad rbt = \mathtt{SENTINEL} \vee \big(\mathtt{redCorrect}(rbt, \mathtt{RED}) \wedge \mathtt{sameBlacks}(rbt)\big)$$

The predicate $\mathtt{redCorrect}(rbt, col)$ (col is the color of the parent node) specifies the first two properties, $\mathtt{sameBlacks}(rbt)$ the last. Both are defined recursively over the structure of the tree, as is $\mathtt{isOrdered}(rbt)$. The definition of redCorrect is given by the following axioms.

$$\vdash \mathtt{redCorrect}(\mathtt{SENTINEL}, col)$$

$$\vdash \mathtt{redCorrect}(\mathtt{node}(e, \mathtt{BLACK}, left, right), col) \leftrightarrow \\ \quad \mathtt{redCorrect}(left, \mathtt{BLACK}) \wedge \mathtt{redCorrect}(right, \mathtt{BLACK})$$

$$\vdash \mathtt{redCorrect}(\mathtt{node}(e, \mathtt{RED}, left, right), col) \leftrightarrow \\ \quad col = \mathtt{BLACK} \wedge \mathtt{redCorrect}(left, \mathtt{RED}) \wedge \mathtt{redCorrect}(right, \mathtt{RED})$$

For sameBlacks, the *black heights* of the children of a node are compared.

$$\vdash \mathtt{sameBlacks}(\mathtt{SENTINEL})$$

$$\vdash \mathtt{sameBlacks}(\mathtt{node}(e, col, left, right)) \leftrightarrow \\ \quad\quad \mathtt{countBlacks}(left) = \mathtt{countBlacks}(right) \\ \quad \wedge \mathtt{sameBlacks}(left) \wedge \mathtt{sameBlacks}(right)$$

The function countBlacks calculates the number of black nodes on the path to the minimal element. Note that this path is chosen arbitrary and any other path could be used for the calculation when used within the recursive definition of sameBlacks.

$$\vdash \mathtt{countBlacks}(\mathtt{SENTINEL}) = 0$$
$$\vdash \mathtt{countBlacks}(\mathtt{node}(e, col, left, right)) = \\ \quad \mathtt{countBlacks}(left) + (col = \mathtt{BLACK} \supset 1 \;;\; 0)$$

While $\mathtt{isOrdered}(rbt)$ is maintained quite easily by the operations of RBTree, e.g., **insert#** adds the new element directly at a position that maintains the order property, the fixing mechanisms for re-establishing $\mathtt{isRbtree}(rbt)$ after insertion or removal are complex. So the main proof effort is to show that these mechanisms are actually correct. In order to keep the proof size manageable, the procedures are split into several subroutines, which are augmented with contracts to prove their correctness in isolation.

```
  rbtree_insertFixup#()
    auxiliary
    precondition  $curPath \in rbt \wedge \texttt{isRbtreeI}(rbt, curPath) \wedge \texttt{isOrdered}(rbt)$;
    postcondition  $\texttt{isRbtree}(rbt) \wedge \texttt{isOrdered}(rbt) \wedge \texttt{elems}(rbt) = \texttt{elems}(rbt\grave{})$;
  {
1   let $continue$ = ?, $left$ = ?, $parLeft$ = ?, $uncleRed$ = ? in {
2     rbtbasic_parentIsRed#(; ; $continue$);
3     // $continue := (curPath \neq \texttt{[]} \wedge rbt\texttt{[}curPath\texttt{.butlast].color} = \texttt{RED})$
4     while $continue$ do {
5       rbtbasic_isLeft#(; ; $left$); // $left := (curPath\texttt{.last} = \texttt{LEFT})$
6       rbtbasic_parentIsLeft#(; ; $parLeft$);
7       // $parLeft := (curPath\texttt{.butlast.last} = \texttt{LEFT})$
8       if $parLeft$ then {
9         rbtbasic_setRightUncleBlack#(; ; $uncleRed$);
10        // $uncleRed := (rbt\texttt{[}curPath\texttt{.butlast.butlast].right.color} = \texttt{RED})$
11        // $rbt\texttt{[}curPath\texttt{.butlast.butlast].right.color} := \texttt{BLACK})$
12        if $uncleRed$ then
13          rbtbasic_insertFixup1L#()
14        else
15          rbtbasic_insertFixup2L#($left$);
16      }
17      else {
18        ... // symmetric case for right parent
19      };
20      rbtbasic_parentIsRed#(; ; $continue$);
21      // $continue := (curPath \neq \texttt{[]} \wedge rbt\texttt{[}curPath\texttt{.butlast].color} = \texttt{RED})$
22    };
23    rbtbasic_setRootBlack#(); // $rbt\texttt{.color} := \texttt{BLACK}$
24  }
  }
```

Figure 3.12: RBTree procedure for restoring the red-black tree properties after an insertion.

As an example, Figure 3.12 lists the procedure **rbtree_insertFixup#** that fixes the tree after an insertion. It is an auxiliary procedure, so it cannot be called by client components but only by procedures of RBTree. The routine starts at the insertion point (at *curPath*) and traverses the tree step by step upwards to the root, rotating and recoloring nodes along the way. This is done with a loop (lines 4-22) as long as the parent of *curPath* is RED or the root is not reached (line 2 and 20, p.butlast returns the path p without its last segment, which in turn can be selected via p.last). In each iteration different modifications around *curPath* are made depending on whether the node at *curPath* is a left child (lines 5 and 15), its parent is a left child (lines 6 and 8), or its uncle is RED (lines 9 and 12).

Basically each case performs different recolorations and rotations as shown exemplary in Fig. 3.13 for the case when the parent is a left child and the uncle is BLACK. In this case up to two rotations are performed: potentially a left-rotation at the parent of *curPath* (lines 1-4) and then a right-rotation of its grandparent (line 9) alongside recolorations of the parent and grandparent (lines 5 and 7). A rotation rearranges a subtree as shown in Fig. 3.14. Here, the situation before and after the execution of line 9 in Fig. 3.13 is depicted. The parent P and grandparent GP of

```
rbtbasic_insertFixup2L#($left$)
  auxiliary
  precondition  isRbtreeI($rbt, curPath$) $\wedge$ isOrdered($rbt$) $\wedge \ldots$
  postcondition   isRbtree($rbt$) $\wedge$ isOrdered($rbt$)
                  $\wedge$ elems($rbt$) = elems($rbt$`) $\wedge$ $curPath$ = $curPath$`.butlast;
{
  if $\neg$ $left$ then {
    rbtbasic_up#(); // $curPath$ := $curPath$.butlast
    rbtbasic_rotateLeft#(); // left-rotation at $curPath$
  };
  rbtbasic_setColorParent#(BLACK);
  // $rbt$[$curPath$.butlast].color := BLACK
  rbtbasic_setColorGrandparent#(RED);
  // $rbt$[$curPath$.butlast.butlast].color := RED
  rbtbasic_rotateRightGrandparent#();
  // right-rotation at $curPath$.butlast.butlast
}
```

Figure 3.13: Exemplary insertion fixup procedure of RBTree.

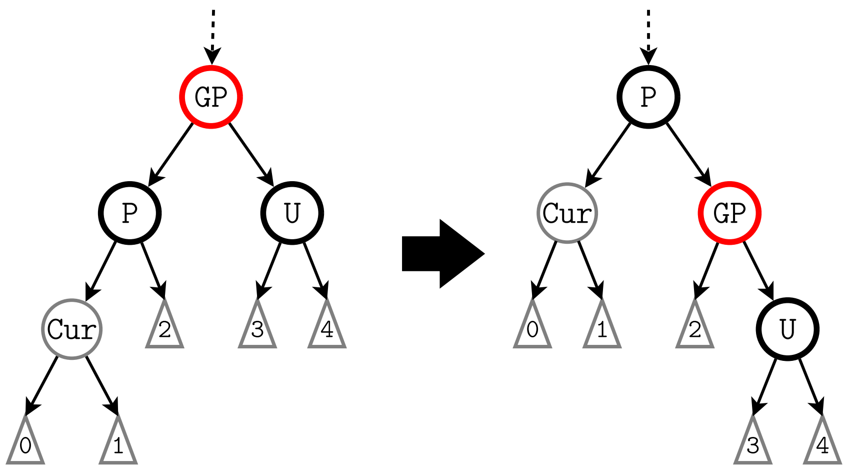


Figure 3.14: Right-rotation at the grandparent (GP) of the current node (Cur).

the node Cur at *curPath* have already been recolored accordingly, and now a right-rotation at GP is performed. This moves Cur up one level (*curPath* is shortened by one as well) and makes P to the new root of the rotated subtree. In this example, the loop body of **rbtree_insertFixup#** would end with a parent of *curPath* that is already BLACK and thus the fixing of *rbt* is completed.

For the verification of **rbtree_insert#**, the auxiliary routines in Fig. 3.12 and Fig. 3.13 are augmented with contracts. When **rbtree_insertFixup#** is called, the tree hast to be in order (isOrdered(rbt)) and *curPath* must point to a node within the tree ($curPath \in rbt$). The tree may violate the red-black tree properties, i.e., isRbtree(rbt) does not have to hold, but only at the insertion point. To express this circumstance, a weaker version of isRbtree is specified for the insertion invariant proofs (the corresponding predicates are indicated with an attached `I`).

$$\vdash \texttt{isRbtreeI}(rbt,\ p) \leftrightarrow$$
$$\texttt{sameBlacks}(rbt) \wedge \texttt{redCorrectI}(rbt, (p = \texttt{[]} \supset \texttt{BLACK}\ ;\ \texttt{RED}),\ p)$$

After insertion, the `sameBlacks` property is still valid since **rbtbasic_add#** (line 19 in Fig. 3.10) adds the new node colored `RED`. On the other hand, the `redCorrect` property might have been violated by the insertion (if the parent of the inserted node is `RED`, then there are two consecutive `RED` nodes). The weaker predicate $\texttt{redCorrectI}(rbt, col, p)$ states that rbt complies with `redCorrect` except for the node at path p, which is `RED` and might have a `RED` parent.

$$\vdash \texttt{redCorrectI}(\texttt{SENTINEL},\ col,\ p)$$

$$\vdash \texttt{redCorrectI}(\texttt{node}(e, col, left, right),\ col_0,\ \texttt{[]}) \leftrightarrow$$
$$col = \texttt{RED} \wedge \texttt{redCorrect}(left, \texttt{RED}) \wedge \texttt{redCorrect}(right, \texttt{RED})$$

$$\vdash \texttt{redCorrectI}(\texttt{node}(e, \texttt{BLACK}, left, right),\ col,\ \texttt{LEFT + } p) \leftrightarrow$$
$$\texttt{redCorrectI}(left, \texttt{BLACK}, p) \wedge \texttt{redCorrect}(right, \texttt{BLACK})$$

$$\vdash \texttt{redCorrectI}(\texttt{node}(e, \texttt{RED}, left, right),\ col,\ \texttt{LEFT + } p) \leftrightarrow$$
$$\texttt{redCorrectI}(left, \texttt{RED}, p) \wedge \texttt{redCorrect}(right, \texttt{RED}) \wedge col = \texttt{BLACK}$$

// symmetric cases for the path `RIGHT +` p

The col argument is again used to pass the color of the parent for the recursive definition of the predicate. For the case when $p = \texttt{[]}$, i.e., when p points to the root, the value `BLACK` is used for col as the root is then `RED` and needs to be recolored. Otherwise the value `RED` enforces that the root node is colored `BLACK`.

The proof then involves showing that after each iteration of the insertion fixup loop, $\texttt{isRbtreeI}(rbt,\ curPath)$ holds again for the shortened $curPath$ or that consistency of rbt is restored and $\texttt{isRbtree}(rbt)$ holds, e.g., after the execution of the procedure in Fig. 3.13. Note that $\texttt{isOrdered}(rbt)$ is maintained by all auxiliary procedures as rotations do not affect the order of the tree.

Fixing up the tree after removal performs similar tasks, however, more cases have to be considered. For the invariant proofs, analogous predicates must be defined that allow the tree to be characterized during removal when some red-black tree properties are violated at a specific location in the tree. In contrast to `isRbtreeI`, the corresponding predicate $\texttt{isRbtreeD}(rbt,\ p)$ also allows a violation of the `sameBlacks` property at path p besides the violation of `redCorrect`.

In order to show that the red-black tree implementation can be used as an implementation of sets, the refinement `RBTree(RBTBasic)` $\leq$ `RBSet` is proven by the following forward simulation.

data refinement `RBTree(RBTBasic)` $\leq$ `RBSet`
abstraction relation $rbs = \texttt{elems}(rbt)$

where the function `elems` calculates the set of elements stored in the red-black tree.

$$\vdash \texttt{elems}(\texttt{SENTINEL}) = \emptyset$$

$$\vdash \texttt{elems}(\texttt{node}(e,\ col,\ l,\ r)) = \big(\texttt{elems}(l) \cup \texttt{elems}(r)\big)\ \texttt{++}\ e$$

This simple abstraction allows to encode the set modifications of `RBSet` into the contracts of `RBTree`. For example, the contract of **rbtree_insert#** in Fig. 3.10 states that *elem* is added to the elements of the tree ($\texttt{elems}(rbt) = \texttt{elems}(rbt') \texttt{ ++ } elem$ where rbt' denotes the value of *rbt* just before the execution of the procedure). This modification happens within **rbtbasic_add#**, the contracts of all other modifying auxiliary procedures, e.g., **rbtree_insertFixup#**, ensure that they do not change set of elements stored in the tree ($\texttt{elems}(rbt) = \texttt{elems}(rbt')$). Similar contracts are given for the other interface procedures which allow to prove the refinement mainly by applying these contracts. Note that the refinement proofs do not require the invariant $\texttt{isRbtree}(rbt)$ (an unbalanced tree would also refine a set correctly) but do require $\texttt{isOrdered}(rbt)$ since it could not be proven that the search within the tree is correct (and thus the correctness of **rbtree_remove#**, **rbtree_lookup#**, **rbtree_getMin#**, and **rbtree_getMax#** could not be shown).

Proving the refinement proof obligations of Thm. 1 together with the invariant proof obligations according to Lem. 1 ensures that the component `RBTree`(`RBTBasic`) is a correct red-black tree implementation maintaining all properties of red-black trees.

Theorem 3 (Correctness of Algebraic Red-Black Trees)**.** *The sequential component* `RBTree`(`RBTBasic`) *is a correct non-destructive implementation of red-black trees.* □

For none of the proofs it is necessary to reason about the heap implementation. In particular, the main invariant properties `isOrdered` and `isRbtree` are proved solely over algebraic trees. What remains to prove is that a pointer-based implementation is a correct refinement of `RBTBasic`.

Exchanging Algebraic Red-Black Trees with a Heap Implementation

Since the component `RBTHeap` is a refinement of `RBTBasic`, it implements the same interface for fine-grained manipulations of the tree data structure. While `RBTBasic` uses two paths to point to the locations tree manipulations are performed, the implementation in `RBTHeap` uses references that point to a heap storing individual tree nodes. The nodes of the implementation use parent pointers, so shortening or lengthening one of the two paths by one (which are operations of `RBTBasic`) can be implemented by simply dereferencing a pointer.

More precisely, the state of `RBTHeap` contains a heap *rbh* (see Sec. 2.5) and a pointer *rootRef* to the root of the tree, together with pointers *curRef* and *auxRef* matching *curPath* and *auxPath*, respectively.

component `RBTHeap`
state rbh : heap(rbnode), $rootRef$: ref, $curRef$: ref, $auxRef$: ref

The heap stores nodes of type rbnode, which contain an element and a color like the nodes of rbtree but use references that point to their left and right subtrees. A parent pointer is added to allow efficient traversal upwards in the tree.

data rbnode = `node(. .elem` : tord; `. .color` : rbcolor; `. .parent` : ref;
`. .left` : ref; `. .right` : ref)

```
  **rbtbasic_rotateRight#**$(p)$
    **auxiliary**
    **precondition** $p$ + LEFT $\in rbt$;
  {
1   **let** $rbt_0 = rbt$[$p$] **in**
2   **let** $rbt_1 = rbt_0$.left **in**
3   **let** $rbt_2 =$ node($rbt_0$.elem, $rbt_0$.color, $rbt_1$.right, $rbt_0$.right) **in**
4     $rbt$[$p$] $:=$ node($rbt_1$.elem, $rbt_1$.color, $rbt_1$.left, $rbt_2$)
  }
```

Figure 3.15: Procedure for right-rotations at a path p of component `RBTBasic`.

Most of the operations of `RBTBasic` are just single assignments, for example recolorings of the node at *curPath* or one of its relative, or changes of *curPath* or *auxPath*. `RBTHeap` implements these operations analogously with lookups at *curRef* and updates of *curRef* or *auxRef* by following the parent- or child-pointers. So instead of selecting a subtree at *curPath*, which requires a traversal of the complete path, the node at *curRef* is accessed by dereferencing the pointer ($rbh[curRef]$). Analogously, instead of adjusting the paths, e.g., with *curPath* $:=$ *curPath* + `RIGHT`, pointers are updated by following the references of the current node, e.g., with *curRef* $:=$ *rbh*`[`*curRef*`].right`. The only more complex operations are rotations used in the fixing routines like the one shown in Fig. 3.14. The `RBTBasic` interface provides operations for rotations at different locations (at *curPath*, *auxPath*, or one of their relatives), all of which use the auxiliary procedure listed in Fig. 3.15 or a symmetric version for left-rotations. The operation takes a path p as an argument and performs a right-rotation at this location. For example, the procedure **rbtbasic_rotateRightGrandparent#** used in Fig. 3.13 calls this operation with the argument *curPath*`.butlast.butlast`. It selects the subtree at p and builds the rotated subtree which is then inserted at p again (the program *rbt*`[`p`]` $:=$ rbt_0 is an abbreviation for *rbt* $:=$ *rbt*`[`p, rbt_0`]`, which replaces *rbt*`[`p`]` with rbt_0 in *rbt*).

Figure 3.16 shows the corresponding implementation of `RBTHeap`. Instead of a path, it takes a reference *ref* as an input, so **rbtheap_rotateLeftGrandparent#** calls the operation with *rbh*`[`*rbh*`[`*curRef*`].parent].parent`. The heap implementation performs the rotation by updating the pointers of the node at *ref* as well as those of its parent and left child. First, the link between the node at *ref* and its new left child is established (lines 2-4). Then the link between the new root of the subtree (*lRef*) and its new parent is created (lines 6-14). And finally, *ref* is linked to *lRef* as its new right child (lines 16 and 17). In contrast to the algebraic variant in Fig. 3.15 (the assignment in line 4 would copy the whole tree *rbt*), all updates are destructive. For example in C, the assignment in line 2 corresponds to a statement `ref->left = lRef->right` where both `ref` and `lRef` as well as the fields `left` and `right` are pointers to a struct `rbnode`.

```
**rbtheap_rotateRight#**$(ref)$
  **auxiliary**
  **precondition** $ref \in rbh \land rbh$[$ref$].left $\in rbh$;
{
  let $lRef = rbh$[$ref$].left in {
    $rbh$[$ref$].left := $rbh$[$lRef$].right;
    if $rbh$[$lRef$].right $\neq$ null then {
      $rbh$[$rbh$[$lRef$].right].parent := $ref$;
    };
    if $lRef \neq$ null then $rbh$[$lRef$].parent := $rbh$[$ref$].parent;
    if $rbh$[$ref$].parent $\neq$ null then {
      if $ref = rbh$[$rbh$[$ref$].parent].right then {
        $rbh$[$rbh$[$ref$].parent].right := $lRef$;
      } else {
        $rbh$[$rbh$[$ref$].parent].left := $lRef$;
      };
    } else {
      $rootRef$ := $lRef$;
    };
    $rbh$[$lRef$].right := $ref$;
    if $ref \neq$ null then $rbh$[$ref$].parent := $lRef$;
  }
}
```

Figure 3.16: Procedure for right-rotations at a reference *ref* of component `RBTHeap`.

The refinement is proven using Separation Logic (see Sec. 2.5) and the following abstraction that does not refer to any red-black tree properties.

data refinement `RBTHeap` $\leq$ `RBTBasic`

abstraction relation $\mathtt{abs}(rootRef,\ \mathtt{null},\ rbt)(rbh)$

$$\land\ rbh[rootRef,\ curPath] = curRef$$
$$\land\ rbh[rootRef,\ auxPath] = auxRef$$

The references *curRef* and *auxRef* must point to the same locations in tree as *curPath* and *auxPath* ($rbh[ref,\ p]$ returns the reference that is reached when the pointer structure in *rbh* is traversed following the path p, starting at *ref*). The heap predicate $\mathtt{abs} : (\mathsf{ref} \times \mathsf{ref} \times \mathsf{rbtree(tord)}) \to \mathsf{heap(rbnode)} \to \mathsf{bool}$ abstracts the pointer tree in *rbh* starting at *rootRef* to the algebraic tree *rbt*. The second ref argument is used to specify the parent of the root, for the complete tree the parent pointer of the root node should be `null`. `abs` is defined recursively over the structure of *rbt*:

$$\vdash \mathtt{abs}(rootRef,\ pRef,\ \mathtt{SENTINEL})(rbh) \leftrightarrow rootRef = \mathtt{null} \land rbh = \emptyset$$

$$\begin{aligned}\vdash\ & \mathtt{abs}(rootRef,\ pRef,\ \mathtt{node}(e,\ col,\ left,\ right))(rbh) \leftrightarrow \\ & \quad \exists\ lRef,\ rRef. \\ & \qquad \big(\ (rootRef \mapsto \mathtt{node}(e,\ col,\ pRef,\ lRef,\ rRef)) \\ & \qquad\ * \mathtt{abs}(lRef,\ rootRef,\ left) \\ & \qquad\ * \mathtt{abs}(rRef,\ rootRef,\ right)\big)(rbh)\end{aligned}$$

For a SENTINEL, the heap *rbh* must be empty. This strong condition ensures the absence of memory leaks since one has to prove that all nodes have been deallocated when they are removed from the tree. For a node, the heap is separated into three disjoint parts as usual: a root node containing the same element and color as the algebraic node as well as two trees that abstract to the left and right algebraic subtree, respectively.

The proof exploits that each operation modifies at most one location inside of the tree. This allows to split up the abstraction at this location, prove that the operation has the expected local behavior (e.g., that it rotates the referenced subtree correctly), and then merge the abstraction again with the updated subtree. For this, two fundamental theorems were formulated. The first theorem splits the abstraction of a tree *rbt* at a path *p*.

$$\begin{aligned} \vdash p \in rbt \to (\mathtt{abs}(rootRef,\ pRef,\ rbt)(rbh) \leftrightarrow \\ \exists\ pthRef,\ pPthRef.\quad\quad\quad\quad\quad\quad\quad \\ \big(\ \mathtt{abspath}(rootRef,\ pRef,\ rbt,\ p,\ pthRef,\ pPthRef) \\ *\ \mathtt{abs}(pthRef,\ pPthRef,\ rbt\,\mathtt{[}p\mathtt{]})\big)(rbh)) \end{aligned}$$

The auxiliary heap predicate abspath is a weaker version of abs: the tree represented in *rbh* must match *rbt* except for the subtree starting at *p*. The references *pthRef* and *pPthRef* are used to fix the references of the root of the subtree and its parent. Thus, the split off subtree $rbt[p]$ can be abstracted separated using abs with *pthRef* as root and *pPthRef* as parent reference. Conversely, the second theorem reconnects a detached subtree rbt_0 at path *p* with the original tree *rbt*.

$$\begin{aligned} \vdash\quad & \big(\ \mathtt{abspath}(rootRef,\ pRef,\ rbt,\ p,\ pthRef,\ pPthRef) \\ & *\ \mathtt{abs}(pthRef,\ pPthRef,\ rbt_0)\big)(rbh) \\ \to\ & \mathtt{abs}(rootRef,\ pRef,\ rbt\,\mathtt{[}p, rbt_0\mathtt{]})(rbh) \end{aligned}$$

The theorem allows to attach arbitrary trees rbt_0 (they do not necessarily have to be related to the originally separated subtree $rbt\,\mathtt{[}p\mathtt{]}$), however, rbt_0 typically results from a simple modification of $rbt\,\mathtt{[}p\mathtt{]}$ like a recoloration or a rotation.

The abstraction relation uses ordinary conjunctions, which supports updating the second and third conjuncts on heap updates via suitable rewrite rules: modifications happen either at one of the two paths or below them (in the latter case no update is necessary at all). Using the abspath heap predicate, rules like the following can be defined for assignments to the heap (here given for the case that path *p* is a prefix of path p_0 and $p_0 \in rbt$).

$$\frac{\big(\mathtt{abspath}(r_0, r_1, rbt, p_0, r_2, r_3)\ *\ r \mapsto nd\ *\ hP\big)(rbh),\ \ rbh\,\mathtt{[}r_0,\ p\mathtt{]} = r\ \vdash \langle\!|\alpha|\!\rangle\,\varphi}{\begin{array}{r}\big(\mathtt{abspath}(r_0, r_1, rbt, p_0, r_2, r_3)\ *\ r \mapsto nd_0\ *\ hP\big)(rbh), \\ rbh\,\mathtt{[}r_0,\ p\mathtt{]} = r\end{array} \vdash \langle\!| rbh\,\mathtt{[}r\mathtt{]} := nd;\ \alpha |\!\rangle\,\varphi}$$

The rule propagates the formula $rbh[r_0,\ p] = r$ (which typically corresponds to the second or third conjunct of the abstraction relation) over an assignment to the heap location *r*, exploiting that the location is separated from the part of the tree that *p* points into. On the other hand, most Separation Logic based provers, e.g., VeriFast [68] or Viper [86], support separating conjunction only, which would require to define several versions of `abs` with additional paths and references as arguments, depending on which of them is contained in a subtree.

Proving the refinement `RBTHeap` $\leq$ `RBTBasic` with the forward simulation given above guarantees memory-safety of the heap implementation (the lookup and update operations of heaps are partial with domains $\lambda\ h, r.\ r \neq$ `null` $\wedge\ r \in h$, so the program proofs also ensure the absence of invalid pointer dereferencing), and by compositionality of refinement (Thm. 2) together with Thm. 3 we get the correctness of the destructive red-black tree implementation `RBTree`(`RBTHeap`).

Theorem 4 (Correctness of Destructive Red-Black Trees)**.** *The sequential component* `RBTree`(`RBTHeap`) *is a correct, memory-safe destructive implementation of red-black trees.*

□

Related Work

There are several other works that do verification of red-black trees. [5, 44, 95] are complete verifications of algebraic (nondestructive) implementations, that correspond to the upper refinement `RBTree`(`RBTBasic`) $\leq$ `RBSet`.

Partial verifications, where the emphasis is on automation, are [22] (insertion without establishing `sameBlacks`) and [89] (just proving the `isOrdered` property).

The only complete verification of destructive code we are aware of is described in [7]. The implementation is directly in Java, the main routines use recursion and no parent pointers, so it is somewhat less efficient than our C implementation which does not need a recursion stack. In addition to verifying the main operations shown in the `RBSet` interface, a concurrent routine for merging red-black trees, which is part of the NLNet Labs Name Server Daemon [90] and uses lists as an intermediate representation, was verified with VerCors [12]. Verification is directly done using Separation Logic, so the proof has to mix red-black tree properties and properties of pointer structures. The proofs of the recursive implementation in VerCors require fewer auxiliary functions and predicates than we needed for proving our iterative implementation in KIV: we have to give loop invariants over the whole tree *rbt* and hence use auxiliary predicates to express something like "*rbt* is valid except for a violation of `redCorrect` at path *p*" while the recursive contracts in VerCors only have to consider the (sub-)tree that is passed as an argument. Hence, we have also given plenty of inductive lemmas, while almost none were necessary for the proof in VerCors.

With VerCors being an automatic verifier backed by an SMT solver, proofs are guided by adding suitable annotations to the programs instead of directly interacting with a GUI during a proof as in KIV. Overall, the user input necessary for the final proof seems less in VerCors than in KIV. From the data given, the effort was somewhat higher than with our approach, however, the authors do not mention how this effort is distributed between the core red-black tree verification and the verification of the merging algorithm, which makes an exact comparison difficult.

Partial verification of destructive code can also be found in [36] (C-Code), [8] (automatic analysis of a specific encoding as graph transformations) and [34] (SPARKS, a subset of Ada). All three of these have analyzed insertion only and left away the `sameBlacks` property. The last is interesting since it uses an array-representation of red-black trees that would be suitable for real-time use (the array only needs to be large enough to hold all tree nodes). It should not be too difficult to replace our heap-based representation in the lower refinement with their array-based one, exploiting that we do not have to re-verify any of the invariants of red-black trees to do this.

Chapter 4

Flashix: A Verified File System for Flash Memory

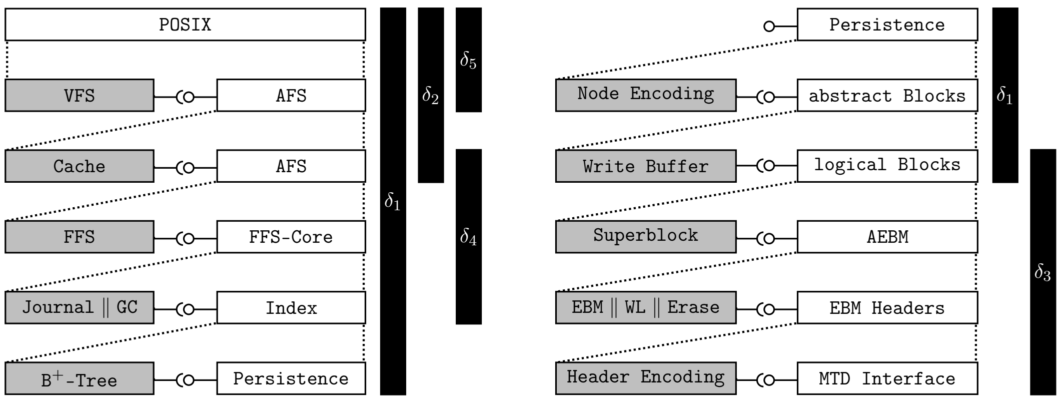
Summary Flashix is a long-term project with the goal of developing a verified, crash-safe file system for flash memory. In the course of development, a deep component hierarchy with over a dozen major refinements was created, using the modularization approach presented in the previous chapter. The file system was purely sequential in the first project phase and did not use any caches. Thus, the second phase of the project was mainly about adding such performance-oriented aspects in the aftermath.
This chapter gives an overview of the complete component hierarchy. It emphasizes the non-local extensions, i.e., extensions that affect multiple layers of the hierarchy, this thesis contributes as part of the second project phase. Furthermore, the original models of the top layers of the hierarchy are introduced as a basis for the extensions presented in the following chapters.

Contents

Publications The Flashix model summary in this chapter is based on [13]. Parts of the individual layer presentations were published in [14, 13, 15], nevertheless, the detailed models in this chapter - shown in their original state - are primarily based on the work of Ernst [38].

4.1 The Flashix Component Hierarchy

The Flashix file system is structured into a deep hierarchy of components, connected by incremental refinements and subcomponent relationships following the approach presented in Sec. 3.1. The pattern of Fig. 3.3 has been applied extensively to build a hierarchy containing over ten major data refinement steps (dashed lines) as shown in Fig. 4.1. Specification components are depicted in white and implementation components in gray. Combining all implementation components then results in the



(a) Upper layers of Flashix. (b) Lower layers of Flashix.

Figure 4.1: Component hierarchy of the Flashix file system with extensions δ_i: δ_1: order-preserving caching; δ_2: non-order-preserving caching; δ_3: concurrent wear leveling & erase; δ_4: concurrent garbage collection; δ_5: external concurrency.

final implementation of the file system.

As a starting point for this work, the basic hierarchy of Flashix, including most of the components and refinements of Fig. 4.1, was already given. The main focus of the work was to add multiple extensions, mainly comprising *caching* and *concurrency*, to this existing hierarchy while invalidating as little proof work as possible. This goal is contrasted by the fact that most extensions are *non-local*, i.e., while they can be implemented primarily locally in a component, they also impact surrounding components. For example, they can change the crash behavior of higher-level components or they result in lower-level components being called concurrently.

Fig. 4.1 additionally shows these non-local extensions as black bars, ranging over the affected layers and labeled with an identifier δ_i. All layers and the non-local extensions of these layers will shortly be introduced in this section. The remaining sections of this chapter will then present the models of the upper layers in more detail (in their initial form), which gives a basis for Ch. 6 and Ch. 7, where the non-local extensions are presented and discussed.

POSIX The top layer `POSIX` of Fig. 4.1a is a formal specification of the POSIX standard [100]. It defines the interface and the functional correctness requirements of the file system. Here, the state of the file system is given by a directory tree where leaves store file identifiers, and a mapping of file identifiers to the corresponding file contents, represented by a sequence of bytes. An indirection between file identifiers and file content is necessary to allow hard links, where the same file is present in several directories. *Structural operations*, i.e., operations that modify the directory tree like creating/deleting directories or (un)linking files, are defined on paths. *Content operations*, such as reading or writing parts of the content of a file, work directly on file identifiers.

MTD Interface The bottom layer `MTD Interface` of the hierarchy in Fig. 4.1b is a formal specification of the Linux MTD Interface (Memory Technology Devices)

[51]. It acts as a lower boundary of the file system and provides low-level operations to erase flash blocks and to read or write single pages within flash blocks. Preconditions ensure that calls to these operations comply with the characteristics of flash memory, i.e., that pages are only written as a whole and that pages are only written sequentially within a block. Additionally, it formalizes assumptions about hardware failures or the behavior of the flash device in the event of a crash. More details on the formal model of `MTD Interface` can be found in [98].

Virtual File System Switch In a first refinement step, the `POSIX` model is refined by a Virtual File System Switch (`VFS`) that uses an abstract specification of the core file system (`AFS`). Similar to the Linux Virtual Filesystem, the `VFS` component implements the resolution of paths to individual file system objects, permission checks, and the management of open files. Basically, the `AFS` provides an interface analogous to the `POSIX` interface but on the level of file system objects instead of paths. This specification abstracts completely from any flash-specific concepts and thus the `VFS` is not limited to be used exclusively with flash file systems. Details of the `POSIX` specification are given in Sec. 4.2, and the sequential models of `VFS(AFS)` are explained in Sec. 4.3. Both sections are based on the earlier publications [41, 42].

The most recent non-local extension to Flashix was to allow concurrent calls to the file system interface, which required implementing a locking concept for the `VFS` (δ_5 in Fig. 4.1). The approach taken focuses on enabling parallel access to file contents, in particular we want to allow arbitrary concurrent reads as well as concurrent writes to different files. Therefore, we chose a fine-grained locking strategy for files, whereas we applied a coarse-grained strategy for the directory tree. This means that each file is protected by an individual reader-writer lock while a single reader-writer lock is used for the entire directory tree. It should be noted, that parallel traversal of the directory tree is still possible as long as no *structural operation* is performed. Thus, we think this is a good trade-off between development or verification effort and performance gain. The existing sequential versions of `VFS` and `AFS` were augmented with locks and ownerships respectively and it was proven that the interleaved implementation of `VFS` is linearizable and deadlock-free using atomicity refinement as explained in Sec. 7.2. Specifics of this extension are presented in Sec. 8.2.

High-Level Cache `AFS` is refined by the actual Flash File System (`FFS`). Additionally, `AFS` is refined by a `Cache` component that caches data structures used at the interface of the core file system. This layer was introduced as another non-local extension (δ_2 in the figure). The cache is integrated as a decorator [48], i.e., it wraps around the `AFS` in the sense that it uses `AFS` as a subcomponent and also implements the interface of `AFS`. This allows the file system to be used both with and without `Cache`. The main goal of this integration was to allow *write-back* caching of content operations. However, write-back caching can have significant effects on the crash behavior of a system. Using this sort of file system caches required to define a novel correctness criterion (given in Sec. 5.1) and significant proof work to show that Flashix complies with it (presented in Sec. 6.1 together with the new models). The extension was also published in [14] and [15].

Flash File System, Journal & Index The `FFS` was the layer at which the development of the Flashix file system started in [111]. It introduces concepts specific to flash memory and to log-structured file systems. Updates to file system objects must be performed out-of-place and atomically. For this purpose, the `FFS` is built upon an efficient `Index`, implemented by a wandering $\texttt{B}^{+}$`-Tree`, and a transactional `Journal`. Both are specified abstractly in the component `FFS-Core`. Updates are encapsulated in nodes and grouped into transactions that are then written to a log. To keep track of the latest versions of objects, the locations of them on the flash memory are stored in the `Index`. The index exists in two versions, one persisted on flash and one in RAM. Updates on the index are initially performed only in RAM in order to improve performance as these update are quite costly to perform on flash. Only during *commits*, which are executed regularly, the latest version of the index is written to flash. The transactional `Journal` ensures that, in the event of a crash, the latest version of the RAM index can be reconstructed. This can be done by replaying the uncommitted entries in the log starting from the persisted index on flash. In doing so, incomplete transactions are discarded to comply to the atomicity properties expected by the `VFS`.

Another crucial mechanism implemented in this layer is *garbage collection.* Due to their out-of-place nature, updates to the file system leave garbage data behind. This data must first be deleted before the storage space it occupies can be used again. But since flash blocks can only be erased as a whole, garbage collection chooses suitable blocks for deletion (preferably blocks with a high percentage of garbage), transfers remaining valid data of that block to another one, and finally triggers the erasure of the block. This mechanism is not triggered explicitly by calls to the file system, instead it must be performed periodically to ensure that the file system does not run out of space. In earlier versions of the file system, garbage collection was triggered by the FUSE integration after each top-level POSIX call. With extension δ_4, garbage collection was extracted into a separate thread (indicated by the $\|$ symbol in Fig. 4.1). This (internal) concurrency extension is presented in Sec. 8.3.

Node Encoding Both the transactional `Journal` and the $\texttt{B}^{+}$`-Tree` write nodes on the flash device. The `Node Encoding` component is responsible for serializing these nodes to bytes before they can be written to flash. It also keeps track of the allocation of erase blocks and, for each block, the number of bytes still referenced by live data, i.e., by nodes of the index or nodes that store current versions of file system objects. This information is used to determine suitable blocks for garbage collection. Besides that, the layer ensures that writing of nodes appears to be atomic to the `Journal` and `Index`. It detects partially written nodes that may occur through crashes or hardware failures and takes care of them. A more in-depth view on these components and the sequential garbage collection is given in [40].

Write Buffer All serialized nodes pass a `Write Buffer`, representing the first non-local extension δ_1 of Flashix. This buffer cache tackles the restriction that flash pages can only be written sequentially and as a whole. It caches all incoming writes and only issues a page write once a page-aligned write is possible, i.e., the write requests have reached the size of one flash page in total. Otherwise, padding nodes would have to be used in order to write partially filled pages, which both would increase the absolute number of writes to flash and the amount of wasted space on the flash device. Introducing such an *order-preserving* write-back cache (written data leaves the cache in the same order as it entered it) also affects the crash behavior of the file system. In [97] we gave a suitable crash-safety criterion as well as a modular verification methodology for proving that systems satisfy this criterion. While this extension is already covered in detail by the work of Pfähler [96], Chapter 6 recaps the most important points of this extension in the context of the extension δ_2.

Superblock The `Superblock` component is responsible for storing and accessing the internal data structures of the file system. A specific part of the flash device is reserved for this data. They are written during a *commit* only, since persisting each update would have a significant negative impact on the performance of the file system. A critical task of this layer is to ensure that commits are performed atomically using a data structure called *superblock*.

Erase Block Manager Finally, the Erase Block Manager (`EBM`) provides an interface similar to the one of `MTD Interface` (read, write, erase). However, the `EBM` introduces an indirection of the *physical blocks* of the flash device to *logical blocks* and all of its interface operations address logical blocks only. These logical blocks are allocated on-demand and mapped to physical blocks. The indirection is used to move logical blocks transparently from one physical block to another one which is necessary to implement *wear leveling*. Wear leveling ensures that within some bounds all blocks are erased the same number of times. This is necessary to maximize the life time of the flash memory, as erasing a flash block repeatedly wears it out, making it unusable. To ensure a bound, the number of performed erases is stored in an *erase counter*. Wear leveling finds a logical block that is mapped to a physical block with low erase count and re-maps it to a block with high erase count. Since a logical block with low erase count typically contains a lot of *stale data* that has not been changed for some time and therefore is not likely to change soon, the number of erases is kept at the same level and the lifetime of the flash device increases.

The `EBM` uses the `Header Encoding` component for the serialization and deserialization of administrative data, most important an inverse mapping stored in the physical blocks containing the numbers of the logical blocks they are mapped to.

A sequential version of the Erase Block Manager is explained in detail in [98]. But similar to garbage collection, wear leveling has to be performed regularly without being triggered by the user. So we adjusted the `EBM` to run wear leveling in a separate thread as well, and another thread is used to perform the erasure of blocks asynchronously, too (extension δ_3). This was the first concurrency extension in Flashix and triggered the development of a verification methodology for introducing concurrency to a sequential refinement hierarchy. [108] illustrates the general approach using a simplified version of the concurrent `EBM`. Pfähler performed the actual extension as part of his thesis [96], in which the full version of the models is presented.

As part of this extension, red-black trees were also introduced to the `EBM`. Now, two trees are used to give efficient access to the erase counters of the flash blocks; particularly, the blocks with the highest and lowest erase counts can be found very quickly ($O(log\ n)$). Before, this information was stored in an array (with the length

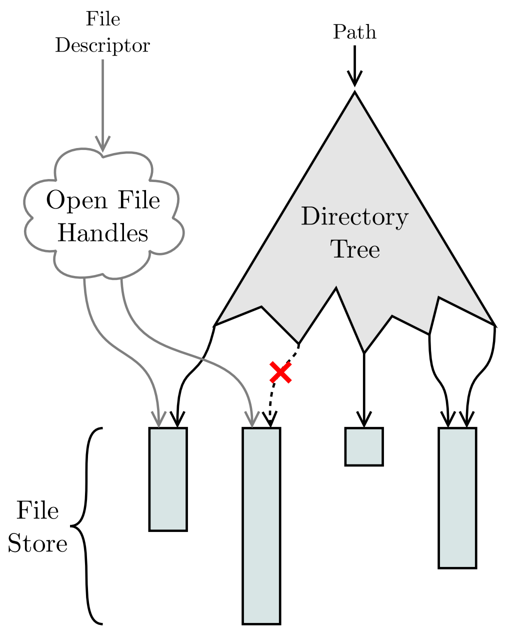


Figure 4.2: `POSIX` representation of the file system.

being the number of erase blocks of the flash hardware), so it was required to search the whole array each time to find the minimum/maximum ($O(n)$). Verification was attempted for the initial, non-modularized version, but it has never been finished due to the complex nature of the pointer implementation. Therefore, the implementation was modularized lately using an approach for *Separating Separation Logic* which allowed to prove the correctness with reasonable effort (see Sec. 3.2 and [106]).

4.2 The POSIX Specification

The Portable Operating System Interface (POSIX) [100] is a collection of standards for maintaining compatibility between operating systems, particularly UNIX-like operating systems. It defines various C-APIs for system- and user-level applications, including file systems. This section describes a formal model for file systems, extracted from the POSIX standard. The model originates from earlier work of the research group [41, 42] and defines the functional correctness requirements for the Flashix file system.

Modeling the File System Tree & File Contents

On an abstract level, the core part of a file system can be seen as a tree structure containing links to file contents, see Fig. 4.2. Starting from the root, the tree is built from nested *directories*, where each directory has named *entries*: an entry is either another directory or a link to a *file* containing *data*. Since files cannot link back to the tree, directories form the inner nodes of the tree while files form the leaves.

File systems typically allow references from multiple directories to the same file, called *hard links*. In order to model this circumstance, an indirection between the leaf nodes and the actual data has to be introduced. Instead of storing file contents directly in the tree, they are kept separately and can be accessed via a unique *file identifier* (FID).

Algebraically, the tree can be specified as a free data type tree, similarly to the rbtree type in Sec. 3.2.

$$\begin{aligned}\textbf{data}\quad \mathsf{tree} &= \mathtt{fnode}(.\ .\mathtt{fid} : \mathsf{fid})\\ &\mid \mathtt{dnode}(.\ .\mathtt{meta} : \mathsf{metadata};\ .\ .\mathtt{entries} : \mathsf{map}(\mathsf{string}, \mathsf{tree}))\end{aligned}$$

Directories are represented by dnodes ("Directory Nodes") that contain a finite mapping from entry names (given as strings) to corresponding subtrees. Unlike red-black trees, a directory tree is not a binary tree, so each directory node can contain an arbitrary number of entries. Links to files in the tree are modeled by fnodes ("File Nodes") containing just the file identifier fid of the linked file. As the fid type is only used internally in the POSIX model, it is not specified further except that its carrier set is infinite, so the creation of new files is not restricted.

The content of files is given by a sequence of bytes, stored in an array. An alias type buffer is introduced to abbreviate the instantiation of the polymorphic type array with byte and the constructor mkarray is renamed to mkbuf.

$$\textbf{type}\quad \mathsf{buffer} \equiv \mathsf{array}(\mathsf{byte}) \quad \textbf{with}\quad \mathtt{mkarray} \mapsto \mathtt{mkbuf}$$

Again, it is not necessary to give a detailed specification of the byte type since a file system implementation does not have to manipulate individual bytes of user data. Only a dedicated constant ZERO of type byte is used for filling unoccupied ranges within buffers. The content buffer of a file is wrapped into a object of type fdata.

$$\textbf{data}\quad \mathsf{fdata} = \mathtt{file}(.\ .\mathtt{meta} : \mathsf{metadata};\ .\ .\mathtt{content} : \mathsf{buffer})$$

Directories and files also contain some metadata, stored in dnodes and files, respectively. The abstract type metadata is used as a placeholder for administrative information of a file system node. For example, it is used to store the owner of a file, together with read, write, and execute permissions. We also use some placeholder predicates pr(*user*, *md*), pw(*user*, *md*), and px(*user*, *md*) to state if a user *user* (of unspecified type user) has the respective permission to access a node with metadata *md*. This formalization is inspired by [61] and is sufficient to implement correct permission checking within the file system.

For accessing file contents as a user, *file handles* store the file identifier fid of a file together with an *access mode* mode and the current position pos withing the content buffer. The mode determines how the file content can be accessed via this handle: it can be either read-only, write-only, or both read and write.

$$\begin{aligned}\textbf{data}\quad \mathsf{fhandle} &= \mathtt{fh}(.\ .\mathtt{fid} : \mathsf{fid};\ .\ .\mathtt{mode} : \mathsf{mode};\ .\ .\mathtt{pos} : \mathsf{nat})\\ \textbf{data}\quad \mathsf{mode} &= \mathtt{MODE\_R} \mid \mathtt{MODE\_W} \mid \mathtt{MODE\_RW}\end{aligned}$$

File handles are created when a user *opens* a file. The user can then access the file via the handle multiple times, i.e., perform several reads or writes, until the user *closes* the file again (see the following subsections). Open file handles are exposed to users of the file system via *file descriptors*, which are unique natural numbers assigned to the handles.

Altogether, this results in a state consisting of a *directory tree root*, a *file store fs* storing file data under the corresponding file identifier, and *open file handles ofh* using file descriptors as keys (cf. Fig. 4.2).

component `POSIX`
state $root : \mathsf{tree}, \quad fs : \mathsf{map}(\mathsf{fid}, \mathsf{fdata}), \quad ofh : \mathsf{map}(\mathsf{nat}, \mathsf{fhandle})$

Manipulating the File System Structure

Individual nodes of the directory tree are accessed via *paths*. A file system path is sequence of directory entry names, so an alias type path abbreviating lists of strings is introduced. For better distinction, the empty path is represented by the constant ε renaming the empty list `[]`.

type $\mathsf{path} \equiv \mathsf{list}(\mathsf{string})$ **with** $\texttt{[]} \mapsto \varepsilon$

In the following, *valid* paths, i.e., paths that lead to a node within the directory tree, are of particular interest, since path accesses to a tree are only well-defined for such paths. A valid path is described by the predicate $. \in . : \mathsf{path} \times \mathsf{tree}$, defined recursively by the following axioms.

$$\vdash \varepsilon \in t$$

$$\vdash \neg\ str + p \in \texttt{fnode}(fid)$$

$$\vdash str + p \in \texttt{dnode}(md, entries) \leftrightarrow str \in entries \land p \in entries\texttt{[}str\texttt{]}$$

Manipulations of the directory tree by *structural operations* use paths as inputs to determine which part(s) of the tree should be modified. The set of structural operations comprises creating new directories or files, removing existing directories, creating or deleting hard links to files, or moving files or directories to other locations within the tree. In order to model these operations, three algebraic functions $.\texttt{[}.\texttt{]} : \mathsf{tree} \times \mathsf{path} \to \mathsf{tree}$, $.\texttt{[}.\texttt{]} : \mathsf{tree} \times \mathsf{path} \times \mathsf{tree} \to \mathsf{tree}$, and $. \texttt{--} . : \mathsf{tree} \times \mathsf{path} \to \mathsf{tree}$ are defined to select the subtree at a path, to exchange the subtree at a path, and to remove a subtree at a path, respectively. All three functions are defined similarly, e.g., the following axioms define the update function.

$$\vdash t\texttt{[}\varepsilon, t_0\texttt{]} = t_0$$

$$\vdash \texttt{dnode}(md, entries)\texttt{[}str\texttt{'}, t_0\texttt{]} = \texttt{dnode}(md, entries\texttt{[}str, t_0\texttt{]})$$

$$\vdash p \neq \varepsilon \land str \in entries \to \texttt{dnode}(md, entries)\texttt{[}str + p, t_0\texttt{]} = \texttt{dnode}(md, entries\texttt{[}str, entries\texttt{[}str\texttt{]}\texttt{[}p, t_0\texttt{]]})$$

Note that the operation here is defined over the structure of paths, however, further distinguishes between a path with one segment (second axiom, str`'` abbreviates the singleton path $str + \varepsilon$) and a path with at least two segments (third axiom, the precondition $p \neq \varepsilon$ ensures that the axiom does not clash with the second axiom as $str + p$ would also match on $str + \varepsilon$ otherwise). Furthermore, note that the operation is only defined for paths p that are either valid or "nearly" valid in the sense that the *parent path*, i.e., the path with the last segment dropped (written p`.parent`),

```
posix_mkdir#($p$, $md$, $user$; ; $err$)
  interface
{
  $root$[$p$] := dnode($md, \emptyset$);
}

posix_rmdir#($p$, $user$; ; $err$)
  interface
{
  $root$ := $root$ -- $p$;
}

posix_readdir#($p$, $user$; $names$; $err$)
  interface
{
  $names$ := dom $root$[$p$].entries;
}

posix_rename#($p_0$, $p_1$, $user$; ; $err$)
  interface
{
  let $t_0 = root$[$p_0$], $t_1 = root$[$p_1$],
      $exists = p_1 \in root$
  in
    $root$ := ($root$ -- $p_0$)[$p_1, t_0$];
    if   $exists \wedge \neg\ t_1$.dir?
       $\wedge \neg\ t_1$.fid $\in ofh$.fids
       $\wedge \neg\ t_1$.fid $\in root$.fids
    then
      $fs$ := $fs$ -- $t_1$.fid;
}
```

```
posix_create#($p$, $md$, $user$; ; $err$)
  interface
{
  choose $fid$ with $\neg\ fid \in fs$ in
    $root$[$p$] := fnode($fid$);
    $fs$[$fid$] := file($md$, mkbuf(0));
}

posix_link#($p_0$, $p_1$, $user$; ; $err$)
  interface
{
  let $fid = root$[$p_0$].fid in
    $root$[$p_1$] := fnode($fid$);
}

posix_unlink#($p$, $user$; ; $err$)
  interface
{
  let $fid = root$[$p$].fid in
    $root$ := $root$ -- $p$;
    if   $\neg\ fid \in ofh$.fids
       $\wedge \neg\ fid \in root$.fids
    then
      $fs$ := $fs$ -- $fid$;
}
```

Figure 4.3: Structural operations of the POSIX component (without error handling).

points to a directory within the tree. Otherwise, the lookup on *entries* in the third axiom would be undefined. Allowing such paths results in the operation not only overwriting existing subtrees but also attaching trees to the existing structure (when $\neg\ str \in entries$ in the second axiom).

Fig. 4.3 lists the complete set of structural operations of the POSIX component. The operations do not have any preconditions as they form the user-level interface of the file system, and hence it could not be assured that preconditions hold every time an operation is called. Instead, operations return an error code *err* of type error as output, signaling whether the operation was successful, a hardware error occurred, or some given inputs were invalid (which corresponds to the violation of an implicit precondition). The exact error handling will be omitted in the following listings. However, the POSIX component uses a generic non-deterministic approach for handling such errors, respectively precondition violations, introduced later in this section.

Due to the use of an algebraic tree for the directory structure combined with the algebraic operations given above, most declaration bodies consist of a handful of assignments. The creation of directories (procedures **posix_mkdir#**) simply adds a new `dnode` with given metadata *md* and without any entries to the directory tree at the path *p*. Similarly, **posix_rmdir#** removes the directory node at path *p* from the tree. The operation **posix_readdir#** reads the names of all entries of a directory at path *p* (the algebraic function `dom` *mp* returns the *domain* of a map *mp*) and returns them to the caller.

Files are created by the operation **posix_create#**, which adds a new `fnode` at *p* analogously to **posix_mkdir#**. Additionally, a new entry must be allocated in *fs* to store the content of the file (initialized with the empty buffer `mkbuf`(0)) and the given metadata *md*. The file identifier is chosen non-deterministically, with the only restriction being that it is *fresh*, i.e., currently not allocated in *fs*. Existing files can be linked from other locations within *root* with **posix_link#**. It takes a path p_0 pointing to an existing file and creates an `fnode` at another path p_1 containing the same file identifier *fid*. Since this allows multiple hard links to point to the same file, there is no explicit operation for removing a file. Only individual links can be removed via **posix_unlink#**. As **posix_rmdir#**, the operation removes the `fnode` at *p* from the tree. However, it also checks if this was the last link pointing to the file (*root*`.fids` collects all file identifiers of `fnode`s in *rootRef*) and if the file therefore needs to be removed from *fs*.

One small detail that needs to be considered (and Fig. 4.2 also hints at) is that both *root* and *ofh* can link to files in *fs*. If a link to a currently opened file is removed (indicated by the red cross in the figure), the file must be kept in the files store, even if there is no link from the directory tree left. Such files are called *orphans* or *orphaned files*. As long as an orphaned file is not closed, its content must be accessible via the file handle since an application could still read (or even write) it. A typical example would be a package upgrade of an application that is still running. When the binary file of the running application is overwritten, the process still references to the old version of the file so that the file system is prevented from deleting its content on disk. So **posix_unlink#** deletes the entry of *fid* from *fs* only if there is no link from the directory tree ($\neg$ *fid* $\in$ *root*`.fids`) and there is no open file handle left pointing to *fid* ($\neg$ *fid* $\in$ *ofh*`.fids`). Consequently, the operation for closing a file acts the same way and potentially removes a file from *fs* (see Fig. 4.5 later this section).

Arguably the most complex structural operation is **posix_rename#**. The operation takes a subtree at a path p_0, which could be a single file or a whole directory including entries, and moves it to another location given by the path p_1. This is modeled by selecting the tree at p_0, removing the tree from *root*, and inserting the tree at p_1 again. However, the path p_1 is allowed to point to an existing file or directory at the beginning of the operation, so moving the tree at *root*`[`p_0`]` to this path overwrites the old tree *root*`[`p_1`]`. This is explicitly allowed by the POSIX standard when some conditions are respected, e.g., files can only overwrite files or directories can only be overwritten by other directories and if they are empty (have no entries). Thus, **posix_rename#** must also consider the case that the move has overwritten a `fnode` and that this node was the last link to the file. Then the file is removed from the file store, too.

Accessing Metadata and File Contents

Since both files and directories store metadata, the operations for fetching and updating them are used uniformly for both types of nodes. Figure 4.4 lists the respective

```
posix_writemeta#(p, md, user; ; err)
  interface
{
  if root[p].dir? then
    root[p].meta := md;
  else let fid = root[p].fid in
    fs[fid].meta := md;
}
```

```
posix_readmeta#(p, user; md,
                 nlink, sz; err)
  interface
{
  if root[p].dir? then
    md := root[p].meta;
    nlink :=   # root[p].subdirs
           + $(p = \varepsilon \supset 1; 2)$;
    sz := # root[p].entries;
  else let fid = root[p].fid in
    md := fs[fid].meta;
    nlink := count(fid, root.fids);
    sz := # fs[fid].content;
}
```

Figure 4.4: Metadata operations of the `POSIX` component (without error handling).

operations **posix_readmeta#** and **posix_writemeta#**, again using a path p as input for identifying a file system node.

Updating the metadata of a node with **posix_writemeta#**just writes the input *md* to the respective field of the node. As directories store their metadata in the tree while files store them in the file store, the operation must distinguishes between the type of node p points to (t`.dir?` checks whether t is a `dnode`). For directories, the metadata is replaced directly in the *root*`[`p`]`. For files, the metadata is updated in the entry of *fs* referenced by the file identifier in *root*`[`p`]`.

Retrieving metadata of a file or directory with **posix_readmeta#** returns the metadata *md* of the node, but also some additional information (in Linux, the operation corresponds to the operations *getattr* resp. *getxattr*, which also require some structural information about the node): the outputs *nlink* and *sz* return the number of links to the node and its size. While the metadata is read directly from the corresponding node in *root* or the file in *fs*, *nlink* and *sz* are not stored explicitly but must be calculated. The size of a directory is defined by the number of its entries, and the set of links to a directory is given by the reverse links of its subdirectories as each directory has a link to its parent via ".." (t`.subdirs` selects all entries of t that are directories), a reflexive link via ".", and the link from the parent directory (each directory has precisely one parent directory[1], except the root directory, which has none). The size of a file is given by the length of its content buffer, i.e., by the number of bytes stored in the file, and the number of links to a file is given by counting all hard links pointing to it, i.e., by all `fnode`s in *root* containing the file identifier *fid*.

Contrary to structural and metadata operations, the contents of files are not accessed via paths but via *file descriptors* referencing *file handles* managed in *ofh*. The operations for creating, destroying, and manipulating file handles are listed in Fig. 4.5.

Before the content of a file can be read or written, the file must be opened to create a file descriptor using the operation **posix_open#**. Opening a file takes a path p pointing to a `fnode` of the file to be opened and allocates a new file handle

[1]Flashix does not support multiple hard links on directories.

```
posix_open#(p, mode, user; fd; err)
  interface
{
  let fid = root[p].fid in
    choose n with ¬ n ∈ ofh in
      ofh[n] := fh(fid, mode, 0);
      fd := n;
}

posix_close#(fd, user; ; err)
  interface
{
  let fid = ofh[fd].fid in
    ofh := ofh -- fd;
    if   ¬ fid ∈ ofh.fids
      ∧ ¬ fid ∈ root.fids
    then
      fs := fs -- fid;
}
```

```
posix_seek#(fd, whence, n, user; ; err)
  interface
{
  let fh = ofh[fd] in
    if whence = SEEK_CUR then
      n := fh.pos + n;
    else if whence = SEEK_END then
      n := # fs[fh.fid].content + n;
    ofh[fd].pos := n;
}
```

Figure 4.5: File handle operations of the POSIX component (without error handling).

with the corresponding file identifier. The handle is stored in *ofh* under a fresh descriptor n, which is chosen non-deterministically and returned to the caller (the implementation of POSIX will resolve this non-determinism). The handle is initialized at position 0, i.e., at the beginning of the file content, with the *mode* passed by the caller determining how the file can be accessed via the file descriptor n (it can be either MODE_R, MODE_W, or MODE_RW).

During the lifetime of a file handle, its fid and mode cannot be changed. What does change, however, is its position pos: reading from or writing to a file automatically moves the current position forward (see Fig. 4.6 later this section), but it can also be changed explicitly with the **posix_seek#** operation. While the former mechanism is useful when a file is read or written entirely from the beginning to the end, setting the file handle to a custom location enables, for example, appending writes to files without overwriting existing content. Therefore, seeking takes a file descriptor *fd*, an offset n, and a *seek flag whence* as input. The seek flag determines the position in the file where the offset n should be applied.

data seekflag = SEEK_SET | SEEK_CUR | SEEK_END

The position can either be set absolutely to n when *whence* = SEEK_SET, the offset n can be added to the current position of the handle when *whence* = SEEK_CUR, or it can be added to the end of the file, i.e., the length of its content buffer # *fs*[*fh*.fid].content, when *whence* = SEEK_END. Note that in all cases, the position of the file handle is set to n at the end of the operation, but in the latter two cases, the value of n is updated locally beforehand (these updates are not visible to the caller as n is an input parameter).

Finally, a file is closed with the operation **posix_close#** by removing the entry of the passed file descriptor *fd* from *ofh*, thus deleting the corresponding file handle. Since the file could have become an orphan during the time it was opened, the

```
**posix_write#**($fd$, $buf$, $user$; $n$; $err$)
  **interface**
  **precondition** $n \leq \# \ buf$
{
  **let** $fh = ofh$[$fd$], $fdata = fs$[$ofh$[$fd$].fid] **in**
    **choose** $m$ **with** $m \leq n$ **in**
      $n := m$;
      $fdata$.content := splice($buf$, 0, $fdata$.content, $fh$.pos, $n$);
      $fs$[$fh$.fid] := $fdata$;
      $ofh$[$fd$].pos := $fh$.pos + $n$;
}
```

Figure 4.6: Write operation of the `POSIX` component (without error handling).

operation checks this circumstance and also removes the file from the file store *fs* if necessary.

Writing to a file is done with the operation **posix_write#** shown in Fig. 4.6. It takes a file descriptor *fd* and a buffer *buf* containing the data to be written as input. The referenced parameter n is used twofold: it gives the requested number of bytes as an input and returns the number of actually written bytes as output. This distinction is necessary as the POSIX standard allows writes (and similarly reads) to write only a prefix of bytes. Such partial writes can, for example, occur when there is insufficient space available on the disk or flash device. Then the operation signals success via *err* nevertheless, but the caller can detect incomplete writes by the value of n. This behavior of writes is particularly interesting for adding caches to the file system as it can be exploited to formulate suitable crash-safety criteria (see Chapter 6). The declaration implements this characteristic by choosing a value m less or equal to n non-deterministically and copying m bytes from the input buffer *buf* to the content of the file referenced by *fd*.

Copying bytes from one buffer to another is modeled with the algebraic function `splice`, which effect is depicted in Fig. 4.7. `splice`($buf_0, m_0, buf_1, m_1, n$) yields the buffer buf_1 in which, starting from position m_1, n bytes have been overwritten by the bytes of buf_0 starting from position m_0. The function `splice` considers two special cases: first, if the bytes to copy reach beyond the size of the target buffer, i.e., $\# \ buf_1 < m_1 + n$, the buffer is resized to $m_1 + n$ to fit all copied bytes (this situation is shown in Fig. 4.7); and second, if the target offset is larger than the target buffer size, i.e., $\# \ buf_1 < m_1$, the buffer is resized as in the first case but the gap between $\# \ buf_1$ and m_1 is filled with `ZERO` bytes.

posix_write# uses `splice` to copy the first n (resp. m) bytes from the input buffer (the source offset is 0) to the `content` buffer of the file, starting from the current position of the file handle *fh*. Thus, the position has to be set with **posix_seek#** properly before calling **posix_write#**. Finally, the position of the handle is incremented by the number of bytes copied so that a subsequent write can directly append data.

The read operation **posix_read#** functions nearly symmetrical. Bytes are copied from the `content` buffer of the file referenced by the descriptor *fd* to the output buffer *buf*. Again, the POSIX standard allows the operation to read fewer bytes than requested, modeled by a non-deterministic choice. In particular, it is

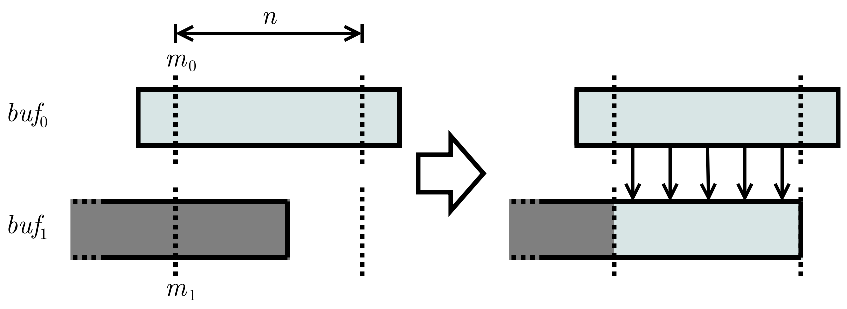


Figure 4.7: Extending copy between buffers with $\texttt{splice}(buf_0, m_0, buf_1, m_1, n)$.

```
**posix_read#**($fd$, $user$; $buf$, $n$; $err$)
  **interface**
  **precondition** $n \leq \#\ buf$
{
  **let** $fh = ofh[fd]$, $fdata = fs[ofh[fd].\texttt{fid}]$ **in**
    **choose** $m$ **with** $m \leq n \land fh.\texttt{pos} + m \leq \#\ fdata.\texttt{content}$ **in**
      $n := m$;
      $buf := \texttt{copy}(fdata.\texttt{content},\ fh.\texttt{pos},\ buf,\ 0,\ n)$;
      $ofh[fd].\texttt{pos} := fh.\texttt{pos} + n$;
    **ifnone**
      $n := 0$;
}
```

Figure 4.8: Read operation of the `POSIX` component (without error handling).

not possible to read bytes beyond the size of the file content, so reading has to stop when this boundary is reached (specified by the additional choice condition $fh.\texttt{pos} + m \leq \#\ fdata.\texttt{content}$). Note that no bytes can be read for an invalid position of the file handle (when $\#\ fdata.\texttt{content} < fh.\texttt{pos}$). Then n is set to 0, and buf is not modified. Furthermore, the simpler function `copy` is used instead of `splice` since the size of the target buffer does not need to be altered (and should not).

Besides increasing the file size via appending writes, the size of a file can also be changed by the operation **posix_truncate#** listed in Fig. 4.9. Although it modifies the content of a file, it uses a path p as input instead of a file handle[2]. A truncation resizes the `content` buffer of the a file at p to the new absolute size n. If the file previously was larger than n, the extra data beyond the new size is lost. If it was shorter, it is extended, and the extended part is filled with `ZERO`s.

[2]Linux also supports an operation *ftruncate* that works on file handles, however, the FUSE library used for the integration of Flashix into Linux does only support the path variant.

```
posix_truncate#($p$, $n$, $user$; ; $err$)
  interface
{
  let $fid = root$[$p$].fid in
    $fs$[$fid$].content := resize($fs$[$fid$].content, $n$);
}
```

Figure 4.9: POSIX operation for resizing of files (without error handling).

Error Model & Crash Behavior

All POSIX operations return an error code *err* indicating whether the operation was executed successfully, the operation was called with invalid arguments, or a low-level error occurred. Thus, the error type is defined as an enumeration type, where each (non-successful) enumeral indicates the cause for a failing operation.

$$\textbf{data}\quad \mathsf{error} \equiv \texttt{ESUCCESS} \mid \texttt{ENOSPC} \mid \texttt{EIO} \mid \texttt{EROFS} \mid \ldots$$

With ESUCCESS signaling the successful execution of an operation (non-failing executions always return ESUCCESS), the remaining error codes can be partitioned into two categories. High-level errors are typically returned when an operation could not be performed because the execution would yield an inconsistent file system state, or the outcome of the operation is not determined for a given input. For example, creating a file at a path that already points to an existing node would unintentionally overwrite the node, and reading from a file with an already closed file handle does not provide a definitive position within the file content. Such situations are often detected in the upper layers of the file system as they require an abstract view of the file system state. On the other hand, low-level errors usually emerge from the underlying flash hardware, e.g., when no space is left on the storage medium, when writing or reading a flash page fails, or when the lower file system levels switch to a read-only mode due to a failed transaction. Since these errors are independent of the arguments of operation calls, the higher levels of the Flashix hierarchy (which are relevant for this thesis primarily) assume that any subcomponent operation can randomly return an error code. Therefore, the class of low-level errors $\lfloor err \rfloor$ is specified as all errors that are not exclusively returned by the upper layers.

$$\begin{aligned} \vdash \lfloor err \rfloor \leftrightarrow \quad & err \neq \texttt{ESUCCESS} \wedge err \neq \texttt{EEXISTS} \wedge err \neq \texttt{ENOENT} \\ & \wedge\, err \neq \texttt{EISDIR} \wedge err \neq \texttt{ENOTDIR} \wedge err \neq \texttt{ENOTEMPTY} \\ & \wedge\, err \neq \texttt{EBADFD} \wedge err \neq \texttt{EACCESS} \end{aligned}$$

Using this error model, the approach for specifying the error behavior of POSIX requires giving a *precondition predicate* pre-op($\underline{inp}$, err) for each operation op that defines possible errors *err* in the current state for given inputs $\underline{inp}$. At the beginning of the operation, an error code satisfying this predicate is chosen, and the actual operation is only performed if it is ESUCCESS. Fig. 4.10 shows the **posix_create#** operation of Fig. 4.3 but extended with this mechanism.

```
**posix_create#**$(p,\ md,\ user;\ ;\ err)$
  **interface**
{
  **choose** $err_0$ **with** pre-create$(p, md, user, root, fs, ofh, err_0)$ **in** $err := err_0$;
  **if** $err =$ ESUCCESS **then**
    **choose** $fid$ **with** $\neg\ fid \in fs$ **in**
      $root$[$p$] $:=$ fnode($fid$);
      $fs$[$fid$] $:=$ file($md$, mkbuf(0));
}
```

Figure 4.10: POSIX operation for creating files with error handling.

The definition of the precondition predicate is given by axioms covering every error, e.g., the axioms for `pre-create` include the following ones (besides others).

$$\vdash \texttt{pre-create}(p, md, user, root, fs, ofh, \texttt{ESUCCESS}) \leftrightarrow$$
$$\neg\ p \in root \wedge p\,\texttt{.parent} \in root \wedge\ root\texttt{[}p\texttt{.parent].dir?}$$
$$\wedge\ \texttt{pw}(user, p\,\texttt{.parent}, root, fs) \wedge \texttt{px}(user, p\,\texttt{.parent}, root, fs)$$

$$\vdash \texttt{pre-create}(p, md, user, root, fs, ofh, \texttt{EEXISTS}) \leftrightarrow p \in root$$

$$\vdash \neg\ \texttt{pre-create}(p, md, user, root, fs, ofh, \texttt{EBADFD})$$

$$\vdash \lfloor err \rfloor \to \texttt{pre-create}(p, md, user, root, fs, ofh, err)$$

The first axiom defines successful operation runs where all input arguments are valid for the current state. For file creation, the target path must not point to an existing node, but the parent of the target must point to an existing directory and the user must have write and execute permissions on the parent. Other axioms, like the second, define error cases when one condition is violated: file creation returns an `EEXISTS` error if the path points to an existing node. Some error will never be returned by an operation, e.g., file creation cannot return `EBADFD` since no file descriptor is accessed during the operation (third axiom). And finally, low-level errors can always occur (last axiom). Implementations of the **posix_create#** operation must then respect these specification and are only allowed to return errors when the predicate evaluates to **tt**, as otherwise, the refinement cannot be proven.

The crash behavior of the uncached version of `POSIX` is given by the crash predicate `posix-crash`, defined over the state *root*, *fs*, and *ofh*.

component `POSIX`
crash predicate `posix-crash`$(root^\backprime, fs^\backprime, ofh^\backprime, root, fs, ofh)$

Basically, the directory tree should be immune to crashes, i.e., the *root* after a crash (and subsequent recovery) should be identical to the tree $root^\backprime$ before the crash. On the other hand, all open file handles in $ofh^\backprime$ are lost during the crash since they are only kept in main memory, so *ofh* is empty after the crash. Like the directory tree, file data in $fs^\backprime$ should not be lost due to a crash. However, as there may be orphaned files at the moment of the crash and the links from the open file handles are lost, these files must be removed since they are no longer accessible. Together, this behavior can be defined by the following axiom.

$$\begin{aligned}\vdash \texttt{posix-crash}(root_0, fs_0, ofh_0, root_1, fs_1, ofh_1) \leftrightarrow &\\ & ofh_1 = \emptyset \land root_1 = root_0\\ \land\ & (\forall\, fid.\ fid \in fs_1 \leftrightarrow fid \in root_1\texttt{.fids} \land fid \in fs_0)\\ \land\ & (\forall\, fid.\ fid \in fs_1 \to fs_1\texttt{[}fid\texttt{]} = fs_0\texttt{[}fid\texttt{]})\end{aligned}$$

The predicate combines the effect of the crash (data in volatile memory is lost) and recovery mechanisms (on reboot, an implementation must actively search and delete orphaned files from persistent storage). Consequently, the recovery procedures of `POSIX` is in principle a no-op. Note that this specification approach is justified since `POSIX` is a specification component, and hence there is no need to model complex recovery mechanisms algorithmically when a simple algebraic specification is sufficient.

4.3 The Virtual File System Switch

The task of `VFS` is to implement `POSIX` operations, like creating or deleting files and directories or opening files and writing buffers to them, by elementary operations on individual nodes that represent a single directory or file. Each of these nodes is identified by a natural number $ino \in \mathsf{ino}$, where $\mathsf{ino} \simeq \mathbb{N}$. The operations on single nodes are implemented by each file system separately and we specify them via the `AFS` ("abstract file system") interface.

AFS: The Interface of VFS to File Systems

The state of `AFS` is specified as abstractly as possible by two heaps with disjoint domains to store directories and files.

$$\begin{aligned}&\textbf{component}\ \texttt{AFS}\\ &\textbf{state}\quad dirs : \mathsf{iheap(dir)}, \quad files : \mathsf{iheap(file)}\end{aligned}$$

where

$$\begin{aligned}&\textbf{type}\quad \mathsf{iheap} \equiv\ \mathsf{heap}\quad \textbf{with}\quad \mathsf{ref} \mapsto \mathsf{ino},\ \texttt{null} \mapsto 0\\ &\textbf{data}\quad \mathsf{dir}\ =\ \texttt{dir}(.\ \texttt{.meta} : \mathsf{metadata};\ .\ \texttt{.nlink} : \mathsf{nat};\ .\ \texttt{.size} : \mathsf{nat};\\ &\qquad\qquad\qquad .\ \texttt{.nsubdirs} : \mathsf{nat};\ .\ \texttt{.entries} : \mathsf{map(string, ino)})\\ &\textbf{data}\quad \mathsf{file}\ =\ \texttt{file}(.\ \texttt{.meta} : \mathsf{metadata};\ .\ \texttt{.nlink} : \mathsf{nat};\ .\ \texttt{.size} : \mathsf{nat};\\ &\qquad\qquad\qquad .\ \texttt{.content} : \mathsf{map(nat, buffer)})\end{aligned}$$

Both directories and files store metadata `meta` that is mainly used to handle access rights (cf. Sec. 4.2), an explicit counter `nlink` of their hard links (for directories, this is always 1 except for the root directory), and their `size` (for directories, this is the number of directory entries). The `entries` of a directory are contained in the directory itself in the form of a mapping from entry names to inode numbers, and the `nsubdirs` field additionally stores how many entries are directories. These inode numbers identify another directory or file in the file system, so they also identify a dir or file in *dirs* or *files*, respectively.

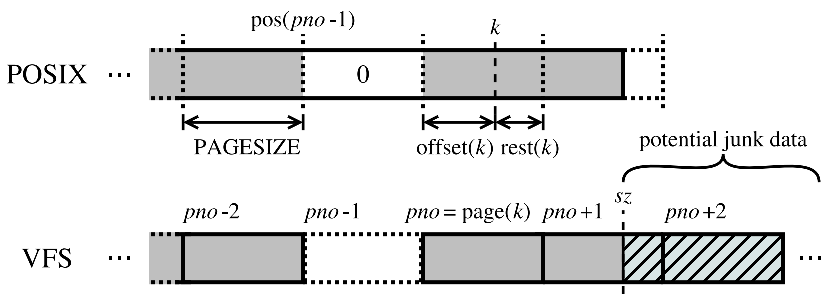


Figure 4.11: Representation of file contents in POSIX and in VFS.

This linking between nodes is the reason that heaps are used for *dirs* and *files* as they form a tree pointer structure similar to the red-black trees presented in Sec. 3.2. Thus, the refinement $\texttt{VFS(AFS)} \leq \texttt{POSIX}$ can also be proven using Separation Logic, as shown later this section. The heaps use inode numbers instead of references (AFS is not an implementation component, so the algebraic heap nodes do not have to be transformable into an actual heap data structure for generating code for Flashix), so a type iheap is used for *dirs* and *files* where the ref type is instantiated with the ino and the null constant is instantiated with 0.

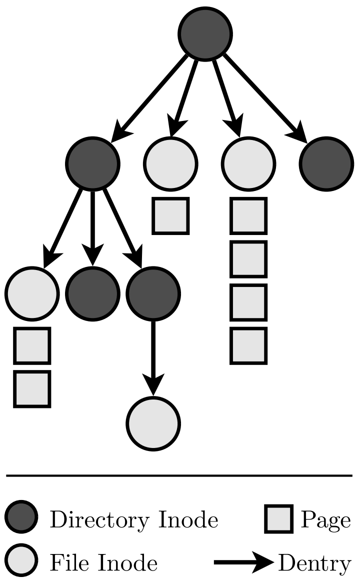


Figure 4.12: VFS representation of the file system tree.

Details on the representation of a file are shown in Fig. 4.11. The uniform representation as a sequence of bytes is broken up into an explicit file size and several pages stored in the content map. Each buffer is an array of size PAGE_SIZE. Byte k of a file is accessed via $\texttt{offset}(k)$ in $\texttt{page}(k)$, which are the remainder and quotient when dividing k by PAGE_SIZE. The function $\texttt{rest}(k)$ is used to denote the length of the rest of the page above $\texttt{offset}(k)$. We have $\texttt{rest}(k) = \texttt{PAGE\_SIZE} - \texttt{offset}(k)$, when the offset is non-zero. Otherwise, $\texttt{rest}(k) = 0$, k is (page-)*aligned*, and predicate $\texttt{aligned}(k)$ is true. The start of page *pno* is at $\texttt{pos}(\mathit{pno}) = \mathit{pno} * \texttt{PAGE\_SIZE}$. The pages are stored as a map, a missing page, e.g., page $\mathit{pno} - 1$ in the figure, indicates that the page contains zeros only. This sparse representation allows the creation of a file with large size without allocating all the pages immediately (which is important, e.g., for streaming data). Another important detail is that there may be irrelevant data beyond the file size. It is possible that the page $\texttt{page}(sz)$ at the file size sz contains random junk data (hatched part of the page) above $\texttt{offset}(sz)$ instead of just zeros. Extra (hatched) pages with a page number larger than $\texttt{page}(sz)$ are possible as well. Allowing such junk data is necessary for efficient recovery from a crash: writing data at the end of a file is always done by writing pages first, and finally incrementing the size. If a crash happens in between, then removing the extra data when rebooting would require to scan all files, which would be prohibitively expensive.

With this data representation, `AFS` offers a number of operations that are called by `VFS`, using parameters of type inode ("Index Node") and dentry ("Directory Entry") as input and output (passed by reference). These data structures are used to build the file system tree representation of `VFS` as shown in Fig. 4.12. An inode represents an existing node of the tree (it is used for both directories and files) and has the form

$$\textbf{data}\quad \mathsf{inode} = \mathtt{inode}(\,.\,.\mathtt{ino} : \mathsf{ino};\ .\,.\mathtt{meta} : \mathsf{metadata};\ .\,.\mathtt{isdir} : \mathsf{bool};\ .\,.\mathtt{nlink} : \mathsf{nat};\ .\,.\mathtt{size} : \mathsf{nat})$$

Each inode contains the inode number `ino` of the represented file or directory node and its metadata `meta`. The boolean `isdir` distinguishes between directories and files, and the `nlink`-field gives the number of hard links for a file (`nlink` $= 1$ for a directory). The `size`-field stores the file size for files and the number of entries for a directory.

Dentries form the edges of the file system tree by linking a parent directory to one of its children. Normal dentries store the `name` of the entry and the inode number `target` of the corresponding child. Negative dentries, on the other hand, are used to indicate that an entry with `name` does not exist in the directory.

$$\begin{aligned}\textbf{data}\quad \mathsf{dentry} &= \mathtt{dentry}(\,.\,.\mathtt{name} : \mathsf{string};\ .\,.\mathtt{target} : \mathsf{ino})\\ &\mid \mathtt{negdentry}(\,.\,.\mathtt{name} : \mathsf{string})\end{aligned}$$

Directories are only affected by *structural operations*, and hence *dirs* is only modified by those as well. For example, the operation **afs_create#** shown in Fig. 4.13 is used to add a new file node to the file system tree. The operation creates this file with the parent node *pinode* under the name *dent*.`name` and with metadata *md*. For this, a fresh inode number *cino* is allocated in *files* (*cino* must not be allocated in either *dirs* or *files*), and a new empty file $\mathtt{file}(md, 1, 0, \emptyset)$ is stored under *cino*. After creation, the file has exactly one hard link (from the parent directory *pinode*), size 0, and empty content. The parent directory must also be updated: a link from *dent*.`name` to *cino* is added to the `entries` of the parent directory and its `size` is increased by 1. To ensure that existing inodes and dentries are still valid after the operation for the use in `VFS`, the operation also updates affected data structures at the end of the operation: the `target` of *dent* is set to the new *cino* and the size of the parent inode *pinode* is increased accordingly. In addition, the inode *cinode* of the created directory is also created and returned.

Fig. 4.13 also shows the approach for specifying errors in `AFS`. All `AFS` operations are allowed to non-deterministically (**or**) fail, i.e., return a low-level error $\lfloor err \rfloor$ (cf. error model of `POSIX` in Sec. 4.2). The procedure **fail#** is used for this purpose.

$$\textbf{fail\#}(;\,; err)\ \{\ \textbf{choose}\ err_0\ \textbf{with}\ \lfloor err_0 \rfloor\ \textbf{in}\ err := err_0\ \}$$

This allows the implementations in the lower layers to return suitable errors, which cannot be specified on this level of abstraction. The implementation will resolve the nondeterminism to success whenever possible.

Of course, it must be ensured that the operation does not have some unwanted side effects, e.g., accidentally overwriting an existing file or directory (if there is already an entry with *dent*.`name` in *dirs*`[`*pino*`].entries`), hence suitable preconditions are checked when the operation is called. For file creation, the passed parent inode

```
afs_create#(md; pinode, cinode, dent; err)
  interface
  precondition   valid-negdentry(pinode.ino, dent, dirs, files)
                 ∧ valid-dir-inode(pinode, dirs, files)
{
  {
    let pino = pinode.ino in
      choose cino with ¬ cino ∈ dirs ∧ ¬ cino ∈ files ∧ cino ≠ 0 in
        files := (files ++ cino)[cino, file(md, 1, 0, ∅)];
        dirs[pino].entries[dent.name] := cino;
        dirs[pino].size := dirs[pino].size + 1;
        dent := dentry(dent.name, cino);
        cinode := inode(cino, md, false, 1, 0);
        pinode.size := pinode.size + 1;
  } or {
    fail#(; ; err);
  }
}
```

Figure 4.13: AFS operation for creating a file.

pinode must match a directory in *dirs* (given by the predicate valid-dir-inode) and *dent* must be a valid-negdentry for the directory *pinode*, i.e., *dent* must be a negdentry containing a name that is not in the entries of *dirs*[*pino*]. These preconditions are established by the surrounding VFS operations (here **vfs_create#**).

In order to for VFS to get the required inodes and dentries for AFS operations, AFS must also provide interface operations for reading the respective data structures from the file system. Fig. 4.14 shows the operation **afs_iget#** that retrieves an inode for a requested inode number *ino*. The operation may only be called when it is ensured that *ino* is a valid inode number, i.e., there is a node allocated under *ino* in either *dirs* or *files*. In the successful case, it uses the algebraic function getinode to retrieve the necessary data from the respective heap.

```
afs_iget#(ino; inode; err)
  interface
  precondition  ino ∈ dirs ∨ ino ∈ files
{
  {
    inode := getinode(ino, dirs, files);
    err := ESUCCESS;
  } or {
    fail#(; ; err);
  }
}
```

Figure 4.14: AFS operation for reading inodes.

$$\vdash \mathtt{getinode}(ino, dirs, files) = \\ (ino \in dirs \supset \mathtt{inode}(ino, dirs\mathtt{[}ino\mathtt{]}\mathtt{.meta}, \mathtt{true}, dirs\mathtt{[}ino\mathtt{]}\mathtt{.nlink}, \\ dirs\mathtt{[}ino\mathtt{]}\mathtt{.nsubdirs}, dirs\mathtt{[}ino\mathtt{]}\mathtt{.size}) \\ ; \mathtt{inode}(ino, files\mathtt{[}ino\mathtt{]}\mathtt{.meta}, \mathtt{false}, files\mathtt{[}ino\mathtt{]}\mathtt{.nlink}, 0, \\ files\mathtt{[}ino\mathtt{]}\mathtt{.size}))$$

On the other hand, **afs_lookup#** listed in Fig. 4.15 is used to look up entries within a directory. Given the inode number *pino* of a valid parent node, i.e., a directory node allocated in *dirs*, the operation checks whether there is an entry in the parent directory with the requested name. The reference parameter *dent* is used

```
afs_lookup#($pino$; $dent$; $err$)
  interface
  precondition valid-parent-ino($pino$, $dirs$, $files$)
{
  {
    if $dent$.name ∈ $dirs$[$pino$].entries then
      $dent$ := dentry($dent$.name, $dirs$[$pino$].entries[$dent$.name]);
      $err$ := ESUCCESS;
    else
      $dent$ := negdentry($dent$.name); $err$ := ENOENT;
  } or {
    fail#(; ; $err$);
  }
}
```

Figure 4.15: AFS operation for looking up dentries.

to pass the requested entry name (usually as a negdentry when called) and return a valid dentry with the inode number of the child node if the entry was found. Otherwise, a negdentry and the error code ENOENT are returned to signal that there is no corresponding entry in the directory.

The core *content operations* of AFS, i.e., the operations for modifying file content, are listed in Fig. 4.16. Recall that, unlike structural operations, they only access *files* and do not read or update *dirs*.

afs_readpage# reads the content of the page with number *pno* into a buffer *pbuf*. The file is determined as the inode number of a file inode *inode*. If the page does not exist, the buffer is set to all zeros (abbreviated as $\perp$), and the *exists* flag is set to false. The flag is ignored by VFS but will be relevant when caches are added (see Chapter 6).

afs_writepage# writes the content of *pbuf* to the respective page *pno*. Note that the page is allowed to be beyond the current file size (which is not modified in this operation). Both page operations are only defined for valid files (thus, *inode* must represent an existing file node in *files*) and must be called with a *pbuf* of suitable size (all pages in the file system must have equal size PAGE_SIZE).

The file size is updated with the operation **afs_writesize#**. The operation is only called during VFS writes when the content is extended. Hence, it is only allowed to be called when the given *sz* is greater than the current files size (the precondition valid-file-inode ensures that *inode*.size is the current size of the file in *files*). Furthermore, the operation does not consider whether potential junk pages become visible by the size increase.

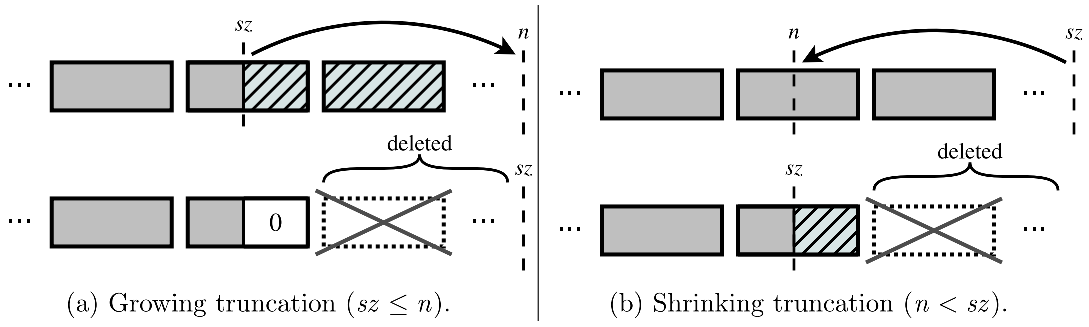
With the truncation operations shown in Fig. 4.18, the file size can be updated (both increased or decreased), including removing potential junk pages. The operation **afs_truncate#** is used for implementing **posix_truncate#**. It changes the file size to an explicit value n, checking that there is no junk data that would end up being part of the file below the new file size. This operation first discards all pages above the minimum sz_T of n and the old sz: The expression *content* upto sz_T keeps pages below sz_T only. For efficiency, the operation is *asymmetric* and distinguishes two cases, shown in Fig. 4.17. The first case in Fig. 4.17a is when the new size n is at least the old size sz. In this case, the page page(sz) may contain junk

```
**afs_readpage**#($inode$, $pno$; $pbuf$, $exists$; $err$)
  **interface**
  **precondition** valid-file-inode($inode, dirs, files$) $\land$ # $pbuf$ = PAGE_SIZE
{
  {
    $exists$ := $pno \in files$[$inode$.ino].content; $err$ := ESUCCESS;
    **if** $exists$ **then**
      $pbuf$ := $files$[$inode$.ino].content[$pno$]
    **else**
      $pbuf$ := $\bot$;
  } **or** {
    **fail**#(; ; $err$);
  }
}

**afs_writepage**#($inode$, $pno$, $pbuf$; ; $err$)
  **interface**
  **precondition** valid-file-inode($inode, dirs, files$) $\land$ # $pbuf$ = PAGE_SIZE
{
  {
    $files$[$inode$.ino].content[$pno$] := $pbuf$; $err$ := ESUCCESS;
  } **or** {
    **fail**#(; ; $err$);
  }
}

**afs_writesize**#($inode$, $sz$; ; $err$)
  **interface**
  **precondition** valid-file-inode($inode, dirs, files$) $\land$ $inode$.size < $sz$
{
  {
    $files$[$inode$.ino].size := $pbuf$; $err$ := ESUCCESS;
  } **or** {
    **fail**#(; ; $err$);
  }
}
```

Figure 4.16: AFS operations for reading and writing pages, and updating the file size.



(a) Growing truncation ($sz \leq n$). (b) Shrinking truncation ($n < sz$).

Figure 4.17: Effects of a truncation to n on a file with size sz.

data, which must be overwritten by zeros since this range becomes part of the file. Overwriting the part above $sz_T = sz$ with zeros is the result of the function call `truncate`(*content*`[`*pno*`]`, *sz*). This call can be avoided if the part is empty or if the old size was aligned. The second case in Fig. 4.17b is when the new file size is less than the old. Then, the page above the new file size simply becomes junk, and it does not need to be modified. The implementation of the **afs_truncate#** operation, therefore, avoids writing pages to persistent storage whenever this is possible[3].

afs_writebegin# is an optimized version of **afs_truncate#** for the case $n = sz$. It is called at the start of writing content to a file in `VFS` and makes sure that writing beyond the old file size will not accidentally create a page that contains junk. By having a dedicated operation (which is typically called significantly more often than **afs_truncate#**), the implementation `FFS` of `AFS` can avoid writing unnecessary inode updates to the journal.

VFS: Implementing Path Lookup, Permission Checks & Paging

Most `VFS` operations, in particular all structural operations, must traverse the directory tree using paths to the involved nodes before the respective `AFS` operation can be invoked. This is done by the auxiliary operation **vfs_walk#** listed in Fig. 4.19. Starting from an *ino*, it incrementally steps down one segment *p*`.head` of the path at a time by calling the `AFS` operation **afs_lookup#** (see Fig. 4.15). Before calling **afs_lookup#**, **vfs_maylookup#** verifies that the required permissions for reading the entries of *ino* are present. This includes to read the inode of *ino* from `AFS` using **afs_iget#** (see Fig. 4.14) in order to access its metadata. If the access is granted, **afs_lookup#** checks whether an entry with the name of the requested dentry exists in the current directory *ino* and updates *dent* with the inode number of the entry *cino* if so. After successful traversal, the required inodes are loaded (for example, the parent inode *pinode* for **afs_create#**), checks specific to the operation are performed, and the `AFS` operation is invoked.

Note that the operation is declared with an extensive contract, specifying all possible outcomes of the operation: a low-level error $\lfloor err \rfloor$ could occur; `ENOENT` is returned if the requested path is not a valid path starting from the initial inode number (recall that $ino^{\grave{}}$ denotes the value of the reference parameter *ino* at the time the operation was called); `EACCESS` is returned if the requested path runs through a

[3]Deleting a page does not write it, but adds a "page deleted" entry to the journal.

```
**afs_truncate**#($n$; $inode$; $err$)
  **interface**
  **precondition** valid-file-inode($inode, dirs, files$)
{
  {
    **let** $ino = inode$.ino, $sz = inode$.size **in**
      **let** $content = files$[$ino$].content, $sz_T$ = min($n, sz$), $pno$ = page($sz$),
          $aligned$ = aligned($sz$), $modify$ = false
      **in**
        $modify := sz \leq n \land pno \in content \land \neg\ aligned$;
        **if** $modify$ **then**
          $content$[$pno$] := truncate($content$[$pno$], $sz$);
        $files$[$ino$].content := $content$ upto $sz_T$;
        $files$[$ino$].size := $n$, $inode$.size := $n$; $err$ := ESUCCESS;
  } **or** {
    **fail**#(; ; $err$);
  }
}

**afs_writebegin**#($inode$; ; $err$)
  **interface**
  **precondition** valid-file-inode($inode, dirs, files$)
{
  {
    **let** $ino = inode$.ino, $sz = inode$.size **in**
      **let** $content = files$[$ino$].content, $pno$ = page($sz$), $aligned$ = aligned($sz$)
      **in**
        **if** $pno \in content \land \neg\ aligned$ **then**
          $content$[$pno$] := truncate($content$[$pno$], $sz$);
        $files$[$ino$].content := $content$ upto $sz_T$; $err$ := ESUCCESS;
    }
  } **or** {
    **fail**#(; ; $err$);
  }
}
```

Figure 4.18: Truncation operations of AFS.

```
vfs_walk#($p$, $user$; $ino$; $err$)
  auxiliary
  precondition  valid-parent-ino($ino$, $dirs$, $files$)
  postcondition   $\lfloor err \rfloor$
                $\lor$ ($err$ = ENOENT $\land \neg$ ispath($p$, $ino'$, $dirs$, $files$))
                $\lor$ ($err$ = EACCESS
                    $\land \exists\ p_0.\ p_0 \sqsubseteq p$.parent
                        $\land$ ispath($p_0$, $ino'$, $dirs$, $files$)
                        $\land \neg$ px($user$, $p_0$, $ino'$, $dirs$, $files$))
                $\lor$ ($err$ = ESUCCESS $\land$ ispath($p$, $ino'$, $ino$, $dirs$, $files$)
                    $\land$ px($user$, $p$.parent, $ino'$, $dirs$, $files$))
{
  $err$ := ESUCCESS;
  while $p \neq \varepsilon \land err$ = ESUCCESS do
    choose $inode$ in { vfs_maylookup#($ino$, $user$; $inode$; $err$) };
    if $err$ = ESUCCESS then let $dent$ = negdentry($p$.head) in
      afs_lookup#($ino$; $dent$; $err$);
      if $err$ = ESUCCESS then
        $ino$ := $dent$.target; $p$ := $p$.tail;
}
```

Figure 4.19: VFS operation for traversing the directory tree, i.e., *walking* along a path.

node for which *user* has not the required execute permissions ($p_0 \sqsubseteq p_1$ denotes that p_0 is a prefix of p_1); and ESUCCESS is returned if p is a valid path, starting from ino' and leading to *ino*, and *user* has execute permissions for all nodes along the path. This contract is formulated in such a way that it can be applied in all main VFS operation proofs, which avoids that an complex invariant proof has to be performed for each operation that uses **vfs_walk#**.

Figure 4.20 shows how a complete structural operation of VFS is composed, exemplary for file creation. The tree is traversed with **vfs_walk#**, starting from the root identified by the constant ROOT_INO and targeting the directory in which the file should be created (p.parent). If walking was successful, the operation **vfs_maycreate#** checks if the file can be created in the reached directory and returns a respective error when this is not the case (matching the possible errors of the POSIX precondition predicate pre-create from the last section). The operation works similarly to **vfs_maylookup#**, and VFS contains a handful of such auxiliary operations (vfs_may...) for checking if an operation of VFS can be executed successfully (provided that no low-level error occurs). However, most of them can be used for multiple operations, e.g., **vfs_maycreate#** is also used for **vfs_mkdir#**. Finally, the corresponding AFS operation is called to perform the actual file creation.

Besides structural operations, VFS also has to implement content operations. Like for POSIX, these operations do not work on paths but on file descriptors, which identify file handles storing access modes and positions for opened files. VFS uses a data structure very similar to *ofh* of POSIX for managing opened files.

component VFS **using** AFS

state *of* : map(nat, fhandle), *maxfd* : nat

The state variable *of* maps file descriptors (represented by natural numbers) to file

```
vfs_create#($p$, $md$, $user$; ; $err$)
  interface
{
  if $p = \varepsilon$ then $err$ := EEXISTS
  else let $ino$ = ROOT_INO, $dent$ = negdentry($p$.last), $p_0 = p$.parent,
           $inode$ = ?, $cinode$ = ?
  in
    vfs_walk#($p_0$, $user$; $ino$; $err$);
    if $err$ = ESUCCESS then
      vfs_maycreate#($ino$, $dent$, $user$; $inode$; $err$);
    if $err$ = ESUCCESS then
      afs_create#($md$; $inode$, $cinode$, $dent$; $err$);
}
```

Figure 4.20: VFS operation for the creation of files.

handles: the type fhandle is nearly identical to the analogous type used in POSIX, however, it uses inode numbers ino as identifiers instead of the anonymous fid type.

data fhandle = fh(. .ino : ino; . .mode : mode; . .pos : nat)

Hence, the file handle operations (opening and closing a file, or seeking within a file) are implemented quite similar to their specification counterparts in POSIX. Consider the **vfs_open#** operation given in Fig. 4.21 as an example. Traversing the tree is again done using **vfs_walk#**, starting from ROOT_INO. When walking was successful, the operation **vfs_mayopen#** checks if the reached node is a file and if *user* has the permissions to open the file with *mode*.

Instead of choosing the file descriptor *fd* non-deterministically, the VFS implementation uses an additional state variable *maxfd* to ensure that no file descriptor is used twice. File descriptors are issued in ascending order, and *maxfd* stores the descriptor that will be used next. Therefore, **vfs_open#** increments *maxfd* after it was used for the current request.

```
vfs_open#($p$, $mode$, $user$; $fd$; $err$)
  interface
{
  let $ino$ = ROOT_INO, $inode$ = ? in
    vfs_walk#($p$, $user$; $ino$; $err$);
    if $err$ = ESUCCESS then
      vfs_mayopen#($ino$, $mode$, $user$;
                        $inode$; $err$);
    if $err$ = ESUCCESS then
      $of$[$maxfd$] := fh($ino, mode, 0$);
      $fd := maxfd$;
      $maxfd := maxfd + 1$;
}
```

Figure 4.21: VFS operation for opening a file.

Writing a buffer *buf* of length n to an opened file is implemented in VFS with the operation **vfs_write#** shown in Fig. 4.22. Like **posix_write#** (see Fig. 4.6), the operation takes a file descriptor *fd*, a *user*, and a buffer *buf* as input. The parameter n contains the requested number of bytes to write. But since the POSIX specification also allows to write only an initial segment of *buf*, the operation modifies n at the end to return the number of bytes that were actually written (lines 19 and 21, respectively). Note that the operation does not perform any permission checks for *user* as these have already been performed when the file was opened. Thus, the operation only checks whether *fd* is a valid file descriptor created with a mode that allows writing (lines 2 and 3). After these checks,

```
   vfs_write#(fd, user, buf; n; err)
     interface
     precondition $n \le \# buf$
   {
1    $err :=$ ESUCCESS;
2    if $\neg fd \in of$ then $err :=$ EBADFD
3    else if $of$[fd].mode $\ne$ MODE_W $\wedge$ $of$[fd].mode $\ne$ MODE_RW then $err :=$ EBADFD
4    else choose inode in
5      afs_iget#(of[fd].ino; inode; err);
6      if $err =$ ESUCCESS then
7        if inode.isdir then $err :=$ EISDIR
8        else let $start = of$[fd].pos, $end = of$[fd].pos $+ n$, $written = 0$ in
9          afs_writebegin#(inode; ; err);
10         if $err =$ ESUCCESS then
11           vfs_writeloop#(inode, start, end, buf; ; written, err);
12           if $written \ne 0$ then $err :=$ ESUCCESS;
13         if $err =$ ESUCCESS then let $sz = inode$.size in
14           $end := start + written$;
15           if $sz < end$ then
16             afs_writesize#(inode, end; ; err);
17           if $err =$ ESUCCESS then $n := written$
18           else
19             $n := (start \le sz \supset sz - start; 0)$;
20             $err :=$ ESUCCESS;
21           $of$[fd].pos $:= start + n$;
   }
```

Figure 4.22: Write operation of the VFS component.

the inode of the file identified by *fd* is loaded using **afs_iget#** (line 5). The inode is necessary for further calls to AFS and to determine the file's current size, which is used to determine if the file size must be increased at the end of the write (lines 15 and 17). As final preparation, **afs_writebegin#** is called to ensure that junk data above the file size is removed.

Writing data then is done by splitting up the the buffer (from *start* to *end*) into pieces that align with page boundaries in the auxiliary operation **vfs_writeloop#** (listed in Fig. 4.23). The *start* and *end* positions are extracted from the file handle *of*[*fd*], and the variable *written* keeps track of how many bytes have been written already. For each page-aligned piece of the buffer, **vfs_writepage#** is called. This operation still has the full buffer *buf* as input and writes *n* bytes of the buffer, starting at position *written*. The *n* bytes are placed into the page *pno*, starting at *offset*. **vfs_writeloop#** calculates the arguments for **vfs_writepage#** using the `offset` and `rest` functions, so that the sum of *n* and *offset* always is less or equal to `PAGE_SIZE` (see lines 4 and 5 of Fig. 4.23 and the precondition of Fig. 4.24). If the sum is less than the page size (i.e., when either $offset \ne 0$ or $n \ne$ `PAGE_SIZE`), the old page must be loaded into a page buffer *pbuf* by calling **afs_readpage#** (lines 2 and 3 in Fig. 4.24), and the relevant part of *buf* must be copied into the page using the function `copy` before the page is written via **afs_writepage#** (lines 6 and 7). Note that *offset* is set to `offset`(*curPos*) and can thus be non-zero only in the first iteration of the loop since subsequent iterations will always write full pages (except

```
vfs_writeloop#($inode, start, end, buf; ; written, err$)
  auxiliary
  precondition    valid-file-inode($inode, dirs, files$)
                  $\wedge\ start < end \wedge end - start \leq \#\ buf$
{
  $err$ := ESUCCESS, $written := 0$;
  while $err$ = ESUCCESS $\wedge\ start + written \neq end$ do
    let $curPos = start + written$ in
    let $offset$ = offset($curPos$), $pno$ = page($curPos$), $rest$ = rest($curPos$) in
    let $n$ = min($end - curPos, rest$) in
      vfs_writepage#($inode, pno, buf, written, offset, n; ; err$);
      if $err$ = ESUCCESS then $written := written + n$;
}
```

Figure 4.23: Auxiliary operation of VFS for breaking down a POSIX write into page writes.

```
vfs_writepage#($inode, pno, buf, written, offset, n; ; err$)
  auxiliary
  precondition    valid-file-inode($inode, dirs, files$)
                  $\wedge\ written + n \leq \#\ buf \wedge offset + n \leq$ PAGE_SIZE
{
  $err$ := ESUCCESS;
  let $pbuf = \perp$, $exists$ = false in
    if $n <$ PAGE_SIZE then
      afs_readpage#($inode, pno; pbuf, exists; err$);
    if $err$ = ESUCCESS then
      $pbuf$ := copy($buf, written, pbuf, offset, n$);
      afs_writepage#($inode, pno, pbuf; ; err$);

}
```

Figure 4.24: Auxiliary operation of VFS for writing a page.

for the last iteration, in which only the beginning of a page may be written).

Since all calls of AFS operations can return an error and writing stops in this case, the number *written* of bytes actually written is returned from **vfs_writeloop#**. It is finally used in **vfs_write#** to modify n. If the file size has been increased, which is the case when $start + written$ is bigger than the old file size sz, **vfs_write#** adjusts it by calling **afs_writesize#** (lines 13-16). As long as at least one byte was written, **vfs_write#** always returns ESUCCESS (line 12), even when increasing the file size was necessary but did not succeed (line 20). In this case, the operation signals that just the range from *start* to the initial size sz has been written (line 19). Finally, the file handle is updated by moving the position to the end position of the write (line 24), so that subsequent appending writes do not have to move it manually with **vfs_seek#**.

Reading a buffer from the content of a file with **vfs_read#** (not shown) is implemented quite similarly to **vfs_write#**: the requested range (determined by the current position of the file handle and the requested number of bytes to read) is partitioned into page-sized and page-aligned chunks by the auxiliary operation **vfs_readloop#**, which reads these pages via repeated calls of **afs_readpage#**.

```
**vfs_truncate#**$(p, n, user; ; err)$
  **interface**
{
  **let** $ino$ = ROOT_INO, $inode$ = ? **in**
    **vfs_walk#**$(p, user; ino; err)$;
    **if** $err$ = ESUCCESS **then**
      **vfs_mayopen#**$(ino,$ MODE_W$, user; inode; err)$;
    **if** $err$ = ESUCCESS **then**
      **afs_truncate#**$(n; inode; err)$;
}
```

Figure 4.25: Truncate operation of VFS for modifying the size of a file.

Of course, **vfs_read#** does not need to care about junk pages, so there is no need to call **afs_writebegin#** at the beginning of the operation, and there is no need to update the file size either.

The implementation of truncations in VFS (shown in Fig. 4.25) resembles the implementation of structural operations more than other content operations. As the operation takes a path as input instead of a file descriptor, it must first traverse the directory tree with **vfs_walk#**. It then checks via **vfs_mayopen#** if the node reached is a file and if *user* has permission to modify the file (MODE_W). If the checks were successful, the truncation is performed with a call to **afs_truncate#**.

Writing and truncating are the central two operations affected when write-back caching for files is added (together with a non-trivial implementation of a synchronization operation). We will see in Chapter 6 that, when adding caches, it is crucial for correctness that VFS implements writing by traversing the pages from low to high page numbers. We will also find that the data representation of VFS, where all calls are optimized for efficiency, which in particular results in an asymmetric **afs_truncate#** (Fig. 4.17), is one of the main difficulties for adding caches correctly.

Correctness & Crashes

The major part of the state of VFS(AFS) lies in AFS in form of *dirs* and *files*. In particular, the persistent part of the state is defined solely in AFS (to match the characteristics of crash-modularizable components, see Def. 15). Hence, the main invariants and the crash predicate are formulated in AFS while VFS has a crash predicate equal to true (since it is a RAM component).

component AFS
crash predicate $\texttt{afs-crash}(dirs', files', dirs, files)$
invariants $\texttt{afs-cons}(dirs, files)$

The crash predicate afs-crash matches the statements about *root* and *fs* of the predicate posix-crash given in Sec. 4.2.

$$\begin{aligned}\vdash \texttt{afs-crash}(dirs_0, files_0, dirs_1, files_1) \leftrightarrow&\\ &dirs_1 = dirs_0 \setminus \texttt{orphans}(dirs_0, files_0)\\ \wedge\ &files_1 = files_0 \setminus \texttt{orphans}(dirs_0, files_0)\end{aligned}$$

The heaps *dirs* and *files* after a crash result from removing all entries of orphans (the function `orphans`$(dirs_0, files_0)$ returns a `set(ino)` of inode numbers that are allocated in $dirs_0$ resp. $files_0$ but have no links pointing to them). While `POSIX` only needs to consider file orphans, `AFS` also considers directory orphans due to how directories are removed by `VFS(AFS)`. For example, the `AFS` operation called by **vfs_rmdir#** removes the entry from the parent directory (together with decreasing the size and subdirectory count) and decreases the link count of the child directory but does not remove the child directory from *dirs*. This is outsourced to a separate `AFS` operation (**afs_evict#**) that checks whether a file or directory has a link count equal to zero and then removes the node from the respective heap if so. That way, **afs_evict#** can be used in multiple `VFS`, e.g., **vfs_rmdir#**, **vfs_unlink#**, or **vfs_rename#**. This split results in orphaned directories for a short time, but this does not pose an additional challenge since the recovery mechanism of the implementation of `AFS` handles orphaned files and directories uniformly.

The invariant predicate `afs-cons` collects all sorts of consistency conditions over the file and directory heap, as shown in the following excerpt of its definition.

$$
\begin{aligned}
&\vdash \texttt{afs-cons}(dirs, files) \leftrightarrow \\
&\quad \forall\ ino.\ \ \big(ino \in dirs \to \neg\ ino \in files \land dirs\texttt{[}ino\texttt{].nlink} \le 1 \\
&\qquad\qquad\qquad\qquad \land\ dirs\texttt{[}ino\texttt{].size} = \#\ dirs\texttt{[}ino\texttt{].entries} \\
&\qquad\qquad\qquad\qquad \land\ \ldots\ \big) \\
&\qquad\quad \land \big(ino \in files \to \neg\ ino \in dirs \land\ \ldots\ \big) \\
&\qquad\quad \land \big(\forall\ name.\ ino \in dirs \land name \in dirs\texttt{[}ino\texttt{].entries} \to \\
&\qquad\qquad\qquad \texttt{valid-ino}(dirs\texttt{[}ino\texttt{].entries[}name\texttt{]}, dirs, files)\big) \\
&\qquad\quad \land \big(\forall\ n.\ ino \in files \land n \in files\texttt{[}ino\texttt{].content} \to \\
&\qquad\qquad\qquad \#\ files\texttt{[}ino\texttt{].content[}n\texttt{]} = \texttt{PAGE\_SIZE}\big)
\end{aligned}
$$

It ensures that an inode number is not used for both a file and a directory and that the fields of the nodes store the correct values, e.g., that `size` of a directory contains the number of its `entries` (first and second conjunct). Furthermore, it specifies that only valid inode numbers can be targeted by the `entries` of a directory, i.e., all directory entries must be allocated in *dirs* or *files* (third conjunct). Finally, all buffers of the `content` of a file must be valid pages, i.e., have the correct size `PAGE_SIZE` (fourth conjunct).

On the other hand, the invariant of `VFS` ensures consistency of the file descriptors and handles stored in *of*. A descriptor in *of* must not be lower than *maxfd* (since otherwise, the mechanism for issuing new descriptors could be corrupted), and all entries of *of* must point to existing files (and thus, orphaned files are not deleted until they are no longer open).

component `VFS` **using** `AFS`

invariants $\forall\ fd.\ fd \in of \to fd < maxfd \land of\texttt{[}fd\texttt{].ino} \in files$

The refinement of $\texttt{VFS}(\texttt{AFS}) \leq \texttt{POSIX}$ is proven using Separation Logic (see Sec. 2.5), with an abstraction similar to the one of red-black trees (see Sec. 3.2).

$$\begin{aligned}
&\textbf{data refinement } \texttt{VFS}(\texttt{AFS}) \leq \texttt{POSIX} \\
&\textbf{abstraction relation} \quad \mathit{ofh} = \texttt{abs-of}(\mathit{of}) \\
&\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{fs} = \texttt{abs-files}(\mathit{files}) \\
&\qquad\qquad\qquad\qquad\quad \wedge\ \texttt{abs}(\mathit{root}, \texttt{ROOT\_INO})(\mathit{dirs})
\end{aligned}$$

Since *of* of `VFS` is basically identical to *ofh* of `POSIX`, *of* can be mapped directly to *ofh* with the function `abs-of` by concretizing the type fid to ino. The function `abs-files` builds a file store from a file tree: a `POSIX file` can be constructed from an `AFS file` by taking over the metadata and combining all content pages up to the file's size into one continuous buffer. The heap predicate $\texttt{abs} : (\mathsf{tree} \times \mathsf{ino}) \rightarrow \mathsf{iheap}(\mathsf{dir}) \rightarrow \mathsf{bool}$ abstracts the pointer tree in *dirs* starting at `ROOT_INO` to the algebraic directory tree *root*. The predicate is defined analogously to the red-black tree abstraction, except that the recursive case requires separating into an arbitrary number of cases (as a directory can have an arbitrary number of entries). In contrast, binary trees are separated into a fixed number of disjoint parts: the parent node, a left subtree, and a right subtree.

4.4 Related Work

File system correctness has been an active research topic for some time, starting with the earliest formal model of the POSIX standard written in Z by Morgan and Sufrin [83].

NASA's proposal to build a verifiable file system [72] has prompted a large body of related work, covering many aspects of file systems in general and also specific to flash memory. Mechanized models have been developed in [19, 43, 30, 29, 46, 61].

There has also been work on specific issues, e.g., reading and writing files at byte-level has been addressed in [6, 73]. Model checking, the existing VFS code has been abstracted, was been used for proving specific properties, e.g., memory safety or the correct usage of locks [47, 85, 118]. A nice summary of this early work is [77].

While these approaches have made interesting contributions, they all considered specific aspects, specific layers of abstraction, or specific properties only. For example, none of the approaches considers crash-safety or the separation of common functionality (`VFS`) and file system specific parts (`AFS`). By considering particular aspects only, none of them produced executable code.

Apart from Flashix, there are two other approaches that have generated running code for full-featured file systems.

Amani et al. design the flash file system BilbyFS [4, 3, 2] as a case study for their tool Cogent [91] for generating verified C code. The system can also derive specifications for Isabelle/HOL [103]. BilbyFS has a similar but simpler structure as Flashix. For instance it builds on top of the Erase Block Manager (using UBI [51] instead of MTD. It is implemented for the use with the Linux VFS, so it implements an interface similar to `AFS`. Thus, caching on the level of the Virtual File System Switch is not considered in BilbyFS, but lower-level caching mechanism are supported. The verification of the file system is incomplete (only the functional correctness of the operations *iget* and *sync* are proven) and crash-safety has also not been considered so far.

Closest to Flashix is the FSCQ file system by Chen et al. [25, 23], a sequentially implemented file system which is targeted for regular disks with random access, not flash memory (and is therefore much simpler than Flashix, the code size is approx. 4K). Haskell code is derived from the specifications that can be integrated in Linux via FUSE. Like for Flashix, crash-safety was a major concern in the development and verification process of FSCQ. Chen et al. developed *Crash Hoare Logic (CHL)*, an extension of traditional Hoare Logic with crash conditions, and proved the correctness of FSCQ using Coq [10]. FSCQ was also extended by performance oriented features over the years: DFSCQ [24] added caching mechanics to the file system, and CFSCQ [20, 21] introduces an additional layer at the bottom of the file system performing concurrent I/O operations to the disk. These additions follow the same motivations as the caching and concurrency extensions in Flashix, however, they use different approaches and are not as extensive as the ones presented in Chapter 6 and Chapter 7. Recent extensions gear towards security aspects like confidentiality in SFSCQ [64] or data integrity in IFSCQ [113], which are currently not covered by Flashix.

Chapter 5

Order-Preserving Caches in File Systems

Summary File systems typically implement some kind of low-level buffer cache to reduce the number of I/O-operations on the underlying storage medium. As the first non-local extension to the Flashix file system, the `Write Buffer` was integrated into the hierarchy, which buffers writes until a complete page can be written. The use of such buffer caches has an impact on the crash behavior of the layers above them. In the event of a crash, the contents of caches are lost as they are only located in RAM. Thus, crash-safety criteria of components with underlying buffer caches must guarantee that losing the buffered data does not affect the structural integrity of the file system.
This chapter gives a brief summary of the `Write Buffer` extension, which is part of the work of Pfähler [96] and resulted in an extension of the semantics of sequential components to cope with the properties of such order-preserving caches. Instead of using a state-based approach for specifying crash effects, an operations-based approach is chosen, where crashes are explained by alternative, crash-free histories that differ only slightly from the original ones. Based on this, the crash-safety criterion *Quasi-Sequential Crash Consistency* for buffered file systems is formulated on the level of `POSIX`: basically, a crash reverts a postfix of the last executed operations.
This chapter extends Pfähler's work to concurrent components by giving a history-based definition of retractions and Quasi-Sequential Crash Consistency: re-executing one operation is generalized to the re-execution of a set of pending operations. The re-formulation is used in Chapter 6 to show compatibility of buffer caches with non-order-preserving high-level caches.

Contents

Publications Crash-safety of order-preserving caches and the correctness of the `Write Buffer` integration is published in [97], and it is also covered in more detail in Pfähler's thesis [96]. The re-formulation of Quasi-Sequential Crash Consistency in terms of histories was performed as part of the publication [15].

5.1 Retracting Components

In Sec. 3.1 the concept of hierarchical components was introduced, together with their semantics for the case that they are *non-retracting* and used in a *sequential* system. In this section, the semantics is adjusted to *retracting* components, which are relevant as soon as write-back caches are used. In a retracting component C, not all states are synchronized, i.e., the set of synchronized states $\mathtt{SYNC}^{\mathtt{C}}$ is a real subset of the set of states $\mathtt{S}^{\mathtt{C}}$. Intuitively, this means that the component can be in a state where some (non-recoverable) data resides in volatile memory only, such that a crash in this state results in actual data loss. This cannot be prevented when one wants to benefit from caching, but (critical) systems must at least provide some guarantees that allow rebooting to a consistent state, ideally with as little data loss as possible.

Furthermore, users must be able to trigger mechanisms for ensuring that certain data is not lost during a crash. This is usually achieved by providing *synchronization operations* that synchronize the complete system state or parts of it. For example, file systems implement a *fsync* operation that synchronizes the content of one particular file. Another goal is to give the system users an understanding of the possible outcomes of a crash. This is complicated using the *state-based* approach for specifying the effects of a crash by specifying a *crash predicate* as given in Sec. 3.1. Using caches on lower levels of the hierarchy would require to explicitly distinguish between volatile and persistent state on every abstraction level, which would complicate specifications (and thus also proofs) significantly or sometimes is not even possible in the first place. For example, there is no natural way in the `POSIX` specification of Sec. 4.2 to express that a write to a flash page (which could contain data of multiple, potentially partial `POSIX` operations) is currently cached.

Therefore, the state-based approach is extended by an *operations-based* crash effect: a crashed state can be explained by an alternative sequence of executed operations that may differ slightly from the original one. This is motivated by the buffer cache in the `Write Buffer` component in the lower part of the Flashix hierarchy, which queues writes to the flash storage until a full flash page can be written (see Sec. 5.2). The cache is *order-preserving*, i.e., writes are emitted to the underlying storage in the same order as they were handed to the cache. As all writes pass this buffer cache, the property causes some recent write requests to be lost in the event of a crash. In other words, a crash essentially has the effect of *retracting* several operations, namely those whose data lies entirely in the cache.

The semantics of retracting components can thus be derived from the semantics of non-retracting components by exchanging the crash transitions with *retracting crash transitions* as given by Def. 33. The idea is similar to *Buffered Durable Linearizability* by Izraelevitz et al. [66]: a postfix h_1 of a (concurrent) history $h = h_0 \cdot h_1$ is lost, and pending operations in h_0 can be completed. However, these completions must not alter the state while we allow re-executions that modify the state (and yield other outputs).

Definition 33 (Retracting Crash Transitions). *A transition $s_n \xrightarrow{\lightning} s_{n+1}$ is a retracting crash transition for an era $I, h^{\mathtt{C}}$ of a component* C *and a system $System^{\mathtt{C}}$ with $I = (s_0, \ldots, s_n)$ and $I, h^{\mathtt{C}} \models System^{\mathtt{C}}_{\lightning}$ iff*

1. *The interval $I, h^{\mathtt{C}}$ can be split into two intervals I_0 and I_1 with corresponding*

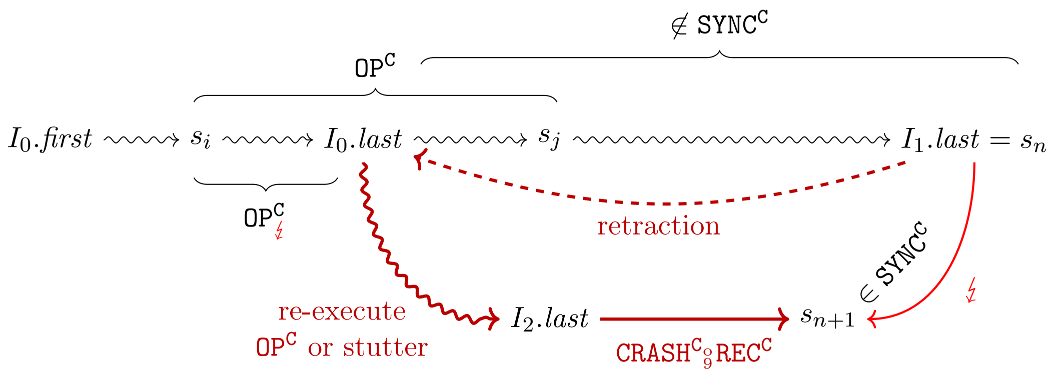

$\notin \mathtt{SYNC}^{\mathtt{C}}$
$\mathtt{OP}^{\mathtt{C}}$
$I_0.\mathit{first}$
s_i
$I_0.\mathit{last}$
s_j
$I_1.\mathit{last} = s_n$
$\mathtt{OP}^{\mathtt{C}}_{↯}$
retraction
$\in \mathtt{SYNC}^{\mathtt{C}}$
↯
re-execute
$\mathtt{OP}^{\mathtt{C}}$ or stutter
$I_2.\mathit{last}$
s_{n+1}
$\mathtt{CRASH}^{\mathtt{C}} ⨾ \mathtt{REC}^{\mathtt{C}}$

Figure 5.1: Constructing a retracting crash transition of a sequential component $\mathtt{C}$.

histories $h^{\mathtt{C}}_0$ and $h^{\mathtt{C}}_1$ where steps of I_1 do not result in synchronized states, i.e., $I = I_0 ⨾ I_1$ and $h^{\mathtt{C}} = h^{\mathtt{C}}_0 \cdot h^{\mathtt{C}}_1$ with $I_1(k) \notin \mathtt{SYNC}^{\mathtt{C}}$ for all $0 < k \leq \#I_1$.

2. *There exists an interval I_2 with a corresponding history $h^{\mathtt{C}}_2$ where $h^{\mathtt{C}}_2$ contains only response events of operations pending in $h^{\mathtt{C}}_0$ and where the concatenation with $I_0, h^{\mathtt{C}}_0$ forms an uninterrupted system run, i.e., $I_0 ⨾ I_2, h^{\mathtt{C}}_0 \cdot h^{\mathtt{C}}_2 \models \mathit{System}^{\mathtt{C}}$.*

3. *The crashed state s_{n+1} can be reached from $I_2.\mathit{last}$ by applying the crash and recovery relations, i.e., $(I_2.\mathit{last}, s_{n+1}) \in \mathtt{CRASH}^{\mathtt{C}} ⨾ \mathtt{REC}^{\mathtt{C}}$.*

The definition formalizes the construction of a retracting crash transition, as visualized in Fig. 5.1 for the sequential case. Given an era with interval I and history $h^{\mathtt{C}}$ of a component $\mathtt{C}$, the construction starts from the last state s_n of I, targeting the crashed state s_{n+1}. A retraction reverts several system steps (it "*jumps back in time*"), splitting I into a kept prefix I_0 and a dropped suffix I_1. I_1 must not contain synchronized states (except for the initial state $I_1.\mathit{first} = I_0.\mathit{last}$) as no synchronized steps are reverted. Similar to $\mathtt{CRASH}^{\mathtt{C}}$ and $\mathtt{REC}^{\mathtt{C}}$, $\mathtt{SYNC}^{\mathtt{C}}$ results from building the Cartesian product of $\mathtt{Sync}^{\mathtt{C}}$ with the respective state sets of $\mathtt{C}$'s subcomponents. The retraction may lead to a state in which an operation is currently running, so there may be a *pending operation* in the prefix era $I_0, h^{\mathtt{C}}_0$. Such an interrupted operation execution, like $\mathtt{OP}^{\mathtt{C}}_{↯}$ in Fig. 5.1, can be completed by an completion interval I_2 with history $h^{\mathtt{C}}_2$. The $h^{\mathtt{C}}_2$ is only allowed to complete the pending operation by adding a suitable response but must not contain invocations. We call this a re-execution of the operation as the completed response is allowed to yield a different result than in the original, uninterrupted execution. This is, for example, already relevant on the level of $\mathtt{POSIX}$, where a retraction could result in fewer pages being written in a write operation. Then the re-execution of this write operation must return a corresponding smaller value n of written bytes in order to match the resulting state. Alternatively, a pending operation can also stutter, which allows the operation to have no effect, e.g., when all modifying persisting steps of the operation were retracted (lie between $I_0.\mathit{last}$ and s_j). Finally, the residual state-based crash effect, i.e., the crash relation $\mathtt{CRASH}^{\mathtt{C}}$ with subsequent recovery $\mathtt{REC}^{\mathtt{C}}$, is applied to $I_2.\mathit{last}$, yielding the crashed state s_{n+1}. Recall that this state is guaranteed to be a synchronized state since all states targeted by the recovery relation $\mathtt{REC}^{\mathtt{C}}$ are in $\mathtt{SYNC}^{\mathtt{C}}$ by Def. 7, so following crashes will not retract past this recovered state.

Note that Def. 33 is explicitly not limited to sequential systems but can be applied for runs of an arbitrary $\mathit{System}^{\mathtt{C}}$. Furthermore, $I_2, h_2^{\mathtt{C}}$ is not restricted to the completion of one operation as in Fig. 5.1 but may complete several pending operations. This liberal definition will be exploited for the semantics of concurrent (retracting) systems in Def. 40. However, Fig. 5.1 depicts the sequential case using the system $\mathit{System}^{\mathtt{C}} \equiv \{\ \mathtt{OP}^{\mathtt{C}}\ \}^*$ from Def. 27, where at most one operation can be pending in $h_0^{\mathtt{C}}$. Accordingly, Def. 34 gives the adjusted semantics for sequential, retracting components.

Definition 34 (Semantics of Sequential, Retracting Components)**.** *The semantics of a sequential, retracting component* C *is given by the labeled transition system* $C = (\mathtt{S}^{\mathtt{C}}, \mathtt{INIT}^{\mathtt{C}}, \mathcal{L}^{\mathtt{C}} \cup \{\tau\}, \rightarrow_{\curvearrowleft}^{seq})$*. The state transition relation* $\rightarrow_{\curvearrowleft}^{seq}$ *is determined by the set of observable system runs* $(I, h^{\mathtt{C}}) \in \mathit{runs}(C)$ *satisfying the conditions (1)-(5) and (7) of Def. 27 and*

6a. Each crash transition $I_i.\mathit{last} \xrightarrow{\lightning_{i+1}} I_{i+1}.\mathit{first}$ *with* $i < n$ *is a retracting crash transition for the era* $I_i, h_i^{\mathtt{C}}$ *and* $\mathit{System}^{\mathtt{C}} \equiv \{\ \mathtt{OP}^{\mathtt{C}}\ \}^*$ *according to Def. 33.*

The invariant proof obligation for retracting components must be adjusted slightly compared to the ones of non-retracting components since they must now ensure that the states reached by initialization and recovery procedure executions are in fact synchronized according to Def. 7.

Lemma 4 (Invariants of Sequential, Retracting Components)**.** *For a sequential, retracting component* C*, the proof obligations (2) and (3) of Lem. 1 together with the following proof obligations ensure termination and the absence of exceptions, establish the sequential invariant* $\mathit{inv}(\underline{s})$*, guarantee that* C *calls its subcomponents' interface operations only if their preconditions are satisfied.*

1a. $\vdash \langle\!|\mathbf{init}\#(\underline{x};\,;\underline{s},\underline{z})|\!\rangle\,(\varphi(\underline{s},\underline{z}) \rightarrow \mathit{inv}(\underline{s}) \wedge \mathit{synced}(\underline{s}))$

4a. $\mathit{inv}(\underline{s}_0),\ \mathit{crash}(\underline{s}_0, \underline{s}) \vdash \langle\!|\mathbf{recover}\#(\underline{x};\underline{s};\underline{z})|\!\rangle\,(\varphi(\underline{s},\underline{z}) \rightarrow \mathit{inv}(\underline{s}) \wedge \mathit{synced}(\underline{s}))$

□

Similarly, the proof obligations for data refinement must be extended by the additional obligation given in Thm. 5. The obligation restricts A to having fewer synchronized states than C so that all retractions on C are also allowed on A. Or in other words, the concrete component C is only allowed to "*jump back*" less than the abstract component A.

Theorem 5 (Data Refinement of Sequential, Retracting Components)**.** *The refinement* $\mathtt{C} \leq \mathtt{A}$ *of sequential, retracting components* A *and* C*, where* C *is compatible with* A*, is implied by the forward simulation* $\mathit{abs}(\underline{s}^{\mathtt{A}}, \underline{s}^{\mathtt{C}})$ *satisfying the conditions (1)-(5) of Thm. 1 and the following one.*

6. $\vdash \mathit{synced}^{\mathtt{A}}(\underline{s}^{\mathtt{A}}) \wedge \mathit{abs}(\underline{s}^{\mathtt{A}}, \underline{s}^{\mathtt{C}}) \rightarrow \mathit{synced}^{\mathtt{C}}(\underline{s}^{\mathtt{C}})$

□

Finally, *compositionality* of data refinement can be generalized to retracting components when the extended proof obligations of Lem. 4 and Thm. 5 are employed.

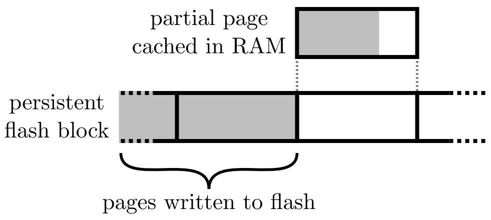


Figure 5.2: Buffering page writes.

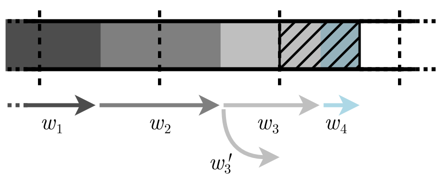


Figure 5.3: Alternative re-execution of a partially buffered write.

Theorem 6 (Compositionality of Data Refinement)**.** *Given two sequential (potentially retracting) components* `C` *and* `A` *with* `C` $\leq$ `A`*, i.e., where* `C` *is a valid refinement of* `A`*, and a crash-modularizable component* `M(A)` *that uses* `A` *as a subcomponent, then* `A` *can be substituted by* `C` *in* `M` *without affecting* `M`*'s correctness, i.e.,* `M(C)` $\leq$ `M(A)` *holds.*

Proof of Thm. 6. See the second case of the proof of Thm. 1 in [96]. □

5.2 Buffering Partial Writes

The Flashix file system features an order-preserving buffer cache that requires switching to the operations-based crash specification presented in the previous section. It is implemented in the `Write Buffer` component, which is located in the lower part of the hierarchy (see Fig. 4.1). The cache deals with the limitation that flash blocks can only be written sequentially and in page-sized chunks. Since the written data can be of arbitrary size, a write is mostly not *aligned* with a flash page. In order to perform such an non-aligned write, the partial page has to be filled with some *padding data* (typically zero bytes) to create a buffer of valid size.

However, this mechanism compromises the performance and space efficiency of the file system significantly as more pages have to be written in total and a substantial amount of storage space is occupied by padding data.

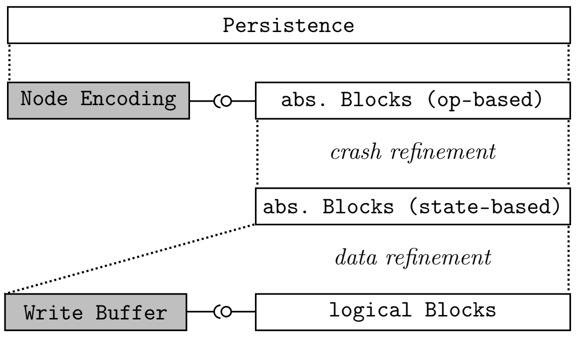


Figure 5.4: Crash refinement of the `Write Buffer` component.

The `Write Buffer` deals with this problem by queuing writes in a page-sized buffer in RAM, as shown in Fig. 5.2. As long as the data in sum does not reach the size of a flash page, no write to the flash hardware is emitted. When the page boundary is crossed, all buffered writes are persisted with a single page write. Then the buffer "*moves*" one page further and caches subsequent writes until the next page in the block can be written.

Naturally, the addition of this cache affects the crash behavior since data written by higher-level operations can still reside (partially) in the buffer, even if the operation has returned already. But because all writes must pass through the buffer (and

`Write Buffer` is not used concurrently), the component introduces *retracting crash transitions* as given by Def. 33.

The switch from the state-based view to the operations-based view is done using another kind of refinement, which we call *crash refinement*. We therefore have an additional refinement step in the hierarchy, as shown in Figure 5.4. While this kind of refinement is not the focus of this thesis, the general idea is briefly presented in the following as it has implications for the high-level caches presented in the remainder of this chapter. A more detailed explanation can be found in [97] and in Pfähler's thesis [96], where he also presents the complete models of the relevant components.

Going bottom-up, the `Write Buffer` component, which has an explicit distinction between the persistent state in `logical Blocks` and the volatile state of the buffer cache, is abstracted via data refinement to `abstract Blocks`, which does not have this distinction but uses still the state-based view on crashes. The component abstracts a *logical erase block (LEB)* as a dynamically-sized buffer *leb*. A write to the block then simply appends the written data to *leb*.

component `abstract Blocks (state-based)`

state leb : buffer, ...

crash predicate $leb = leb' \downarrow \texttt{EB\_PAGE\_SIZE}$

The crash predicate of the component specifies the effect of loosing the buffered page by trimming *leb* to the next lower page-aligned size. The constant `EB_PAGE_SIZE` defines the size of hardware pages (note that it generally does not coincide with the size `PAGE_SIZE` of `VFS` pages), and the algebraic function $buf \downarrow n$ resizes (shrinks) the buffer *buf* to the greatest possible length m with $m \leq \# \; buf \land m \; \texttt{mod} \; n = 0$. Fig. 5.3 shows an example with four write operations $w_1, \ldots, w_4$ using this state representation. The vertical dashed lines depict the page boundaries, and the arrows, together with the color shades, denote the content that was written by the respective write operation. A crash in this state removes the hatched part at the end of the shown section of the block since the `Write Buffer` caches this page.

A crash refinement step abstracts this explicit crash specification to an implicit operations-based specification. The resulting component is identical regarding state representation and operations, but the crash predicate is simplified to identity.

component `abstract Blocks (operations-based)`

state leb : buffer, ...

crash predicate $leb = leb'$

synchronized predicate $\texttt{aligned}(leb, \texttt{EB\_PAGE\_SIZE})$

A synchronized predicate is introduced to restrict the domain of the crash predicate to states where the content of an erase block is aligned with the pages, and a crash has no effect (leaves the content unchanged). For the example in Fig. 5.3, this synchronized state is reached by retracting the last two operations w_3 and w_4 and subsequently re-executing w_3 so that it only writes to the page boundary. Note that the re-execution w_3' is always possible as the flash hardware can always refuse to write a page.

The operations-based component `abstract Blocks` is then used as a subcomponent of `Node Encoding`. However, the specification of synchronized states propagates

```
**posix_fsync#**($fd$, $user$; ; $err$)
  **interface**
{
  **choose** $err_0$ **with** pre-fsync($fd, user, root, fs, ofh, err_0$) **in** $err := err_0$;
  $psynced := (err = \texttt{ESUCCESS})$;
}

**posix_create#**($p$, $md$, $user$; ; $err$)
  **interface**
{
  **choose** $err_0$ **with** pre-create($p, md, user, root, fs, ofh, err_0$) **in** $err := err_0$;
  ...
  $psynced :=$ false;
}
```

Figure 5.5: Specification of synchronized states in POSIX operations.

all way up the refinement hierarchy: on each layer, a synchronized predicate must be given. On higher levels, synchronized states cannot be identified precisely as the state abstracts from the hardware structure, e.g., *dirs* and *files* of AFS do not partition their contents in flash pages. For this reason, *ghost state variables*[1] are used in specification components to specify when a state must be synchronized, i.e., when its implementations must ensure that the buffer cache is empty and all operations are persisted entirely. For example, the POSIX component (see Sec. 4.2) realizes this by a boolean ghost variable *psynced*.

```
**component** POSIX
...
**ghost state**   $psynced$ : bool
**synchronized predicate**   $psynced$
```

The POSIX operations then set these synchronized flags explicitly, depending on whether their implementations (potentially) write something to the buffer cache or should synchronize it. POSIX, for example, defines an operation *fsync* for synchronizing individual files. A corresponding operation is added to the POSIX component (shown in Fig. 5.5 at the top) and is used trigger the synchronization of the Write Buffer. In POSIX, this operation does nothing but set *psynced* to true if the successful execution was chosen. Nevertheless, its implementation propagates a *fsync* call down to the Persistence component, which synchronizes the Write Buffer by writing a *padding node* with exactly the size to reach a page boundary. That way, successful execution of **posix_fsync#** guarantees that all buffered operations were persisted and thus cannot be reverted by crashes. Other POSIX operations, like **posix_create#** shown in Fig. 5.5 at the bottom, pessimistically set *psynced* to false as they could have produced a non-synchronized state by writing something to the buffer cache.

[1]Ghost state variables are used for verification purposes only and are not allowed to influence the outputs of operations or the regular state of a component. This is checked syntactically by KIV. When code is generated, all ghost state variables and statements accessing them are removed.

While the *fsync* operation is used here for the synchronization of a low-level buffer cache, which is completely independent of a specific file descriptor *fd* and the associated file, its actual purpose will be relevant for the integration of high-level caching discussed in the following section. We will see that the operation (more precisely, its implementation) is crucial for the correctness and crash-safety of the caching mechanisms of the `Cache` extension.

Using this extended version of the `POSIX` specification, a correctness & crash-safety criterion can be defined for file systems using a buffer cache similar to the `Write Buffer`. Our criterion *Quasi-Sequential Crash Consistency (QSCC)* expands on the work of Bornholt et al. [17], which defines the criterion *Sequential Crash-Consistency* using the idea of retracting crashes. QSCC extends the criterion by allowing re-execution of pending operations.

Definition 35 (Quasi-Sequential Crash Consistency (QSCC))**.** *A file system is quasi-sequential crash consistent iff it refines the* `POSIX` *component given in Sec. 4.2 extended by a synchronization operation and a synchronized predicate as presented in this section.*

In [97] and the thesis of Pfähler [96], it was shown that this criterion holds for the Flashix implementation that does not use the `Cache` layer introduced in the context of this thesis. Adding non-order-preserving high-level caches to a file system does affect the crash-safety, as we will see in Sec. 6.1. However, we will argue in Sec. 6.4 that both kinds of caches and thus the respective crash-safety criteria can be combined.

Chapter 6

Crash-Safe Caching in File Systems

Summary The second caching extension to the Flashix file system was the addition of the `Cache` layer, a collection of caches for high-level data structures used in `VFS`. These caches are *non-order-preserving*, i.e., writes may be reordered within the caches before they are persisted, which complicates reasoning about crash-safety significantly.
As a central contribution of this thesis, this chapter presents the `Cache` extension in detail. The layer is integrated modularly into the existing hierarchy by applying the *Decorator* pattern. The functional correctness of the layer primarily depends on a correct, non-trivial implementation of the synchronization routine. A crucial aspect of this implementation is the proper handling of *asymmetric truncations* to files (cf. Fig. 4.17): due to the efficient data representation of `VFS`, which is based on the Linux VFS implementation, a correct aggregation of cached truncations is not evident.
This chapter introduces a novel crash-safety criterion for non-order-preserving caches. *Write-Prefix Crash Consistency* not only guarantees consistency after recovering from crashes, but it also gives an intuitive explanation for the crash effects in terms of histories: crashes yield shorter writes (*prefix-writes*) for files that have not been fully synchronized. For synchronized files, Write-Prefix Crash Consistency ensures that all written data is persisted and thus not lost in a crash.
Proofs are given that Flashix with the `Cache` extension satisfies Write-Prefix Crash Consistency. As a result of the complex nature of the caching layer, these proofs are challenging as they require a forward simulation over multiple layers of the hierarchy. Furthermore, it is shown that high-level caches are compatible with order-preserving caches, e.g., that `Cache` can be used on top of the `Write Buffer`.

Contents

Publications The `Cache` addition, the correctness of non-order-preserving caches, and the correctness proofs for Flashix are published in [14, 15]. The compatibility with order-preserving caches is published in [15].

6.1 Caching of High-Level Data Structures

In Linux, the implementation of the Virtual File System Switch is enormously complicated. In particular, it is noticeably more complex than the `VFS` model of Flashix. One reason for this difference is that the Linux VFS also includes several caches for the data structures frequently used to communicate with the underlying file systems. For example, it employs caches for inodes, dentries, and file content pages (see Sec. 4.3).

Initially, Flashix was developed without such caches for high-level data structures. However, it is inevitable to use mechanisms similar to the Linux VFS caches to achieve comparable performance. Integrating caches into the existing `VFS` component would result in the need to reprove the whole layer from scratch while at the same time its complexity is increased significantly. Therefore, a new layer between the `VFS` and the `FFS` is introduced to add such caches to the Flashix hierarchy with minimal impact on the surrounding layers, visualized in Fig. 6.1.

This layer is implemented as a *Decorator* [48], i.e., it implements the same interface, namely the one of `AFS`, and delegates calls from `VFS` to `FFS`, which also has `AFS` as its interface. The `VFS` communicates with a Cache Controller (`Cache`), which in turn communicates with the `FFS` and manages caches for inodes (`ICache`), dentries (`DCache`), pages (`PCache`), and an auxiliary cache for truncations (`TCache`).

The `ICache`, `DCache`, and `PCache` components internally store maps from unique identifiers to the corresponding data structures. They all offer interfaces to `Cache` for adding resp. updating, reading, and deleting cache entries. `Cache` is responsible for processing requests from `VFS` by either delegating these requests to `FFS` or fulfilling them with the help of the required caches. It also has to keep the caches consistent with data stored on flash, i.e., update cached data when changes to corresponding data on flash have been made.

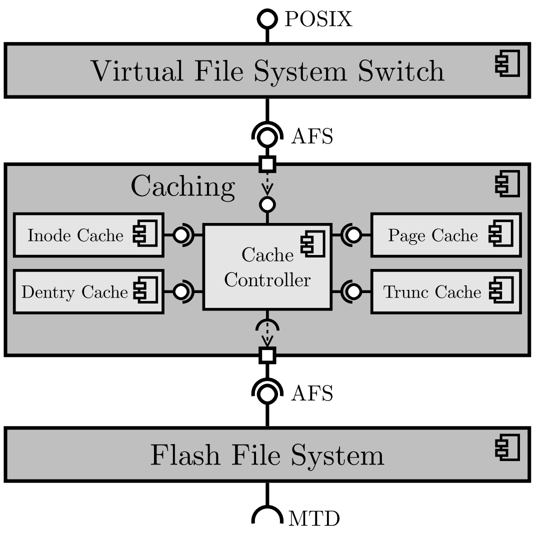


Figure 6.1: Flashix component hierarchy with caching layer.

Write-Through Caching of Structural Operations

Similar to the Linux VFS, information about the file system structure is cached in a write-through manner to speed up read accesses. This includes dentries in `DCache` as well as structural fields of inodes in `ICache` like `size` of directory inodes or `nlink`. By writing changes to the file system structure (through the cache directly) to flash memory, the integrity of the file system tree after a crash is ensured since structural operations are usually highly dependent on one another and affect multiple data objects. For example the creation of a file can only take place after the parent directory was created and linking of a file results in changes to the file itself as well

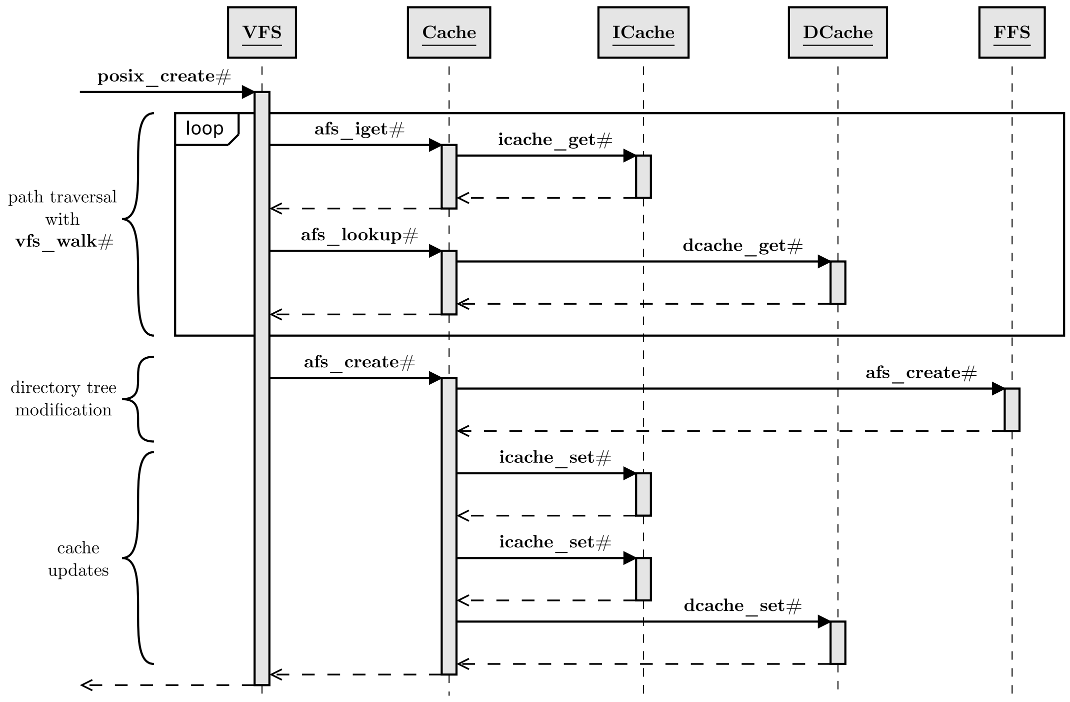


Figure 6.2: Interaction of `VFS`, `FFS`, and the caching layer during a **posix_create#** operation.

as to the directory in which the link is created.

Nevertheless, adding such write-through caches has a noticeable impact on the performance of structural operations as well. Consider for example the typical sequence arising from the **posix_create#** operation in Fig. 6.2, which is representative for all structural operations. As explained in Sec. 4.3, the initial path traversal comprises multiple reads of inodes and dentries during **vfs_walk#** (see Fig. 4.19). This is done by calling the `AFS` interface operations **afs_iget#** and **afs_lookup#**, which typically results in slow reads from flash via the `FFS` when no cache is used. However, with the caching layer, these requests can potentially be handled by the `ICache` and the `DCache`, as seen in the example sequence and in the respective implementations listed in Fig. 6.3. If a *cache miss* occurs, i.e., a requested data object is not cached and the operation **icache_get#** resp. **dcache_get#** returns *hit* = `false`, the object must still be read from flash, however, it is then added to the cache for future queries via **icache_set#** or **dcache_set#**. `Cache` passes the actual update calls of `VFS` simply to `FFS` (**afs_create#** in the example), and hence the file system is still modified persistently. But the caching layer takes advantage of the fact that `FFS` can return all modified or created data structures during the operation (for **afs_create#** in Fig. 4.13, these are *pinode*, *cinode*, and *dent*) and adds these data structures to the caches or updates them in the caches respectively (with the **icache_set#** and **dcache_set#** operations). This significantly reduces the number of reading accesses to the flash storage for everyday workloads such as extracting an archive, which requires traversing the full path to the parent directory each time a node is created.

```
cache_iget#($ino$; $inode$; $err$)
  interface
  precondition  $ino \in dirs \lor ino \in files$
{
  $err$ := ESUCCESS;
  let $hit$ = false, $dirty$ = false in
    icache_get#($ino$; $inode, dirty$; $hit$);
    if $\neg$ $hit$ then
      afs_iget#($ino$; $inode$; $err$);
      if $err$ = ESUCCESS then
        icache_set#($inode, dirty$);
}
```

```
cache_lookup#($pino$; $dent$; $err$)
  interface
  precondition
    valid-parent-ino($pino, dirs, files$)
{
  $err$ := ESUCCESS;
  let $hit$ = false in
    dcache_get#($pino$; $dent$; $hit$);
    if $\neg$ $hit$ then
      afs_lookup#($pino$; $dent$; $err$);
      if $err$ = ESUCCESS then
        dcache_set#($pino, dent$);
}
```

Figure 6.3: Cache operations for reading inodes and dentries.

```
pcache_set#($ino, pno, pbuf, dirty$)
  interface
  precondition  # $pbuf$ = PAGE_SIZE
{
  let $key$ = pkey($ino, pno$) in
    $pcache$[$key$] := pentry($pbuf, dirty$);
}

pcache_markclean#($ino, pno$)
  interface
  precondition  pkey($ino, pno$) $\in pcache$
{
  let $key$ = pkey($ino, pno$) in
    $pcache$[$key$].dirty := false;
}
```

```
pcache_get#($ino, pno$; $pbuf, dirty$; $hit$)
  interface
{
  let $key$ = pkey($ino, pno$) in
    $hit$ := $key \in pcache$;
    if $hit$ then
      $dirty$ := $pcache$[$k$].dirty;
      $pbuf$ := $pcache$[$k$].page;
}

pcache_delete#($ino, pno$)
  interface
{
  $pcache$ := $pcache$ -- pkey($ino, pno$);
}
```

Figure 6.4: Core operations of the PCache component.

Write-Back Caching of Content Operations

Compared to structural operations, updates to file data can be considered mostly in isolation. This means that in particular reads and writes to different files do not interfere with each other. Therefore we allow write-back caching of POSIX operations that modify the content of a file, namely **posix_write#** and **posix_truncate#** (cf. Fig. 4.6 and Fig. 4.9, respectively). Hence, the Cache component does not forward page writes to the FFS and instead only stores the pages in PCache. Updates to the size of a file are also performed in ICache only as garbage data could be exposed in the event of a power cut otherwise. To distinguish between up-to-date data and cached updates, entries of PCache or ICache include an additional *dirty* flag. For PCache, this results in a mapping from inode numbers and page numbers (wrapped in the tuple type pcache−key) to entries consisting of a page-sized buffer and a boolean flag (wrapped in the tuple type pcache−entry).

```
component PCache
state  $pcache$ : map(pcache−key, pcache−entry)
```

```
cache_writepage#(inode, pno, pbuf; ; err)
  interface
  precondition ... ∧ # pbuf = PAGE_SIZE
{
  err := false;
  let ino = inode.ino, dirty = true in
    pcache_set#(ino, pno, pbuf, dirty);
}
```

```
cache_writesize#(inode, sz; ; err)
  interface
  precondition ... ∧ inode.size < sz
{
  err := false;
  inode.size := sz;
  let dirty = true in
    icache_set#(inode, dirty);
}
```

Figure 6.5: `Cache` operations for writing pages and updating file sizes.

where

$$\begin{aligned} &\textbf{data} \quad \mathsf{pcache{-}key} = \mathtt{pkey}(\texttt{. .ino} : \mathsf{ino};\ \texttt{. .pageno} : \mathsf{nat}) \\ &\textbf{data} \quad \mathsf{pcache{-}entry} = \mathtt{pentry}(\texttt{. .page} : \mathsf{buffer};\ \texttt{. .dirty} : \mathsf{bool}) \end{aligned}$$

Fig. 6.4 lists the central operations of the `PCache` component using the state *pcache*. The components `ICache` and `DCache` are defined analogously. `ICache` stores a mapping *icache* : map(ino, icache−entry) from inode numbers to entries containing the inode and a dirty flag, and `DCache` uses a mapping *dcache* : map(dcache−key, dentry) from directory inode numbers and entry names to dentries (only real dentries are stored, no `negdentry`s).

$$\begin{aligned} &\textbf{data} \quad \mathsf{icache{-}entry} = \mathtt{ientry}(\texttt{. .inode} : \mathsf{inode};\ \texttt{. .dirty} : \mathsf{bool}) \\ &\textbf{data} \quad \mathsf{dcache{-}key} = \mathtt{dkey}(\texttt{. .ino} : \mathsf{ino};\ \texttt{. .name} : \mathsf{string}) \end{aligned}$$

Note that no dirty flag is necessary for the entries of *dcache* since they are always identical to their persisted counterparts. The auxiliary cache `TCache` records the minimal truncation size `tsize` of a file since the last synchronization as well as its current persisted size `fsize` using a mapping *tcache* : map(ino, tcache−entry).

$$\textbf{data} \quad \mathsf{tcache{-}entry} = \mathtt{tentry}(\texttt{. .tsize} : \mathsf{nat};\ \texttt{. .fsize} : \mathsf{nat})$$

Writing pages or file sizes results in putting the new data *dirty* in the particular caches. These operations of the controller component `Cache` are shown in Fig. 6.5.

On the other hand, reading pages returns the page in question stored in `PCache` or, if it has not been cached yet, it tries to read it from flash (Fig. 6.6 lines 2 and 10, respectively). But reading from flash yields the correct result only if there was no prior truncation that would have deleted the relevant page (lines 6-8). This is the case when an entry for this file exists in `TCache`, and when applying this truncation would delete the requested page (if *pno* is beyond the cached truncate size sz_T or the current persisted size of the file sz_F). If reading the page from flash is correct and the page actually stores any relevant data (*exists* is true), the resulting page is stored *clean* in `PCache` to handle repeated read requests (lines 11-13).

For truncations of files, there are several steps `Cache` needs to perform. They are implemented by the operation **cache_truncate#** shown in Fig. 6.7, and the operation **cache_writebegin#** (not shown). First, when an actual user truncation is executed, `ICache` needs to be updated by setting the size to the size the file is truncated to (lines 11 and 12). Second, cached pages beyond *sz* resp. *n*

```
**cache_readpage#**($inode, pno; pbuf, exists; err$)
  **interface**
  **precondition**   valid-file-inode($inode, dirs, files$)
                  $\wedge$ # $pbuf$ = PAGE_SIZE
{
  **let** $ino = inode$.ino, $hit$ = false, $dirty$ = false **in**
    **pcache_get#**($ino, pno; pbuf, dirty; hit$);
    **if** $hit$ **then** { $exists$ := true; $err$ := ESUCCESS }
    **else let** $sz_T = 0,\ sz_F = 0$ **in**
      **tcache_get#**($ino; sz_T, sz_F; hit$);
      **if** $hit \wedge$ min($sz_T, sz_F$) $\leq$ pos($pno$) **then**
        $pbuf := \perp$; $exists$ := false; $err$ := ESUCCESS;
      **else**
        **afs_readpage#**($inode, pno; pbuf, exists; err$);
        **if** $err$ = ESUCCESS $\wedge$ $exists$ **then**
          $dirty$ := false;
          **pcache_set#**($ino, pno, pbuf, dirty$);
}
```

Figure 6.6: Cache operation for reading pages.

have to be removed from PCache (line 5), and the truncate sizes in TCache have to be updated (line 10). For this purpose, the two subcomponents provide dedicated truncation operations **pcache_truncate#** and **tcache_update#**, respectively. **tcache_update#** aggregates multiple truncations by caching the minimal truncate size n for each file only. Additionally, the persisted size sz of a file is stored in TCache to determine whether it is allowed to read a page from flash in **cache_readpage#**. **pcache_truncate#** ensures that no truncated data is cached by removing all entries pkey(ino, pno) from PCache where $sz_T \leq$ pos(pno). Finally, if the truncate is growing, i.e., $sz \leq n$, the page at size sz may need to be filled with zeros (lines 6-8). The auxiliary operation **cache_truncpage#** shown in Fig. 6.8 is used to determine if this page is existent. This is the case if the page is either cached in PCache or can be read from flash. However, reading from flash is tried only if the page would not have been deleted already by a truncation, which is checked by comparing the page position with the cached truncate sizes in TCache. If necessary, the page returned by **cache_truncpage#** is then filled with zeros beyond offset(sz) using the truncate function, and the result is stored in PCache.

Like the corresponding AFS operation, **cache_writebegin#** performs a truncation with the current file size. Hence, its implementation is almost identical to the one shown in Fig. 6.7, except that the file size must not be updated, so the statements in lines 11 and 12 are not performed. Because n is equal to sz, the condition $sz \leq n$ in line 6 is obsolete and sz_T is always sz.

Synchronization of File Contents

The synchronization of files, i.e., transferring cached updates to the persistent storage, is coordinated by Cache, too. Clients can use the POSIX *fsync* operation to trigger synchronization of a specific file.

The implementation of *fsync* in Cache is shown in Fig. 6.9. The general idea of this implementation is to split the process into multiple phases: first, all pages that

```
cache_truncate#($n$; $inode$; $err$) {
  interface
  precondition valid-file-inode($inode$, $dirs$, $files$)
{
  let $ino = inode$.ino, $sz = inode$.size in, $sz_T$ = min($n$, $inode$.size) in
  let $pno$ = page($sz_T$), $pbuf = \bot$, $hit$ = false, $dirty$ = true in
    cache_truncpage#($inode$, $pno$; $pbuf$; $hit$, $err$);
    if $err$ = ESUCCESS then
      pcache_truncate#($ino$, $sz_T$);
      if $hit \wedge sz \leq n \wedge \neg$ aligned($sz$) then
        $pbuf$ := truncate($pbuf$, $sz$);
        pcache_set#($ino$, $pno$, $pbuf$, $dirty$);
      tcache_update#($ino$, $n$, $sz$);
      $inode$.size := $n$;
      icache_set#($inode$, $dirty$);
}
```

Figure 6.7: Cache operation for user truncations of file contents.

```
cache_truncpage#($inode$, $pno$; $pbuf$; $hit$, $err$)
  auxiliary
  precondition valid-file-inode($inode$, $dirs$, $files$) $\wedge$ # $pbuf$ = PAGE_SIZE
{
  $err$ := false;
  let $dirty$ = false in
    pcache_get#($ino$, $pno$; $pbuf$, $dirty$; $hit$);
    if $\neg$ $hit$ then let $sz_T = 0$, $sz_F = 0$ in
      tcache_get#($ino$; $sz_T$, $sz_F$; $hit$);
      if $\neg$ $hit$ $\vee$ pos($pno$) < min($sz_T$, $sz_F$) then
        afs_readpage#($inode$, $pno$; $pbuf$; $hit$, $err$);
}
```

Figure 6.8: Cache auxiliary operation for determining a page for truncation.

would have been truncated since the last synchronization are removed from flash; then, a VFS *write* is mimicked by persisting all *dirty* pages in PCache and updating the file size to the size stored in ICache if necessary. More precisely, **cache_fsync#** performs the following steps.

1. The file is truncated to the minimal truncate size sz_T since its last synchronization using **afs_truncate#**.

2. Potential garbage is removed by a call of **afs_writebegin#**.

3. All cached *dirty* pages are written in ascending order with **afs_writepage#**.

4. The file size is updated with **afs_writesize#**.

As we will see in Sec. 6.2, it is crucial for crash-safety that synchronization is executed in this order. Particularly, the order in which pages are persisted is essential for the formulation of an expressive crash-safety criterion.

The **cache_fsync#** operation uses the auxiliary operations listed in Fig. 6.10 to implement this process. The operation **cache_fbegin#** is responsible for synchronizing truncations and for preparing the subsequent writing of pages and updating

```
**cache_fsync#**($inode$; ; $err$)
  **interface**
  **precondition** valid-file-inode($inode, dirs, files$)
{
  **let** $sz_F = 0,\ dosync =$ false **in**
    **cache_fbegin#**($inode$; $sz_F$; $dosync, err$);
    **if** $err =$ ESUCCESS $\wedge$ $dosync$ **then cache_fpages#**($inode$; ; $err$);
    **if** $err =$ ESUCCESS $\wedge$ $dosync$ **then cache_finode#**($inode, sz_F$; ; $err$);
    **if** $err =$ ESUCCESS **then afs_fsync#**($inode$; ; $err$);
}
```

Figure 6.9: File synchronization operation of `Cache`.

of the file size in **cache_fpages#** and **cache_finode#**, respectively. When using this synchronization strategy, it is sufficient to aggregate multiple truncations by truncating to the minimal size the file was truncated to, and only if this minimal truncation size is lower than the current file size on flash. As truncation is the only possibility to delete pages (except for deleting the file as a whole), this **afs_truncate#** call deletes all obsolete pages. The following **afs_writebegin#** call ensures that the entire file content beyond sz_T resp. sz_F is zeroed so that writing pages and increasing the file size on flash is possible safely. Since `AFS` enforces an initial **afs_writebegin#** before writing pages or updating the file size, and `Cache` is a refinement of `AFS`, it is guaranteed that there are dirty pages in `PCache` or dirty inodes in `ICache` only if there is an entry in `TCache` for the file that is being synchronized. Hence there is nothing to do if **tcache_get#** returns `false` in *hit* (which is directly stored in *dosync* by the call in **cache_fbegin#**).

If there is dirty data to persist (*dosync* was set to true), **cache_fpages#** iterates over all possibly cached pages of the file delegating the synchronization of individual pages to **cache_fpage#**. Similar to the implementation of **vfs_write#** explained in Sec. 4.3, this iteration is executed bottom-up, starting at page 0 up to the maximal page m cached in `PCache` (returned by **pcache_maxpage#**). **cache_fpage#** checks for a given page number *pno* if the respective page of the file is cached and *dirty*. If this is the case, it persists the *dirty* page with **afs_writepage#** and marks it *clean* in `PCache` after writing it successfully.

After all pages have been synchronized successfully, **cache_finode#** updates the file size with **afs_writesize#** if the cached size is greater than the persisted size sz_F. Finally, **cache_fsync#** synchronizes the lower levels as well by calling **afs_fsync#** to ensure that all updates to the file have actually been written completely to flash as dictated by the POSIX standard.

6.2 Crash-Safety of Non-Order-Preserving Caches

Due to the modular approach, verifying the functional correctness of the integration of high-level caches into Flashix as shown in Fig. 4.1 requires to prove a single additional data refinement `Cache(AFS)` $\leq$ `AFS` only. For better distinction between both `AFS` variants in the hierarchy, we will use `AFS`$_\texttt{P}$ for the *persistent* `AFS` used as subcomponent by `VFS`, and `AFS`$_\texttt{C}$ for the *cached* `AFS` used by `VFS`. Accordingly, elements of the components (state variables, operations, ...) will be marked with

```
cache_fbegin#($inode$; $sz_F$; $dosync$, $err$)
  auxiliary
{
  $err$ := false;
  let $ino = inode$.ino, $sz_T = 0$ in
    tcache_get#($ino$; $sz_T, sz_F$; $dosync$);
    if $dosync$ then
      if $sz_T < sz_F$ then
        afs_truncate#($sz_T$; $inode$; $err$);
        $sz_F := sz_T$;
      if $err$ = ESUCCESS then
        afs_writebegin#($inode$; ; $err$);
      if $err$ = ESUCCESS then
        tcache_delete#($inode$.ino);
}

cache_finode#($inode, sz_F$; ; $err$)
  auxiliary
{
  let $sz = inode$.size in
    if $sz_F < sz$ then
      afs_writesize#($inode, sz$; ; $err$)
    else
      $err$ := false;
}

cache_fpages#($inode$; ; $err$)
  auxiliary
{
  $err$ := false;
  let $ino = inode$.ino, $pno = 0$, $m = 0$ in
    pcache_maxpage#($ino$; ; $m$);
    while $err$ = ESUCCESS $\wedge$ $pno \leq m$ do
      cache_fpage#($inode, pno$; ; $err$);
      $pno := pno + 1$;
}

cache_fpage#($inode, pno$; ; $err$)
  auxiliary
{
  let $hit$ = false, $dirty$ = false,
      $ino = inode$.ino, $pbuf = \bot$
  in
    pcache_get#($ino, pno$; $pbuf, dirty$; $hit$);
    if $hit \wedge dirty$ then
      afs_writepage#($inode, pno, pbuf$; ; $err$);
      if $err$ = ESUCCESS then
        pcache_markclean#($ino, pno$);
}
```

Figure 6.10: Auxiliary operations for file synchronization in Cache.

subscripts $_\mathtt{P}$ or $_\mathtt{C}$, respectively.

The following forward simulation R is used for the refinement proofs.

data refinement Cache(AFS$_\mathtt{P}$) $\leq$ AFS$_\mathtt{C}$

$$\textbf{abstraction relation} \quad dirs_\mathtt{C} = dirs_\mathtt{P} \wedge files_\mathtt{C} = ((files_\mathtt{P} \downarrow tcache) \oplus pcache) \oplus icache$$

The abstraction states that the abstract state of AFS$_\mathtt{C}$ can be constructed from the concrete state of Cache(AFS$_\mathtt{P}$) by applying all cached updates to the persistent AFS$_\mathtt{P}$ state. Basically, it encodes the process of synchronizing all files: all files are pruned at their cached truncate size ($\_ \downarrow tcache$), their pages are overwritten with their cached contents ($\_ \oplus pcache$), and the file sizes are updated with the corresponding cached ones ($\_ \oplus icache$). As no structural operations are cached, $dirs_\mathtt{C}$ and $dirs_\mathtt{P}$ are identical.

Theorem 7 (Functional Correctness of High-Level Caches)**.** *The sequential component* Cache *is a correct cache decorator implementation of the component* AFS*, i.e.,* Cache(AFS) $\leq$ AFS *holds.*

□

While AFS$_\mathtt{C}$ functionally matches the original specification of AFS, it is easy to see that AFS$_\mathtt{C}$ differs quite heavily from AFS$_\mathtt{P}$ in terms of its crash behavior. A crash in AFS$_\mathtt{P}$ just has the effect of removing orphaned files (see Sec. 4.3), i.e., those files that are not accessible from the file system tree anymore but still opened in VFS for reading/writing at the event of the crash. However, if there are pending writes that have not been synchronized yet, a crash in $\texttt{AFS}_\texttt{C}$ additionally may revert parts of these writes as all data only stored in the volatile state of Cache is lost.

Like for order-preserving caches (cf. Sec. 5.1 and Sec. 5.2), this crash effect cannot be expressed explicitly (using a *crash relation*) on higher levels of the hierarchy, where there is no distinction between persistent and cached state any more. This is already the case for $\texttt{AFS}_\texttt{C}$, and thus for POSIX, too. So instead of verifying crash-safety in a state-based manner, we want to explain the effects of a crash by constructing an alternative run where losing cached data does not have any effect on the state of $\texttt{AFS}_\texttt{C}$ (nevertheless, the residual state-based crash effect of removing orphans is still present). If such an alternative run can always be found, crash-safety holds since all regular (non-crashing) runs of $\texttt{AFS}_\texttt{C}$ yield consistent states, and thus a crash results in a consistent state as well.

However, the semantics of retracting components (see Def. 34) is not sufficient for the use of these non-order-preserving caches. For example, consider a sequence of two writes, where the first one writes a file f_1 and the second one writes to another file f_2. If now f_2 is synchronized, e.g., via *fsync*, and a crash occurs before a synchronization of f_1, the second write cannot be retracted (since it is persisted already), while the first one had no persistent effect and would thus have to be retracted. But obviously, this cannot be achieved by retracting a postfix of the original history, so Quasi-Sequential Crash Consistency (see Def. 35) cannot hold.

Write-Prefix Crash Consistency

Therefore, a new correctness criterion is required: in natural language, the criterion *Write-Prefix Crash Consistency* can be formulated as follows.

> *A file system is write-prefix crash consistent (WPCC) iff a crash keeps the directory tree intact and for each file f a crash has the effect of retracting all write and truncate operations to f since the last state it was synchronized and re-executing them, potentially resulting in writing prefixes of the original runs.*

This property results from the fact that files are synchronized individually by the *fsync* operation. Thus, all runs of operations that modify the content of a file, either cached or persistent, can be decoupled from runs of structural operations or operations accessing the content of other files.

The criterion is formulated in the granularity of POSIX operations, so we formalize *Write-Prefix Crash Consistency* in terms of POSIX histories, focusing on content operations. Since content operations always affect only one file (identified by a file identifier *fid* in POSIX), we restrict histories h to histories $h|_{\mathit{fid}}$ that contain only the content operations to *fid*. These include write, read, truncation, and synchronization operations so that $h|_{\mathit{fid}}$ consists of matching pairs of the form

$$\big(\mathit{inv}(\texttt{write}, \mathit{fid}, \mathit{buf}, n),\ \mathit{res}(\texttt{write}, n', \mathit{err})\big)$$
$$\big(\mathit{inv}(\texttt{read}, \mathit{fid}, n),\ \mathit{res}(\texttt{read}, n', \mathit{buf}, \mathit{err})\big)$$
$$\big(\mathit{inv}(\texttt{truncate}, \mathit{fid}, n),\ \mathit{res}(\texttt{truncate}, \mathit{err})\big)$$
$$\big(\mathit{inv}(\texttt{fsync}, \mathit{fid}),\ \mathit{res}(\texttt{fsync}, \mathit{err})\big)$$

Each invocation is marked with the targeted file *fid* and each response contains the error flag *err* recording whether the operation was successful, i.e., no error occurred and $err = \texttt{ESUCCESS}$, or failed. n' denotes the potentially altered value of the reference parameter n and hence records how many bytes were written or read, respectively.

Note that we omit the input argument *user* in the invocation events since it is not relevant for crash-safety reasoning (invalid user permissions are reflected by error outputs in the corresponding response events).

WPCC will use *write-prefix histories* to define *write-prefix crash transitions* giving an alternative explanation for crashed runs. A notion for *prefix operations* is needed to define such histories.

Definition 36 (Prefix Operations). *For a completed content operation* (e_{inv}, e_{res})*, a corresponding prefix operation is a tuple* (e_{inv}, e'_{res}) *where* e'_{res} *results from changing the outputs of* e_{res} *in the sense that*

- *operations that were originally successful, i.e., returned* $err = \texttt{ESUCCESS}$*, may fail in the prefix operation, i.e., have an output* $err \neq \texttt{ESUCCESS}$ *(this includes that the remaining outputs may differ as well, e.g., the buf of a read response could still be filled with nothing but zeros), and*
- `write` *operations, that have written* m *bytes originally, may write less bytes* $m' \leq m$ *in the* prefix write*.*

Note that we do not define prefixes of read operation other than failed executions (that did not read anything at all). While it would be feasible to allow reads that have read some but not all bytes of the original operation, this is not necessary as reads do not affect the observable behavior of following operations and hence can simply considered as failed or omitted altogether when considering crash-safety.

Write-prefix histories should contain prefixes for non-synchronized operations only. For this, a history h can be split in an non-synchronized prefix $h_{>sync}$ and a synchronized postfix $h_{\leq sync}$.

Definition 37 ((Non-)Synchronized Histories). *A history* h *can be split in two histories* $h_{\leq sync}$ *and* $h_{>sync}$ *with* $h = h_{\leq sync} \cdot h_{>sync}$ *where* $h_{>sync}$ *is the maximal postfix of* h *not containing an event* $res(\texttt{fsync}, \texttt{ESUCCESS})$*.*

Thus, $(h|_{fid})_{\leq sync}$ contains all synchronized `write`, `read`, and `truncate` operations as well as all complete, successful `fsync` operations of *fid*. $(h|_{fid})_{\leq sync}$ either ends with an $res(\texttt{fsync}, \texttt{ESUCCESS})$, or it is empty (when $h|_{fid}$ does not contain any successful response of `fsync`). Conversely, $(h|_{fid})_{>sync}$ contains all pending `fsync` operations of *fid* and all `write`, `read`, and `truncate` operations of *fid* that are not (fully) synchronized. Both histories, however, may contain failed `fsync` operations.

Definition 38 (Write-Prefix Crash Transitions). *A transition* $s_n \xrightarrow{\lightning} s_{n+1}$ *is a write-prefix crash transition for an era* I, h *of the component* `POSIX` *and a system* $System^{\texttt{POSIX}}$ *with* $I = (s_0, \ldots, s_n)$ *and* $I, h \models System^{\texttt{POSIX}}_{\lightning}$ *iff there exists a legal write-prefix history* h' *with corresponding interval* I' *where*

1. $I'.first = s_0$ *and* I', h' *forms an uninterrupted system run, i.e.,* $I', h' \models System^{\texttt{C}}$*,*

2. *h' contains invocations and responses of the same operations in the same order as h with the exception that all pending operations of h are completed in h',*

3. *completed structural operations in h are identical in h',*

4. *for each file fid, $h'|_{fid} = (h|_{fid})_{\leq sync} \cdot h''|_{fid}$ and $h''|_{fid}$ is derived from $(h|_{fid})_{>sync}$ by replacing* `write`*,* `read`*, and* `truncate` *operations with matching prefix operations, and by completing pending operations, and*

5. *the crashed state s_{n+1} can be reached from $I'.last$ by applying the crash and recovery relations, i.e., $(I'.last, s_{n+1}) \in \mathtt{CRASH}^{\mathtt{POSIX}} \mathbin{;} \mathtt{REC}^{\mathtt{POSIX}}$.*

Write-Prefix Crash Consistency is then given by the sequential semantics of the `POSIX` component given by Def. 27 resp. Def. 34, but instantiated with write-prefix crash transitions according to Def. 38.

Definition 39 (Write-Prefix Crash Consistency (WPCC))**.** *A file system is write-prefix crash consistent iff it refines the* `POSIX` *component given in Sec. 4.2 extended by a synchronization operation and a synchronized predicate as presented in Sec. 5.2, and each crash transition $I_i.last \xrightarrow{\lightning_{i+1}} I_{i+1}.first$ with $i < n$ is a write-prefix crash transition for the era I_i, h_i and $System^{\mathtt{POSIX}} \equiv \{\ \mathtt{OP}^{\mathtt{POSIX}}\ \}^*$ according to Def. 38.*

The main point of this criterion is that interrupted runs (represented by legal histories h) can be explained by alternative, uninterrupted legal histories h' (condition 1 of Def. 38). Conditions 2-4 of Def. 38 ensure that h' is again well-formed and as close as possible to the original era h: the same operations must be executed in the same order and with the same inputs, structural operations must yield the same results (and hence, yield the same file system tree), only non-synchronized content operations may have slightly different results. The final condition 5 applies the residual, state-based crash effect like for retracting crash transitions (cf. Def. 33).

Note that our definition completes all pending operations while general linearizability completes some and deletes the remaining pending operations. This simplification is possible since each `POSIX` operation has a completion that fails non-deterministically and does not change the state, so the completed operation is equivalent to not executing it.

Furthermore, fixing the returned errors and the number of written/read bytes in the responses resolves all non-determinism (`POSIX` is non-deterministic just in succeeding or failing of operations and in writing or reading prefixes of the requested number of bytes, see Sec. 4.2)[1]. This property makes it possible to argue solely with histories about crashes since a history determines the complete observable run of the system for a given initial state. However, while both system runs (the original run and a corresponding write-prefix run) yield the same `POSIX` state, we will see in the following section that the internal representation of the file system may be slightly different.

[1]The creation of a file also chooses fresh file identifier non-deterministically, however, these choices are unaffected by content operations and are thus negligible for the considerations here.

6.3 Proving Crash-Safety of High-Level Caches

Given the WPCC criterion of Def. 39, this section presents the proof that Flashix is crash-safe, i.e., that the following theorem holds.

Theorem 8 (Flashix satisfies WPCC)**.** *The Flashix file system with the* `VFS` *implementation given in Sec. 4.3, extended by the* `Cache` *component of Sec. 6.1, satisfies WPCC.*

The basic proof idea for Thm. 8 is to consider a system run with history $h = h_0 \cdot \lightning$ with just a single crash $\lightning$ (and empty final era), construct a write-prefix era h_0' satisfying Def. 39, and show that $h' = h_0'$ is legal. We also prove that runs of $h_0 \cdot \lightning$ and h_0' yield identical `POSIX` states, which guarantees that legal write-prefix histories for histories with an arbitrary number of crashes can be constructed inductively.

Whereas WPCC could be formulated purely in terms of `POSIX` histories, we now have to consider implementation runs for proving, in particular the ones of $\mathtt{VFS(Cache(AFS_P))}$. We exploit that the lower levels of Flashix can be assumed to be atomic w.r.t. to crashes. In particular, this still holds for $\mathtt{AFS_P}$, i.e., a crash during an $\mathtt{AFS_P}$ operation has the same effect as a crash either directly before or directly after the operation. The implementation by a transactional journal ensures that, and thus crashes in $\mathtt{Cache(AFS_P)}$ have to be contemplated only between $\mathtt{AFS_P}$ operations. The proofs, therefore, revolve around constructing matching alternative runs in $\mathtt{AFS_C}$ for all possible crashes in $\mathtt{Cache(AFS_P)}$. In a next step, these runs have to be lifted to $\mathtt{VFS(AFS_C)}$ runs via the refinement $\mathtt{Cache(AFS_P)} \leq \mathtt{AFS_C}$. Finally, the refinement $\mathtt{VFS(AFS_C)} \leq \mathtt{POSIX}$ then ensures that they are also runs of `POSIX`.

As it turns out, for an arbitrary file *fid*, the only critical case is when a crash occurs during the execution of **cache_fsync#** for this file. In all other cases, updates to the content of *fid* have been stored in cache only, thus the persistent content of *fid* in $\mathtt{AFS_P}$ is unchanged since the last successful execution of **cache_fsync#** for *fid*. So if the crash is outside of **cache_fsync#**, we can choose a $\mathtt{VFS(AFS_C)}$ run in which all unsynchronized writes and truncates to *fid* have failed and hence have not written or deleted any data. Constructing such runs is always possible as $\mathtt{AFS_C}$ is *crash-neutral*, i.e., all operations of $\mathtt{AFS_C}$ are specified to have a run that fails without any changes to the state (see Sec. 3.1 and Sec. 4.3). In terms of histories, all operations in $(h_0|_{\mathit{fid}})_{>\mathit{sync}}$ are replaced by failed prefix operations.

However, showing that *WPCC* holds for crashes during **cache_fsync#** is hard. The remainder of the section shows why caching of asynchronous truncations is one major difficulty when proving Thm. 8, the approach for constructing write-prefix histories when crashes occur during synchronization, and how it was proved in KIV that such write-prefix histories always yield a valid write-prefix run.

Truncations and Caching

Initially, our goal was to prove this property locally on the level of $\mathtt{AFS_C}$ resp. of `Cache` and $\mathtt{AFS_P}$ only. For example, one approach was to construct matching prefix runs of $\mathtt{AFS_C}$ by commuting and merging operation calls. While we will not go into the many pitfalls we ran into, the main problem with these approaches was the synchronization of aggregated truncates, as states resulting from an interrupted

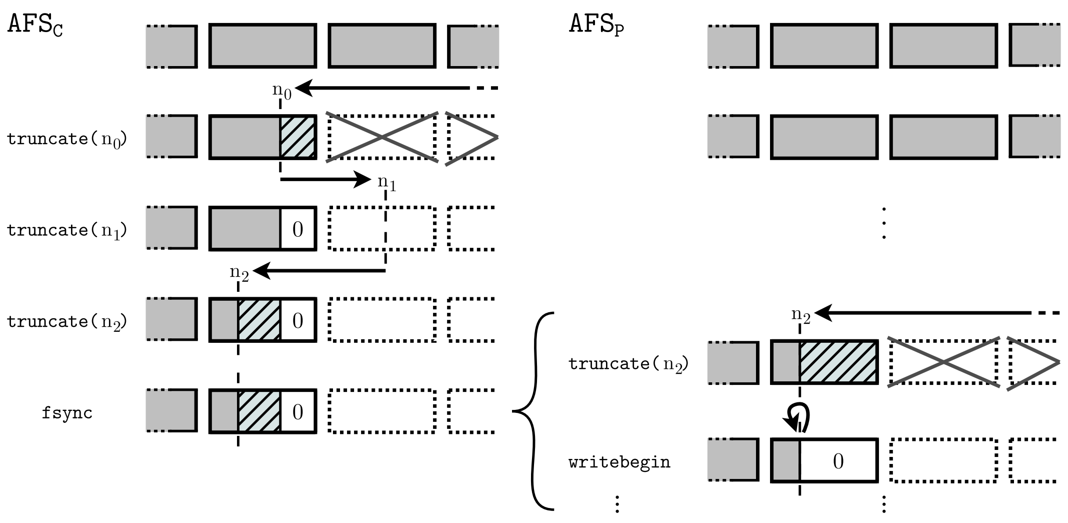


Figure 6.11: Effect of a sequence of **afs_truncate#** operations and a following **afs_fsync#** on the states of one file in AFS$_C$ (left) and AFS$_P$ (right), including intermediate states of AFS$_P$ during **vfs_fsync#**. The state of Cache is omitted.

synchronization in Cache could not be reconstructed by any combination of VFS prefixes from the corresponding AFS$_C$run.

For example, given the sequence of three **afs_truncate#** calls followed by an **afs_fsync#** call as visualized in Fig. 6.11, starting with a synchronized file, i.e., the contents (and sizes) of the affected file are equal in AFS$_C$ and AFS$_P$. Considering this run in AFS$_C$ on the left, the first truncation shrinks the file to a new size n_0 deleting all pages above page(n_0). Since aligned(n_0) is false, rest(n_0) bytes of junk data remain in page(n_0) for the moment. This junk data is removed not before the second truncation as it increases the file size then to n_1 and the remainder of page(n_0) is filled with zeros. Then finally, the third truncation shrinks the file again to n_2 with $n_2 < n_0$ but page(n_0) = page(n_2), which yields a mixed page containing valid data, junk, and zeros.

These truncations do not have any effect on the persistent state of AFS$_P$ as Cache handles all requests. Conversely, a call to **afs_fsync#** in AFS$_C$ leaves its state unchanged but its implementation Cache triggers a number of calls to AFS$_P$. First, the file is truncated to n_2, the minimal truncation size since its last synchronized state. Second, junk data above n_2 is removed with **afs_writebegin#** to prepare a potential synchronization of pages beyond n_2.

Comparing the state after **afs_writebegin#** in AFS$_P$ with the state after all truncations in AFS$_C$, one can see that the sizes and the valid part of the content match but there is some junk data left in AFS$_C$ that is not in AFS$_P$. In fact, if a crash occurs in a state after this **afs_writebegin#** call and before the synchronization of page(n_2) with **afs_writepage#**, we cannot construct a VFS prefix run of AFS$_C$ that yields exactly the state of AFS$_P$. However, the abstraction from VFS(AFS$_C$) to POSIX (see Secvfs) ignores bytes written beyond the file size anyway, and the implementation Cache(AFS$_P$) may at most remove more junk data than AFS$_C$, so the implementation actually matches our crash-safety criterion under the POSIX abstraction as intended.

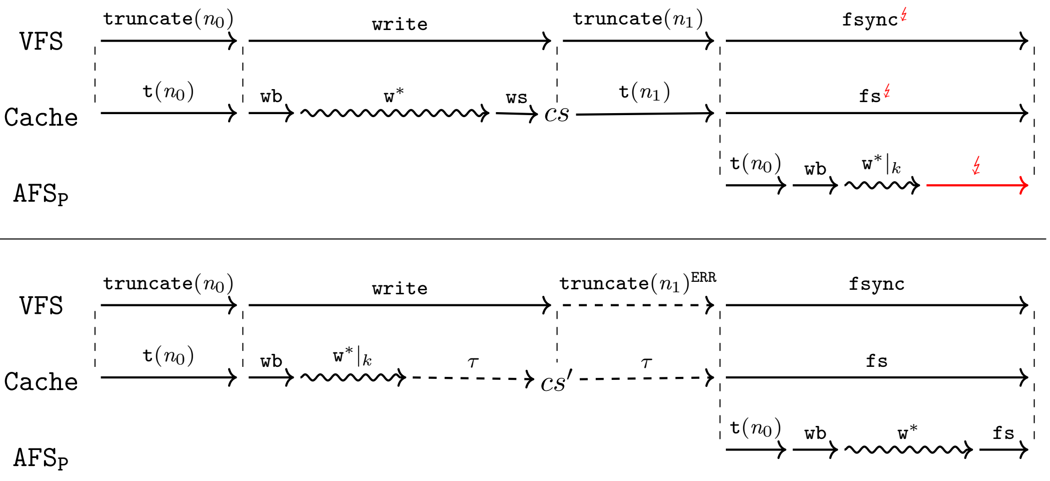


Figure 6.12: Construction of a write-prefix run (lower half) matching a run with a crash ↯ in `fsync` that occurs just before writing page k (upper half).

But in order to prove this, we need to explicitly consider runs of $\texttt{AFS}_\texttt{C}$ in the context of `VFS`.

Write-Prefix Histories for Crashed Synchronizations

To find a legal write-prefix history that satisfies Def. 39 for an arbitrary history that crashed during **cache_fsync#**, one has to consider at which point the synchronization was interrupted. We will now omit arguments of operations and abbreviate the `AFS`/`Cache` operations **writebegin#**, **writepage#**, **writesize#**, **truncate#**, and **fsync#** with `wb`, `w`, `ws`, `fs`, and `t`, respectively. Given the implementation of **cache_fsync#** in Fig. 6.9 and Fig. 6.10, a typical run yields a $\texttt{AFS}_\texttt{P}$ call sequence of the form `t wb w`* `ws fs`. Because of the crash-atomicity of $\texttt{AFS}_\texttt{P}$, effectively two cases need to be addressed, namely a crash occurs

1. between `t` and `wb` or
2. between persisting pages $k-1$ and k with `w`.

Two additional cases are crashes before `t` or after `ws` (the `fs` call to $\texttt{AFS}_\texttt{P}$ can be ignored for now, it only is relevant when additional low-level caches are used as discussed in Sec. 6.4). These can be viewed as crashing before resp. after the complete **cache_fsync#** operation since no persistent changes happen in these ranges. We also do not explicitly consider crashes immediately after `wb` or before `ws` as separate cases, but instead we handle these as variants of case 2.

For case 1, finding a write-prefix history is quite obvious. As **cache_fsync#** only executed a single persisting truncation to sz_T, only truncate operations to sz_T in $(h_0|_{fid})_{>sync}$ have been synchronized. Thus, only these operations remain unchanged in $(h_0'|_{fid})_{>sync}$, truncate operations to sizes n greater than sz_T are replaced by failed prefix operations. Similarly, all write operations are replaced by failed prefix operations as no pages have been persisted in **cache_fsync#**.

Verifying case 2 requires more effort. As an example consider the crashed run shown in the upper half of Fig. 6.12. The run contains **vfs_truncate#** and

vfs_write# calls, followed by an interrupted synchronization with **vfs_fsync#** (denoted by the $\_^{\lightning}$ superscript in the figure). One can see that **vfs_truncate#** triggers just a single `t` call to $\mathtt{AFS}_\mathtt{C}$ resp. `Cache`, whereas **vfs_write#** yields a call sequence of the form `wb` $\mathtt{w}^*$ `ws` (cf. Figures 4.22, 4.23, and 4.24). The **vfs_truncate#** and **vfs_write#** operations are performed in `Cache` only, so calls to $\mathtt{AFS}_\mathtt{P}$ are performed not until synchronization. The synchronization crashes after an ascending sequence $\mathtt{w}^*|_k$ of page writes, which contains only writes to pages $< k$.

A possible write-prefix run (and therefore a write-prefix history) is shown in the lower half of Fig. 6.12. As for case 1, the write-prefix run contains successful executions of **vfs_truncate#** calls to the minimal truncate size. In the example, this is the size n_0, so the first truncation is performed as before. For the second truncation to n_1 on the other hand, we choose a failing run of **vfs_truncate#** (failing operations are marked with $\_^{\mathtt{ERR}}$), which results in a stutter step τ in `Cache`, i.e., no operation is executed in `Cache`. So the first truncate operation is carried over unchanged to $(h'_0|_{fid})_{>sync}$ while the second truncate operation is replaced by a failed prefix operation.

The main aspect of *WPCC* is that non-synchronized **vfs_write#** executions write just as far as the interrupted **vfs_fsync#** was able to persist pages in the write-prefix run. Hence, the alternative **vfs_write#** execution successfully performs `wb` and a prefix of the original sequence $\mathtt{w}^*$, namely the prefix of writes $\mathtt{w}^*|_k$ to pages $< k$. All other writes to pages $\geq k$ are again replaced by stutter steps τ in `Cache`. When constructing the write-prefix history, a write operation with $e_{inv} = inv(\mathtt{write}, fid, buf, n)$ and $e_{res} = res(\mathtt{write}, n', \mathtt{ESUCCESS})$ in $(h_0|_{fid})_{>sync}$ is replaced in $(h'_0|_{fid})_{>sync}$ by the prefix operation (e_{inv}, e'_{res}) where

$$e'_{res} \quad = \quad \begin{cases} e_{res} & \text{if} \quad pos + n < \mathtt{min}(\mathtt{pos}(k), sz) \\ res(\mathtt{write}, 0, \lfloor err \rfloor) & \text{if} \quad \mathtt{min}(\mathtt{pos}(k), sz) \leq pos \\ res(\mathtt{write}, \mathtt{min}(\mathtt{pos}(k), sz) - pos, \mathtt{ESUCCESS}) & \text{otherwise} \end{cases}$$

Depending on the range the original **vfs_write#** has written to, the restricted sequence $\mathtt{w}^*|_k$ may be empty or the full sequence $\mathtt{w}^*$. However, because the alternative run does not execute updates of the file size via `ws`, not all bytes written by $\mathtt{w}^*|_k$ become visible on the level of `POSIX`, but only bytes written below the persisted file size sz and below $\mathtt{pos}(k)$. Hence, the prefix operation (e_{inv}, e'_{res}) writes the same number of bytes as the original operation (first case), fails and writes no bytes at all (second case, an arbitrary low-level error $\lfloor err \rfloor$ is returned), or writes bytes up to $\mathtt{min}(\mathtt{pos}(k), sz)$ (third case).

With a complete write-prefix history constructed this way, a full, successful **vfs_fsync#** run has the same effect as the crashed execution of the original run (except for differences in junk data resulting from the problematic nature of synchronizing truncations discussed earlier this section, see Fig. 6.11). Thus, the pending operation $e_{inv} = inv(\mathtt{fsync}, fid)$ can be completed with the response $e_{res} = res(\mathtt{fsync}, \mathtt{ESUCCESS})$.

From Write-Prefix Histories to Write-Prefix Runs

We now want to prove that write-prefix histories, as constructed in the previous section, are legal and that the corresponding write-prefix runs yield the same `POSIX`

$$\begin{tikzcd}
as'_0 \arrow[rr, "\mathtt{w}_\mathtt{A}(i)"] \arrow[dd, dash, "R"'] & & \exists\ as'_1 \arrow[dd, dash, "R"] & \\
& as_0 \arrow[rr, "\mathtt{w}_\mathtt{A}(i)"] \arrow[dd, dash, "R"] & & as_1 \arrow[dd, dash, "R"] \\
cs'_0 \arrow[rr, "\mathtt{w}_\mathtt{C}(i)"] \arrow[dr, dash, "\cong^k"'] & & \exists\ cs'_1 \arrow[dr, dash, "\cong^k"'] & \\
& cs_0 \arrow[rr, "\mathtt{w}_\mathtt{C}(i)"] & & \exists\ cs_1
\end{tikzcd}$$

(a) Successful $\mathtt{AFS}_\mathtt{C}$ run for $i < k$.

$$\begin{tikzcd}
as'_0 \arrow[rr, dashed, "\mathtt{w}_\mathtt{A}(i)^{\mathtt{ERR}} \ / \ \tau"] \arrow[dd, dash, "R"'] & & as'_0 \arrow[dd, dash, "R"] & \\
& as_0 \arrow[rr, "\mathtt{w}_\mathtt{A}(i)"] \arrow[dd, dash, "R"] & & as_1 \arrow[dd, dash, "R"] \\
cs'_0 \arrow[rr, dashed, "\tau"] \arrow[dr, dash, "\cong^k"'] & & cs'_0 \arrow[dr, dash, "\cong^k"'] & \\
& cs_0 \arrow[rr, "\mathtt{w}_\mathtt{C}(i)"] & & \exists\ cs_1
\end{tikzcd}$$

(b) Failing/stuttering $\mathtt{AFS}_\mathtt{C}$ run for $i \geq k$.

Figure 6.13: Commuting diagrams of a $\mathtt{w}$ run writing page i.

states as their original crashed runs. This is done with a forward simulation $\cong^k \subseteq CS \times CS$ over Cache(AFS$_\mathtt{P}$) states CS using commuting diagrams. The relation $\cong^k$ links all vertically aligned states in Fig. 6.12.

$$\begin{aligned}
\cong^k \quad \equiv \quad & ((\mathit{files}_\mathtt{P} \downarrow \mathit{tcache}) \oplus \mathit{pcache}|_k)\,.\,\mathtt{seq}(\mathit{ino}) \\
= \ & (((\mathit{files}'_\mathtt{P} \downarrow \mathit{tcache}') \oplus \mathit{pcache}') \oplus \mathit{icache}')\,.\,\mathtt{seq}(\mathit{ino})
\end{aligned}$$

where $\mathit{pcache}|_k$ restricts *pcache* to entries for pages $i < k$ and $\mathit{files}\,.\,\mathtt{seq}(\mathit{ino})$ extracts the content of the file *ino* as a sequence of bytes up to the current size of *ino* in *files*. Intuitively, two Cache states *cs* and *cs'* are $cs \cong^k cs'$ if a synchronization interrupted at page k of *cs* yields the same content (up to the file size) as a complete synchronization of *cs'*. Note that $cs \cong^k cs'$ enforces implicitly that the file size of *ino* is identical in *cs* and *cs'* and hence the cached truncate sizes in *tcache* and *tcache'*, as well as the cached size in *icache'*, must be equal.

For **writepage#** calls the commuting diagrams as shown in Fig. 6.13 in the bottom plane are required. **writepage#** operations of AFS$_\mathtt{C}$ and Cache(AFS$_\mathtt{P}$) are denoted $\mathtt{w}_\mathtt{A}$ and $\mathtt{w}_\mathtt{C}$, respectively. When writing a page $< k$, re-executing this operation maintains $\cong^k$ (Fig. 6.13a). In contrast, writing pages $\geq k$ maintains $\cong^k$ if the alternative run stutters (Fig. 6.13b). Since VFS is defined on AFS$_\mathtt{C}$, these commuting properties must be lifted from Cache(AFS$_\mathtt{P}$) to AFS$_\mathtt{C}$ in order to construct commuting diagrams for VFS(AFS$_\mathtt{C}$) runs. This is why the commuting diagrams are extended by R-corresponding AFS$_\mathtt{C}$ runs, yielding the front and back sides of Fig. 6.13 (R is given by the *abstraction relation* of the refinement Cache(AFS$_\mathtt{P}$) $\leq$ AFS$_\mathtt{C}$ shown in Sec. 6.2). So in addition we show that, given a run $as_0 \xrightarrow{\mathtt{w}_\mathtt{A}(i)} as_1$ as it is part of **vfs_write#**, there is an R-corresponding run $cs_0 \xrightarrow{\mathtt{w}_\mathtt{C}(i)} cs_1$ of Cache(AFS$_\mathtt{P}$). Conversely, we have to show that the resulting alternative run of Cache(AFS$_\mathtt{P}$) can be lifted to an R-corresponding run of AFS$_\mathtt{C}$ as well. Depending on the operation, up to two versions of this lifting are necessary if the run is stuttering: an AFS$_\mathtt{C}$ run that stutters (τ transition from as'_0 in Fig. 6.13b) and a failing run of the AFS$_\mathtt{C}$ operation ($\mathtt{w}_\mathtt{A}(i)^{\mathtt{ERR}}$ transition from as'_0 in Fig. 6.13b). For **writepage#**, the former is used to skip writes of pages $> k$ while the latter is required to stop the loop of **vfs_write#** when trying to write page k.

In KIV, these commuting diagrams have been proven using the sequent based weakest-precondition calculus presented in Sec. 2.4. For example, we get the following proof obligation for Fig. 6.13a.

$$\begin{aligned}
&inv_{\mathtt{A}}(as_0),\ inv_{\mathtt{C}}(cs_0),\ inv_{\mathtt{A}}(as_0'),\ inv_{\mathtt{C}}(cs_0'),\\
&R(as_0, cs_0),\ R(as_0', cs_0'),\ cs_0 \cong^k cs_0',\\
&\langle \textbf{afs\_writepage\#}(inode, pno, pbuf; as_0; err)\rangle(err = \mathtt{ESUCCESS} \wedge as_1 = as_0),\\
&pno \leq k\\
&\vdash \exists\ cs_1, cs_1', as_1'.\\
&\qquad\qquad \langle \textbf{cache\_writepage\#}(inode, pno, pbuf; cs_0; err)\rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad (err = \mathtt{ESUCCESS} \wedge cs_1 = cs_0)\\
&\qquad \wedge \langle \textbf{cache\_writepage\#}(inode, pno, pbuf; cs_0'; err)\rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad (err = \mathtt{ESUCCESS} \wedge cs_1' = cs_0')\\
&\qquad \wedge \langle \textbf{afs\_writepage\#}(inode, pno, pbuf; as_0'; err)\rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad (err = \mathtt{ESUCCESS} \wedge as_1' = as_0')\\
&\qquad \wedge inv_{\mathtt{A}}(as_1) \wedge inv_{\mathtt{C}}(cs_1) \wedge inv_{\mathtt{A}}(as_1') \wedge inv_{\mathtt{C}}(cs_1')\\
&\qquad \wedge R(as_1, cs_1) \wedge R(as_1', cs_1') \wedge cs_1 \cong^k cs_1'
\end{aligned}$$

Recall that the calculus requires that all non-local variables accessed in the body of a procedure are appropriate parameters while the declarations in our components (as shown in Sec. 4.3 and Sec. 6.1) do not have state variables as explicit parameters, Hence, the state variables of a component (in the example, the vectors as_0, cs_0, as_0', or cs_0', respectively) are added automatically as reference parameters to the signature of its operations (as they can always read or update their component's state). Having the states as reference parameters also makes modifications to the states visible in postconditions of program formulas. For example, the equation $as_1 = as_0$ in the antecedent program formula fixes the $\mathtt{AFS_C}$ state reached by the **afs_writepage#** run to as_1. This state can then be referenced in other formulas of the sequent, e.g., in the conjunct $R(as_1, cs_1)$ of the succedent formula.

In order to construct a valid alternative `VFS` run, analogous commuting diagrams for `wb`, `ws`, and `t` have been proven, not all commuting diagrams were necessary for each operation though. The proofs of commuting diagrams for **vfs_write#** and **vfs_truncate#** then are based upon the step by step application of these commutative properties.

Fig. 6.14 shows the construction of a **vfs_write#** write-prefix execution as an example. At the bottom there is the $\mathtt{Cache(AFS_P)}$ run cs_0 to cs_n resulting from the original **vfs_write#** execution $\mathtt{write_{VFS}}$. We proved for an arbitrary k (i.e., for every possible crash occurrence within a **cache_fsync#** matching case 2 given earlier this section) that there is an execution $\mathtt{write'_{VFS}}$ yielding $\mathtt{Cache(AFS_P)}$ states cs_0' ... cs_{i+1}' which preserve $\cong^k$. Regardless of k, the initial `wb` call always needs to be executed successfully in $\mathtt{write'_{VFS}}$ in order to get a valid execution. So we use a commuting diagram analogous to Fig. 6.13a to get the states cs_1' and as_1'. Then we repeatedly append the commuting diagram for `w` of Fig. 6.13a for all writes to pages less than k. This yields the state cs_{i+1}' (and a R-corresponding as_{i+1}') for which $cs_{i+1} \cong^k cs_{i+1}'$ obviously still holds, since so far the write-prefix execution

$$\mathtt{write}'_{\mathtt{VFS}}$$

$$\begin{tikzcd}
as'_0 \arrow[rr, "\mathtt{wb}_{\mathtt{A}}"] \arrow[dd, dash, "R"] & & as'_1 \arrow[rr, squiggly] \arrow[dd, dash, "R"] & & as'_i \arrow[rr, "\mathtt{w}_{\mathtt{A}}(k-1)"] \arrow[dd, dash, "R"] & & as'_{i+1} \arrow[rr, dashed, "\mathtt{w}_{\mathtt{A}}(k)^{\mathtt{ERR}}"] \arrow[dd, dash, "R"] & & as'_{i+1} \arrow[rr, squiggly] \arrow[dd, dash, "R"] & & as'_{i+1} \arrow[rr, dashed, "\tau"] \arrow[dd, dash, "R"] & & as'_{i+1} \arrow[dd, dash, "R"] & \\
& & & & & & & & & & & & & \\
cs'_0 \arrow[rr, "\mathtt{wb}_{\mathtt{C}}"] \arrow[dr, dash, "\cong^k"] & & cs'_1 \arrow[rr, squiggly] \arrow[dr, dash, "\cong^k"] & & cs'_i \arrow[rr, "\mathtt{w}_{\mathtt{C}}(k-1)"] \arrow[dr, dash, "\cong^k"] & & cs'_{i+1} \arrow[rr, dashed, "\tau"] \arrow[dr, dash, "\cong^k"] & & cs'_{i+1} \arrow[rr, squiggly] \arrow[dr, dash, "\cong^k"] & & cs'_{i+1} \arrow[rr, dashed, "\tau"] \arrow[dr, dash, "\cong^k"] & & cs'_{i+1} \arrow[dr, dash, "\cong^k"] & \\
& cs_0 \arrow[rr, "\mathtt{wb}_{\mathtt{C}}"'] & & cs_1 \arrow[rr, squiggly] & & cs_i \arrow[rr, "\mathtt{w}_{\mathtt{C}}(k-1)"'] & & cs_{i+1} \arrow[rr, "\mathtt{w}_{\mathtt{C}}(k)"'] & & cs_{i+2} \arrow[rr, squiggly] & & cs_{n-1} \arrow[rr, "\mathtt{ws}_{\mathtt{C}}"'] & & cs_n
\end{tikzcd}$$

$$\mathtt{write}_{\mathtt{VFS}}$$

Figure 6.14: Lifting commuting diagrams to VFS, exemplary for a **vfs_write#** run (omitting the original $\mathtt{AFS}_{\mathtt{C}}$ run for readability).

does not differ from the original one. For the preservation of $\cong^k$, neither a write to a page greater than or equal to k nor a size update may be performed in the write-prefix execution, so $\mathtt{write}'_{\mathtt{VFS}}$ must stutter for further $\mathtt{w}$ and $\mathtt{ws}$ steps in $\mathtt{write}_{\mathtt{VFS}}$. In order to achieve a valid **vfs_write#** execution, the operation has to be aborted due to an error in $\mathtt{w}$ or $\mathtt{ws}$, respectively (see Figures 4.22, 4.23, and 4.24). Hence, we append the failing variant of the commuting diagram of Fig. 6.13b to end the **vfs_writeloop#**, and we continue with repeating stuttering steps of the τ variant of Fig. 6.13b until the state cs_{n-1} after **vfs_writeloop#** is reached. Finally, if the file size was increased in the original execution, one last stuttering commuting diagram for **writesize#** has to be applied. This commuting diagram is analogous to the one for **writepage#** in Fig. 6.13b: depending on whether a page is written beyond the file size sz in the write-prefix execution, this diagram either yields a failed transition $\mathtt{ws}_{\mathtt{A}}(i)^{\mathtt{ERR}}$ (if $\mathtt{pos}(k) \leq sz$ as shown in Fig. 6.14) or a stuttering τ transition (if $sz < \mathtt{pos}(k)$). Note that state cs'_{i+1} of the write-prefix execution after writing page $k-1$ remains unchanged as only stutter steps τ are taken and hence all cs_j with $i+1 \leq j \leq n$ satisfy $cs_j \cong^k cs'_{i+1}$.

For the construction of a write-prefix run for **vfs_truncate#**, only one modifying step needs to be considered, namely the call of $\mathtt{t}$. As shown in Fig. 6.12 in the lower half, **vfs_truncate#** calls may occur re-executed or failed in the write-prefix run, depending on whether they decrease the file size or they increase the file size. Hence, commuting diagrams analogous to Fig. 6.13a and Fig. 6.13b (only the variant with $\mathtt{t}_{\mathtt{A}}(i)^{\mathtt{ERR}}$) are necessary for $\mathtt{t}_{\mathtt{A}}$ and $\mathtt{t}_{\mathtt{C}}$. While re-executing only the minimal truncation would be sufficient to get a complete write-prefix run, the forward simulation $\cong^k$ requires that the truncate sizes in *tcache* and *tcache'* are identical after each step. As a consequence, every update of the minimal truncate size in the original run (when $n < \mathtt{min}(sz_T, sz_F)$) must result in the same update in the write-prefix run. So for such **vfs_truncate#** calls, the successful run is chosen while for all others (where $n \geq \mathtt{min}(sz_T, sz_F)$), the failed run is chosen.

Considering the final states of the runs shown in Fig. 6.12, *tcache*, $pcache|_k$, *tcache'*, *pcache'*, and *icache'* do not contain any *dirty* data for *ino*, and so applying them to $files_{\mathtt{P}}$ resp. $files'_{\mathtt{P}}$ does not have any effect. Consequently, in theses states $cs \cong^k cs'$ reduces to $files_{\mathtt{P}}.\mathtt{seq}(ino) = files'_{\mathtt{P}}.\mathtt{seq}(ino)$, which is exactly the property we wanted to achieve, namely identical `POSIX` states.

□

All in all, the verification of the crash-safety properties alone (not including earlier attempts) comprises approx. 300 theorems. The proofs were significantly more complex and required about ten times more interactions than the proofs for functional correctness (only 79% of the proof steps could be automated compared to 98%). Most of the time was spent proving the commuting diagrams for $\cong^k$ on the level of `Cache`(`AFS`$_{\mathtt{P}}$) since many different cases have to be considered. Lifting these to `AFS`$_{\mathtt{C}}$ could be done mainly by reusing the commuting diagrams for R together with some auxiliary lemmata over the `Cache`(`AFS`$_{\mathtt{P}}$) and `AFS` operations, which in turn enabled proving the commuting diagrams for `VFS`(`AFS`$_{\mathtt{C}}$) without major issues.

6.4 Crash-Safe Combination of Caches

In the previous sections, Write-Prefix Crash Consistency was considered without using other caches simultaneously. However, the high-level caches implemented in `Cache` are built on top of the buffer cache implemented in `Write Buffer`. Thus, the crash-safety criterion and the correctness of the `Cache` component must be considered in the context of retractions as introduced in Sec. 5.1.

In particular, we want to show that the crash effects of stacked caches can be aggregated: the crash effect of the lower-level buffer cache is applied first, and the effect of the high-level caches is applied afterward to the resulting state.

Theorem 9 (Compatibility of WPCC and QSCC)**.** *Write-Prefix Crash Consistency is compatible with the use of order-preserving caches in lower levels of the implementation that satisfy Quasi-Sequential Crash Consistency.*

Proof of Thm. 9. For compatibility of WPCC with QSCC, it is relevant that `AFS`$_{\mathtt{P}}$ operations do not have different executions: re-executing an `AFS`$_{\mathtt{P}}$ operation will lead either to the same result or to an error without any state change, which is equivalent to not executing the operation at all. Therefore, the effect of a crash is just to retract some of the final `AFS`$_{\mathtt{P}}$ operations. In the context of Def. 39, prefix-histories are not built based on the original eras I_i, h_i but on prefixes $I_i^{\sqsubseteq}, h_i^{\sqsubseteq}$ of them. This means that for each file, some of its final `AFS`$_{\mathtt{P}}$ operations are retracted as well, and thus $h'_i|_{fid}$ is also constructed based on a prefix $h_i^{\sqsubseteq}|_{fid}$ of $h_i|_{fid}$. For the crashed run of Fig. 6.12, the existence of the low-level cache will result in less persisting `AFS`$_{\mathtt{P}}$ operations being executed when a crash happens during **vfs_fsync#**. However, this either is the same scenario as before (case 2 in Sec. 6.3 with $k' \leq k$ instead of k) or results in a simpler crash (case 1 in Sec. 6.3 or in a state before **vfs_fsync#** started).

Note that executing `fs` at the end of **cache_fsync#** is crucial since it synchronizes the `Write Buffer`: As each successful **vfs_fsync#** operation yields a synchronized state $s \in \mathtt{SYNC}^{\mathtt{VFS}}$, a crash cannot retract `AFS`$_{\mathtt{P}}$ operations called before the response of **vfs_fsync#**. Thus, $(h_i^{\sqsubseteq}|_{fid})_{\leq sync} = (h_i|_{fid})_{\leq sync}$ and $(h_i^{\sqsubseteq}|_{fid})_{>sync}$

simply contains shorter prefixes than $(h_i|_{fid})_{>sync}$, i.e., more failed operations and writes with fewer bytes written, which is entirely in line with Def. 39.

□

6.5 Related Work

Bornholt et al. [17] provide a model for POSIX-compliant file system that includes the specification of crashes. They introduce several *crash consistency models*, e.g., for the ext4 file system, and validate these models against real file system implementations with a tool called FERRITE. Crashes are modeled by maintaining a sequence of updated events: the state after a crash is constructed by choosing a prefix of the event sequence, reordering them according to the respective consistency criterion, and applying the resulting event sequence to the initial or last recovered state, respectively. Our criterion *Quasi-Sequential Crash Consistency* for order-preserving write-back caches (see Sec. 5.2) is inspired by Bornholt's *Sequential Crash Consistency*. Compared to Sequential Crash Consistency, QSCC allows re-executions of pending operations after the retraction, which may produce different events by yielding different states and outputs. Furthermore, the `POSIX` model of Flashix specifies a residual state-based crash effect by closing all opened files and removing orphans (cf. Sec. 4.2), which is omitted in [17]. Another restriction is that [17] does not support truncations, which are the crux of the matter when giving crash consistency criteria for non-order-preserving caches like *Write-Prefix Crash Consistency* as shown in Sec. 6.2 and Sec. 6.3.

BilbyFS [4, 3, 2] by Amani et al. implements caching mechanisms and gives a specification of the *sync* operation on the level of `AFS`. These mechanisms are similar to the buffer cache implemented in Flashix in form of the `Write Buffer` component presented in Sec. 5.2. The functional correctness of this order-preserving cache is proven, for which pending writes that are buffered in-memory are specified explicitly as a sequence of *file-system transformations*. However, the verification of crash-safety properties for the buffer cache and recovery mechanisms still remains future work. Non-order-preserving caches have not been considered so far, and caching on the level of `VFS`, as discussed in sections 6.1-6.3, has not been addressed either since BilbyFS is designed for the use with the Linux VFS.

DFSCQ [23, 24] employs caching that are non-order-preserving. Similar to our approach, structural updates to the file system tree are persisted in order using a sequential log. However, it allows writes to *bypass* the log, so that content updates can commute arbitrarily with structural operations. Furthermore, DFSCQ uses a page cache but does not specify an order in which cached pages are written to persistent store. Therefore, it is not provable that a crash leads to a POSIX-conforming alternate run. Instead, a weaker crash-safety criterion is satisfied, called *metadata-prefix* specification: it is proved that a consistent file system results from a crash, where a prefix of the *metadata operations* (they correspond to the term *structural operations* used in this thesis) took place, and some subset of the page writes has been executed.

Like in BilbyFS, the effects of the in DFSCQ employed caches are modeled using a state-based approach. *Tree sequences* store a snapshot of the file system tree for each transaction, i.e., each structural operation. In addition, possible contents of individual pages are specified. A crash then selects a file system tree from the sequence and assigns one of its possible contents to each page individually. DFSCQ also implements the *fsync* operation and guarantees that all file data (metadata and content) is persisted after successful execution and cannot be reverted by a crash.

In the context of Flashix, the weaker criterion of DFSCQ should be provable for any (functional correct) implementation of VFS caches, since we ensure that all $\mathtt{AFS}_\mathtt{P}$ operations are atomic (calls can never overlap) and the refinement proof of $\mathtt{VFS} \leq \mathtt{POSIX}$ has lemmas for all AFS operations, that ensure that even these (and not just the VFS operations) preserve the abstraction relation to a consistent file system. Indeed, no stronger result is possible for the original VFS implementation in Linux, which offers many caching strategies, and does not even protect files from concurrent writes with mixed results.

For applications, when WPCC is not satisfied, the only strategy to achieve crash-safety is typically to write a new version of the whole file, call *fsync* on the new version, and atomically replace the old with the new version (using the *rename* operation). While this is efficient for small files, e.g. text files modified by an editor, it is inefficient for larger files.

Chapter 7

From Sequential to Concurrent Systems

Summary Introducing concurrency into sequential systems is a complex task. When operations are executed in parallel, the semantics of components changes as runs now consist of interleaved steps of multiple threads. Thus, the correctness of components is affected as well, and sequential reasoning about components does not suffice anymore.

This chapter introduces the semantics for *concurrent components* as an adaptation of the semantics of sequential components. Correctness of concurrent components is shown by proving *linearizability* using *atomicity refinement*: atomic program statements are incrementally combined to larger ones by applying a technique called *reductions* and thread-local reasoning with a *rely-guarantee calculus*. To facilitate this process, programs are augmented with locking instructions and *ownership* for the state of components is specified, which allows to partially automate atomicity reasoning.

The methodology presented in this chapter is, to a large extent, a summary of the work of Pfähler [96]. This chapter serves as a basis for Ch. 8, where the methodology is applied to introduce concurrency on the top levels of the Flashix file system, namely in the form of a concurrent garbage collection mechanism and concurrent executions of `POSIX` operations.

Contents

Publications The concept of concurrent components and the verification methodology for proving linearizability via atomicity refinement presented in this chapter is mainly adopted from the work of Pfähler [96]. The publication [108] summarizes the main aspects of the approach, the presentation of the rely-guarantee calculus is based on the publication [107].

7.1 Rely-Guarantee Calculus

For the verification of concurrent programs, the logic is based on the idea of having *programs as formulas.* For this, the semantics of expressions e (cf. Sec. 2.1) is generalized from $[\![e]\!](v)$ to $[\![e]\!](I)$ using an interval I (see semantics of programs in Sec. 2.3) instead of a single state v. The expressions considered so far refer to the initial valuation $I(0)$ of the interval only. The extended semantics makes it possible to view a program α with free variables $\underline{z}$ as a formula $[: \underline{z} \mid \alpha\ ]$ (expression with boolean result) which returns **tt** iff the semantics of α includes the interval I. That α has a temporal property φ is then simply expressed as the implication $[: \underline{z} \mid \alpha\ ] \to \varphi$ or, in terms of sequents, as $[: \underline{z} \mid \alpha\ ] \vdash \varphi$.

The resulting calculus (described in detail in [110]) is again based on symbolic execution of programs as well as temporal formulas. It is strong enough to define *rely-guarantee (RG)* formulas as abbreviations of temporal logic formulas and formally derive the rules of rely-guarantee calculus. Initially, rely-guarantee reasoning was introduced by Jones [69, 70]. Xu et al. extended the method with reasoning about deadlock- and divergence-freedom later in [117]. Since rely-guarantee calculus is predominantly used in the practical verification of programs in KIV, rely-guarantee formulas were added explicitly to the KIV system, and the rules for them were derived. While KIV supports various temporal logic formulas, like the standard formulas $\Box\ \varphi$, $\Diamond\ \psi$, $\ldots$ of linear temporal logic (LTL) [99], we will focus on rely-guarantee formulas in the following since they are used for reasoning about the correctness of concurrent components.

Variables are now partitioned into flexible variables, that may be modified by concurrent programs, and static variables, that always have the same value in all states $I(0), I'(0), \ldots$ of an interval. For the remainder of this section, we use y to denote a static and z for a flexible variable[1]. Flexible variables are allowed in quantifiers, but not as parameters of λ-expressions. Flexible variables z can also be used in primed or double primed form in predicate logic expressions (z' or z'', respectively). $[\![z']\!](I)$ and $[\![z'']\!](I)$ are defined as $I'(0)(z)$ and $I(1)(z)$, except for the case where the interval consists of a single state, i.e., $\#\ I = 0$. For such an *empty* interval the value of both is $I(0)(z)$. Formulas like $z' = z$ or $z'' \geq z'$ therefore talk about the relation of the first program step (z is not changed) and about the first environment step (z is not decremented). They are used as guarantee and rely formulas that constrain program and environment steps. We write φ' and φ'' for predicate logic formulas where one resp. two extra primes are added to every free variable that is flexible.

The rely-guarantee method reduces reasoning about concurrent systems, in which multiple threads execute programs interleaved, to thread-local reasoning about single programs. Therefore, the properties of programs are expressed by distinguished formulas over a sequential program α of some thread that executes atomic steps. These alternate with environment steps, where one environment step is an arbitrary

[1]In KIV, the convention is used that flexible variables start with uppercase letters while all others are static.

sequence of steps of other threads.

$$[:\ \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''),\ \mathit{guar}(\underline{z}, \underline{z}'),\ \mathit{inv}(\underline{z}),\ \alpha\ ](\varphi\ ;\ \underline{\xi})$$
$$\langle:\ \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''),\ \mathit{guar}(\underline{z}, \underline{z}'),\ \mathit{inv}(\underline{z}),\ \mathit{runs}(\underline{z}),\ \alpha\ \rangle(\varphi\ ;\ \underline{\xi})$$

Using these formulas, partial and total correctness are expressed as sequents.

$$\psi \vdash [:\ \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''),\ \mathit{guar}(\underline{z}, \underline{z}'),\ \mathit{inv}(\underline{z}),\ \alpha]\ \varphi \tag{1}$$
$$\psi \vdash \langle:\ \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''),\ \mathit{guar}(\underline{z}, \underline{z}'),\ \mathit{inv}(\underline{z}),\ \mathit{runs}(\underline{z}),\ \alpha\rangle\ \varphi \tag{2}$$

Like for the wp-formulas, we leave away the default exception condition $\underline{\xi} \equiv$ **default** :: `false` that forbids any exceptions for the final state.

Formula (1) asserts that if the precondition ψ holds in the initial state, program steps of α will not violate the guarantee *guar* (and do not throw exceptions), and the final state will not violate the postcondition φ unless an earlier environment step violated the rely *rely*. The formula implies that partial correctness holds: if all environment steps are rely steps, all program steps will be guarantee steps and in final states the postcondition will hold.

Total correctness, given by formula (2), guarantees two additional properties. First, the program α is guaranteed to terminate when the rely is never violated. Second, in all states where predicate *runs* holds, the next program step is guaranteed not to be blocked. The additional *runs*-formula is used to verify *deadlock-freedom*: when an interleaved program satisfies total correctness with *runs* = `true`, the program is deadlock-free, i.e., there is always at least one of its threads not currently waiting to acquire a lock. Conversely, when *runs* = `false` is chosen, the formula guarantees *divergence-freedom* only: the program α is allowed to block arbitrarily but must only perform finitely many non-blocking steps.

Note that the invariant formula *inv* specifies conditions that must hold in each state, regardless of whether a program or an environment step produced the state. However, this is just an abbreviation for adding the conditions to both the rely and guarantee formulas. For example, in [117], the RG assertion (1) would be written as

$$\alpha\ \underline{\mathbf{sat}}\ \{\psi,\ \mathit{rely}(\underline{z}', \underline{z}'') \wedge (\mathit{inv}(\underline{z}') \to \mathit{inv}(\underline{z}'')),\ \mathit{guar}(\underline{z}, \underline{z}') \wedge (\mathit{inv}(\underline{z}) \to \mathit{inv}(\underline{z}')),\ \varphi\}$$

Since both wp- and RG-formulas are defined based on the interval semantics of programs, they can be transformed into each other. This property can be exploited, for example, to apply lemmas formulated in wp-calculus to RG-proofs.

$$[\alpha]\ \varphi \equiv [:\ \underline{z} \mid \underline{z}'' = \underline{z}',\ \mathtt{true},\ \mathtt{true},\ \alpha]\ \varphi$$
$$\langle\!|\alpha|\!\rangle\ \varphi \equiv \langle:\ \underline{z} \mid \underline{z}'' = \underline{z}',\ \mathtt{true},\ \mathtt{true},\ \mathtt{true},\ \alpha\rangle\ \varphi$$

In the formulas, $\underline{z}$ are the flexible variables that occur free in α (all variables except those bound by **let** or **choose**) or free in φ.

Symbolic execution using the rely-guarantee calculus resembles symbolic execution in wp-calculus. The extra effort needed when proving RG formulas can be seen when looking at the rule for assignment listed in Fig. 7.1.

For easier notation, the rule assumes that all variables of the frame assumption $\underline{z}$ are assigned. We also omit side conditions for possible exceptions when evaluating $\underline{t}$. These are the same as in wp-calculus (see Sec. 2.4).

$$\frac{\begin{array}{l}\Gamma^{\underline{y}_0}_{\underline{z}},\ \mathit{inv}(\underline{y}_0),\ \underline{y}_1 = \underline{t}^{\underline{y}_0}_{\underline{z}} \vdash \mathit{guar}(\underline{y}_0, \underline{y}_1) \wedge \mathit{inv}(\underline{y}_1),\ \Delta^{\underline{y}_0}_{\underline{z}} \\ \Gamma^{\underline{y}_0}_{\underline{z}},\ \underline{y}_1 = \underline{t}^{\underline{y}_0}_{\underline{z}},\ \mathit{rely}(\underline{y}_1, \underline{z}) \vdash \langle : \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''), \mathit{guar}(\underline{z}, \underline{z}'), \mathit{inv}(\underline{z}), \mathit{runs}(\underline{z}),\ \alpha\rangle\ \varphi,\ \Delta^{\underline{y}_0}_{\underline{z}}\end{array}}{\Gamma\ \vdash \langle : \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''), \mathit{guar}(\underline{z}, \underline{z}'), \mathit{inv}(\underline{z}), \mathit{runs}(\underline{z}),\ \underline{z} := \underline{t}; \alpha\rangle\ \varphi,\ \Delta}$$

Figure 7.1: RG-calculus rule for assignments.

$$\frac{\begin{array}{l}\Gamma,\ \mathit{inv}(\underline{z}),\ \mathit{runs}(\underline{z}) \vdash \psi,\ \Delta \\ \Gamma^{\underline{y}}_{\underline{z}},\ \mathit{rely}(\underline{z}, \underline{y}) \\ \quad \vdash \langle\!|\alpha\{\underline{z} \mapsto \underline{y}\}|\!\rangle \left(\langle : \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''), \mathit{guar}(\underline{z}, \underline{z}'), \mathit{inv}(\underline{z}), \mathit{runs}(\underline{z}),\ \underline{z} := \underline{y}; \beta\rangle\ \varphi\right),\ \Delta^{\underline{y}}_{\underline{z}}\end{array}}{\Gamma\ \vdash \langle : \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''), \mathit{guar}(\underline{z}, \underline{z}'), \mathit{inv}(\underline{z}), \mathit{runs}(\underline{z}),\ \textbf{atomic}\ \psi\ \{\alpha\}; \beta\rangle\ \varphi,\ \Delta}$$

Figure 7.2: RG-calculus rule for atomic blocks.

In the semantics, symbolic execution of the assignment reduces the interval $I = (I(0), I(0)_b, I'(0), I(1), \ldots)$ to the one shorter interval $(I(1), \ldots)$. The values of variables $\underline{z}$ in states $I(0)$ and $I'(0)$ are now stored in two vectors $\underline{y}_0$ and $\underline{y}_1$ of fresh static variables, while the remaining program α again starts with the values of the variables in $\underline{z}$. The other formulas of the sequent (collected in Γ and Δ) now hold for $\underline{y}_0$, the values stored in $\underline{y}_1$ are equal to the ones of the terms $\underline{t}^{\underline{y}_0}_{\underline{z}}$. As the assignment rule shows, the RG-calculus has two differences to wp-calculus. First, there is an additional premise asserting that executing the assignment satisfies the guarantee (which is usually simple). Second, the main premise needs two vectors of fresh variables instead of one to store old values: one before and one after the assignment but before the environment step.

A crucial program construct for the verification methodology of this chapter is the **atomic** block. Fig. 7.2 shows the corresponding RG-calculus rule. The first premise guarantees that the guard of the atomic block is satisfied (and thus, the program does not block) when *runs* holds. For the second premise, the assumptions Γ and Δ are substituted with static variables $\underline{y}$ referring to the state before arbitrary many blocking steps. Nothing is known about these steps apart from that they are *rely* steps. The block α itself is executed in wp-calculus, calculating a new state $\underline{y}$. The actual program step is then performed by assigning the resulting state to the flexible frame variables $\underline{z}$, exploiting that wp-formulas can have program formulas as postconditions.

The remaining rules of RG-calculus (e.g., the invariant rules for partial and total correctness) look very similar to wp-calculus. Again, the only difference is that an additional premise is generated, ensuring that the step is a guarantee step.

Individual rely-guarantee proofs for single threads can be combined to a rely-guarantee property of a concurrent system. The crucial property that needs to hold for this to work is that the relies and guarantees must be *compatible*: the guarantee of each thread $\mathit{guar}_{\mathit{tid}}$ must imply the relies $\mathit{rely}_{\mathit{tid}'}$ of other threads $\mathit{tid}' \neq \mathit{tid}$. For the components used in this thesis, where all threads are known to execute the same operations, the guarantee can be chosen to be $\mathit{guar}_{\mathit{tid}} \equiv \bigwedge_{\mathit{tid}' \neq \mathit{tid}} \mathit{rely}_{\mathit{tid}'}$, the weakest guarantee possible that is trivially compatible.

7.2 Semantics & Correctness of Concurrent Components

In the previous chapters, components were only considered in a *sequential context*. Sec. 3.1 introduces sequential components with a state-based crash specification and Sec. 5.1 extends the component semantics with retracting crash transitions, an operations-based crash specification.

Def. 40 adjusts the semantics of components further to consider runs of a concurrent system. The concurrent system is given by a weak-fair interleaving (using the **forall**$\|$ construct, see Sec. 2.3) of sequential runs of an arbitrary number of threads $tid \in Tid$. Crashes now partition system runs into *concurrent eras* so that the retractions of crash transitions may yield states in which several operations are pending. Thus, multiple pending operations can be re-executed.

Definition 40 (Semantics of Concurrent, Retracting Components)**.** *The semantics of a concurrent, retracting component* $\mathtt{C}$ *is given by the labeled transition system* $C = (\mathtt{S}^{\mathtt{C}}, \mathtt{INIT}^{\mathtt{C}}, \mathcal{L}^{\mathtt{C}} \cup \{\tau\}, \rightarrow^{con}_{\curvearrowleft})$*. The state transition relation* $\rightarrow^{con}_{\curvearrowleft}$ *is determined by the set of observable system runs* $(I, h^{\mathtt{C}}) \in runs(C)$ *satisfying the conditions (1)-(3) and (7) of Def. 27 and*

4b. Each era $I_i, h_i^{\mathtt{C}}$ *with* $i < n$ *forms an interrupted run of the concurrent system, i.e.,* $I_i, h_i^{\mathtt{C}} \models \textbf{forall}\| \; tid \; \textbf{with} \; tid \in Tid \; \textbf{do} \; \{\{ \; \mathtt{OP}^{\mathtt{C}} \; \}^* \; ; \; \mathtt{OP}^{\mathtt{C}}_{\lightning}\}$*.*

5b. The last era $I_n, h_n^{\mathtt{C}}$ *forms an uninterrupted run of the concurrent system, i.e.,* $I_n, h_n^{\mathtt{C}} \models \textbf{forall}\| \; tid \; \textbf{with} \; tid \in Tid \; \textbf{do} \; \{ \; \mathtt{OP}^{\mathtt{C}} \; \}^*$*.*

6b. Each crash transition $I_i.last \xrightarrow{\lightning_{i+1}} I_{i+1}.first$ *with* $i < n$ *is a retracting crash transition for the era* $I_i, h_i^{\mathtt{C}}$ *and* $System^{\mathtt{C}} \equiv \textbf{forall}\| \; tid \; \textbf{with} \; tid \in Tid \; \textbf{do} \; \{\mathtt{OP}^{\mathtt{C}}\}^*$ *according to Def. 33.*

For concurrent components, the wp-calculus proof obligations of Lem. 1 alone are not sufficient to ensure termination and correct usage of subcomponents since they only consider executions in an empty environment. Therefore, we use the rely-guarantee method presented in the previous section to formulate suitable proof obligations. This requires an extended specification for concurrent components.

concurrent component $\mathtt{C}$
thread id tid
invariants $inv(\underline{s})$
rely condition $rely_{tid}(\underline{s}', \underline{s}'')$
deadlock freedom $runs_{tid}(\underline{s})$

An explicit, constant thread identifier tid of type threadid is given, which is used to distinguish the currently considered thread from environment threads syntactically. tid is used similar to *ghost state*, i.e., it cannot be read from concrete operations but is used for specification purposes only. Instead of a sequential invariant (which must hold in between operation executions), a *concurrent invariant* $inv(\underline{s})$ over the component state $\underline{s}$ can be given. This invariant is used in RG formulas to establish properties that must hold in every state of concurrent system runs. Similarly, a *rely condition* $rely_{tid}(\underline{s}', \underline{s}'')$ over the primed and double primed component state specifies a universal rely for the component, which is used in all RG formulas over all its operations. When no invariant or rely condition is specified, the formula `true` is used as default value. Guarantee conditions for components are not specified by hand: since all operations that manipulate the state are known, and they all use the same rely condition, the guarantee is defined automatically as

$$guar_{tid}(\underline{s}, \underline{s}') \equiv \forall\ tid'.\ tid' \neq tid \rightarrow rely_{tid'}(\underline{s}, \underline{s}')$$

Finally, a *runs predicate* over the component state can be given for proving *deadlock freedom.* If no formula is given, the default value `false` is used, which only ensures *divergence-freedom.* Using these formulas, Lem. 5 shows the proof obligations that must be shown for all concurrent components.

Lemma 5 (Invariants of Concurrent Components)**.** *For a concurrent component* `C`*, the proof obligations 1 and 4 of Lem. 1 and the following proof obligations are proven.*

2a. $pre_j(\underline{x}, \underline{y}, \underline{s}),\ inv(\underline{s})$
$\vdash \langle \colon rely_{tid}(\underline{s}', \underline{s}''),\ guar_{tid}(\underline{s}, \underline{s}'),\ inv(\underline{s}),\ runs_{tid}(\underline{s}),\ \mathbf{op}_j\#(\underline{x}; \underline{y}, \underline{s}; \underline{z})\ \rangle\ \texttt{true}$ *for all* $j \in J$

2b. $pre_j(\underline{x}, \underline{y}, \underline{s}_0),\ inv(\underline{s}_0),\ rely_{tid}(\underline{s}_0, \underline{s}_1) \vdash pre_j(\underline{x}, \underline{y}, \underline{s}_1)$ *for all* $j \in J$

3a. $guard_k(\underline{s}),\ inv(\underline{s})$
$\vdash \langle \colon rely_{tid}(\underline{s}', \underline{s}''),\ guar_{tid}(\underline{s}, \underline{s}'),\ inv(\underline{s}),\ runs_{tid}(\underline{s}),\ \mathbf{iop}_k\#(; \underline{s})\ \rangle\ \texttt{true}$ *for all* $k \in K$

5. $\vdash rely_{tid}(\underline{s}, \underline{s})$

6. $\vdash rely_{tid}(\underline{s}_0, \underline{s}_1) \land rely_{tid}(\underline{s}_1, \underline{s}_2) \rightarrow rely_{tid}(\underline{s}_0, \underline{s}_2)$

7. $inv(\underline{s}) \vdash \exists\ tid.\ runs_{tid}(\underline{s})$ *if* $runs_{tid} \neq \texttt{false}$

The obligations ensure divergence-freedom and the absence of exceptions, establish the concurrent invariant $inv(\underline{s})$*, and guarantee that* `C` *calls its subcomponents' interface operations only if their preconditions are satisfied. Furthermore, they ensure termination of all atomic blocks whenever their guard is satisfied, stability of preconditions over steps of other threads, and the compliance with the rely-guarantee discipline when using the guarantee* $guar_{tid} \equiv \bigwedge_{tid' \neq tid} rely_{tid'}$*. For* $runs_{tid} \neq \texttt{false}$*, obligation 7 also guarantees deadlock-freedom.*

□

Obligations 2a and 3a give the main RG proof obligations for interface and internal operations (corresponding to obligations 2 and 3 of LeminvariantsI). Obligation 2b ensures that the preconditions of interface operations do not get invalidated by steps of other threads (abstracted by the $rely_{tid}$). The predicate logic obligations 5 and 6 ensure that $rely_{tid}$ is a valid rely condition, i.e., it is reflexive and transitive.

Note that for $runs_{tid} = \texttt{false}$, the programs may block arbitrarily at atomic blocks (cf. the first premise of Fig. 7.2). For $runs_{tid} \neq \texttt{false}$, obligation 7 ensures that, at any time, $runs_{tid}$ is true for at least one thread. Thus, at least one thread does not have to wait at an atomic block since $runs_{tid}$ must imply the guards of all atomic blocks in the component due to the obligations 2a and 3a and the **atomic** rule shown in Fig. 7.2.

The standard criterion used to prove correctness of a concurrent implementation $\mathtt{C_{Iv}}$ (we use the subscript $\__{\mathtt{Iv}}$ to denote that a component is concurrent, i.e., has an *interleaved* semantics) with respect to an atomic specification $\mathtt{A_{At}}$ (the subscript $\__{\mathtt{At}}$ denotes that a component is atomic) is *linearizability* [60].

Informally, a concurrent implementation $\mathtt{C_{Iv}}$ with non-atomic operations $\mathtt{OP^C}$ is linearizable to an atomic specification $\mathtt{A_{At}}$ with atomic operations $\mathtt{OP^A}$ if the input/output behavior of each concurrent run can be explained by mapping them to the input/output behavior of some sequential run of $\mathtt{A_{At}}$.

The mapping between a concurrent and a sequential run is as follows: for each concurrent call of an operation $\mathtt{OP^C}$ that is invoked at time t_i and returns at time t'_i, find some point in time l_i with $t_i \leq l_i \leq t'_i$ such that all l_i are different. The point is called the *linearization point* of the operation call. Then construct some sequential run of $\mathtt{A_{At}}$ that executes each corresponding abstract operation $\mathtt{OP^A}$ atomically at time l_i. Note that even for fixed linearization points, this may give several sequential runs if the abstract operations are non-deterministic.

A refinement from $\mathtt{A_{At}}$ to $\mathtt{C_{Iv}}$ then is *linearizable* if for every concurrent run, linearization points and an abstract sequential run can be found such that all operation calls *have the same inputs and outputs.*

The clients of the interface then cannot distinguish the concurrent run from one where each operation call is delayed until time l_i, executes $\mathtt{OP^A}$ atomically, and then is delayed again until time t'_i.

We now define linearizability formally according to Herlihy and Wing [60], and extended the criterion with crash events. Each history h induces an irreflexive partial order $<_h$ on operations, which captures the "real-time" ordering of operations in h.

Definition 41 (Real-Time Order)**.** *The real-time order $<_h$ of operation executions Op_0 and Op_1 on a crash-free history h is defined as*

$$Op_0 <_h Op_1 \quad \textit{iff} \quad \textit{the response of } Op_0 \textit{ precedes the invocation of } Op_1 \textit{ in } h.$$

Two operation executions are executed concurrently in in the history h iff they are not ordered by the order $<_h$.

Linearization then extends a concurrent history by completing pending operations and reorders overlapping operations to get a sequential history.

Definition 42 (Linearizability)**.** *A crash-free history h is linearizable iff it can be extended (by appending response events) to a history h' such that completed(h') is equivalent to some sequential history h'' according to Def. 22, and h'' respects the real-time order of h, i.e., $<_h \subseteq <_{h''}$. Then the history h'' is called a linearization of h.*

Extending h to h' allows to complete pending operations that may have taken effect but have not yet returned their responses to the caller. The restriction to *completed*(h') corresponds to removing the remaining pending operations not yet having an effect. Adhering to $<_h$ ensures that operations are reordered only with operations they are overlapping with.

This criterion can now be lifted from crash-free histories (eras) to histories interrupted by crash events, forming *crash linearizations.*

Definition 43 (Crash Linearizability). *A history $h = h_0 \cdot \lightning_1 \cdot h_1 \cdot \lightning_2 \cdot \ldots \cdot \lightning_n \cdot h_n$ consisting of crash-free era histories $h_0, h_1, \ldots, h_n$ and separated by crash events $\lightning_1, \lightning_2, \ldots, \lightning_n$ is crash linearizable iff for each h_i with $i \leq n$, there exists a linearization h''_i. Then the combined history $h'' = h''_0 \cdot \lightning_1 \cdot h''_1 \cdot \lightning_2 \cdot \ldots \cdot \lightning_n \cdot h''_n$ is called a crash linearization of h.*

Finally, the correctness of a concurrent component $\texttt{C}_{\texttt{Iv}}$ w.r.t. to the atomic component $\texttt{A}_{\texttt{At}}$ can be defined by applying crash linearizability to the observable behavior of the components.

Definition 44 (Crash Linearizability of Components). *A concurrent component $\texttt{C}_{\texttt{Iv}}$ is crash linearizable with respect to a compatible atomic component $\texttt{A}_{\texttt{At}}$ iff for each $h \in Obs(\texttt{C}_{\texttt{Iv}})$ there is a crash linearization h' with $h' \in Obs(\texttt{A}_{\texttt{At}})$.*

Abstracting a concurrent component to an atomic specification is the key concept used for adding concurrency to the (originally) sequential refinement hierarchy of Flashix. Instead of considering concurrent runs of the potentially complex implementation, clients must only cope with sequential runs of the atomic specification. Obviously, this simplifies reasoning about the client component significantly.

As shown in Fig. 7.3, the abstraction process is split in two main steps. First, it is proven via *atomicity refinement* that the concurrent implementation $\texttt{C}_{\texttt{Iv}}$ can be abstracted to an atomic version $\texttt{C}_{\texttt{At}}$. $\texttt{C}_{\texttt{At}}$ has the same data and operations as $\texttt{C}_{\texttt{Iv}}$ but different atomicity. This introduction of atomicity is performed incrementally using *reductions*, as shown in the remainder of this section. Using the intermediate component $\texttt{C}_{\texttt{At}}$ has the advantage that a potential change of the data representation from $\texttt{C}_{\texttt{Iv}}$ to $\texttt{C}_{\texttt{At}}$ is completely decoupled from reasoning about concurrency resp. atomicity. Thus, the second abstraction step can be performed using data refinement as presented in Sec. 3.1 and Sec. 5.1, respectively.

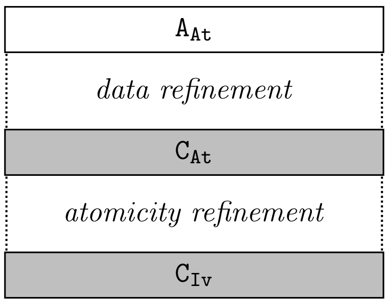


Figure 7.3: Abstracting a concurrent implementation $\texttt{C}_{\texttt{Iv}}$ to an atomic specification $\texttt{A}_{\texttt{At}}$.

Note that linearizability alone does not imply termination (resp. divergence-freedom) and deadlock-freedom. Hence, the reduction methodology presented in the following is not sufficient to prove these properties. However, the proof obligations of Lem. 5 cover these aspects (deadlock-freedom only if a sufficient runs predicate is given), so they are proven for all concurrent components. We call these components *strong linearizable* in the following (when fair locking implementations are used, the criterion would coincide with the classification *starvation-free* in [59]).

7.3 Proving Linearizability with Atomicity Refinement

The strategy for atomicity refinement follows Lipton's [80] idea of combining atomic statements to larger ones. The idea is that a thread executing two atomic steps at_1 and at_2 with an environment step in between is often equivalent to first executing the environment step, then at_1 and at_2 with no intermediate environment step. In this case the two steps can be merged together to form one atomic step.

Reverting the order of first executing at_1 and then an environment step is possible if all steps of other threads, that could be a part of the environment step, commute to the right with at_1 in the sense that executing them in both orders gives the same final state. Fig. 7.4 shows an example where the environment step consists of two steps at_m and at_n of other threads. The original run is shown at the bottom, and the alternative run, which allows to execute at_1 and at_2 as one atomic step, at the top. The intermediate states of the runs are different but they reach the same final state. Commutation of a step with environment steps is often trivial, for example if the step itself accesses local variables only. If at_1 commutes to the right with all steps of other threads, it is called a *right mover*. Dually, if at_2 commutes to the left with environment steps, it is called *left mover*. In this case a combined atomic step can be done before the first environment step.

This fundamental idea was revisited by Pfähler in his thesis [96], where he defined a reduction calculus for the programs used in concurrent, retracting components. The rules are similar to the ones given by Elmas et al. [37], which are an extension of Lipton's approach. In the following, the methodology of Pfähler will be recapped as a basis for the next chapter, where the methodology is applied to various layers of Flashix.

$$\begin{tikzcd}
 & s'_1 \arrow[r, "at_n"] & s'_2 \arrow[dr, "at_1"] \arrow[drr, bend left, "\textbf{atomic}\ \{at_1; at_2\}"] & & \\
s_0 \arrow[ur, bend left, "at_m"] \arrow[r, "at_1"] & s_1 \arrow[r, "at_m"] & s_2 \arrow[r, "at_n"] & s_3 \arrow[r, "at_2"] & s_4
\end{tikzcd}$$

Figure 7.4: The atomic step at_1 commutes to the right of the environment steps at_m and at_n.

Note that the calculus given by Pfähler also covers crash-safety. For this, he presents a more general reduction approach distinguishing between $\mathcal{R}$-retractable and $\mathcal{R}$-introducible programs ($\mathcal{R}$ is a generic binary relation over states, this relation is then instantiated with the crash relation $\lightning$) which slightly affects the reasoning in theory. However, this distinction is irrelevant for the practical application in Flashix since all in-memory steps and all atomic operations of specification components are proven to be *crash-neutral* (see Lem. 2 and Lem. 3, or Thm. 5 and Thm. 6 in [96]), which corresponds to Pfähler's introducibility criterion. Therefore, the following presentations will ignore this generalization and only show how the calculus is applied practically.

To apply reductions, we have to extract the atomic statements, called *atoms*, of a component first. These atoms can then be commuted to build bigger atomic blocks.

Definition 45 (Atoms of Programs)**.** *The set of atoms $At(\alpha)$ of a (sequential) program α contains a program for every non-stuttering transition of α, defined as the smallest set satisfying the following equations.*

$$At(\underline{x} := \underline{t}) = \{\underline{x} := \underline{t}\}$$
$$At(\alpha; \beta) = At(\alpha) \cup At(\beta)$$
$$At(\textbf{if } \varepsilon \textbf{ then } \alpha \textbf{ else } \beta) = \{bv := \varepsilon\} \cup At(\alpha) \cup At(\beta)$$
$$At(\textbf{choose } \underline{x} \textbf{ with } \varphi \textbf{ in } \alpha \textbf{ ifnone } \beta) = At(\alpha) \cup At(\beta)$$
$$\cup \{\textbf{choose* } \underline{y}' \textbf{ with } \varphi_{\underline{x}}^{\underline{y}'} \textbf{ in } \underline{x} := \underline{y}',\ bv := \texttt{true} \textbf{ ifnone } bv := \texttt{false}\}$$
$$At(\textbf{choose } x \textbf{ with } \varphi \textbf{ in } \alpha \textbf{ ifnone abort}) = At(\alpha) \cup At(\beta) \cup \{x :\in \varphi\}$$

$$\begin{aligned}
&At(\textbf{let } \underline{x} = \underline{t} \textbf{ in } \alpha) = \{\underline{x} \coloneqq \underline{t}\} \cup At(\alpha)\\
&At(\textbf{while } \varepsilon \textbf{ do } \alpha) = \{bv \coloneqq \varepsilon\} \cup At(\alpha)\\
&At(\textbf{proc\#}(\underline{t}; \underline{u}; \underline{v})) = \{\underline{y}' \coloneqq \underline{t}\} \cup At(\gamma)\\
&At(\textbf{atomic } \varepsilon\ \{\alpha\}) = \{\textbf{atomic } \varepsilon\ \{\alpha\}\}
\end{aligned}$$

where **proc#** *is a procedure with non-atomic declaration* **proc#**$(\underline{x}; \underline{y}; \underline{z})\{\gamma\}$*, and the variables* bv*,* $\underline{y}'$ *are globally fresh. The set of atoms* $At($`C`$)$ *of a component* `C` *is given by the union of atoms for the programs of every interface and internal operation of* `C`*.*

Def. 45 assumes w.l.o.g. that programs do not introduce local variables that shadow other program variables, including state variables of the component `C` in which they are defined. This is exploited in the sense that introduced local variables $\underline{x}$ of **choose** or **let** do not have to be renamed in the resulting atoms. Since all local variables are disjoint from the state variables of a component `C`, the local variables of an atom are determined by the difference of its free variables and the state variables of `C`.

Def. 45 is primarily defined on implementation programs (cf. Def. 16) since reductions are performed on implementation components while we consider specification components to be atomic already (after all, the goal is to show that a concurrent implementation component is linearizable w.r.t. to an atomic specification). However, the definition also covers cases for **atomic** blocks and **choose** statements. The former is necessary because atoms must be calculated repeatedly in different stages of the reduction: after the first reduction, some atoms usually have been combined into larger **atomic** blocks already, which are reduced further in subsequent iterations. The latter is relevant, for example, for the allocation of new heap locations (cf. Sec. 3.2). Since **let** is an abbreviation of **choose**, its case could also be covered by the one of **choose** but the tailored definition for **let** simplifies the commutation proofs. Similarly, an optimized case for **choose** with only a single introduced variable and an aborting **ifnone** case is defined, as this is the typical way of using choice programs: the choice-assignment $x :\in \varphi$ assigns a random value satisfying the condition φ to the variable x. Note that atoms make all branch decisions explicit by assigning the results of evaluating conditions to a globally fresh boolean variable bv (and similarly, the choice of values for $\underline{x}$ in **choose**s). As a result, commutations must also maintain these decisions. Furthermore, note that the equation for procedure calls applies only to procedures with non-atomic bodies. For procedures with atomic bodies, the atomic block is lifted outside of the call such that the equation for atomic blocks applies. Since a concurrent component mainly calls interface operations of its subcomponents, which are usually atomic specifications, this is the primary way calls are treated in atoms.

The atoms of a program α can be combined to *atom sequences*, i.e., the sequences of atoms that are executed during a execution of α. In the context of concurrent components, these sequences are always always finite since Lem. 4 guarantees divergence-freedom of all programs of the component. Thus, while loops and recursive procedures are guaranteed to perform only a finite number of non-blocking steps, and this number directly corresponds to the length of the atom sequence.

$$\frac{\begin{array}{l}\text{for all } \beta \in At(\mathtt{C}) : \\ \mathit{tid}' \neq \mathit{tid},\ \underline{y}_0 = \underline{y},\ \mathit{inv}^{\mathtt{C}}(\underline{s}^{\mathtt{C}}),\ \varphi_{\beta'} \\ \vdash [\beta']\Big(\varphi_\alpha \to [\alpha]\big(\langle \alpha\{\underline{y} \mapsto \underline{y}_0\}\ ;\ \beta'\{\underline{y} \mapsto \underline{y}_0\}\rangle\ \underline{y}_0 = \underline{y}\big)\Big)\end{array}}{\mathit{inv}^{\mathtt{C}} \vdash \alpha : \mathbb{L}} \quad \alpha \in At(\mathtt{C}) \qquad (\mathbb{L}\text{-Mover})$$

$$\frac{\begin{array}{l}\text{for all } \beta \in At(\mathtt{C}) : \\ \mathit{tid}' \neq \mathit{tid},\ \underline{y}_0 = \underline{y},\ \mathit{inv}^{\mathtt{C}}(\underline{s}^{\mathtt{C}}),\ \varphi_\alpha \\ \vdash [\alpha]\Big(\varphi_{\beta'} \to [\beta']\big(\langle \beta'\{\underline{y} \mapsto \underline{y}_0\}\ ;\ \alpha\{\underline{y} \mapsto \underline{y}_0\}\rangle\ \underline{y}_0 = \underline{y}\big)\Big)\end{array}}{\mathit{inv}^{\mathtt{C}} \vdash \alpha : \mathbb{R}} \quad \alpha \in At(\mathtt{C}) \qquad (\mathbb{R}\text{-Mover})$$

$$\frac{\mathit{inv}^{\mathtt{C}} \vdash \alpha : \mathbb{L} \quad \mathit{inv}^{\mathtt{C}} \vdash \alpha : \mathbb{R}}{\mathit{inv}^{\mathtt{C}} \vdash \alpha : \mathbb{B}} \quad \alpha \in At(\mathtt{C}) \qquad (\mathbb{B}\text{-Mover})$$

Figure 7.5: Calculus rules for inferring Left-, Right-, and Both-Movers. The derived atom β' results from the atom β by replacing *tid* with *tid*$'$ and local variables $\mathit{free}(\beta) \setminus \underline{s}^{\mathtt{C}}$ with fresh ones.

Consider now the following simple program α as an example.

$$\alpha \quad \equiv \quad \textbf{if}\ \varphi\ \textbf{then}\ \{x := t_0;\}\ \textbf{else}\ \{x := t_1;\}\ ;\ y := f(x);$$

The program has the set of atoms $At(\alpha) = \{bv := \varphi,\ x := t_0,\ x := t_1,\ y := f(x)\}$ according to Def. 45. Based on these atoms, α has two possible atom sequences, determined by the result of evaluating the **if** condition φ.

$$bv := \varphi;\ x := t_0;\ y := f(x);$$
$$bv := \varphi;\ x := t_1;\ y := f(x);$$

In order to reduce a program to an atomic block, all its atom sequences must be reducible, i.e., the atoms of a sequence must move in a way that they can “meet” at one point. This “meeting point” is an atom that marks the (or one possible) *linearization point* of the program execution: all preceding atoms of the sequence must move to the right, and all subsequent atoms of the sequence must move to the left.

In order to determine such mover properties, the calculus rules of Fig. 7.5 are applied. The rules $\mathbb{L}$-Mover and $\mathbb{R}$-Mover are used for proving that an atom $\alpha \in At(\mathtt{C})$ of the component C is a left or right mover, respectively. Therefore, one has to show that α commutes to the left (resp. to the right) of *all* atoms $\beta \in At(\mathtt{C})$ (including α itself) that are executed by another thread $\mathit{tid}' \neq \mathit{tid}$. In the derived atom β', the thread identifier *tid* is replaced with a different identifier *tid*$'$, and local variables of (all free variables except for $\underline{s}^{\mathtt{C}}$) are renamed to avoid clashes with the local variables of α. The variable vector $\underline{y}$ comprises all free variables of α and β',

$$\frac{\Gamma \ \vdash \Psi(L),\ \Delta \qquad \Psi(L) \vdash \langle : \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''), \mathit{guar}(\underline{z}, \underline{z}'), \mathit{inv}(\underline{z}), \mathit{runs}(\underline{z}),\ \alpha\rangle\ \varphi}{\Gamma \ \vdash \langle : \underline{z} \mid \mathit{rely}(\underline{z}', \underline{z}''), \mathit{guar}(\underline{z}, \underline{z}'), \mathit{inv}(\underline{z}), \mathit{runs}(\underline{z}),\ L\!:\!\text{-}\ \alpha\rangle\ \varphi,\ \Delta}$$

Figure 7.6: RG-calculus rule for assertions at label L.

including the shared state variables $\underline{s}^{\mathtt{C}}$. Atoms that are both left- and right-movers can be declared as both-movers (rule $\mathbb{B}$-Mover). If an atom does not move in any way, we call it a *non-mover* (abbreviated $\mathbb{A}$).

Proving mover properties is supported by two additions to the calculus rules of Fig. 7.5. First, the concurrent invariant $\mathit{inv}^{\mathtt{C}}(\underline{s}^{\mathtt{C}})$ of the component $\mathtt{C}$ can be assumed since the proof obligations of Lem. 4 ensure that the invariant is maintained in all possible interleaved runs of the component (we therefore write $\mathit{inv}^{\mathtt{C}} \vdash \alpha : \mathbb{L}/\mathbb{R}/\mathbb{B}$ for the statement that α is a left-/right-/both-mover under the invariant $\mathit{inv}^{\mathtt{C}}$).

Second, assertions φ_α and $\varphi_{\beta'}$ that hold for the respective atoms α and β are added as assumptions to rules. These assertions can be given for each atomic statement (i.e., for each *atom*) of a concurrent component. During the rely-guarantee proofs, it is then proven that the assertions always hold when the statement is executed (thus, they must be stable over the rely of the component). For the specification of assertions, programs are provided with labels L, each identifying one atom. For example, the program α from above is labeled as follows.

```
L1  if φ then                    assertions
L2     x := t0                     L1 : ψ1
    else                           L2, L3 : ψ2
L3     x := t1;                    L2 → L4 : ψ3
L4  y := f(x);                     L4 : ψ4
```

Note that labels L_2, L_3, and L_4 mark atoms that directly consist of the assignments of the program, but label L_1 identifies the atom $\mathit{bv} := \varphi$ build from the test of the conditional. As shown on the right of α, assertions can be given for individual labels (ψ_1 and ψ_4), for multiple distinguish labels (ψ_2), or for ranges of labels (ψ_3).

When symbolic execution of an RG formula reaches a labeled statement, the calculus rule of Fig. 7.6 is applied (the program $L\!:\!\text{-}\ \alpha$ denotes the program α at label L). The rule produces two premises: First, all given assertions for L must follow from the current assumptions Γ and Δ. Here, $\Psi(L)$ abbreviates the conjunction over all assertions specified for L. In the example, the combined assertion for label L_3 would be $\Psi(L_3) = \psi_2 \wedge \psi_3$. Second, the symbolic execution is continued regularly at α, but now the assertions $\Psi(L)$ can be assumed for this.

Using the assertions established this way, reduction proofs often become trivial. Assertions of both atoms are often contradictory, which means that the initial interleaving of $\mathbb{L}$-Mover or $\mathbb{R}$-Mover could not occur in the first place. As we will see later, this is especially the case when locks are used, e.g., two threads cannot be inside a critical region protected by a mutex at the same time. Thus, the respective statements do not have to move over each other, and the corresponding rule can be applied when assertions for both atoms are given that the respective threads hold the mutex.

Giving assertions has another advantage. As shown with the calculus rules in

Fig. 7.1 and Fig. 7.2, symbolic execution with rely-guarantee adds lots of predicate logic assumptions to the sequent. Particularly, each steps adds an *rely*-step to the antecedent, which adds up quite quickly to sequents containing hundreds of individual formulas. This complicates reasoning as one can easily lose the overview of the goal, but also slows down automation performed by the proof system since the number of formula combinations that have to be checked for simplifications/rewriting grows exponentially. For large programs, it can even go as far as being practical unusable.

Hence, we combined the symbolic execution approach with the one used, for example, in IO-Automata [81], where individual proofs for each possible step of the automaton are performed. The second premise of the calculus rule in Fig. 7.6 not only adds $\Psi(L)$ to the antecedent but also drops the other assumptions Γ and Δ, as they usually refer to older states, keeping only the rely-guarantee formula[2]. Of course, this requires giving assertions that describe the current state as precisely as possible to be able to continue the proof from there on successfully. In practice, the user can decide for each step individually (by declaring the assertions to be *weakening* or not) whether the assertions should be established in the strong fashion shown in Fig. 7.6 or whether old assumptions should be kept. This allows for a mixed proof approach in which weakening assertions regularly establish a concise sequent between short sequences of symbolically executed statements. Practice has shown that using this approach significantly improves conducting and maintaining proofs.

When mover properties are determined for the atoms of an atom sequence, the sequence can be combined into an atomic block if it is *reducible* according to Def. 46.

Definition 46 (Reductions of Programs)**.** *A program α of a component* C *is reducible if every finite atom sequence $\alpha_1; \alpha_2; \ldots; \alpha_n$ of α can be split into a prefix sequence $\alpha_1; \alpha_2; \ldots; \alpha_i$ and postfix sequence $\alpha_{i+1}; \ldots; \alpha_n$ for some $0 \leq i \leq n$ where*

1. *$inv^{\mathtt{C}} \vdash \alpha_j : \mathbb{R}$ holds for all $0 < j \leq i$,*
2. *$inv^{\mathtt{C}} \vdash \alpha_j : \mathbb{L}$ holds for all $i + 1 < j \leq n$,*
3. *$inv^{\mathtt{C}} \vdash \alpha_{i+1} : \mathbb{L}$ or $inv^{\mathtt{C}} \vdash \alpha_{i+1} : \mathbb{A}$ holds, and*
4. *the atoms α_j with $1 < j \leq n$ never block, i.e., have the guard* `true`*.*

Going back to the example program α with corresponding atom sequences, the following mover properties could be inferred.

$$\{bv := \varphi\} : \mathbb{R};\ \{x := t_0\} : \mathbb{R};\ \{y := f(x)\} : \mathbb{A};$$
$$\{bv := \varphi\} : \mathbb{R};\ \{x := t_1\} : \mathbb{L};\ \{y := f(x)\} : \mathbb{A};$$

It is easy to see that the first sequence matches all conditions of Def. 46 (note that the prefix and postfix sequences are allowed be empty), and thus it can be combined to an atomic sequence. However, the second sequence does not satisfy the required properties as a suitable split cannot be found. While the first two atoms can be combined by splitting the sequence at $i = 1$, the last atom cannot be added to the atomic prefix since it cannot move to the left as required by condition 2 of Def. 46

[2]The implementation of this rule in KIV does not drop all formulas naively, but keeps some by determining a set of formulas that may still be relevant heuristically. Nevertheless, the majority of formulas are removed.

(it is a non-mover $\mathbb{A}$). Thus, it must be shown that the atom $\{x := t_1\}$ also moves to the right (and therefore is a both-mover $\mathbb{B}$) to reduce the program α to

$$\textbf{atomic } \{\textbf{if } \varphi \textbf{ then } \{x := t_0;\} \textbf{ else } \{x := t_1;\} \ ; \ y := f(x);\}$$

Such reduction steps can then be applied to a concurrent component by Thm. 10.

Theorem 10 (Atomicity Refinement of Concurrent Components)**.** *Given a reducible program α of a divergence-free, concurrent component* `C`*, then wrapping α in an atomic block is sound, i.e., the refinement*

$$\mathtt{C}\{\alpha \mapsto \textbf{atomic } \varphi \ \{\alpha\}\} \leq \mathtt{C}$$

holds where the guard φ is chosen to be the guard of the first statement of α if it is an atomic block, or `true` *otherwise.*

$\square$

Reduction of a program is not possible if an atom in one of its atom sequences has a guard other than `true` and is not the sequence's leading atom (see 4 of Def. 46). Thm. 11 copes with that problem by giving a way to remove guards from atomic blocks. When a concurrent invariant could be established that implies the guard, i.e., the guard always holds outside of atomic blocks, it can be replaced with the guard `true`.

Theorem 11 (Removal of Atomic Guards)**.** *If a concurrent component* `C` *has a concurrent invariant $inv^{\mathtt{C}}$ that implies the guard φ of an atomic block, i.e., $inv^{\mathtt{C}} \vdash \varphi$, then the guard can be removed, i.e.,* $\mathtt{C}\{\textbf{atomic } \varphi \ \{\alpha\} \mapsto \textbf{atomic } \{\alpha\}\} \leq \mathtt{C}$.

$\square$

7.4 Locking & Ownership

Concurrent components must employ mechanisms to ensure correctness of the algorithms implemented. For example, a typical goal is to ensure that the implementation is *data race free*, i.e., that two threads cannot write simultaneously to a shared resource. The approach used in this thesis is the one of *lock-based concurrency*: critical regions in programs are surrounded by locks to restrict access to the protected resources.

Using locks also facilitates the reduction of a concurrent component $\mathtt{C_{Iv}}$ to its atomic counterpart $\mathtt{C_{Iv}}$. When a resource is protected properly by a lock, i.e., all accesses to that resource require acquiring the corresponding lock, atoms of $\mathtt{C_{Iv}}$ that read or write the resource are trivially movers since no other thread can access the resource concurrently.

By and large, Flashix uses two kinds of locks: *mutexes* and *reader/writer locks.* Both are used to acquire *ownership* of a particular resource (in the context of components, some part of the state). We specify ownership with a free data type owner.

$$\textbf{data} \quad \mathsf{owner} = \mathtt{readers}(.\ \mathtt{.tids} : \mathsf{set(threadid)}) \mid \mathtt{writer}(.\ \mathtt{.tid} : \mathsf{threadid})$$

Ownership is either exclusive, i.e., a single thread `tid` can be a `writer` of a resource, or shared, i.e., multiple threads are `readers` of a resource. The idea is that exclusive

```
mutex_lock#(tid; mtx) { atomic (mtx = free) {mtx := locked(tid)}; }
mutex_unlock#(tid; mtx) { if* mtx = locked(tid) then mtx := free else abort; }

rwlock_rlock#(tid; rwl)
  { atomic (rwl.read? ∧ ¬ tid ∈ rwl.tids) {rwl := rlock(rwl.tids ++ tid)}; }
rwlock_wlock#(tid; rwl)
  { atomic (rwl.read? ∧ rwl.tids = ∅) {rwl := wlock(tid)}; }
rwlock_runlock#(tid; rwl)
  { if* rwl.r?(tid) ∧ ¬ rwl.w?(tid) then rwl := rlock(rwl.tids -- tid) else abort; }
rwlock_wunlock#(tid; rwl)
  { if* rwl.w?(tid) then rwl := rlock(∅) else abort; }
```

Figure 7.7: Declarations for Mutex and Reader/Writer-Lock Operations.

ownership allows the (single) `writer` to read and modify the owned resource arbitrarily. In contrast, shared ownership grants access to the resource to several `readers`, which can only read the resource though. Crucially, there cannot be a writer and a reader of the resource simultaneously. For a given owner o, the predicate o`.w?`(tid) checks if *tid* is the current `writer`, and the predicate o`.r?`(tid) checks if *tid* has read permissions.

Locks are also defined as free data types. Following the idea of ownership, a mutex acquires exclusive ownership only, while an rwlock can be used to acquire shared or exclusive ownership.

```
data  mutex =  free | locked(. .tid : threadid)
data  rwlock = rlock(. .tids : set(threadid)) | wlock(. .tid : threadid)
```

A mutex *mtx* is either `free` (no thread has ownership) or `locked` (a single thread has exclusive ownership). A reader/writer lock *rwl* is *write-locked* (`wlock`) by a single thread or *read-locked* (`rlock`) by a set of threads. *rwl* is *free* if the set of reading threads is empty, i.e., rwl = `rlock`$(\emptyset)$. Mutexes *mtx* and reader/writer locks *rwl* can be translated into its corresponding owner with the function *mtx*`.owner` and *rwl*`.owner`, e.g., `free.owner` = `readers`$(\emptyset)$ and `wlock`(tid) = `writer`(tid).

These data types are not manipulated directly in programs. Instead, they are only accessed via the locking procedures given in Fig. 7.7. These correspond directly to locking operations of the widely used POSIX Threads (*pthreads*) library and are thus mapped to these operations when code is generated. The declarations use the **atomic** construct to model the expected behavior of locking and unlocking. Acquiring a mutex *mtx* blocks until *mtx* is `free` and then instantly locks *mtx* for the current thread *tid*, while releasing a mutex simply sets it to `free`. Note that **mutex_unlock#** only terminates regularly if *mtx* was acquired by the calling thread, otherwise it **abort**s. For reader/writer locks, there are distinct procedures for acquiring and releasing a lock *rwl* as reader or writer, respectively. **rwlock_rlock#** adds the current thread to the set of readers as soon *rwl* is a `rlock` (checked by the postfix predicate `.read?`). **rwlock_wlock#** blocks until *rwl* is an `rlock` without any readers and then changes `rlock` to a `wlock`. Conversely, **rwlock_runlock#** removes the current thread from the set of readers (*rwl*`.r?`(tid) and *rwl*`.w?`(tid) checks whether the thread *tid* has read resp. write permissions according to *rwl*),

```
concurrent component C using A

thread id  $tid$ : threadid
state  $mtx$ : mutex,  $m$ : nat
ownership
  $m$ owned by  $mtx$.owner

c_operation#(...)
  interface
  precondition ...
{
      ...
$L_i$   mutex_lock#($tid$; $mtx$);
      a_wacquire#();
      ...
      /* assert $mtx$.owner.w?($tid$) */
$L_j$   $m := m + 1$;
      ...
$L_k$   a_operation#(...);
      ...
      a_release#();
$L_l$   mutex_unlock#($tid$; $mtx$);
      ...
}
```

```
atomic component A

thread id  $tid$ : threadid
state  $n$ : nat
ghost state  $o$ : owner

a_wacquire#()
  ghost interface
{
  if* $o$ = readers($\emptyset$) then $o$ := writer($tid$);
}

a_release#()
  ghost interface
{
  if* $o$.r?($tid$) then $o$ := unlock($o$, $tid$);
}

a_operation#(...)
  interface
  precondition  $o$.w?($tid$) $\wedge$ ...
{
  $n := f(n)$;
}
```

Figure 7.8: Usage of locking and ownership in client- and subcomponents.

and **rwlock_wunlock#** resets rwl to an empty reader lock $\texttt{rlock}(\emptyset)$. Note that the given implementation of reader/writer locks is *non-reentrant*, i.e., calling a locking operation while the thread already holds the lock is illegal. The declarations reflect this circumstance by then blocking infinitely (the guards $\neg\ tid \in rwl.\texttt{tids}$ and $rwl.\texttt{tids} = \emptyset$ will never change be satisfied since the thread cannot be unlocked by other threads).

For the reduction of concurrent components, ownership and locks are used in multiple ways, particularly hierarchies of components are built. Fig. 7.8 shows an abstract, minimal example to illustrate these applications. First, consider the concurrent component C on the left in isolation. It defines an natural number m as state, together with a mutex mtx. This mutex is used to protect the variable m, i.e., all operations of C must have mtx acquired whenever they read or write m. For example, the operation **c_operation#** increments m at label L_j, for which mtx is acquired with **mutex_lock#** at L_i before and released with **mutex_unlock#** at L_l afterward. Thus, the assertions $L_{i+1} \to L_l : mtx = \texttt{writer}(tid)$ can be established, which include $L_j : mtx = \texttt{writer}(tid)$ in particular. Other operations must lock similarly when they read from or write to m, so the same assertion $mtx = \texttt{writer}(tid)$ holds for these statements. For different threads, the assertions obviously contradict each other ($\texttt{writer}(tid) \neq \texttt{writer}(tid')$ if $tid \neq tid'$), and thus the statements trivially commute. Since L_j only accesses m, it also commutes with all remaining statements, which do not access m, and so L_j is a both-mover $\mathbb{B}$.

This reasoning is quite generic for protected state variables. Hence, concurrent components provide a mechanism to reduce the specification and verification effort for this kind of ownership: an explicit owner can be given for state variables, e.g., the **owned by** clause of C specifies that *mtx* (more precisely, the extracted owner $\_mtx$.owner) is the owner of m. When such a clause is given (called an *ownership annotation*), all statements accessing the owned state variable are provided with an assertion checking that the required ownership is present. In the example, the assertion *mtx*.owner.w?(*tid*) must be proven at L_j since the statement writes to m (note that o.w?(*tid*) subsumes o.r?(*tid*)). The assertions are then proven in the rely-guarantee proofs, together with the assertions given by the user. As a consequence, atoms that only access *owned* state, like the one of L_j, are automatically inferred to be both-movers.

Furthermore, a generic rely condition can be generated for an ownership annotation x **owned by** o.

$$\begin{aligned}\vdash\ & \mathit{rely}_{tid}(o', o'', x', x'') \leftrightarrow \\ & \qquad (o'.\texttt{r?}(\mathit{tid}) \rightarrow x' = x'') \\ & \quad \wedge\ (o'.\texttt{r?}(\mathit{tid}) \leftrightarrow o''.\texttt{r?}(\mathit{tid})) \\ & \quad \wedge\ (o' = \texttt{writer}(\mathit{tid}) \leftrightarrow o'' = \texttt{writer}(\mathit{tid}))\end{aligned}$$

The formula states that the value of x is unchanged over the rely step, i.e., other threads do not modify x, if the thread *tid* has read permission for the field. Moreover, other threads do not remove (or add) read or write permissions from the thread *tid*.

Besides plain state variables, such *ownership assertions* can also be given for *access forms* of state variables, as they can also be directly assigned to. Basically, these include field selections of data types (x.sel) or selections of array and map locations (*ar*[n] resp. *mp*[k]). The rely conditions for owned access forms slightly differ from the generic rely shown above. They must take definedness conditions into account, e.g., only allocated locations in maps can be owned. However, custom relies are often necessary for more complex state compositions, so they will be introduced in the respective sections individually if relevant.

When subcomponents are used, ownership serves another purpose. It is used to encode that a (transitive) client component has acquired a lock and that hence certain operations cannot occur concurrently. Consider again Fig. 7.8 as an example. The component C uses an atomic component A as subcomponent, where A is potentially implemented by another concurrent component. A provides an interface operation **a_operation#** that calculates some new value $f(n)$ for the state variable n. The implementation of **a_operation#** can be more complex and may require that the operation is not called concurrently with certain other operations of the component to guarantee linearizability. For this reason, A uses an owner ghost state variable o to enforce a certain locking scheme in client components. An *ownership precondition* o.w?(*tid*) is added to **a_operation#** and all operations that could conflict when called concurrently. Since this precondition can only be satisfied by at most one thread at a time (there can be only one writer), concurrent calls to these operations are not possible.

C establishes these ownership preconditions by employing proper locking. In order to formally establish the precondition of **a_operation#**, **c_operation#** has to acquire the ownership by calling the ghost operation **a_wacquire#**. Analogously, the ownership must be released by calling the symmetrical ghost operation **a_release#**

after **a_operation#** was executed. However, **a_wacquire#** only sets the `writer` accordingly if no other thread has acquired ownership already (it is no actual locking operation). Thus, the client `C` is obliged to ensure that no other thread is able to hold ownership when **a_wacquire#** is called. This is again achieved by enclosing the calls to the ghost operations (and the call to **a_operation#** at L_k itself) by the locking range of *mtx*. The ownership conditions (particularly, the ownership preconditions) are then propagated downwards via refinement (the concurrent implementation must specify a corresponding ownership ghost state), and can be used for the reductions of the implementation.

An important aspect of the locking operations listed in Fig. 7.7 is that they have intrinsic mover properties. It can be shown that acquiring a mutex or reader/writer lock is always a right-mover $\mathbb{R}$, and symmetrically, releasing a mutex or reader/writer lock is a left-mover $\mathbb{L}$.

mutex_lock# : $\mathbb{R}$	**mutex_unlock#** : $\mathbb{L}$
rwlock_rlock# : $\mathbb{R}$	**rwlock_runlock#** : $\mathbb{L}$
rwlock_wlock# : $\mathbb{R}$	**rwlock_wunlock#** : $\mathbb{L}$

Analogously, it can usually be shown that ownership require and release operations, like **a_wacquire#** and **a_release#**, are also right- and left-movers, respectively.

Combining all the mover properties obtained for the example so far, the locking range $L_i \to L_l$ can be reduced to one atomic block (assuming that all omitted statements also move accordingly). The guard ($mtx =$ `free`) of the leading **mutex_lock#** call is lifted to the surrounding block, which prevents the block from being combined with a preceding atomic block at first. However, when locks are used in a hierarchical fashion, i.e., locking ranges adhere to a strict order, all locking ranges of *mtx* in `C` should be reduced to atomic blocks. Then the trivial concurrent invariant $mtx =$ `free` can be established, enabling the application of Thm. 11 and thus the removal of the guard.

Since the atoms of the reduced component are more coarse-grained, additional concurrent invariants may be established to facilitate further reductions. The reduction process is then repeated until all operations have atomic bodies, and thus the component can be considered atomic as well.

Chapter 8

Concurrency in File Systems

Summary This chapter presents another central contribution of this thesis, answering the question of how concurrency can be added correctly to higher levels of an originally sequential file system. Using the sequential Flashix hierarchy as starting point, concurrency is employed on various layers.
External concurrency is introduced by allowing concurrent user calls to the file system interface. A concurrent implementation of `VFS` is given that is linearizable to the original `POSIX` specification by realizing a suitable locking strategy. The structural integrity of the file system is ensured by restricting tree manipulations to be executed sequentially only, while writing and reading of file contents is allowed to be executed concurrently. An efficient mechanism for managing locks for dynamically allocated objects is realized and used by `VFS` for the locking of individual files. The particular locking strategy was chosen so that caches are still used in a crash-safe way, i.e., WPCC still holds for the concurrent implementation.
Internal concurrency of Flashix is extended further by moving the garbage collection mechanism implemented in the `Journal` layer into a separate thread, similar to what was already done with wear leveling in `EBM`. That way, users do not have to wait for these internal administrative tasks to finish after the execution of `POSIX` operations.
All concurrency extensions have been proven to be correct w.r.t. their atomic specifications by applying the methodology presented in Ch. 7.

Contents

Publications The methodology used to specify and prove the concurrency extensions of this chapter is published in [108]. The presentation of concurrent garbage collection is based on the publication [13].

8.1 Locking of Dynamically Allocated Objects

Similar to the integration of high-level caches (see Chapter 6), the main goal for allowing concurrent calls to the file system interface was to improve the performance of *content operations*, i.e., writing or reading the contents of files. Since content operations to different files affect different parts of the high-level state (`AFS`), a natural decision is to allow concurrent executions of these operations. In particular, writes to different files can be executed concurrently in `VFS`, as we will see in the following Sec. 8.2.

However, concurrent writes to the same file must be prohibited if the file system should adhere to correctness criteria such as *strong linearizability* (see Sec. 7.2) and *Write-Prefix Crash Consistency* (see Sec. 6.2). Concurrent writes to the same file could yield arbitrary interleaved calls to **afs_writepage#**, potentially resulting in mixed file content. Obviously, no *linearization point* can be found then, neither can a *write-prefix history*.

So, the concurrent implementation of `VFS` must employ locking mechanisms to prevent such concurrent executions but still allow harmless concurrent executions, i.e., concurrent reads or writes targeting different files. This can be achieved by using locking on the granularity of files, i.e., one lock restricts access to one file. As reading from a file in parallel is non-critical, using reader/writer locks (see Sec. 7.2) for this is an obvious choice. Hence, the state of `VFS` could be extended by a map *flocks* storing an rwlock for each ino of a file.

```
concurrent component VFS using AFS
state    ..., $flocks$ : map(ino, rwlock)
```

This, however, has one problem. Because the map *flocks* itself is an in-memory data structure that can be accessed concurrently, it must also be protected by a lock. Furthermore, lock entries in *flocks* must be allocated and deallocated dynamically during runtime, e.g., when a new file is created, an existing file is removed, or when a file is accessed the first time after a remount (allocating locks for all files at recovery is not reasonable as it would require to scan the entire file system). As a result, locking a file *ino* would be implemented as follows, for example.

```
mutex_lock#($tid$; $mtx$);
if $\neg$ $ino \in flocks$ then
  rwlock_init#(; $flocks$[$ino$]);
rwlock_wlock#($tid$; $flocks$[$ino$]);
mutex_unlock#($tid$; $mtx$);
```

Before the reader/writer lock for *ino* can be acquired by the current thread *tid*, it must be checked whether the corresponding lock is already allocated in *flocks*. If not, it has to be initialized first (the operation **rwlock_init#** creates an `rlock` with empty `tids`). The crux of the matter is that arbitrary steps of other threads can be performed between the conditional check $\neg$ *ino* $\in$ *flocks*, the initialization of the lock, and the actual acquisition of the lock with **rwlock_wlock#**. In order to prevent data races regarding the lock entry for *ino* (notably, no other thread should delete it in the meantime), the whole process must be protected, e.g., by a mutex *mtx* (read-locking is not sufficient since the domain of *flocks* is potentially modified).

Using this locking mechanism, the system can quickly get into unnecessary blocking of threads. As soon as the requested lock cannot be acquired immediately, the thread stays at the **rwlock_wlock#** call, still holding *mtx*. While the thread is waiting, no other thread can enter the locking range of *mtx*, and thus other content operations are blocked as no other file can be locked. To avoid such situations, we use a more elaborate mechanism for managing file locks, inspired by the Linux VFS.

Separating Locks from Protected Maps

The basic concept for solving this problem is twofold. First, the actual rwlock objects are not stored directly in a map. Instead, a map is only used to store a mapping from inode numbers to references that point to heap locations storing the corresponding locks. This way, a lock reference can be retrieved from the map (while locking it), and the targeted rwlock can be acquired afterward only via this reference. Second, the map also stores counters for the number of threads currently accessing the lock of the respective file, similar to the concept of *reference counting* (but with at most one reference to a lock per thread). The idea of this counter is that, as long as it is non-zero, some thread has the corresponding rwlock acquired or wants to acquire it. Thus, entries must not be deleted when threads are still using the associated lock (indicated by a counter greater than zero), which makes holding a lock over the actual (potentially blocking) lock operation, as it has to be done in the example above, unnecessary.

In order to detach this mechanism from the already complex VFS implementation, it is moved into a separate component CLocking. Since the implementation has to handle concurrent accesses to heaps, an atomic abstraction ALocking is used as interface for VFS. Hence, we get the resulting refinement hierarchy shown in Fig. 8.2 as instance of the generic atomicity refinement pattern of Fig. 7.3.

Instead of heap and map types, the abstract (atomic) component ALocking uses total functions for storing the *mapping* from inode numbers to references and the allocated *rwlocks*.

atomic component ALocking
thread id *tid* : threadid
state *mapping* : ino → lock−entry, *rwlocks* : ref → rwlock

where

data lock−entry = lentry(. .ref : ref; . .cnt : nat)

As the function *mapping* is total, it cannot directly be seen whether a lock for an *ino* is already allocated. Thus, non-allocated entries will store the null reference to distinguish them from allocated ones. The *rwlocks* function stores *free* reader/writer locks rlock(∅) as the default value, but only those with an *rwlocks* entry pointing to them are relevant.

The interface of ALocking offers operations for read-/write-locking and unlocking of files. However, they do not have an inode number *ino* as input but a reference *r*. The operations must be called with valid references pointing to an allocated rwlock only. Thus, clients must have a possibility to look up the lock-reference for a given *ino*.

Fig. 8.1 lists the two operations interface operations for managing such references. The operation **alocking_getlock#** on the left takes an inode number *ino* as input and returns a reference *r* pointing to the corresponding lock. If there is currently no

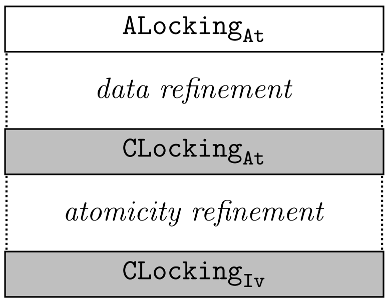
```
**alocking_getlock#**$(ino;;r)$
  **interface**
  **precondition** $\neg\ tid \in accessed(ino)$
{
  **if** $mapping(ino).\mathtt{ref} = \mathtt{null}$ **then** {
    **choose** $r_0$ **with**
      $r_0 \neq \mathtt{null}\ \wedge\ \neg\ \mathtt{mapped}(r_0, mapping)$
    **in**
      $mapping(ino) := \mathtt{lentry}(r_0, 1)$;
      $r := r_0$;
  } **else** {
    **let** $refcnt = mapping(ino)$ **in**
      $mapping(ino).\mathtt{cnt} := refcnt.\mathtt{cnt} + 1$;
      $r := refcnt.\mathtt{ref}$;
  };
  $accessed(ino) := accessed(ino)$ ++ $tid$;
}
```

```
**alocking_droplock#**$(ino)$
  **interface**
  **precondition**
      $tid \in accessed(ino)$
    $\wedge\ mapping(ino).\mathtt{ref} \neq \mathtt{null}$
    $\wedge\ \mathtt{idle}(tid, mapping(ino).\mathtt{ref}, rwlocks)$
{
  **let** $cnt = mapping(ino).\mathtt{cnt}$ **in**
    **if** $cnt = 1$ **then** {
      $mapping(ino) := \mathtt{lentry}(\mathtt{null}, 0)$
    } **else** {
      $mapping(ino).\mathtt{cnt} := cnt - 1$;
    };
  $accessed(ino) := accessed(ino)$ -- $tid$;
}
```

Figure 8.1: `ALocking` operations for accessing references to reader/writer locks.

lock allocated for *ino*, i.e., the entry *mapping*(*ino*) points to `null`, a fresh reference r_0 is chosen and returned (the predicate $\mathtt{mapped}(r_0, mapping)$ checks whether there is an entry in *mapping* storing r_0 as `ref`). The entry of *mapping*(*ino*) is then set to the new reference r_0 and initialized with the $\mathtt{cnt} = 1$: *tid* is the only thread accessing the lock at the moment. Note that an initialization of the rwlock in $rwlocks(r_0)$ is not necessary as it has the suitable default value $\mathtt{rlock}(\emptyset)$. When a lock is already allocated for *ino*, i.e., $mapping(ino).\mathtt{ref} \neq \mathtt{null}$, the reference stored in the entry is returned, and the counter is incremented: one more thread accesses the lock now.

Recall that `ALocking` is an *atomic* specification. Hence, the body of **alocking_getlock#** is executed in a single atomic step, which is why no locking is necessary within the operation. The implementation `CLocking` will have to use locks to realize this atomic behavior.



$\mathtt{ALocking}_{\mathtt{At}}$
data refinement
$\mathtt{CLocking}_{\mathtt{At}}$
atomicity refinement
$\mathtt{CLocking}_{\mathtt{Iv}}$

Figure 8.2: Locking refinement $\mathtt{CLocking}_{\mathtt{Iv}} \leq \mathtt{ALocking}_{\mathtt{At}}$.

The operation **alocking_droplock#** on the right of Fig. 8.1 is the counterpart to **alocking_getlock#**. It takes the *ino* of an *accessed* lock as input (the precondition $mapping(ino).\mathtt{ref} \neq \mathtt{null}$ ensures that a rwlock for *ino* is allocated) and decrements the counter for the lock entry. If the count would be decremented to 0, i.e., *tid* is the last thread accessing the lock for *ino*, it is directly "deallocated" by setting the `ref` of the entry to `null`. Again, the rwlock in *rwlocks* does not have to be updated since no other thread is allowed to have it acquired and *tid* is only allowed to call **alocking_droplock#** if it has released the lock before (the predicate $\mathtt{idle}(tid, r, rwlocks)$ in the precondition states that *tid* has not locked the rwlock at $rwlocks(r)$).

Besides *mapping*, both operations also manipulate a function *accessed*, which is a *ghost state variable* of `ALocking`.

$$\textbf{ghost state} \quad \mathit{accessed} : \mathsf{ino} \to \mathsf{set}(\mathsf{threadid})$$

It maps inode numbers to a set of thread identifiers, specifying explicitly which threads are accessing a rwlock. Thus, the value directly corresponds to the `cnt` of the *mapping* in the sense that the size $\#\ \mathit{accessed}(\mathit{ino})$ is always equal to the count $\mathit{mapping}(\mathit{ino})$`.cnt` for all inode numbers *ino*. This property is also specified by the predicate *accessed* `#=` *mapping* as part of the invariant of `ALocking`.

$$\begin{aligned}
&\textbf{atomic component } \texttt{ALocking}\\
&\textbf{invariants } \mathit{accessed} \texttt{ \#= } \mathit{mapping} \ \land\ \texttt{inj}(\mathit{mapping})\\
&\textbf{rely condition} \quad \texttt{rwlock-rely}_{\mathit{tid}}(\mathit{rwlocks}', \mathit{rwlocks}'')\\
&\qquad\qquad\qquad \land\ \texttt{accessed-rely}_{\mathit{tid}}(\mathit{accessed}', \mathit{accessed}'')\\
&\qquad\qquad\qquad \land\ \texttt{mapped-rely}_{\mathit{tid}}(\mathit{mapping}', \mathit{accessed}', \mathit{mapping}'', \mathit{accessed}'')
\end{aligned}$$

The second part of the invariant $\texttt{inj}(\mathit{mapping})$ enforces that the function *mapping* is *injective*, which apparently is a fundamental requirement for the mechanism to work correctly.

The component also defines some *rely conditions*. Note that these are not relevant for the correctness of the isolated component (since it is atomic), but are necessary for the integration as a subcomponent into `VFS`. While they could be proven in `VFS` in principle, it is way less effort to prove them directly in the component because they only need to be proved for runs of `ALocking` operations rather than entire `VFS` operations.

The rely for *rwlocks* is relatively standard for reader/writer locks but lifted to functions. When the thread *tid* has read- or write-locked a location *r* in *rwlocks* (specified by the predicates `rlocked` and `wlocked`), the thread will keep the lock over the rely step. On the other hand, other threads also do not lock a location for *tid*.

$$\begin{aligned}
&\vdash \texttt{rwlock-rely}_{\mathit{tid}}(\mathit{rwlocks}', \mathit{rwlocks}'') \leftrightarrow\\
&\qquad \forall\ r.\quad \big(\texttt{rlocked}(\mathit{tid}, r, \mathit{rwlocks}') \leftrightarrow \texttt{rlocked}(\mathit{tid}, r, \mathit{rwlocks}'')\big)\\
&\qquad\qquad \land\ \big(\texttt{wlocked}(\mathit{tid}, r, \mathit{rwlocks}') \leftrightarrow \texttt{wlocked}(\mathit{tid}, r, \mathit{rwlocks}'')\big)
\end{aligned}$$

Similarly, when *tid* has "registered" itself as accessing the lock of an *ino*, other threads do not "unregister" *tid*, or vice versa.

$$\begin{aligned}
&\vdash \texttt{accessed-rely}_{\mathit{tid}}(\mathit{accessed}', \mathit{accessed}'') \leftrightarrow\\
&\qquad \forall\ \mathit{ino}.\ \big(\mathit{tid} \in \mathit{accessed}'(\mathit{ino}) \leftrightarrow \mathit{tid} \in \mathit{accessed}''(\mathit{ino})\big)
\end{aligned}$$

Finally, when *tid* accesses the lock of an *ino*, the location for this lock is not changed in *mapping*. In particular, no other thread deallocates the lock of *ino* while *tid* still uses it.

$$\begin{aligned}
&\vdash \texttt{mapped-rely}_{\mathit{tid}}(\mathit{mapping}', \mathit{accessed}', \mathit{mapping}'', \mathit{accessed}'') \leftrightarrow\\
&\qquad \forall\ \mathit{ino}.\ \mathit{tid} \in \mathit{accessed}'(\mathit{ino}) \to\\
&\qquad\qquad\qquad \mathit{mapping}'(\mathit{ino})\texttt{.ref} = \mathit{mapping}''(\mathit{ino})\texttt{.ref}
\end{aligned}$$

```
alocking_rlock#($r$)
  interface
  precondition
      idle($tid, r, rwlocks$)
    $\wedge$ accesses($tid, r, mapping, accessed$)
  postcondition
      rlocked($tid, r, rwlocks$)
    $\wedge$ accesses($tid, r, mapping, accessed$)
{
  rwlock_rlock#($tid; rwlocks(r)$);
}
```

```
alocking_runlock#($r$)
  interface
  precondition
      rlocked($tid, r, rwlocks$)
    $\wedge$ accesses($tid, r, mapping, accessed$)
  postcondition
      idle($tid, r, rwlocks$)
    $\wedge$ accesses($tid, r, mapping, accessed$)
{
  rwlock_runlock#($tid; rwlocks(r)$);
}
```

Figure 8.3: ALocking operations for acquiring and releasing a reader-lock.

The necessity for these relies becomes apparent when looking at the declarations for locking and unlocking an rwlock, as listed for example in Fig. 8.3 for reader-locks (the operations for writer-locks are analogous). The operations just call the respective operations **rwlock_rlock#** and **rwlock_runlock#** for the locks stored at r in *rwlocks*. However, for the calls to be valid, i.e., targeting an allocated rwlock, the precondition accesses must hold.

$$\vdash \texttt{accesses}(tid, r, mapping, accessed) \leftrightarrow$$
$$r \neq \texttt{null} \wedge \exists\ ino.\ mapping(ino)\texttt{.ref} = r \wedge tid \in accessed(ino)$$

The reference r must be a valid mapped referenced in *mapping* and the requesting thread *tid* must be in *accessed* of the corresponding *ino*. Together with the latter two relies, this guarantees that the reference r stays valid after retrieving it via **alocking_getlock#**, i.e., no other thread deallocates the lock at r.

Note that at this abstraction level, where total functions are used, the access $rwlocks(r)$ is actually non-critical in terms of definedness. Nevertheless, it becomes very well critical for the implementation CLocking where the functions are implemented by heaps and maps, as we will see in the following. Thus, the preconditions must be enforced by ALocking in order to enable the refinement $\texttt{CLocking}_{\texttt{At}} \leq \texttt{ALocking}_{\texttt{At}}$.

Concurrent Management of Locks in Heaps

The implementation CLocking is in many aspects quite similar to ALocking. The state of CLocking contains the same mappings, however, functions are now implemented by maps and heaps, for which code can be generated. The heap *rwh* corresponds to *rwlocks*, the maps *cmapping* and *caccessed* correspond to *mapping* and *accessed*, respectively.

concurrent component CLocking

thread id *tid* : threadid

state *cmapping* : map(ino, lock−entry), *rwh* : heap(rwlock), *mtx* : mutex

ghost state *caccessed* : map(ino, set(threadid))

ownership *cmapping*, *caccessed* **owned by** *mtx*.owner

```
clocking_rlock#(r)
  interface
  precondition
      idle(tid, r, rwh)
    ∧ accesses(tid, r, cmapping, caccessed)
  postcondition
      rlocked(tid, r, rwh)
    ∧ accesses(tid, r, cmapping, caccessed)
{
  rwlock_rlock#(tid; rwh[r]);
}
```

```
clocking_runlock#(r)
  interface
  precondition
      rlocked(tid, r, rwh)
    ∧ accesses(tid, r, cmapping, caccessed)
  postcondition
      idle(tid, r, rwh)
    ∧ accesses(tid, r, cmapping, caccessed)
{
  rwlock_runlock#(tid; rwh[r]);
}
```

Figure 8.4: `CLocking` operations for acquiring and releasing a reader-lock.

Since the component is concurrent, a mutex *mtx* is used to prevent data races by protecting *cmapping* and *caccessed*. The ownership annotation specifies this explicitly, generating assertions and inferring movers automatically as presented in Sec. 7.4.

Fig. 8.4 shows the implementation of the read-lock operations listed in Fig. 8.3. The operations are nearly identical to their specifications: they also call either **rwlock_rlock#** or **rwlock_runlock#**, but now for heap locations. The contracts use analogous predicates defined on the concrete heap and map types.

The implementations for accessing references shown in Fig. 8.5 are more interesting as they now have to cope with concurrent calls to the component. Both operations use *mtx* to ensure that these operations cannot be executed in parallel as this would easily yield inconsistent data structures. But note that the locking operations in Fig. 8.4 do not acquire the mutex *mtx* and can thus be called concurrently to each other and to the operations of Fig. 8.5. Particularly, a blocking **rwlock_rlock#** call in **clocking_rlock#** does not affect the locking of other threads since a execution of **clocking_getlock#** or **clocking_droplock#** cannot block once the locking range of *mtx* is entered.

The basic steps of **clocking_getlock#** or **clocking_droplock#** are still the same. The former checks whether *ino* is already mapped: if *ino* is mapped, the operation returns the stored reference from *cmapping* while incrementing the counter of the entry; if *ino* is not yet mapped, an entry with fresh reference and the counter 1 is added to *cmapping*. The latter decrements the counter of *ino* and removes the entry from *cmapping* if *tid* was the last thread accessing *ino*. The updates to *caccessed* are also performed analogously; they are just outsourced to **ghost auxiliary** operations **clocking_addaccess#** and **clocking_rmaccess#** (not shown, they must consider multiple cases as *caccessed* is not total). However, they additionally have to update the heap *rwh* accordingly: the fresh reference r_0 is allocated in *rwh* together with initializing the rwlock at that location via **rwlock_init#**, or the reference *refcnt*.`ref` is deallocated in *rwh* when no other thread accesses the lock, respectively. These are the crucial statements for correctness as they are not protected by *mtx* and thus potentially interfere with acquiring/releasing an individual reader/writer lock (cf. Fig. 8.4).

Therefore, suitable concurrent invariants and rely conditions are established. As for `ALocking`, the invariants establish injectivity of *cmapping* and relate the access counter of a mapped *ino* with the cardinality of the set of accessing threads in

```
clocking_getlock#($ino$; ; $r$)
  interface
  precondition
        $\neg$ $ino \in caccessed$
     $\vee$ $\neg$ $tid \in caccessed$[$ino$]
{
  mutex_lock#($tid$; $mtx$);
  if $\neg$ $ino \in cmapping$ then {
    choose $r_0$ with
      $r_0 \neq$ null $\wedge$ $\neg$ $r_0 \in rwh$
    in
      $rwh$ := $rwh$ ++ $r_0$;
      $cmapping$[$ino$] := lentry($r_0$, 1);
      rwlock_init#(; ; $rwh$[$r_0$]);
      $r$ := $r_0$;
  } else {
    let $refcnt$ = $cmapping$[$ino$] in
    let $cnt$ = $refcnt$.cnt in
      $cmapping$[$ino$].cnt := $cnt + 1$;
      $r$ := $refcnt$.ref;
  };
  clocking_addaccess#($ino$); // ghost
  mutex_unlock#($tid$; $mtx$);
}
```

```
clocking_droplock#($ino$)
  interface
  precondition
        $ino \in caccessed$
     $\wedge$ $caccessed$[$ino$]
     $\wedge$ idle($tid$, $caccessed$[$ino$].ref, $rwlocks$)
{
  mutex_lock#($tid$; $mtx$);
  let $refcnt$ = $cmapping$[$ino$] in
  let $cnt$ = $refcnt$.cnt in
    clocking_rmaccess#($ino$); // ghost
    if $cnt = 1$ then {
      $cmapping$ := $cmapping$ -- $ino$;
      $rwh$ := $rwh$ -- $refcnt$.ref;
    } else {
      $cmapping$[$ino$].cnt := $cnt - 1$;
    };
  mutex_unlock#($tid$; $mtx$);
}
```

Figure 8.5: `CLocking` operations for accessing references to reader/writer locks.

caccessed. For the latter, equality cannot be achieved in the fine-grained concurrent version of `CLocking`. Instead, $caccessed \subseteq cmapping$ states that the cardinality is at most the value of the corresponding counter. Note that the respective updates to *cmapping* and *caccessed* in Fig. 8.5 are arranged so that this property actually holds. Reductions will then increase the granularity such that the stronger equality property can be established. Additionally, $caccessed \subseteq cmapping$ imposes a subset relation between the domains of *caccessed* and *cmapping*. The new invariant `valid-refs`(*cmapping*, *rwh*) ensures that all references stored in *cmapping* are valid for *rwh*, i.e., they are not `null` and are allocated in *rwh*. Again, the order of statements in Fig. 8.5 is chosen so that this holds, e.g., a reference is allocated before it is stored in *cmapping* but deallocated only after its entry is removed from *cmapping*.

concurrent component `CLocking`

invariants $caccessed \subseteq cmapping \wedge \texttt{inj}(cmapping)$

$\wedge\ \texttt{valid-refs}(cmapping, rwh)$

rely condition

$$\texttt{heap-rely}_{tid}(mtx', rwh', rwh'')$$
$$\wedge\ \texttt{accessed-rely}_{tid}(caccessed', caccessed'')$$
$$\wedge\ \texttt{refs-rely}_{tid}(caccessed', cmapping', rwh', cmapping'', rwh'')$$
$$\wedge\ \texttt{last-rely}_{tid}(mtx', caccessed', cmapping', rwh', cmapping'', rwh'')$$

Contrary to `ALocking`, the rely conditions of `CLocking` are primarily necessary for the correctness of the component itself, more precisely, for guaranteeing linearizability. The $\texttt{accessed-rely}_{tid}$ is analogously to the one of ALocking, stating that no thread is unintentionally added to or removed from a set of accessing threads. The $\texttt{heap-rely}_{tid}$ copes with *rwh* not being protected directly by *mtx*: while the values of allocated references can be updated without acquiring *mtx*, the domain of *rwh* is only manipulated when *mtx* is held.

$$\begin{aligned}&\vdash \texttt{heap-rely}_{tid}(mtx', rwh', rwh'') \leftrightarrow\\ &\qquad mtx' = \texttt{locked}(tid) \rightarrow \texttt{dom}\ rwh' = \texttt{dom}\ rwh''\end{aligned}$$

This is crucial for allocating new rwlocks in **clocking_getlock#**. After a fresh reference r_0 is chosen, the current thread *tid* must rely on that other threads do not allocate r_0 before *tid* allocates it (allocating a reference twice is illegal) and that the reference is not again deallocated before *tid* initializes the location $rwh\texttt{[}r_0\texttt{]}$.

$\texttt{refs-rely}_{tid}$ is mainly relevant for the actual locking and unlocking operations (cf. Fig. 8.4), but it is also for client steps between retrieving a lock reference from **clocking_getlock#** and calls to these operations.

$$\begin{aligned}&\vdash \texttt{refs-rely}_{tid}(caccessed', cmapping', rwh', cmapping'', rwh'')\\ &\quad\leftrightarrow \forall\ ino.\ \big(\quad ino \in caccessed' \wedge tid \in caccessed'\texttt{[}ino\texttt{]}\\ &\qquad\qquad \rightarrow \quad cmapping'\texttt{[}ino\texttt{].ref} = cmapping''\texttt{[}ino\texttt{].ref}\\ &\qquad\qquad\quad \wedge\ (\quad cmapping'\texttt{[}ino\texttt{].ref} \in rwh'\\ &\qquad\qquad\qquad \leftrightarrow cmapping''\texttt{[}ino\texttt{].ref} \in rwh'')\big)\end{aligned}$$

It ensures that, as long as *tid* accesses *ino*, no other thread changes the assigned lock reference for *ino* or deallocates the respective reference. This implies that the accesses predicate, used in the preconditions of **clocking_rlock#** etc., is stable over rely steps so that, in particular, the access $rwh\texttt{[}r\texttt{]}$ stays valid while the thread is waiting for acquiring the lock.

The most specific rely is $\texttt{last-rely}_{tid}$, as it is tailored to one particular situation.

$$\begin{aligned}&\vdash \texttt{last-rely}_{tid}(mtx', caccessed', cmapping', rwh', cmapping'', rwh'')\\ &\quad\leftrightarrow \forall\ ino.\qquad mtx' = \texttt{locked}(tid)\\ &\qquad\qquad\qquad \wedge\ ino \in caccessed'\\ &\qquad\qquad\qquad \wedge\ caccessed'\texttt{[}ino\texttt{]} = \{tid\}\\ &\qquad\qquad \rightarrow\ rwh'\texttt{[}cmapping'\texttt{[}ino\texttt{].ref]} = rwh''\texttt{[}cmapping''\texttt{[}ino\texttt{].ref]}\end{aligned}$$

It guarantees that other threads do not modify the *value* of a heap location, i.e., acquire or release the rwlock stored under a particular reference, when *tid* has locked *mtx* and is the only one accessing the respective rwlock. This property encodes the idea that only *accessing* threads are allowed to acquire or release the corresponding reader/writer locks stored in the heap, but is reduced to the only case it is critical: a thread is the last thread accessing *ino* in **clocking_droplock#**. In this situation, the thread deallocates the reference in $cmapping\texttt{[}ino\texttt{]}$. However, this is only safe if no one has acquired the rwlock stored at that heap location, i.e., it is $\texttt{rlock}(\emptyset)$.

This follows indirectly from $\texttt{last-rely}_{tid}$: a thread *tid* must always consider that another thread tid_0 could be about to deallocate a lock *tid* wants to access. If tid_0 is the only thread in *caccessed*, *tid* must adhere to the rely of tid_0 and is not allowed to

```
clocking_getlock#(ino; ; r)
  interface
  precondition ...
{
ℝ  mutex_lock#(tid; mtx);
𝔹  if ¬ ino ∈ cmapping then {
𝔸     choose $r_0$ with
         $r_0 \neq$ null ∧ ¬ $r_0$ ∈ rwh
      in
𝔸        rwh := rwh ++ $r_0$;
𝔹        cmapping[ino] := lentry($r_0$, 1);
𝔸        rwlock_init#(; ; rwh[$r_0$]);
𝔹        r := $r_0$;
   } else {
𝔹     let refcnt = cmapping[ino] in
𝔹     let cnt = refcnt.cnt in
𝔹        cmapping[ino].cnt := cnt + 1;
𝔹        r := refcnt.ref;
   };
𝔹  clocking_addaccess#(ino);
𝕃  mutex_unlock#(tid; mtx);
}
```

```
clocking_getlock#(ino; ; r)
  interface
  precondition ...
{
ℝ  atomic (mtx = free) {
      mutex_lock#(tid; mtx);
   };
𝔹  if ¬ ino ∈ cmapping then {
𝔸     choose $r_0$ with
         $r_0 \neq$ null ∧ ¬ $r_0$ ∈ rwh
      in
𝔸        rwh := rwh ++ $r_0$;
𝔸        atomic (true) {
            cmapping[ino] := lentry($r_0$, 1);
            rwlock_init#(; ; rwh[$r_0$]);
            r := $r_0$;
         };
   } else {
𝔹     atomic (true) {
         let refcnt = cmapping[ino] in
         let cnt = refcnt.cnt in
            cmapping[ino].cnt := cnt + 1;
            r := refcnt.ref;
      };
   };
𝕃  atomic (true) {
      clocking_addaccess#(ino);
      mutex_unlock#(tid; mtx);
   }
}
```

Figure 8.6: First reduction of the **clocking_getlock#** operation.

modify (acquire or release) the lock in question. Thus, the only way to circumvent this restriction for *tid* is to register itself in *caccessed* as well.

Recall that, besides these custom relies, the ownership annotations for *cmapping* and *caccessed* also add generic rely conditions ensuring that both maps are not modified via rely steps if the current thread holds the mutex *mtx*.

Given this concurrent specification of `CLocking`, we will look at the atomicity refinement from $\texttt{CLocking}_{\texttt{Iv}}$ to $\texttt{CLocking}_{\texttt{At}}$ in the following.

Atomicity Refinement of Lock-Management

Based on the ownership annotations of `CLocking`, mover properties for large part of the component's atoms are inferred directly. All atoms that only access local state or the owned maps *cmapping* and *caccessed* are automatically both-movers $\mathbb{B}$ (the required assertions are proven in the rely-guarantee proofs of `CLocking`). Furthermore, all **mutex_lock#** and **mutex_unlock#** calls are right- and left-movers, respectively. What remains are statements accessing the heap *rwh*, which are initially non-movers $\mathbb{A}$.

For most of the operations, this is already sufficient to reduce their bodies to atomic blocks. The locking and unlocking operations only consist of a single call to a rwlock operation that is already atomic by its declaration (cf. Fig. 7.7). Hence, this atomic block can be lifted outside of the call. **clocking_droplock#** has more steps, however, it has only one non-moving step, namely the heap modification $rwh := rwh$ `--` $refcnt$`.ref`. The remaining statements (resp., their atoms) are all both-movers, except for the surrounding mutex calls, which move inward. This directly complies to Def. 46, splitting the atom sequence at the heap modification (or at an arbitrary point for the atom sequence that does not include the assignment to the heap).

For **clocking_getlock#**, this does not apply. As there are multiple accesses to the heap, the atom sequences of the operation contain several non-moving atoms. Fig. 8.6 on the left shows what mover-properties can be inferred initially. One can see that the atom sequence of the path where the **if**-condition evaluates to **ff**, i.e., when $ino \in cmapping$ is **tt**, is already reducible. For the other path, the sequence contains three non-moving atoms (cf. Def. 45)[1].

1. $\{r_0 :\in r_0 \neq \texttt{null} \ \wedge \ \neg\ r_0 \in rwh\} : \mathbb{A}$
2. $\{rwh := rwh \texttt{ ++ } r_0\} : \mathbb{A}$
3. $\{\textbf{atomic}\ (\texttt{true})\ \{\textbf{rwlock\_init\#}(;;rwh\texttt{[}r_0\texttt{]})\}\} : \mathbb{A}$

A first reduction step produces then the reduced body in Fig. 8.6 on the right. For a complete reduction of the operation, it is sufficient to show that atoms 1 and 2 move to the right, i.e., we apply the calculus rule ℝ-Mover of Fig. 7.5 to prove

1. $\{r_0 :\in r_0 \neq \texttt{null} \ \wedge \ \neg\ r_0 \in rwh\} : \mathbb{R}$
2. $\{rwh := rwh \texttt{ ++ } r_0\} : \mathbb{R}$

Since both atoms only access rwh besides local variables, commutations only need to be shown for atoms accessing rwh as well (all other atoms commute trivially due to program variables being disjoint). Commutations with atoms of **clocking_getlock#** or **clocking_droplock#** follow from holding the mutex mtx at all relevant atoms, assuming that respective assertions are given ($\texttt{locked}(tid)$ is a contradiction to $\texttt{locked}(tid_0)$ for $tid \neq tid_0$).

It remains to show the commutations of the atoms 1 and 2 with the rwlock operations called in Fig. 8.3 (and the according writer-lock operations). Atom 1 commutes to the right because locking/unlocking does not affect the domain of rwh, and thus the same reference r_0 can be chosen. Atom 2 commutes to the right because r_0 is not allocated in rwh while the argument r of the locking/unlocking operation has to be a valid reference in rwh, and heap allocations commute with heap updates if they target different references.

With these mover properties proven, the body of **clocking_getlock#** reduces to one atomic block with top-level guard $mtx = \texttt{free}$ (lifted from the leading **mutex_lock#** call). At this level of atomicity, the guard can be established as a new invariant, facilitating its removal via Thm. 11. Thus, the whole component is atomic, i.e., $\texttt{CLocking}_{\texttt{Iv}} \leq \texttt{CLocking}_{\texttt{At}}$ holds.

[1]The body of **rwlock_init#** is atomic, which is why the atomic block is lifted outside of the call to build the atom.

Finally, the data refinement of the atomic components can be shown by an abstraction relation that maps the state of CLocking directly to the state of ALocking.

$$\begin{array}{ll} \textbf{data refinement } \texttt{CLocking}_{\texttt{At}} \leq \texttt{ALocking}_{\texttt{At}} & \\ \textbf{abstraction relation} & \ \mathit{mapping} = \mathit{cmapping}.\texttt{to-fct} \\ & \wedge\ \mathit{rwlocks} = \mathit{rwh}.\texttt{to-fct} \\ & \wedge\ \mathit{accessed} = \mathit{caccessed}.\texttt{to-fct} \end{array}$$

The algebraic functions _.to-fct construct total functions uniformly from the respective mappings. Depending on whether key is allocated in the concrete map/heap, the created functions map to the store values or to the default values used in ALocking, e.g., *rwh*.to-fct is defined as

$$\mathit{rwh}.\texttt{to-fct} = \lambda\ r.\ (r \in \mathit{rwh} \supset \mathit{rwh}\texttt{[}r\texttt{]}\ ;\ \texttt{rlock}(\emptyset))$$

Using this abstraction, the data refinement proofs are straightforward. Together with the atomicity refinement from above, the concurrent implementation $\texttt{CLocking}_{\texttt{Iv}}$ is proven to be correct w.r.t. the atomic specification $\texttt{ALocking}_{\texttt{At}}$.

Theorem 12 (Correctness of Concurrent Lock Management)**.** *The concurrent component* CLocking *is a correct implementation of the atomic specification* ALocking*, i.e.,* $\texttt{CLocking}_{\texttt{Iv}} \leq \texttt{ALocking}_{\texttt{At}}$ *holds.*

□

The presented implementation of CLocking overcomes the potential problem described at the beginning of the section. Used as a subcomponent, clients obtain references to locks (via **alocking_getlock#**) and then pass these references to the respective locking operations. To achieve atomicity, the implementation of the former operation must use an exclusive locked (a mutex), but this mutex is held only very briefly by each thread. On the other hand, the actual locking operations of CLocking do not require holding the mutex and thus do not block other threads accessing other locks.

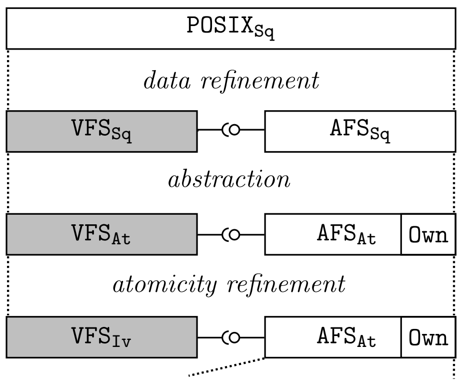
Note that the concrete type ino used to identify individual locks is not relevant to the implemented mechanism. Thus, the components presented in this section can be realized for any other desired type as well. Nevertheless, the components defined with inode numbers ino are used in the concurrent version of VFS for the locking files, as shown in the following section.

8.2 A Concurrent Virtual File System Switch

When allowing concurrent calls to the POSIX interface, the sequential implementation of VFS as given in Sec. 4.3 can no longer guarantee consistency of the file system as concurrent operations could arbitrarily interleave, causing data races. Hence, the existing implementation must be adjusted to avoid such data races. In the case of Flashix, this is done by extending implementation components with locking.

In order to prove that the extended VFS component is correct with respect to its specification POSIX, we apply the methodology presented in Chapter 7, resulting in the refinement hierarchy shown in Fig. 8.7. The concurrent version $\texttt{VFS}_{\texttt{Iv}}$ of VFS still uses `AFS` as a subcomponent, but now extended by *ownership* state and operations enforcing certain concurrency restrictions to client components (denoted by `Own` in the figure, cf. Sec. 7.4). $\texttt{VFS}_{\texttt{Iv}}$ assumes that the specification `AFS` has atomic behavior (denoted by $\texttt{AFS}_{\texttt{At}}$). The implementation of $\texttt{AFS}_{\texttt{At}}$ is responsible for guaranteeing this atomicity, which is usually achieved by using locks as well. Atomicity refinement (see Sec. 7.3) is applied to prove strong linearizability of the concurrent implementation $\texttt{VFS}_{\texttt{Iv}}$ and to reduce it to an atomic version $\texttt{VFS}_{\texttt{At}}$.

When following the pattern of Fig. 7.3, the atomic implementation would be abstracted to an atomic specification using data refinement. This specification is typically also extended by ownership to specify how clients can use it concurrently. However, `VFS` implements the top-level specification `POSIX` of the file system. Thus, ownership extensions to the original sequential version of `POSIX` are not necessary since the file system cannot enforce any concurrency restrictions on its users. The operations of the `POSIX` interface can be called in arbitrary concurrent fashion (we call concurrency induced this way *external concurrency*), and the file system implementation has to restrict the concurrency internally to avoid executions harming the consistency of the system. As a result, the atomic implementation $\texttt{VFS}_{\texttt{At}}(\texttt{AFS}_{\texttt{At}})$ can be abstracted to its original sequential $\texttt{VFS}_{\texttt{Sq}}(\texttt{AFS}_{\texttt{Sq}})$. This abstraction can be done by a trivial data refinement that drops all locking- and ownership-related state variables. Therefore, the original data refinement $\texttt{VFS}_{\texttt{Sq}}(\texttt{AFS}_{\texttt{Sq}}) \leq \texttt{POSIX}_{\texttt{Sq}}$ is entirely unaffected and must not be reproved.



$\texttt{POSIX}_{\texttt{Sq}}$
data refinement
$\texttt{VFS}_{\texttt{Sq}}$
$\texttt{AFS}_{\texttt{Sq}}$
abstraction
$\texttt{VFS}_{\texttt{At}}$
$\texttt{AFS}_{\texttt{At}}$
Own
atomicity refinement
$\texttt{VFS}_{\texttt{Iv}}$
$\texttt{AFS}_{\texttt{At}}$
Own

Figure 8.7: Refinement hierarchy of the concurrent `VFS`.

The remainder of this section gives an overview on how the components $\texttt{VFS}_{\texttt{Sq}}$ and $\texttt{AFS}_{\texttt{Sq}}$ from Sec. 4.3 were extended to $\texttt{VFS}_{\texttt{Iv}}$ and $\texttt{AFS}_{\texttt{At}}$, respectively. For the most part, atomicity refinement of `VFS` is proved analogously to the refinement of `CLocking` described in detail in the previous section, so only some specifics are emphasized in this section.

Specifying Ownership of `AFS`

The basic idea of the locking strategy of `VFS` is to sequentialize *structural operations*, i.e., restrict manipulations of the file system tree to one at a time, but to allow execution of multiple *content operations* in parallel, as long as a file is not manipulated concurrently. This corresponds to using a single lock for the whole tree while using individual locks for files. Hence, respective ownership (ghost) state and annotations are added to `AFS`.

```
atomic component AFS
thread id  tid : threadid
state  dirs : iheap(dir),   files : iheap(file)
```

ghost state $\mathit{odirs} : \mathsf{owner}$, $\mathit{ofdom} : \mathsf{owner}$, $\mathit{ofiles} : \mathsf{map(ino, owner)}$
ownership
dirs **owned by** odirs
files`[`ino`].meta` **owned by** odirs
files`[`ino`].nlink` **owned by** odirs
files`[`ino`].content` **owned by** ofiles`[`ino`]`
files`[`ino`].size` **owned by** ofiles`[`ino`]`

The directory heap *dirs* is owned completely by the owner *odirs*, which leads to the generation of assertions and relies as explained in Sec. 7.4. But *odirs* also specifies ownership for the *structural* fields of *files*, e.g., the field `nlink` that stores the number of hard links for a file (given by ownership annotations for the access form *files*`[`*ino*`].nlink`). On the other hand, the non-structural fields of *files*, i.e., `content` and `size`, of a file *ino* are owned by a corresponding entry in the owner map *ofiles*.

Defining ownership for entries of partial mappings (heaps or maps) generates a slightly different version of the generic ownership rely. The ownership annotations for *files* produce a rely $\texttt{files-rely}_{tid}$ of the form

$$\begin{aligned}
\vdash\ & \texttt{files-rely}_{tid}(\mathit{ofiles}', \mathit{ofiles}'', \mathit{files}', \mathit{files}'') \leftrightarrow \\
& \forall\, \mathit{ino}.\quad \big(\texttt{valid-reader}(\mathit{tid}, \mathit{ino}, \mathit{ofiles}') \leftrightarrow \texttt{valid-reader}(\mathit{tid}, \mathit{ino}, \mathit{ofiles}'')\big) \\
& \qquad\quad \wedge\ \big(\texttt{valid-writer}(\mathit{tid}, \mathit{ino}, \mathit{ofiles}') \leftrightarrow \texttt{valid-writer}(\mathit{tid}, \mathit{ino}, \mathit{ofiles}'')\big) \\
& \qquad\quad \wedge\ \big(\mathit{ino} \in \mathit{files}' \wedge \texttt{valid-reader}(\mathit{tid}, \mathit{ino}, \mathit{ofiles}') \rightarrow \\
& \qquad\qquad\qquad \mathit{ino} \in \mathit{files}'' \wedge\ \mathit{files}'\texttt{[}\mathit{ino}\texttt{].content} = \mathit{files}''\texttt{[}\mathit{ino}\texttt{].content} \\
& \qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{files}'\texttt{[}\mathit{ino}\texttt{].size} = \mathit{files}''\texttt{[}\mathit{ino}\texttt{].size}\big)
\end{aligned}$$

The generated rely universally quantifies over all keys of the mappings and also takes their domains into account. The idea is that the `content` and `size` of a file *files*`[`*ino*`]` is protected, i.e., is not changed over the rely step, when the thread *tid* is (at least) a reader of the corresponding entry in *ofiles*`[`*ino*`]`. Since *ofiles* is a partial mapping, the predicates `valid-reader` and `valid-writer` additionally ensure that *ino* is allocated in *ofiles*.

$$\begin{aligned}
&\vdash \texttt{valid-reader}(\mathit{tid}, \mathit{ino}, \mathit{ofiles}) \leftrightarrow \mathit{ino} \in \mathit{ofiles} \wedge \mathit{ofiles}\texttt{[}\mathit{ino}\texttt{].r?}(\mathit{tid}) \\
&\vdash \texttt{valid-writer}(\mathit{tid}, \mathit{ino}, \mathit{ofiles}) \leftrightarrow \mathit{ino} \in \mathit{ofiles} \wedge \mathit{ofiles}\texttt{[}\mathit{ino}\texttt{]} = \texttt{writer}(\mathit{tid})
\end{aligned}$$

Note that $\texttt{files-rely}_{tid}$ rely also implies that an entry *ino* is not removed (neither from *files* nor from *ofiles*) as long as *tid* has ownership of *ino* before the rely step. Of course, this ownership is also propagated via the rely.

Ownership is acquired by client components of $\texttt{AFS}_{\texttt{At}}$ via additional ghost interface operations. For example, Fig. 8.8 on the right shows the respective operations of acquiring and releasing ownership of a file *ino*. The regular interface operations are extended with ownership preconditions depending on the way they access state. As shown in Fig. 8.8 on the left, content operations like **afs_writepage#** or **afs_readpage#** force callers to be owners of the read or written file, while structural operations like **afs_lookup#** of **afs_create#** require ownership of the whole tree.

```
afs_writepage#(inode, ...)
  interface
  precondition ...
    ∧ valid-writer(tid, inode.ino, ofiles)
{ ... }

afs_readpage#(inode, ...)
  interface
  precondition ...
    ∧ valid-reader(tid, inode.ino, ofiles)
{ ... }

afs_lookup#... )
  precondition ...
    ∧ odirs.r?(tid)
{ ... }

afs_create#( ... )
  precondition ...
    ∧ odirs.w?(tid) ∧  ofdom.w?(tid)
{ ... }
```

```
afs_wacquirefile#(ino)
  ghost interface
{
  if ino ∈ ofiles ∧ ofiles[ino] = readers(∅)
  then ofiles[ino] := writer(tid);
}

afs_racquirefile#(ino)
  ghost interface
{
  if ino ∈ ofiles ∧ ¬ ofiles[ino].owned?
  then
    ofiles[ino] := rlock(ofiles[ino], tid);
}

afs_releasefile#()
  ghost interface
{
  if ino ∈ ofiles ∧ ofiles[ino].r?(tid)
  then
    ofiles[ino] := unlock(ofiles[ino], tid);
}
```

Figure 8.8: Adjustments to AFS for the use in the concurrent VFS component.

Special care has to be taken for the creation and deletion of files (performed by the operations **afs_create#** and **afs_evict#**). As also shown in Fig. 8.8 for create, these operations require an additional ownership of *ofdom* (for other operations, *ofdom* is not relevant). This requirement prevents both operations from being executed concurrently, which is necessary for proving linearizability (creation and deletion of files do not commute with each other). Hence, ownership of *ofdom* ensures that the domain of *files* is not changed by other threads, given as custom rely condition $\texttt{dom-rely}_{tid}$.

$$\begin{aligned}\vdash \texttt{dom-rely}_{tid}(ofdom', ofdom'', files', files'') \leftrightarrow \\ &(ofdom'.\texttt{r?}(tid) \rightarrow \texttt{dom}\ files' = \texttt{dom}\ files'') \\ \wedge\ &(ofdom'.\texttt{r?}(tid) \leftrightarrow ofdom''.\texttt{r?}(tid)) \\ \wedge\ &(ofdom' = \texttt{writer}(tid) \leftrightarrow ofdom'' = \texttt{writer}(tid))\end{aligned}$$

Furthermore, an additional invariant ensures equality of the domains of *ofiles* and *files* to ensure that each entry of *files* has a corresponding owner in *ofiles* (naturally, the original AFS invariants still hold as well).

invariants $\texttt{dom}\ files = \texttt{dom}\ ofiles$
rely condition $\texttt{dom-rely}_{tid}(ofdom', ofdom'', files', files'')$

Linearizable Locking in `VFS`

Based on the extended $\texttt{AFS}_{\texttt{At}}$, $\texttt{VFS}_{\texttt{Iv}}$ implements the desired locking strategy using multiple locks and the abstract locking management component `ALocking` introduced in Sec. 8.1.

> **concurrent component** `VFS` **using** `AFS`, `ALocking`
> **thread id** *tid* : threadid
> **state** *of* : map(nat, fhandle), *maxfd* : nat
> *rwldirs* : rwlock, *rwlfdom* : rwlock, *rwlvfs* : rwlock
> *oflocks* : map(nat, mutex)
> **ownership**
> *maxfd* **owned by** *rwlvfs*`.owner`
> *of*`[`*fd*`]` **owned by** *oflocks*`[`*fd*`].owner`

The reader/writer locks *rwldirs* and *rwlfdom* are used for acquiring the respective ownership *odirs* and *ofdom* of $\texttt{AFS}_{\texttt{At}}$. Ownership for individual files is acquired by the reader/writer locks *rwlocks* outsourced to `ALocking`. For protecting the RAM data structures *maxfd* and *of* of $\texttt{VFS}_{\texttt{Iv}}$, *rwlvfs* and *oflocks* are used. The map *oflocks* stores a mutex for each file descriptor *fd* currently existing, and *rwlvfs* protects the file descriptor counter *maxfd*.

It is crucial for a linearizable and deadlock-free implementation that the locks are used in the same order in each operation. Consider for example, the locking scheme outlined in Fig. 8.9 for the **vfs_create#** and **vfs_write#** operations, representing both structural and content operations.

Structural operations mainly have to acquire ownership of the file system tree only. They usually acquire *rwldirs* at the beginning of the operation, traverse the tree via **vfs_walk#**, perform the actual operation with the respective $\texttt{AFS}_{\texttt{At}}$ call, and then release *rwldirs* at the end of the operation. To meet the ownership preconditions of $\texttt{AFS}_{\texttt{At}}$, ownership is acquired via the respective ghost interface operations of $\texttt{AFS}_{\texttt{At}}$ directly after the corresponding lock was acquired. For *rwldirs*, this is done by calling **afs_wacquiredirs#** as shown in Fig. 8.9 on the left. This call is always successful, i.e., *tid* is guaranteed to be an exclusive `writer` afterward, since *rwldirs* is the only lock that can acquire *odirs* ownership. Formally, this is established by giving a respective invariant (and similarly, for the other lock-ownership relations).

invariants $\texttt{dom}\ \mathit{of} \subseteq \texttt{dom}\ \mathit{oflocks}$
$\land\ \mathit{odirs} \subseteq_{\texttt{own}} \mathit{rwldirs}\ \land\ \mathit{ofdom} \subseteq_{\texttt{own}} \mathit{rwlfdom}$
$\land\ \mathit{ofiles} \subseteq_{\texttt{own}} (\mathit{mapping}, \mathit{rwlocks})$

The predicate $o \subseteq_{\texttt{own}} \mathit{rwl}$ abbreviates $o \subseteq \mathit{rwl}$`.owner`, which states that no thread has more ownership in o than in the derived owner of the lock *rwl*. This corresponds to acquiring ownership only within the respective locking range. For *ofiles*, this

```
vfs_create#(p, md, user; ; err)
  interface
{
  ...
  rwlock_wlock#(tid; rwldirs);
  afs_wacquiredirs#(); // ghost
  vfs_walk#( ... );
  ...
  rwlock_wlock#(tid; rwlfdom);
  afs_wacquirefdom#(); // ghost
  afs_create#( ... );
  afs_releasefdom#(); // ghost
  rwlock_wunlock#(tid; rwlfdom);
  ...
  afs_releasedirs#(); // ghost
  rwlock_wunlock#(tid; rwldirs);
  ...
}
```

```
vfs_write#(fd, user, buf; n; err)
  interface
  precondition  n ≤ # buf
{
  ...
  mutex_lock#(tid; oflocks[fd]);
  let ino = oflocks[fd].ino in {
    alocking_getlock#(ino; ; r);
    alocking_wlock#(r);
    afs_wacquirefile#(ino); // ghost
    ...
    vfs_writeloop#(...);
    ...
    afs_releasefile#(ino); // ghost
    alocking_wunlock#(r);
    alocking_droplock#(ino);
    ...
    of[fd].pos := start + n;
  };
  mutex_unlock#(tid; oflocks[fd]);
  ...
}
```

Figure 8.9: Locking in the concurrent `VFS` component.

property is lifted to the state of `ALocking`.

$$\begin{aligned}\vdash\ & \mathit{ofiles} \subseteq_{\texttt{own}} (\mathit{mapping}, \mathit{rwlocks}) \leftrightarrow \\ & \quad \forall\ \mathit{ino}.\ \mathit{ino} \in \mathit{ofiles}\ \wedge\ \mathit{ofiles}[\mathit{ino}] \neq \texttt{readers}(\emptyset) \rightarrow \\ & \qquad\qquad \big(\quad \mathit{mapping}(\mathit{ino}).\texttt{ref} \neq \texttt{null} \\ & \qquad\qquad\ \wedge\ \mathit{ofiles}[\mathit{ino}] \subseteq_{\texttt{own}} \mathit{rwlocks}(\mathit{mapping}(\mathit{ino}).\texttt{ref})\big)\end{aligned}$$

An additional invariant ($\texttt{dom}\ \mathit{of} \subseteq \texttt{dom}\ \mathit{oflocks}$) ensures that existing file descriptors can always be locked. This is achieved by creating the respective lock for a file descriptor when opening a file (before the entry is added to *of*) and deleting the lock not until the descriptor is removed from *of* when closing of the file. Note that the **vfs_create#** operation also has to acquire the *rwlfdom* lock and the associated *ofdom* ownership for the **afs_create#** call.

Content operations that are called with a file descriptor *fd* as input, such as **vfs_write#** on the right of Fig. 8.9, do not need to lock the directory tree. Instead, they lock the file descriptor for retrieving the inode number *ino* (and the current position) of the targeted file. The actual file is then locked by requesting a lock reference from `ALocking` via **alocking_getlock#**, and then acquiring the the lock with **alocking_wlock#**. Afterward, ownership of the file is acquired in $\texttt{AFS}_{\texttt{At}}$, which enables the following $\texttt{AFS}_{\texttt{At}}$ calls (**afs_writebegin#**, **afs_writepage#**, and **afs_writesize#**) triggered by the write operation. At the end of the actual writing, ownership and lock are released before also dropping the reference to the lock. Finally, the file descriptor mutex is released after the position has been updated.

By using a strict *locking hierarchy*, the reduction of the concurrent component is straightforward. Locking ranges are contracted to atomic blocks incrementally, starting from the innermost locking range towards the outermost one. At each iteration, the guard of the next outer locking range can be established as a new concurrent invariant and thus eliminated (see Thm. 11). Deadlock-freedom follows from this locking scheme as well: for hierarchical employed locks, one thread on the lowest acquired *locking level* can always progress as it either does not need to acquire a further lock or is the first one entering the next locking level.

Together with the original refinement $\mathtt{VFS}_{\mathtt{Sq}}(\mathtt{AFS}_{\mathtt{Sq}}) \leq \mathtt{POSIX}_{\mathtt{Sq}}$ and the trivial abstraction $\mathtt{VFS}_{\mathtt{At}}(\mathtt{AFS}_{\mathtt{At}}) \leq \mathtt{VFS}_{\mathtt{Sq}}(\mathtt{AFS}_{\mathtt{Sq}})$, the reduction proofs establish correctness of the concurrent `VFS` implementation.´

Theorem 13 (Correctness of the Concurrent Virtual File System Switch)**.** *The concurrent implementation of* `VFS` *is a correct, deadlock-free implementation of the sequential* `POSIX` *specification, i.e.,* $\mathtt{VFS}_{\mathtt{Iv}}(\mathtt{AFS}_{\mathtt{At}}) \leq \mathtt{POSIX}_{\mathtt{Sq}}$ *holds.* □

The implementation presented in this section is linearizable to the sequential `POSIX` specification, and it achieves the goal of allowing concurrent writes to different files and arbitrary concurrent reads.

There are still some minor optimizations possible. For example, the implementation `Cache` of `AFS` is currently implemented sequentially. Hence, concurrent calls to $\mathtt{AFS}_{\mathtt{At}}$ are sequentialized by a global mutex surrounding the interface of `Cache`, but a concurrent component that locks each sub-cache (`ICache`, `DCache`, `PCache`, and `TCache`, see Sec. 6.1) exclusively should be unproblematic to realize. Access to the maps used within the `Cache` layer is then still sequential, but we think this would only have a marginal impact on performance. Nevertheless, it could be investigated how thread-safe implementations of maps can be integrated into our approach.

Another potential optimization could be to implement a more fine grained locking strategy for the directory tree, e.g., by adapting the approach of Zou et al. [119]. This would allow for more concurrent path traversals as a structural operation would not have to lock the whole tree exclusively. While this change would certainly bring some performance improvements, we expect them to be negligible in everyday workloads since the performance-critical tasks usually access file contents. Furthermore, the locking strategy in [119] is not compatible with the linearizability criterion if multiple hard links to files are supported (which is the case in Flashix), as Lehner found out in his master thesis [79].

Concurrency & Write-Prefix Crash Consistency

In Ch. 6 we introduced the crash-safety criterion *Write-Prefix Crash Consistency* that copes with non-order-preserving caches like the ones implemented in the `Cache` layer of Flashix, and we proved that Flashix adheres to this criterion. However, the proofs only considered purely sequential runs of `POSIX`. When concurrent `POSIX` calls are allowed, the `VFS` implementation must ensure that concurrent executions do not invalidate WPCC. We argue that the concurrent implementation $\mathtt{VFS}_{\mathtt{Iv}}$ presented in this section achieves this.

Concurrent `POSIX` calls yield runs of the file system with well-formed concurrent histories. At first, these histories may contain arbitrary interleaved operations, particularly multiple content operations to the same file *fid*. Such interleavings would be problematic for WPCC because they would enable a **vfs_write#** to "overtake" a **vfs_fsync#** or vise versa (which could result in mixed file contents after a crash, even if it occurs outside of **vfs_fsync#**). However, $\mathtt{VFS_{Iv}}$ prohibits such situations by using a reader/writer lock per file set at the start of all content operations (via the `ALocking` component, e.g., see Fig. 8.9 on the right). As a consequence, all histories $h_i|_{fid}$ are sequential, i.e., an interleaving of the form $inv_{tid}(\mathtt{op}, fid, \ldots)\ inv_{tid'}(\mathtt{op'}, fid, \ldots)\ res_{tid}(\mathtt{op}, \ldots)\ res_{tid'}(\mathtt{op'}, \ldots)$ is not possible, and thus WPCC as defined in Def. 39 is still applicable.

Theorem 14 (Compatibility of WPCC and with the Concurrent Virtual File System Switch)**.** *Write-Prefix Crash Consistency is compatible with the concurrent implementation* $\mathtt{VFS_{Iv}}$ *of the Virtual File System Switch.*

□

8.3 Concurrent Garbage Collection

Besides allowing concurrent calls to the file system interface as outlined in Sec. 8.2, moving certain internal mechanisms into separate threads also introduces additional concurrency to the file system (we call this *internal concurrency*). Hence, the affected models also have to be modified in order to avoid conflicts resulting from parallel executions of operations.

Internal concurrency was first introduced for the *wear leveling* algorithm in the Erase Block Manager layer (`EBM`, cf. Sec. 4.1) by Pfähler [96]. Moving *garbage collection* to a separate thread is the natural continuation of this work. Thus, the methodology used in [96] is also used for the extension presented in this section. We will not go into technical details (see sections 8.1 and 8.2 for insights into the concrete specification and proof work) but will only outline the high-level approach in the following.

The extension of Flashix with concurrent garbage collection ranges from the `FFS` layer to the `Journal` layer. In the `FFS`, the concurrent operation for garbage collection is introduced (Fig. 8.11) as an *internal operation.* Since it is not part of the interface (it refines **skip**, i.e., it has no visible effect for clients), it cannot be called by any client components. Instead it will be repeated infinitely within its own thread. To ensure that garbage collection is not performed continuously, especially when no more space can be regained, a condition variable *gccond* is used [2]. At the beginning of each iteration, the thread blocks at the **condition_wait#** call until it is signaled by another thread to start. The concrete garbage collection algorithm is specified in the `FFS-Core` and implemented in the `Journal` component, so after being signaled the operation **ffs_core_gc#** is called.

Signaling takes place in all `FFS` operations that may modify the file system state in the sense that either entries are written to the log (and thus space on the flash device is allocated) or garbage is introduced by invalidating allocated space. Such operations, as shown generically in Fig. 8.10, emit a signal to *gccond* after they have updated the `Index`.

[2]Note that condition variables are always coupled to a `mutex`. Here *gccond* is coupled to *gcmtx*. Signaling a condition requires to hold the corresponding mutex. Starting to wait for a signal requires to hold the mutex as well, however, the mutex is released during waiting. As soon as a signal was emitted and the mutex is `free`, the waiting thread acquires the mutex and continues its execution.

```
ffs_operation#(...)
  interface
{
  ...
  $nd_1$ := inodenode($key_1$, ...);
  ...
  rwlock_wlock#( ; corelock);
  ffs_core_wacquire#(); // ghost
  ffs_core_jadd#($nd_1$, ... ; $adr_1$, ... ; err);
  if err = ESUCCESS then {
    ffs_core_istore#($key_1$; $adr_1$);
    ...
    mutex_lock#( ; gcmtx);
    condition_signal#( ; gccond, gcmtx);
    mutex_unlock#( ; gcmtx);
  };
  ffs_core_release#(); // ghost
  rwlock_wunlock#( ; corelock);
  ...
}
```

Figure 8.10: General operation scheme of modifying FFS operations.

```
ffs_gc#()
  internal
{
  mutex_lock#( ; gcmtx);
  condition_wait#( ; gccond, gcmtx);
  mutex_unlock#( ; gcmtx);

  rwlock_wlock#( ; corelock);
  ffs_core_wacquire#(); // ghost
  ffs_core_gc#();
  ffs_core_release#(); // ghost
  rwlock_wunlock#( ; corelock);
}
```

Figure 8.11: Internal garbage collection operation of FFS.

The implementation of **ffs_core_gc#** in the Journal component then first checks whether there is a block which is suitable for garbage collection. If that is the case, all still referenced nodes of this block are collected, these nodes are then written to the journal, and their new addresses are updated in the Index accordingly. Finally, if the referenced data was successfully copied, the block can be marked for *erasure*.

As an additional thread is introduced in the FFS, established ownerships in the VFS/AFS layer (see Sec. 8.2) are not sufficient to prevent data races between the garbage collection thread and other threads. For this reason the reader/writer lock *corelock* is added to the FFS component. It is used to acquire exclusive or shared ownership for the Journal and Index data structures (both are abstracted together as FFS-Core). We did not head for a more fine-grained locking approach since updates usually affect nearly all parts of the state of FFS-Core anyway. However, using reader/writer locks still allows for concurrent read accesses to the Journal.

Modifying operations in the FFS as shown in Fig. 8.10 always follow the same scheme. First, all new or updated data objects are wrapped into nodes ($nd_1, ...$) with an unique key ($key_1, ...$). Depending on the concrete operation, nodes for inodes, dentries, and pages are created (cf. Sec. 4.3). These nodes are then grouped into transactions and appended to the log using the **ffs_core_jadd#** operation. If successful, i.e., the operation returns the error code ESUCCESS, the operation returns the addresses ($adr_1, ...$) where the passed nodes have been written to. Finally, the Index is updated by storing the new addresses of the affected keys via the operation **ffs_core_istore#** [3]. It is crucial that garbage collection is never performed between these calls since this could result in a loss of updates (e.g., when garbage collection moves nodes updated by the operation), potentially yielding an

[3]Some operations, e.g., truncations, also update the index by removing entries from it.

inconsistent file system state. Hence, the locking range of *corelock* must include the **ffs_core_jadd#** as well as all **ffs_core_istore#** calls.

To prove that this locking strategy is in fact correct, i.e., that the interleaved components are linearizable, we again apply *atomicity refinement* (see Sec. 7.3). This results in the expansion of the refinement hierarchy shown in Fig. 8.12. Atomicity refinement could be applied to `Index` and the layers below, too. However, we have not yet implemented a concurrent version of the $\texttt{B}^{+}\texttt{-Tree}$ ([75] presents a verification for concurrent B+ trees using Iris/Coq [10]), as we do not expect noticeable efficiency gains from doing so. So instead, we locked the interface of the `Index` (depicted by 🔒). This means that each call to an `Index` operations **index_operation#** requires the current thread to be an exclusive owner of the `Index` component. In the `Journal`, this is realized by surrounding these calls with a mutex *idxlock* as shown in Fig. 8.13 on the left. Owning a subcomponent exclusively ensures that the subcomponent is only called sequentially and hence allows to directly use the unaltered sequential version of the subcomponent and its refinements (denoted by the subscript $_{\texttt{Sq}}$ in Fig. 8.12).

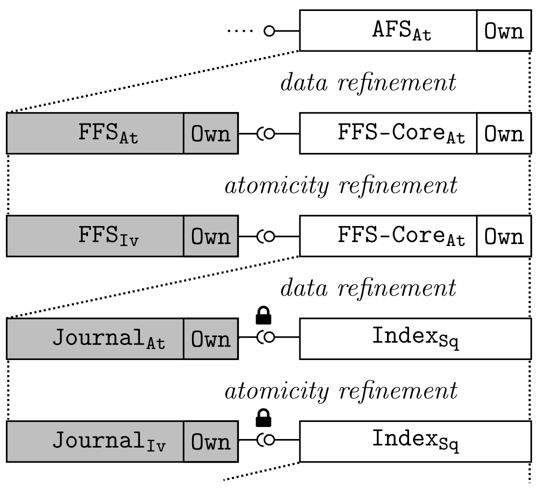

....
$\texttt{AFS}_{\texttt{At}}$
Own
data refinement
$\texttt{FFS}_{\texttt{At}}$
Own
$\texttt{FFS-Core}_{\texttt{At}}$
Own
atomicity refinement
$\texttt{FFS}_{\texttt{Iv}}$
Own
$\texttt{FFS-Core}_{\texttt{At}}$
Own
data refinement
$\texttt{Journal}_{\texttt{At}}$
Own
$\texttt{Index}_{\texttt{Sq}}$
atomicity refinement
$\texttt{Journal}_{\texttt{Iv}}$
Own
$\texttt{Index}_{\texttt{Sq}}$

Figure 8.12: Refinement hierarchy extended by concurrent garbage collection.

`FFS-Core` is augmented with ownership ghost state matching the reader/ writer lock *corelock* of `FFS`. The `FFS` operations acquire and release this ownership according to the locking ranges (see Fig. 8.10 and Fig. 8.11). While the ownership granularity of $\texttt{AFS}_{\texttt{At}}$ (owned directory tree *odirs*, owned files *ofiles*, ...) does not match the state of the `FFS-Core` or the `Journal`, the information about which files etc. are owned when an operation is called (encoded in the preconditions) is still relevant for the `FFS` in order to preserve functional correctness. For example, an owned file must not be removed from the `Index` while its metadata is updated. Therefore, ownership ghost state is added to `FFS` analogously to $\texttt{AFS}_{\texttt{At}}$ and corresponding ownership properties are established. This is sufficient to prove that the interleaved $\texttt{FFS}_{\texttt{Iv}}$ can be reduced to an atomic $\texttt{FFS}_{\texttt{At}}$ via atomicity refinement. The data refinement of the atomic $\texttt{AFS}_{\texttt{At}}$ to the atomic $\texttt{FFS}_{\texttt{At}}$ is basically identical to the original sequential refinement, in addition, it must only be shown that their respective ownerships match.

When proving the atomicity refinement of the `Journal`, it is apparent that `Index` operations together with their surrounding lock calls form atomic blocks like in the center of Fig. 8.13. But as most operations have multiple calls to the `Index`, this is not sufficient to reduce these operations to completely atomic ones. It remains to show that these blocks as well as statements that access the local state of the `Journal` move appropriately (usually they have to be both mover). To prove this, the ownership information of the `FFS-Core` component can be used. The `Journal` is

```
journal_operation#(...)
  interface
{
  ...
  mutex_lock#(; idxlock);
  index_operation_i#(...);
  mutex_unlock#(; idxlock);
  ...
  mutex_lock#(; idxlock);
  index_operation_j#(...);
  mutex_unlock#(; idxlock);
  ...
}
```

```
journal_operation#(...)
  interface
{
  ...
  atomic{
    mutex_lock#(; idxlock);
    index_operation_i#(...);
    mutex_unlock#(; idxlock);
  }
  ...
  atomic{
    mutex_lock#(; idxlock);
    index_operation_j#(...);
    mutex_unlock#(; idxlock);
  }
  ...
}
```

```
journal_operation#(...)
  interface
{
  atomic{
    ...
    mutex_lock#(; idxlock);
    index_operation_i#(...);
    mutex_unlock#(; idxlock);
    ...
    mutex_lock#(; idxlock);
    index_operation_j#(...);
    mutex_unlock#(; idxlock);
    ...
  }
}
```

Figure 8.13: Reduction steps of a `Journal` operation (from left to right).

augmented with ownership properties, operations and preconditions that match those of the `FFS-Core`, and so accesses to the local state can be inferred to be both movers. The information that certain ownership is acquired at the calls of `Index` operations and their associated locking operations allows us to prove that these blocks in fact are movers and, hence, further reduce the operations to be atomic (Fig. 8.13 on the right). Although the proofs are simple, this is quite elaborate since many commutations have to be considered. The data refinement of $\texttt{FFS-Core}_{\texttt{At}}$ to the atomic $\texttt{Journal}_{\texttt{At}}$ then again is basically identical to the sequential refinement. All in all, this guarantees the correctness of the concurrent garbage collection implementation.

Theorem 15 (Correctness of Concurrent Garbage Collection). *The concurrent components* $\texttt{FFS}_{\texttt{Iv}}$ *and* $\texttt{Journal}_{\texttt{Iv}}$ *implementing concurrent garbage collection are a correct, deadlock-free implementation of the atomic* $\texttt{AFS}_{\texttt{At}}$ *specification, i.e., the refinement* $\texttt{FFS}_{\texttt{Iv}}(\texttt{Journal}_{\texttt{Iv}}(\texttt{Index}_{\texttt{Sq}})) \leq \texttt{AFS}_{\texttt{At}}$ *holds.*

□

8.4 Related Work

Verification of concurrent, lock-based systems is of course a very broad topic with lots of important contributions. However, we are not aware of other formal methods that specifically address the question of adding concurrency a posteriori to an existing modular, sequential system, without having to prove the whole system from scratch. Nevertheless, adding concurrency to components of an existing software system to increase efficiency is a recurring software engineering task that should be supported by formal methods.

The method for verifying the correctness of concurrent components presented in the previous chapter, and used to verify the extensions of this chapter, is largely adopted from the work of Pfähler [96]. The original approach of Pfähler added assertions as *program statements* (**assert**) directly to the operation declarations. In this thesis, assertions are given decoupled from declarations in the form of annotations for *labels* identifying statements, which simplifies specification (identical assertions do not need to be added as multiple program statements but can simply be mapped to multiple labels) and improves proof maintenance (altering a declaration invalidates more proofs than changing an annotation).

Rely-guarantee [69, 117] is a commonly used method to reason about concurrent programs from a thread-local perspective. It is used both in Pfähler's and this thesis to establish the assertions given or generated for programs of concurrent programs. Using RG significantly reduces the number of proof obligations that need to be shown compared to automata-based approaches such as the Owicki-Gries-Method [93, 94] or IO-Automata [81], which also require giving concise specifications for all program steps. On the other hand, proof in RG-calculus tend to be relatively cumbersome and produce large goals since the symbolic execution approach accumulates information about previous program states, which have to be propagated (transitively) over rely-steps. Thus, the verification performed in this thesis uses a combination of both approaches: symbolic execution of a program with RG-calculus is "partitioned" into manageable sections by giving complete characterizations of intermediate program states.

The concept of ownership is also used in various other verification systems. For example, the C verification tools VCC [26, 27] and Spec# [67] also use ownership to ensure data race freedom of verified code. These tools typically couple ownership to objects of the programming language while the approach used in this thesis decouples the use of ownership from objects by specifying separate ownership (ghost) state variables, linked to owned objects via rely conditions only. Other verification tools like VeriFast [68], used to verify C and Java programs, use Fractional Permissions [18] to specify and verify similar properties.

Based on Lipton's reduction approach [80] for the verification of parallel programs, the calculus of atomic actions presented by Elmas et al. [37] provides an incremental verification methodology for proving linearizability of concurrent systems. Compared to the adapted approach in this thesis and in [96], [37] provides a more incremental methodology geared towards highly concurrent systems and lock-free algorithms. The approach includes *abstraction* steps into the verification process that transform the initial implementation step-wise into a simplified atomic program. The calculus only addresses partial correctness; divergence- and deadlock-freedom would have to be proven differently. In our approach, on the other hand, divergence- and deadlock-freedom are proved during the rely-guarantee proves establishing assertions. The original program statements remain unchanged while they are gradually grouped into larger atomic blocks. Transforming programs in other ways would require an extra data refinement step; however, our approach currently only supports data refinement of fully atomic components.

There are also some works targeting the verification of concurrency in file systems. Damchoom et al. [30, 29] develop a flash file system by using incremental refinement. Concurrency is verified on a similar level as `AFS` for reading and writing of file content and for wear leveling as well. Synchronization between threads is implicit by the semantics of Event-B [1] models. Events are always executed atomically, which simplifies modeling and verification noticeably but increases the gap to actual running code since no explicit synchronization mechanisms such as locks are necessary.

Chajed et al. [20] adapt the idea of optimistic transactions [76] to add I/O concurrency to the verified FSCQ file system [25, 23]. While the file system itself still runs completely sequential, an additional I/O layer is added at the bottom of the file system that serves as a read cache of the storage device. Calls to this layer assume *optimistically* that the requested data is cached, aborting their execution if it is not. The file system then rolls back its pending transaction while simultaneously, the I/O layer loads the requested data into the cache. The aborted transaction is then retried hoping that all required data was added to the cache in the mean time. [20] reports from an intermediate state of the development where the extension was executable but not yet verified. Since we are not aware of further publications regarding this concurrency extension of FSCQ, it is not apparent whether or how its correctness has been proved.

A very interesting work regarding concurrency on the level of `VFS` is the verification of AtomFS by Zou et al. [119]. AtomFS is a concurrent in-memory file system prototype that is directly programmed in C. It implements the POSIX interface and uses hash tables to store directories and files at an abstraction level similar to our `AFS` specification. Hence, crash-safety is not considered and the implementation is with about 700 lines of code significantly smaller than Flashix. It does, however, use a sophisticated locking scheme in its `VFS` implementation that employs hand-over-hand locking for inodes (lock coupling). Zou et al. proof correctness of the implementation using the theorem prover Coq [10]. A particular challenge for the proof of linearizability was the *rename* operation, which moves directories (whole subtrees). The operation has to lock both the source and target directory but avoid deadlocks. The locking approach could be applied to Flashix as well, where one major task would be to investigate the impact of allowing hard links on the linearizability criterion (this is not covered by AtomFS as hard links are not supported).

Chapter 9

Benchmarking a Verified File System

Summary With most extensions of the second project phase of Flashix, and thus of the contributions of this thesis, targeting efficiency, this chapter provides an insight into how the extensions affect the performance of the Flashix file system. Different workloads representing everyday usage of file systems are run on the generated C code of the Flashix models, integrated into Linux via the FUSE interface. The performance of Flashix in its original, i.e., sequential and non-cached, form is compared with its respective performance after adding caching and concurrency, showing significant improvements.
Furthermore, the performance of Flashix is evaluated against the state-of-the-art flash file system UBIFS, demonstrating competitiveness with real-world (handwritten) implementations. Flashix now covers all performance-relevant concepts, which is reflected in the results; only small deficits remain due to non-optimal code generation.

Publications The evaluation presented in this chapter is based on the publication [13].

To evaluate the performance of the Flashix file system a collection of microbenchmarks were performed. This gives some insight in whether the extensions that were made, especially those described in Chapter 6 and Chapter 8, have an impact on the performance. Furthermore, we want to compare the performance of Flashix with state-of-the-art flash file systems like UBIFS [63].

In order to run Flashix, our code generator produces executable C or Scala code for all implementation components included in the refinement hierarchy (see Fig. 4.1). The generated C implementation, which we are most interested in when considering performance, comprises approximately 18,000 lines of code. Another 2,000 lines are used to integrate Linux via the FUSE (Filesystem in USErspace) library. Algebraic data types are transformed into corresponding C data structures, e.g., doubly-linked lists are used to implement algebraic lists and algebraic maps are realized by hash tables.

All benchmarks were run within a virtualized Linux Mint 19.3 distribution, using 3 Cores of a Intel Core i5-7300HQ CPU and 4,8 GB of RAM. The flash device was simulated in RAM using the NAND simulator (nandsim) integrated into the Linux kernel [87]. The numbers shown in the following represent the mean of 5 benchmark runs in which the mean standard deviation across all runs is below 4.5%

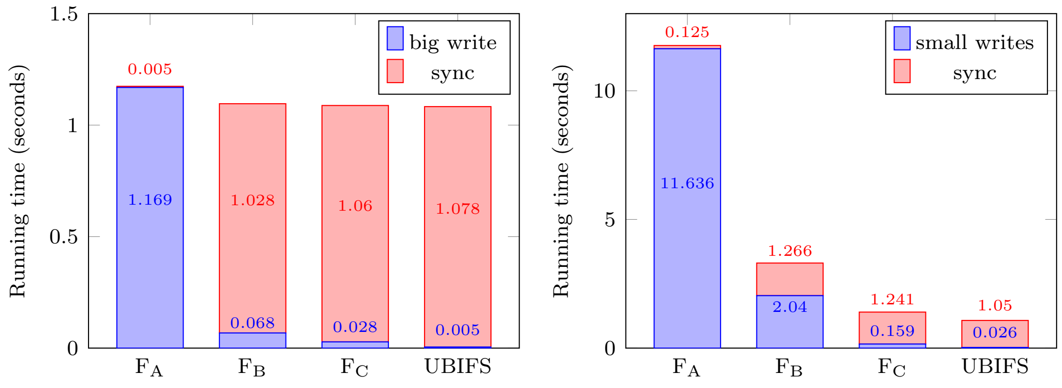


Figure 9.1: Nano write benchmarks on Flashix and UBIFS: big write (left) and small writes (right). Flashix was used in three different configurations: sequentially without VFS cache (F_A), sequentially with VFS cache (F_B), and with VFS cache and concurrency (F_C).

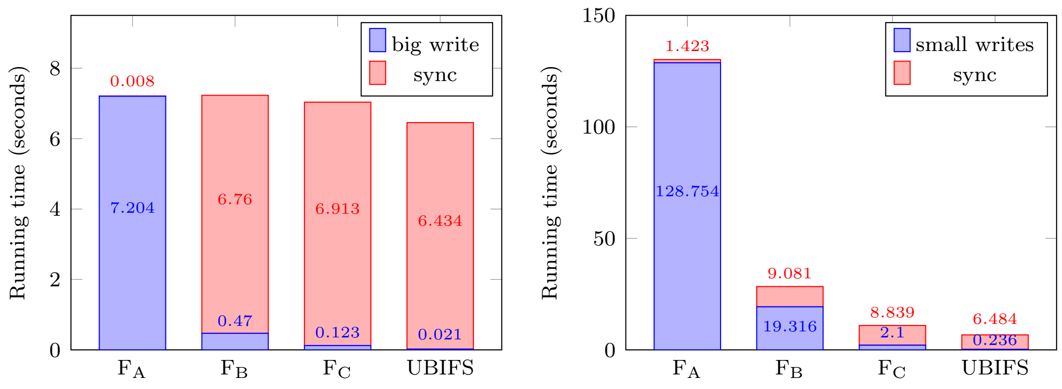
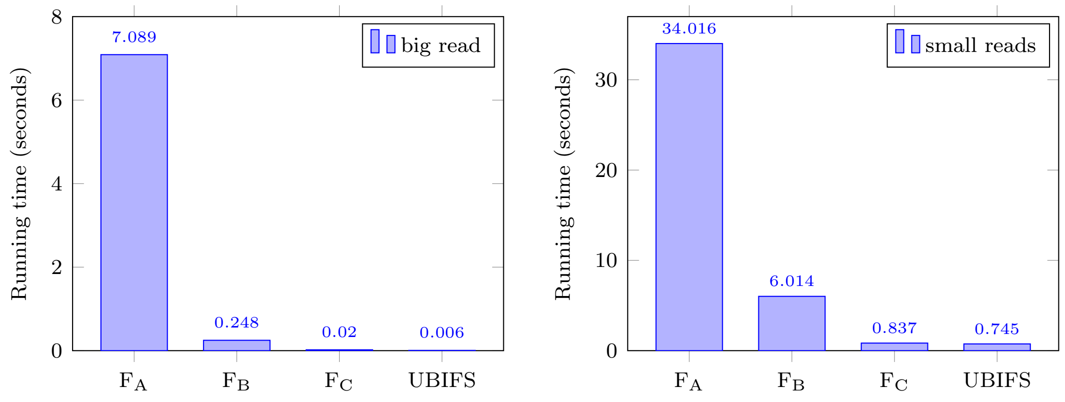


Figure 9.2: Vim write benchmarks on Flashix and UBIFS: big write (left) and small writes (right). Flashix was used in three different configurations: sequentially without VFS cache (F_A), sequentially with VFS cache (F_B), and with VFS cache and concurrency (F_C).

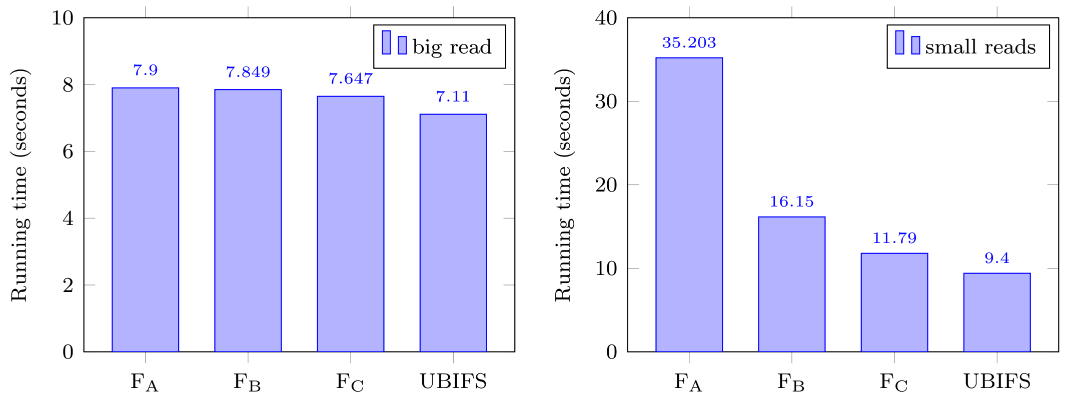
(this translates to a mean deviation in runtime of less than 0.16 seconds).

We chose some small workloads that represent everyday usage of file systems: copying and creating/extracting archives. Copying an archive to the file system results in the creation of a file and writing the content of that one file. Analogously, copying an archive from the file system yields in reading the content of the file. Hence, we call these workloads *big write* and *big read* respectively. On the other hand, extracting an archive results in the creation of a directory structure containing many files. The contents of the created files are written as well, however, these are multiple smaller writes compared to the single big write when copying. Creating an archive from a directory structure on the file system requires to read all directories and files. Hence, we call such workloads *small writes* and *small reads* respectively. As sample data we used archives of the text editors Nano[1] and Vim[2].

[1] nano-2.4.2.tar: approx. 220 elements, 6.7 MB
[2] vim-7.4.tar: approx. 2570 elements, 40.9 MB



(a) Vim read benchmarks with with *hot* caches.



(b) Vim read benchmarks with with *cold* caches.

Figure 9.3: Vim read benchmarks on Flashix and UBIFS: big read (left) and small reads (right). Flashix was used in three different configurations: sequentially without VFS cache (F_A), sequentially with VFS cache (F_B), and with VFS cache and concurrency (F_C).

Fig. 9.1 shows the results of the write benchmarks with Nano. When comparing the uncached configuration (F_A) with the cached configuration (F_B) of Flashix, one can see that adding the `Cache` layer to `VFS` (see Sec. 6.1) has indeed a significant impact on write times (depicted in blue). But as these times do not include persisting the cached data to flash, we enforced synchronization directly afterwards via *fsync* calls (depicted in red). For big writes, the combined runtime of the cached configuration is similar to the uncached one. For small writes though, the combined runtime of F_B is substantially faster since repeated reads to directory and file nodes during path traversal can be handled by the cache.

Moving wear leveling and garbage collection (see Sec. 8.3) into separate threads (F_C) further improves the performance. This is especially noticeable in the small writes workload where the write time can be reduced by about one order of magnitude. In the sequential configurations (F_A and F_B), after each top-level operation it was checked whether garbage collection or wear leveling should be performed. During these checks and potential subsequent executions of the algorithms, other `POSIX`

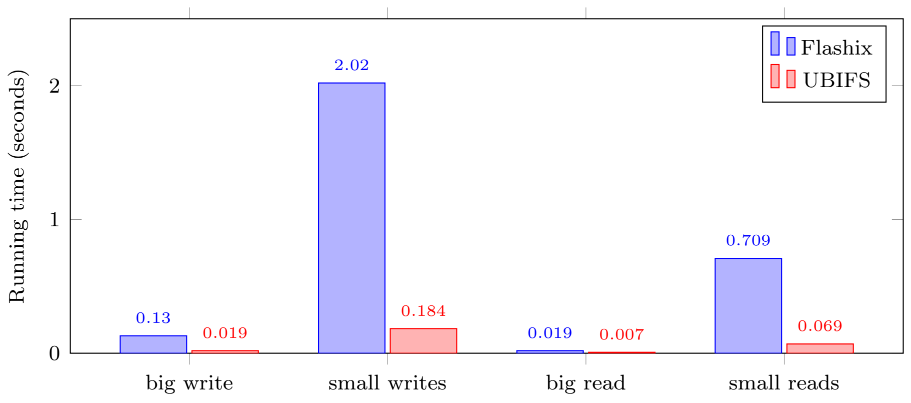


Figure 9.4: Vim benchmarks on Flashix and UBIFS without flash delays.

operation calls were blocked. In the concurrent configuration ($\mathrm{F_C}$) these blocked time can be eliminated for the most part since writing to the cache does not interfere with garbage collection or wear leveling. As small writes trigger considerably more top-level operation calls, this effect is much more noticeable than with big write workloads.

Compared to UBIFS, the current version of Flashix performs as expected. Runtimes of the $\mathrm{F_C}$ configuration are always within the same order of magnitude of those of UBIFS. This also applies for running the benchmarks with a larger archive like Vim shown in Fig. 9.2.

Similar effects can be observed when considering read workloads as shown in Fig. 9.3. Adding caches significantly speeds up both reading a single big file and reading many small files when the caches are *hot* (Fig. 9.3a), i.e., when the requested data is present in the caches. Likewise, moving wear leveling and garbage collection to background processes brings down the runtime by an order of magnitude. When reading from *cold* caches, i.e., when no requested data is present in the caches, the speed up is much more subtle since the main delay results from reading data from flash. As shown in Fig. 9.3b, for big reads there is hardly any improvement from $\mathrm{F_A}$ to $\mathrm{F_B}$ or $\mathrm{F_C}$. However, both expansions have an impact on the runtime of small read workloads for the same reasons as for small write workloads: repeated reads to the directory structure can be handled by the cache, and blocked time for garbage collection and wear leveling can be eliminated. With these additions one can see that Flashix is competitive with UBIFS regarding read performance, too.

Flashix now covers all performance-oriented concepts of realistic file system implementations, which is reflected in the benchmark results. Nevertheless, some minor performance gains could still be achieved by improving our code generator as the generated code is not optimal in allocating/deallocating and copying data structures. The optimization potential becomes apparent when comparing the raw in-memory runtimes of Flashix and UBIFS like in Fig. 9.4. Here we instructed nandsim to not simulate any delays for accessing the simulated NAND memory. The results show that UBIFS is still up to a factor of 10 faster than Flashix for the Vim microbenchmarks (the Nano benchmarks yield similar results). First experiments show, that even simple routines can affect performance noticeably if they are generated inefficiently. For example, we found out that a simple optimization of a routine used in the `Journal` for calculating the required space of a node-list on flash improved the runtime by up to 30% compared to the generated code. Hence, we plan to apply data flow analysis to identify this and other locations where such optimizations can be performed, further closing the gap to state-of-the-art handwritten file systems.

Chapter 10

Summary & Outlook

Summary This chapter summarizes the theoretical and practical results of this thesis, contributing to the formal development of large-scale software systems. The case study of this thesis, the Flashix file system, has evolved significantly due to the results obtained and is now competitive with real-world implementations.

This thesis contributes to the modular development of formally verified software systems. When a software system is developed using formal methods, the primary goal is usually the functional correctness of the resulting system w.r.t. some concise, abstract specification. Hence, the system is designed to be verified easily, which typically includes modularization. Performance or extensibility are often secondary aspects, but quickly become relevant when the system competes with realistic implementations.

This thesis has investigated three prominent concepts used to improve the performance of software systems: destructive heap-based algorithms, caching, and concurrency. It contributes methodologies for facilitating modular verification and integrating them into existing verified component hierarchies while retaining large parts of the original specification and proof work. The methodologies were applied to the Flashix case study, yielding an efficient and realistic flash file system implementation.

Verification of Destructive Heap Algorithms For the verification of efficient destructive algorithms on pointer structures, a modularization concept is given that separates different verification concerns into multiple refinements (Chapter 3): the functional correctness of algorithms can be proved solely on algebraic data types, while complex reasoning about pointer structures (for example, by employing Separation Logic) is limited to fine-grained operations.

The concept is applied to a destructive implementation of red-black trees. Two data refinements are proven, starting from an abstract specification of sets, over an algebraic representation of trees, to a heap-based pointer implementation. Correctness of the resulting implementation is shown by Thm. 4.

Theorem 4 (Correctness of Destructive Red-Black Trees). *The sequential component* `RBTree(RBTHeap)` *is a correct, memory-safe destructive implementation of red-black trees.*

The verified red-black tree implementation is now used for several purposes in Flashix, e.g., for the management of free and used erase blocks in the `EBM`. This extension reduces the untrusted code base of Flashix as an unverified external library was used before.

Besides red-black trees, we think that the modularization concept can also be beneficial to the verification of other pointer data structures. For example, the verification of wandering B+ trees using this approach is currently in progress.

Integration and Correctness of Caching Caching is one of the standard mechanisms to reduce the (perceived) delay of storage-based systems. This thesis provides an implementation and verification strategy for the post hoc integration of caches into a software system (Chapter 6): a cache layer can be added between two existing layers as a *decorator* without adapting or re-validating surrounding components. The functional correctness of the addition can be shown by just one additional refinement proof.

The Flashix file system is extended with a corresponding layer for caching high-level data structures, integrated between the Virtual File System Switch `VFS` and the abstract interface `AFS` of the flash file system. Thm. 7 guarantees the correct decorator integration of the `Cache` component.

Theorem 7 (Functional Correctness of High-Level Caches). *The sequential component* `Cache` *is a correct cache decorator implementation of the component* `AFS`*, i.e.,* `Cache(AFS)` $\leq$ `AFS` *holds.*

The adaption of the decorator pattern is not restricted to this specific application but can be used as a universal technique to support formal development. For example, the approach for modeling and verifying durable opacity for software transactional memory (STM) in [11] uses a similar approach (applied to IO-Automata): an initial specification `A` performs a transaction on persistent memory and is then refined by an implementation `C` that buffers transactions until they can be written entirely to persistent storage specified by `A`.

Crash-Safety of Caching Crash-Safety is a fundamental concern of storage-based systems, particularly when caching mechanisms are employed. File systems, however, often only provide unsatisfactory guarantees in the event of a crash/power cut (e.g., the POSIX standard [100] enforces guarantees only for fully synchronized operations). This thesis contributes a novel crash-safety criterion *Write-Prefix Crash Consistency* for cached file systems with Def. 39 (Chapter 6).

Definition 39 (Write-Prefix Crash Consistency (WPCC)). *A file system is write-prefix crash consistent iff it refines the* `POSIX` *component extended by a synchronization operation, and each crash transition is a write-prefix crash transition.*

The criterion is based on the history-based definition of *write-prefix crash transition* given by Def. 38, which reduces the crash effect to individual files by restricting file histories to writing prefixes of their original executions. The criterion is motivated by the natural way of synchronizing files: cached file content is persisted from low to high pages, imitating a POSIX write operation (which is also the default strategy of the Linux VFS).

For the Flashix file system, a reference implementation is given with the `Cache` component realizing synchronization this way. A strategy for proving the compliance to WPCC is presented and applied to the case study, guaranteeing crash-safety for Flashix, as shown with Thm. 8.

Theorem 8 (Flashix satisfies WPCC). *The Flashix file system with the* `VFS` *implementation, extended by the* `Cache` *component, satisfies WPCC.*

WPCC is a universal correctness criterion for file systems, not specific to flash memory. It gives stronger guarantees than other related criteria, e.g., the *metadata-prefix* specification of Chen et al. [23, 24]. Thus, WPCC gives applications new ways to access the file system in a crash-safe way, e.g., check-sums written before the actual data can be used to detect writes that were not persisted completely.

Cache implementations compliant with WPCC are also compatible with low-level buffer caches that imply the crash-safety criterion *Quasi-Sequential Crash Consistency* introduced by Pfähler [96], as shown by Thm. 9.

Theorem 9 (Compatibility of WPCC and QSCC). *Write-Prefix Crash Consistency is compatible with the use of order-preserving caches in lower levels of the implementation that satisfy Quasi-Sequential Crash Consistency.*

Introducing Concurrency to Sequential Systems Concurrent systems are often designed from the ground up for correct parallel execution of operations. Hence, adding concurrency to a sequential system a posteriori typically requires significant effort. This thesis presents an approach that allows introducing concurrency to specific layers of a sequential refinement hierarchy by moving internal algorithms to separate threads or permitting external parallel calls to the system interface (Chapter 7 and Chapter 8).

This can be done while retaining large parts of the original specifications and proofs: the sequential implementation is augmented with suitable locking instructions guaranteeing *ownership* of parts of the shared state, which is propagated to lower layers via ownership operations and preconditions. Correctness of the adaption is proved by an additional *strong linearizability* proof using *atomicity refinement*, abstracting the concurrent implementation to an atomic version. Since this atomic version is basically identical to the original sequential implementation, existing data refinement proofs can be carried over with little effort.

This thesis contributes several realizations of concurrency concepts for file systems (Chapter 8). An efficient mechanism for managing dynamically allocated locks, i.e., locks for shared objects that can be created and destroyed during runtime of the system, is presented, which avoids unnecessary blocking of other threads. Thm. 12 guarantees the correctness of this mechanism, separated as a modularly usable component `CLocking`.

Theorem 12 (Correctness of Concurrent Lock Management). *The concurrent component* `CLocking` *is a correct, deadlock-free implementation of the atomic specification* `ALocking`*, i.e.,* $\texttt{CLocking}_{\texttt{Iv}} \leq \texttt{ALocking}_{\texttt{At}}$ *holds.*

While the mechanism can be employed universally, in Flashix, the component is used to implement a concurrent implementation of `VFS`, the top layer of the file system. It facilitates an implementation that permits concurrent reads and writes to file contents by locking files individually. The correctness of this approach is ensured by Thm. 13.

Theorem 13 (Correctness of the Concurrent Virtual File System Switch). *The concurrent implementation of* `VFS` *is a correct, deadlock-free implementation of the sequential* `POSIX` *specification, i.e.,* $\texttt{VFS}_{\texttt{Iv}}(\texttt{AFS}_{\texttt{At}}) \leq \texttt{POSIX}_{\texttt{Sq}}$ *holds.*

Since this concurrency extension creates calls of interleaved operations to `Cache`, compatibility of WPCC with concurrency must be considered. This thesis provides a compliant implementation and locking scheme for `VFS` and `Cache`, as shown by Thm. 14.

Theorem 14 (Compatibility of WPCC and with the Concurrent Virtual File System Switch). *Write-Prefix Crash Consistency is compatible with the concurrent implementation* $\texttt{VFS}_{\texttt{Iv}}$ *of the Virtual File System Switch.*

Additional internal concurrency was introduced in Flashix with a concurrent variant of *garbage collection* in the `FFS` and `Journal` layers. Thus, the administrative work of garbage collection is moved to the background so that users do not have to wait for it to be completed. Thm. 15 guarantees correct integration of this concurrent algorithm in the hierarchy.

Theorem 15 (Correctness of Concurrent Garbage Collection). *The concurrent components* $\texttt{FFS}_{\texttt{Iv}}$ *and* $\texttt{Journal}_{\texttt{Iv}}$ *implementing concurrent garbage collection are a correct, deadlock-free implementation of the atomic* $\texttt{AFS}_{\texttt{At}}$ *specification, i.e., the refinement* $\texttt{FFS}_{\texttt{Iv}}(\texttt{Journal}_{\texttt{Iv}}(\texttt{Index}_{\texttt{Sq}})) \leq \texttt{AFS}_{\texttt{At}}$ *holds.*

The verification methodology for concurrent components used in this thesis is one of the key factors for successfully extending the refinement hierarchy with concurrency. Verifying correctness by abstracting a concurrent component to an atomic one first (*atomicity refinement*) and then re-using the results of sequential reasoning (*data refinement*) is much more feasible than proving linearizability simultaneously with functional correctness.

The approach is not restricted to file systems but can be used universally to verify concurrent systems. It works best for lock-based concurrency with a strict hierarchical locking strategy, as it is used in the Flashix file system. Support for other types of concurrency could still be improved, e.g., the CAS (compare-and-swap) instruction used frequently in lock-free algorithms is not supported at the moment. Verifying more fine-grained locking strategies where locking ranges overlap, like hand-over-hand locking used by Zou et al. [119], is currently quite cumbersome and requires rather much manual proof work. Future work could explore how the methodology could be extended for better automation.

Verification Under Adverse Conditions Verifying large-scale software systems under *adverse conditions*, like when using caching and concurrency, turns out to be a major challenge. When integrating these concepts into existing systems, proving the correctness of these additions is often significantly more laborious than the original verification of the respective components, particularly when considering crashes. For example, proving crash-safety of the caching addition (Chapter 6) was noticeably harder than proving its functional correctness (crash-safety required about ten times more interactions than the proofs for functional correctness). As another example, the core verification of concurrent garbage collection (Chapter 8) was much more extensive than the sequential verification (approx. 90,000 proof steps compared to 18,000 steps). For managing these complexities, rigorous specification and proof engineering efforts were inevitable.

Tool Support In the course of this thesis, tooling support was also improved. The KIV proof system has undergone various enhancements, further improving the support for the proof engineer and increasing the trust in systems verified by KIV. For example, the KIV logic was extended by a polymorphic type system (Chapter 2) that reduces the specification overhead for instantiable data types noticeably. Furthermore, exceptions were integrated into the programming language and the calculus so that the absence of runtime exceptions is verified directly by program proofs (Chapter 2 and (Chapter 7). Rely-guarantee and reduction proofs with KIV are improved by enabling to give assertions as annotations for labeled statements (Chapter 7), which are also used for the generation of proof obligations. The latter has turned out to be another crucial factor for the successful development of large-scale systems. The methods used in this thesis are supported directly by KIV, e.g., there are distinct specification types of defining sequential and concurrent components or refinements, which all generate proof obligations that guarantee correctness automatically. These mechanisms also find regular application in various small- to mid-sized case studies outside of Flashix, for example, in competitions of the VerifyThis series [115].

A Realistic Flash File System Implementation As a result of this thesis, the Flashix file system is the first fully verified and crash-safe file system for flash memory meeting realistic standards. It covers all safety-critical and performance-oriented aspects of real-world file system implementations. It is thus competitive with standard implementations like UBIFS, which are established in the operating system environment (Chapter 9). As discovered in the course of this work, there is still some potential for performance gains due to the limitations of the code generator used. This insight led to a follow-up project that examines how code generation can be improved, e.g., by applying data flow analysis techniques, so that the resulting code is more efficient in sharing and copying data structures.

Bibliography

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 2005.

[2] S. Amani. *A Methodology for Trustworthy File Systems.* PhD thesis, University of New South Wales, Sydney, Australia, 2016.

[3] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. Cogent: Verifying High-Assurance File System Implementations. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–188. Association for Computing Machinery (ACM), 2016.

[4] S. Amani and T. Murray. Specifying a Realistic File System. In *Proc. of Workshop on Models for Formal Analysis of Real Systems*, volume 196 of *Electronic Proc. in Theoretical Computer Science*, pages 1–9. Open Publishing Association, 2015.

[5] A. Appel. Efficient Verified Red-Black Trees. Princeton University, 2011.

[6] K. Arkoudas, K. Zee, V. Kuncak, and M. C. Rinard. Verifying a File System Implementation. In *Proc. of International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 373–390. Springer, 2004.

[7] L. Armborst and M. Huisman. Permission-Based Verification of Red-Black Trees and Their Merging. In *Proc. of International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 111–123. IEEE, 2021.

[8] P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees, 20005.

[9] M. Ben-Ari. The Bug That Destroyed a Rocket. *Special Interest Group on Computer Science Education (SIGCSE) Bulletin*, 33(2):58–59, 2001.

[10] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* Springer, 1st edition, 2010.

[11] E. Bila, J. Derrick, S. Doherty, B. Dongol, G. Schellhorn, and WehrheimH. Modularising Verification of Durable Opacity. *Logical Methods in Computer Science (LMCS)*, 2022. to appear.

[12] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Proc. of International Conference on Integrated Formal Methods (IFM)*, volume 10510 of *LNCS*, pages 102–110. Springer, 2017.

[13] S. Bodenmüller, G. Schellhorn, M. Bitterlich, and W. Reif. Flashix: Modular Verification of a Concurrent and Crash-Safe Flash File System. In *Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday*, volume 12750 of *LNCS*, pages 239–265. Springer, 2021.

[14] S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Integration of Crashsafe Caching into a Verified Virtual File System Switch. In *Proc. of International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *LNCS*, pages 218–236. Springer, 2020.

[15] S. Bodenmüller, G. Schellhorn, and W. Reif. Verification of Crashsafe Caching in a Virtual File System Switch. *Formal Aspects of Computing (FAC)*, 34(1), 2022.

[16] E. Börger and R. F. Stärk. *Abstract State Machines — A Method for High-Level System Design and Analysis.* Springer, 2003.

[17] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and Checking File System Crash-Consistency Models. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 83–98. Association for Computing Machinery (ACM), 2016.

[18] J. Boyland. Checking Interference with Fractional Permissions. In *Proc. of International Static Analysis Symposium (SAS)*, pages 55–72. Springer, 2003.

[19] A. Butterfield and J. Woodcock. Formalising Flash Memory: First Steps. In *Proc. of IEEE International Conference on Engineering Complex Computer Systems (ICECCS)*, pages 251–260. IEEE, 2007.

[20] T. Chajed, A. Chlipala, M. Kaashoek, and N. Zeldovich. Extending a Verified File System with Concurrency. In *Student Research Competition at Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery (ACM), 2017.

[21] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich. Verifying Concurrent, Crash-Safe Systems with Perennial. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, pages 243–258. Association for Computing Machinery (ACM), 2019.

[22] A. Charguéraud. Program verification through characteristic formulae. In *Proc. of ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 321–332. Association for Computing Machinery (ACM), 2010.

[23] H. Chen. *Certifying a Crash-Safe File System.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, United States, 2016.

[24] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleriy, A. Chlipala, M. Kaashoek, and N. Zeldovich. Verifying a High-Performance Crash-Safe File System Using a Tree Specification. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, pages 270–286. Association for Computing Machinery (ACM), 2017.

[25] H. Chen, D. Ziegler, A. Chlipala, N. Zeldovich, and M. F. Kaashoek. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, pages 18–37. Association for Computing Machinery (ACM), 2015.

[26] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proc. of International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 23–42. Springer, 2009.

[27] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local Verification of Global Invariants in Concurrent Programs. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 480–494. Springer, 2010.

[28] Cormen, T. and Leiserson, C. and Rivest, R. and Stein, C. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009.

[29] K. Damchoom and M. Butler. Applying Event and Machine Decomposition to a Flash-Based Filestore in Event-B. In *Proc. of Brazilian Symposium on Formal Methods (SBMF)*, volume 5902 of *LNCS*, pages 134–152. Springer, 2009.

[30] K. Damchoom, M. Butler, and J. R. Abrial. Modelling and Proof of a Tree-Structured File System in Event-B and Rodin. In *Proc. of International Conference on Formal Engineering Methods (ICFEM)*, volume 5256 of *LNCS*, pages 25–44. Springer, 2008.

[31] W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Non-Compositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2012.

[32] W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1998.

[33] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall Series in Automatic Computation. Prentice Hall, 1976".

[34] C. Dross and Y. Moy. Auto-Active Proof of Red-Black Trees in SPARK. In *Proc. of NASA Formal Methods (NFM)*, volume 10227 of *LNCS*, pages 68–83. Springer, 2017.

[35] Eclipse Website. Eclipse Foundation. URL: `https://www.eclipse.org/`.

[36] J. Elgaard, A. Møller, and M. I. Schwartzbach. Compile-Time Debugging of C Programs Working on Trees. In *Proc. of European Symposium on Programming (ESOP)*, volume 1782 of *LNCS*, pages 182–194. Springer, 2000.

[37] T. Elmas, S. Qadeer, and S. Tasiran. A Calculus of Atomic Actions. In *Proc. of Principles of Programming Languages (POPL)*, pages 2–15. Association for Computing Machinery (ACM), 2009.

[38] G. Ernst. *A Verified POSIX-Compliant Flash File System-Modular Verification Technology & Crash Tolerance*. PhD thesis, University of Augsburg, Augsburg, Germany, 2017.

[39] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV - Overview and VerifyThis Competition. *Software Tools for Technology Transfer (STTT)*, 17(6):677–694, 2015.

[40] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a Verified Flash File System: Transactions & Garbage Collection. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 9593 of *LNCS*, pages 73–93. Springer, 2015.

[41] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. A Formal Model of a Virtual Filesystem Switch. In *Proc. of Software and Systems Modeling (SSV)*, volume 102 of *EPTCS*, pages 33–45. Open Publishing Association, 2012.

[42] G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *LNCS*, pages 242–261. Springer, 2013.

[43] M. A. Ferreira, S. S. Silva, and J. N. Oliveira. Verifying Intel Flash File System Core Specification. In *Modelling and Analysis in VDM: Proceedings of the Fourth VDM/Overture Workshop*, pages 54–71. School of Computing Science, Newcastle University, 2008. Technical Report CS-TR-1099.

[44] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In *Proc. of European Symposium on Programming (ESOP)*, volume 2986 of *LNCS*, pages 370–384. Springer, 2004.

[45] L. Freitas, J. Woodcock, and A. Butterfield. POSIX and the Verification Grand Challenge: A Roadmap. In *Proc. of IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 153–162. IEEE, 2008.

[46] L. Freitas, J. Woodcock, and Z. Fu. POSIX file store in Z/Eves: An Experiment in the Verified Software Repository. *Science of Computer Programming (SCP)*, 74(4):238–257, 2009.

[47] A. Galloway, G. Lüttgen, J. T. Mühlberg, and R.I. Siminiceanu. Model-Checking the Linux Virtual File System. In *Proc. of International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403 of *LNCS*, pages 74–88. Springer, 2009.

[48] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

[49] G. Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39:176–210, 1935.

[50] G. Gentzen. Untersuchungen über das logische Schließen II. *Mathematische Zeitschrift*, 39:405–431, 1935.

[51] T. Gleixner, F. Haverkamp, and A. Bityutskiy. UBI - Unsorted Block Images. 2006. URL: `http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf`.

[52] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[53] S. Gouw, F. S. Boer, R. Bubel, R. Hähnle, J. Rot, and D. Steinhöfel. Verifying OpenJDK's Sort Method for Generic Collections. *Journal of Automated Reasoning*, 62(1):93–126, 2019.

[54] L. J. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proc. of Symposium on Foundations of Computer Science (SFCS)*, pages 8–21. IEEE, 1978.

[55] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex Electronic Purses with KIV: From Transactions to a Security Protocol. *Formal Aspects of Computing (FAC)*, 20:41–59, 2008.

[56] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*, pages 99–217. Springer, 2002.

[57] J. He, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In *Proc. of European Symposium on Programming (ESOP)*, volume 213 of *LNCS*, pages 187–196. Springer, 1986.

[58] M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In *Logical Foundations of Computer Science*, volume 363 of *LNCS*, pages 134–145. Springer, 1989.

[59] M. P. Herlihy and N. Shavit. On the Nature of Progress. In *Proc. of International Conference On Principles Of Distributed Systems (OPODIS)*, volume 7109 of *LNCS*, pages 313–328. Springer, 2011.

[60] M. P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[61] W. H. Hesselink and M. I. Lali. Formalizing a Hierarchical File System. *Formal Aspects of Computing (FAC)*, 24(1):27–44, 2012.

[62] C.A.R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50(1):63–69, 2003.

[63] A. Hunter. A Brief Introduction to the Design of UBIFS. 2008. URL: `http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf`.

[64] A. Ileri, T. Chajed, A. Chlipala, F. Kaashoek, and N. Zeldovich. Proving Confidentiality in a File System Using DiskSec. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 323–338. USENIX Association, 2018.

[65] IntelliJ IDEA Website. JetBrains. URL: `https://www.jetbrains.com/idea/`.

[66] J. Izraelevitz, H. Mendes, and M. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proc. of International Symposium on Distributed Computing (DISC)*, volume 9888 of *LNCS*, pages 313–327. Springer, 2016.

[67] B. Jacobs, R. Leino, F. Piessens, and W. Schulte. Safe Concurrency for Aggregate Objects with Invariants. In *Proc. of IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 137–147. IEEE, 2005.

[68] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *Proc. of NASA Formal Methods (NFM)*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

[69] C. B Jones. *Development Methods for Computer Programs Including a Notion of Interference.* Oxford University Computing Laboratory, 1981.

[70] C. B. Jones. Specification and Design of (Parallel) Programs. In *Proc. of IFIP Congress*, pages 321–332. North-Holland, 1983.

[71] C. B. Jones and M. Thomas. The Development and Deployment of Formal Methods in the UK. *Formal Aspects of Computing (FAC)*, 2022. just accepted.

[72] R. Joshi and G.J. Holzmann. A Mini Challenge: Build a Verifiable Filesystem. *Formal Aspects of Computing (FAC)*, 19(2):269–272, 2007.

[73] E. Kang and D. Jackson. Formal Modelling and Analysis of a Flash Filesystem in Alloy. In *Proc. of International Conference Abstract State Machines, B and Z (ABZ)*, volume 5238 of *LNCS*, pages 294 – 308. Springer, 2008.

[74] KIV Website. Institue for Software & Systems Engineering – University of Augsburg. URL: `https://kiv.isse.de`.

[75] S. Krishna, N. Patel, D. Shasha, and T. Wies. Verifying Concurrent Search Structure Templates. In *Proc. of Conference on Programming Language Design and Implementation (PLDI)*, pages 181–196. Association for Computing Machinery (ACM), 2020.

[76] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[77] M. I. Lali. File System Formalization: Revisited. *International Journal of Advanced Computer Science (IJACS)*, 3(12):602–606, 2013.

[78] G. Le Lann. An Analysis of the Ariane 5 Flight 501 Failure - a System Engineering Perspective. In *Proc. of International Conference and Workshop on Engineering of Computer-Based Systems (ECBS)*, pages 339–346. IEEE, 1997.

[79] C. Lehner. Design und Verifikation einer effizienten, modularen Implementierung von Dateisystem-Bäumen. Master thesis, University of Augsburg, Augsburg, Germany, 2021.

[80] R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications of the ACM*, 18(12):717–721, 1975.

[81] N. Lynch and F. Vaandrager. Forward and Backward Simulations – Part I: Untimed Systems. *Information and Computation*, 121(2):214–233, 1995.

[82] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 1st edition, 2003.

[83] C. Morgan and B. Sufrin. Specification of the UNIX Filing System. volume SE-10, pages 128–142. IEEE, 1984.

[84] B. Moszkowski and Z. Manna. Reasoning in Interval Temporal Logic. In *Logics of Programs*, pages 371–382. Springer, 1984.

[85] J.T. Mühlberg and G. Lüttgen. Verifying Compiled File System Code. *Formal Aspecpts of Computing (FAC)*, 24(3):375–391, 2012.

[86] M. Müller, P.and Schwerhoff and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Proc. of International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer, 2016.

[87] Linux MTD: NAND and NAND simulator. URL: `http://www.linux-mtd.infradead.org/faq/nand.html`.

[88] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, 2015.

[89] T. Nipkow. Automatic Functional Correctness Proofs for Functional Search Trees. In *Proc. of 7th International Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 307–322. Springer, 2016.

[90] NLnet Labs Name Server Daemon. NLnet Labs. URL: `https://nlnetlabs.nl/projects/nsd/about`.

[91] L. O'Connor, Z. Chen, C. Rizkallah, V. Jackson, S. Amani, G. Klein, T. Murray, T. Sewell, and G. Keller. Cogent: Uniqueness Types and Certifying Compilation. *Journal of Functional Programming (JFP)*, 31, 2021.

[92] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala.* Artima Incorporation, 3rd edition, 2016.

[93] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6(4):319–340, 1976.

[94] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, 1976.

[95] R. Peña. An Assertional Proof of Red–Black Trees Using Dafny. *Journal of Automated Reasoning*, 64:767–791, 2020.

[96] J. Pfähler. *A Modular Verification Methodology for Caching and Lock-Based Concurrency in File Systems.* PhD thesis, University of Augsburg, Augsburg, Germany, 2018.

[97] J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular Verification of Order-Preserving Write-Back Caches. In *Proc. of International Conference on Integrated Formal Methods (IFM)*, volume 10510 of *LNCS*, pages 375–390. Springer, 2017.

[98] J. Pfähler, G. Ernst, G. Schellhorn, D. Haneberg, and W. Reif. Formal specification of an Erase Block Management Layer for Flash Memory. In *Proc. of Haifa Verification Conference (HVC)*, volume 8244 of *LNCS*, pages 214–229. Springer, 2013.

[99] A. Pnueli. The Temporal Logic of Programs. In *Proc. of Annual Symposium on Foundations of Computer Science (SFCS)*, pages 46–57. IEEE, 1977.

[100] The Open Group Base Specifications Issue 7, IEEE Std 1003.1, 2018 Edition. The IEEE and The Open Group, 2017. URL: `https://pubs.opengroup.org/onlinepubs/9699919799`.

[101] G. Reeves and T. Neilson. The Mars Rover Spirit FLASH Anomaly. In *Proc. of IEEE Aerospace Conference*, pages 4186–4199. IEEE, 2005.

[102] J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[103] C. Rizkallah, J. Lim, Y. Nagashima, T. Sewell, Z. Chen, L. O'Connor, T. Murray, G. Keller, and G. Klein. A Framework for the Automatic Formal Verification of Refinement from Cogent to C. In *Proc. of International Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 323–340. Springer, 2016.

[104] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual (2nd Edition).* Pearson Higher Education, 2004.

[105] G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In *Automated Deduction — A Basis for Applications: Volume III Applications*, pages 165–194. Springer, 1998.

[106] G. Schellhorn, S. Bodenmüller, M. Bitterlich, and W. Reif. Separating Separation Logic – Modular Verification of Red-Black Trees. In *Proc. of International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, volume 13800 of *LNCS*, pages 129–147. Springer, 2022.

[107] G. Schellhorn, S. Bodenmüller, M. Bitterlich, and W. Reif. Software & System Verification with KIV. In *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, volume 13360 of *LNCS*, pages 408–436. Springer, 2022.

[108] G. Schellhorn, S. Bodenmüller, J. Pfähler, and W. Reif. Adding Concurrency to a Sequential Refinement Tower. In *Proc. of International Conference on Rigorous State-Based Methods (ABZ)*, volume 12071 of *LNCS*, pages 6–23. Springer, 2020. Invited Paper.

[109] G. Schellhorn, G. Ernst, J. Pfähler, S. Bodenmüller, and W. Reif. Symbolic Execution for a Clash-Free Subset of ASMs. *Science of Computer Programming (SCP)*, 158:21–40, 2018.

[110] G. Schellhorn, B. Tofan, G. Ernst, J. Pfähler, and W. Reif. RGITL: A Temporal Logic Framework for Compositional Reasoning about Interleaved Programs. *Annals of Mathematics and Artificial Intelligence (AnMAI*, 71:131–174, 2014.

[111] A. Schierl, G. Schellhorn, D. Haneberg, and W. Reif. Abstract Specification of the UBIFS File System for Flash Memory. In *Proc. of Formal Methods (FM)*, volume 5850 of *LNCS*, pages 190–206. Springer, 2009.

[112] R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley, 4th edition, 2011.

[113] D. W. Song, K. Mamouras, A. Chen, N. Dautenhahn, and D. S. Wallach. The Design and Implementation of a Verified File System with End-to-End Data Integrity, 2020. URL: `https://arxiv.org/abs/2012.07917`.

[114] Unified Modeling Language (UML), Version 2.5.1. Object Management Group, 2017. URL: `http://www.omg.org/spec/UML/2.5.1`.

[115] VerifyThis Competition Website. ETH Zürich. URL: `https://www.pm.inf.ethz.ch/research/verifythis.html`.

[116] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4), 2009.

[117] Q. Xu, W.-P. de Roever, and J. He. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing (FAC)*, 9(2):149–174, 1997.

[118] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. volume 24, pages 393–423. Association for Computing Machinery (ACM), 2006.

[119] M. Zou, H. Ding, D. Du, M. Fu, R. Gu, and H. Chen. Using Concurrent Relational Logic with Helpers for Verifying the AtomFS File System. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, pages 259–274. Association for Computing Machinery (ACM), 2019.